Imperial College London

Student Name

____

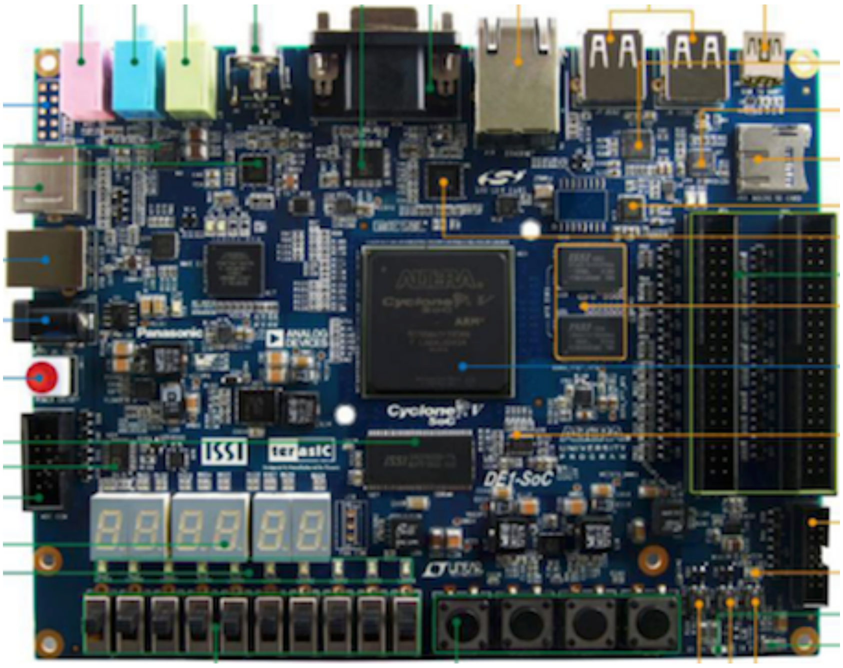
Department of Electrical and Electronic Engineering

Second Year Electronics Laboratory

EXPERIMENT VERI

Digital Design with FPGA and Verilog

11th November – 6th December 2019



V4.3 - PYK Cheung, 5 Nov 2019

Department of Electrical & Electronic Engineering

Imperial College London

2^{nd} Year Laboratory

Experiment VERI: FPGA Design with Verilog (Part 1)

(webpage: www.ee.ic.ac.uk/pcheung/teaching/E2_Experiment /)

Objectives

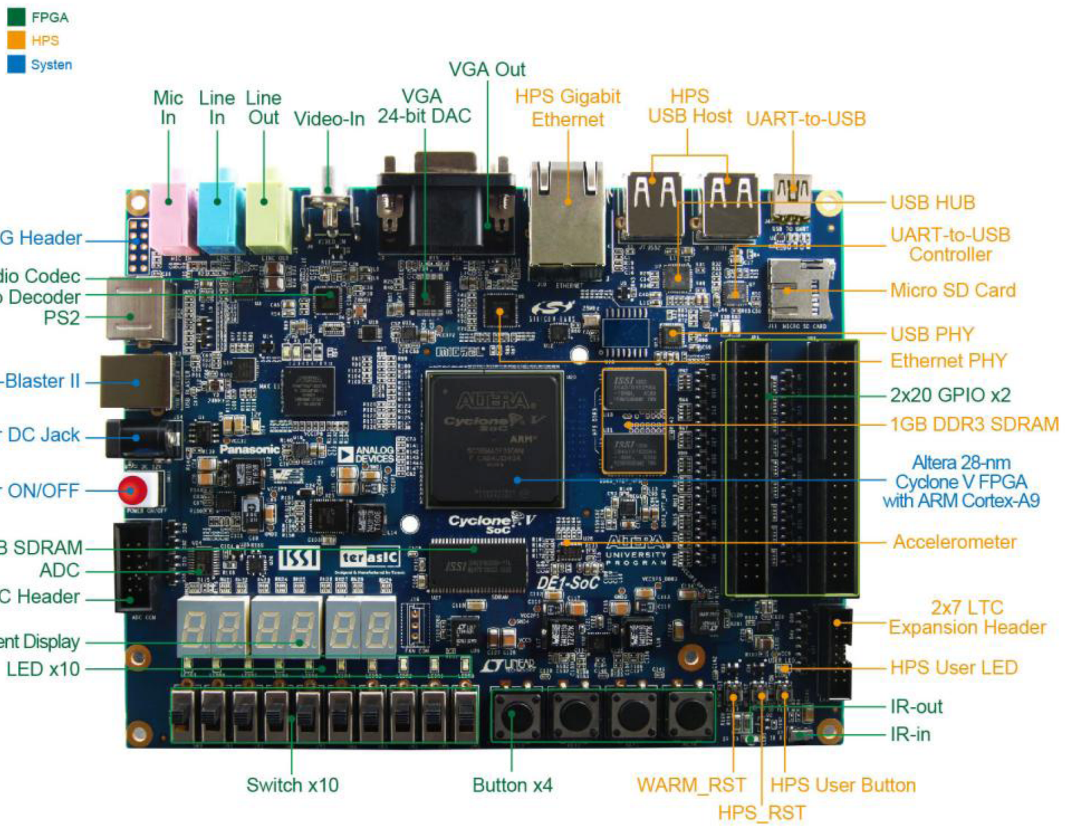
By the end of this experiment, you should have learned:

- How to design digital circuits using Altera's Quartus Prime Design software;
- How to design digital circuits targeting Altera's Cyclone V FPGA using Terasic's DE1-SoC Board;
- How to design digital circuits in efficient, synthesizable Verilog HDL;
- How to evaluate your design in terms of resource utilization and clock speed;
- How to use the DE1-SoC FPGA board with its custom daughter board for analogue I/O functions;
- Have designed something yourself for the Cyclone V FPGA.

Before you start

Before you come to the laboratory, you are expected to:

- Have understood the lectures on Verilog
- Be familiar with the basic architecture inside the FPGA
- Have read through this Laboratory instructions



Both the experimental board and a PC would be made available to you during your allotted period in the second year laboratory. In addition, you may also borrow a DE1-SoC board from Level 1 stores to use at home for one week. If no one else has booked the board, you can renew the borrowing period on a week-by-week basis.

This instruction manual is divided into four parts, one for each week. Each part has its own goals and learning outcomes.

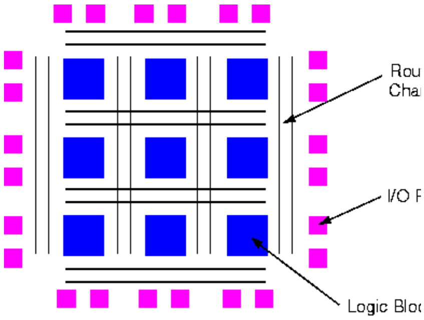
Some students will find this experiment harder or easier than average, depending on your prior experience with digital logic. Therefore the four Lab sessions contain the compulsory s well as optional exercisers. If you fall behind this experiment during any week, it is wise to find a bit of spare time to catch up outside the official laboratory sessions and restrict yourself to the compulsory parts only.

PART I – Schematic to Verilog

1.0 Introduction

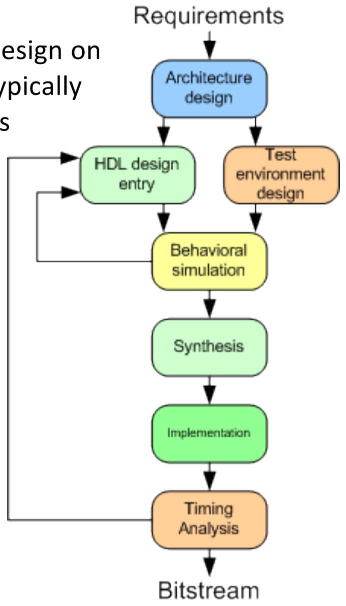
You should have done some background reading before attending the laboratory session as suggested at the Lecture.

FPGAs is a type of programmable logic devices introduced by Xilinx in 1985. It is now the predominant technology for implementing digital logic in low to moderate volume production. The basic structure of an FPGA is shown below. It consists of three main types of resources: 1) Logic Blocks (or Elements); 2) Routing Resources; 3) I/O Pad. For more information about FPGA, see Lecture 1 notes available on the E2 Digital Electronics course webpage.



1.1 Quartus Prime Design Suite

Quartus provides a complete environment for you to implement your design on an Altera FPGA. It supports all aspects of the design flow, which is typically following the flow diagram shown here. The best way to learn Quartus is to go through this experiment step-by-step. After you have learned the basics, you can start to explore other aspects of the Quartus system.



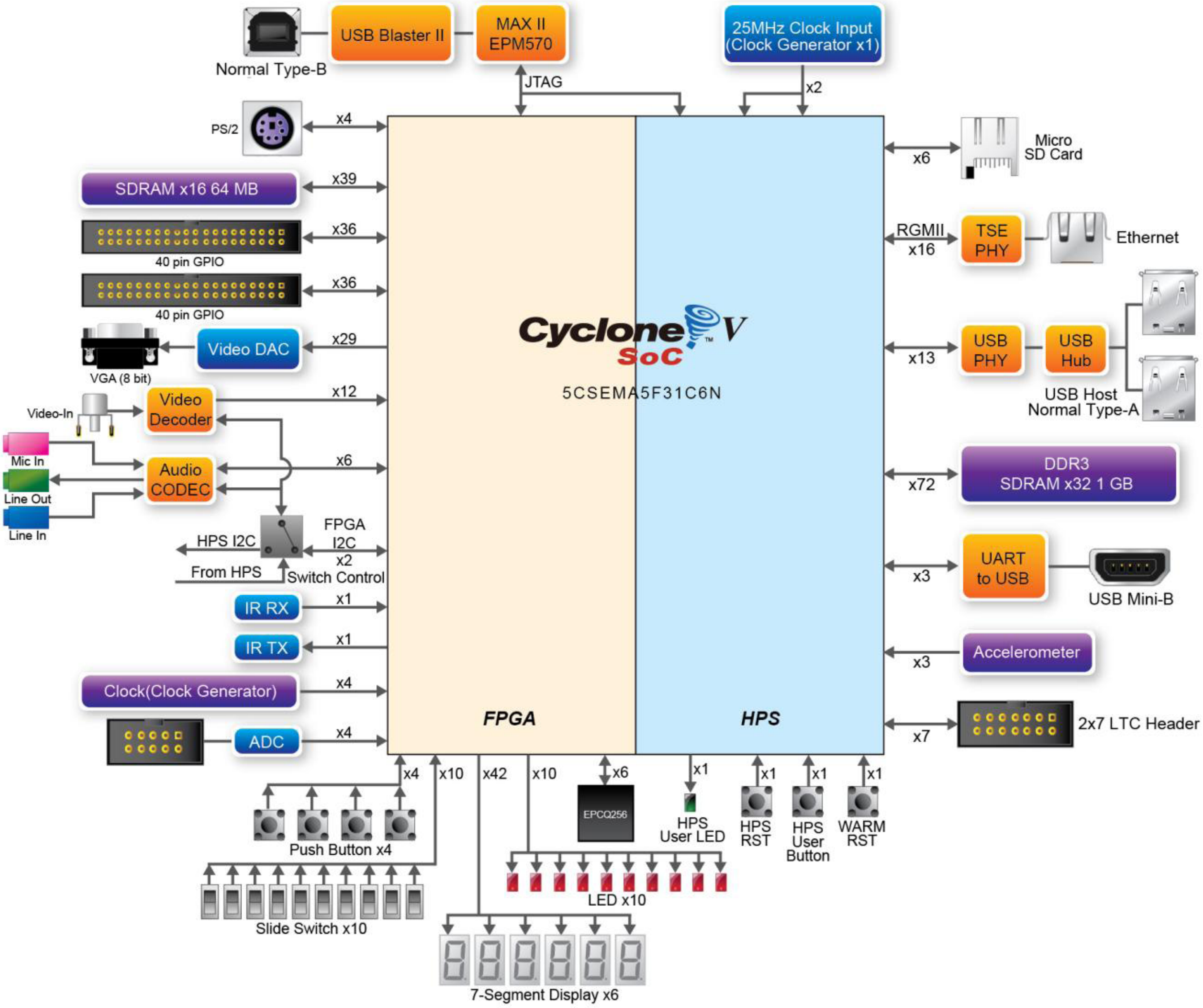
1.2 DE1-SoC Board

DE1-SoC Board is designed and made by Terasic. It is based around a Cyclone V FPGA from Altera. Include on the DE1 board are various I/O devices such as 7-segment LED displays, LED, switches, VGA port, RS232 port, SD card slot etc. A block diagram of the DE1 board is shown below. Although the Cyclone V includes a dual-core ARM processor, we will only be using the FPGA part of the FPGA for this experiment.

1.4 Verilog Hardware Description Language

One of the key learning objective of the 2nd year course in digital logic (E2.1) is for you to learn the Verilog Hardware Description Language (HDL), which is commonly used to specify FPGA and other types of chip designs. An excellent tutorial can be found on:

http://www.asic-world.com/verilog/veritut.html. A Verilog Syntax Summary sheet is provided in Appendix A.



Block Diagram of the DE1-SoC Board

1.5 Using Quartus Prime software and DE1 at home

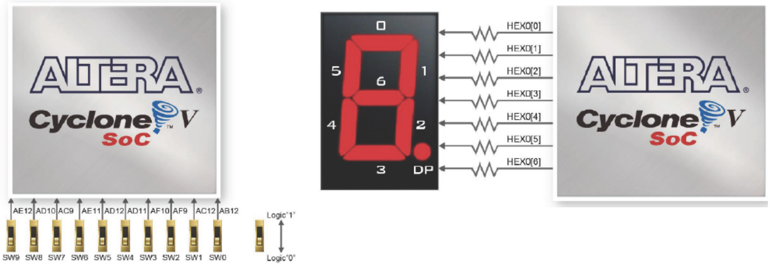
If your own laptop is sufficiently powerful (at least 4GB of RAM) and has plenty of free disk space (at least 1GB of free disk space), you may want to install a copy of the Quartus design software on your own computer. The latest version is Quartus version 16. You may also borrow a DE1-SoC board from the EEE Stores with your ID card. The lending period is one week at a time. You may renew your loan of the board if no one else is on the waiting list. Of course, the DE1-SoC board and the appropriate software are available anytime during working hours in the Level 1 Electronics Lab.

To install your own copy of Quartus, you should go to Altera's website to register, then download the free Quartus Prime Light Edition from: http://dl.altera.com/?edition=web. Note that Quartus and the DE1 board only works with MS Windows or Linux. If you are a Mac user, you would need to run a virtual machine (e.g. VirtualBox, Parallels or VMware), load a version of Windows or Linux, and then run Quartus under that environment.

Plug the DE1 board to a USB port on your computer and turn it ON (red button). It will ask you for a device driver, which can be found in the Quartus software directory ….\drivers. See "**DE1-SoC Getting Started Guide**" from the experiment webpage.

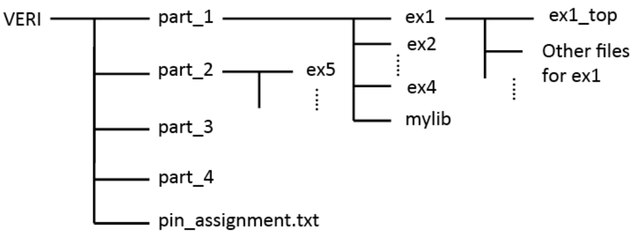
Experiment 1: Schematic capture using Quartus – 7-Segment Display

If you have come to the laboratory session prepared, Part I of experiment VERI should take no more than ONE 3-hour session. It will lead you through the entire design of a 7-segment decoder using schematic entry method. It will use four slide switches on the right (SW3 to SW0) on the DE1 board as input, and display the 4-bit binary number as a hexadecimal digit on the right-most 7-segment display (HEX0).



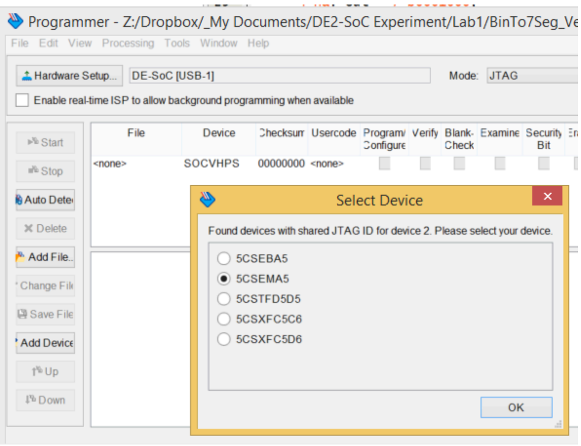
Step 1: Creating a good directory structure

Before you start carrying out any design for this exercise, it would be very helpful if you first create in your home directory a directory structure on the h:\ drive for this experiment. Shown on the right is a possible directory structure that you may choose to create. Each folder is empty for now, but as you progress through the four Lab Sessions, you will be creating each design in each of the folders.



Step 2: See what you are aiming for

Go to the Experiment webpage (see above) and download a copy of the solution for Exercise 1: "**ex1sol.sof**" to your home directory (or wherever that is). Now turn ON the DE1 board.



Programmer - Z:/Dropbox/_My Documents/DE2-SoC Experiment/Lab1/BinTo7Seg_Ve

Hardware Setup... DE-SoC [USB-1] Mode: JTAG

☐ Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/Configure	Verify	Blank-Check	Examine	Security Bit
<none>	SOCVHPS	00000000	<none>	□	□	□	□	□

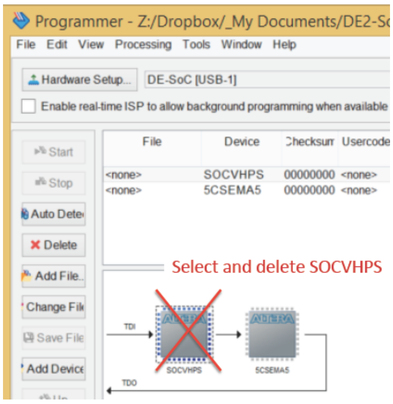
Select Device

Found devices with shared JTAG ID for device 2. Please select your device.

○ 5CSEBA5
● 5CSEMA5
○ 5CSTFD5D5
○ 5CSXFC5C6
○ 5CSXFC5D6

Step 3: Programme the FPGA

Start up Quartus software on your computer. Click command: **Tools > Programmer**. In the popup window, click: **Hardware Setup ….** You should see something like the diagram on the right. Then select: **DE-SOC [USB-1]**. This is to tell Quartus software that you are using the DE1-SoC USB interface to program (or blast) the FPGA. Then click **Auto Detect** button on the left. A window will pop up and you need to select SCSEMA5 radio button to tell the system which type of Cyclone V FPGA chip you are using (which is 5CSEMA5).



Programmer - Z:/Dropbox/_My Documents/DE2-Sc

Hardware Setup... DE-SoC [USB-1]

☐ Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode
<none>	SOCVHPS	00000000	<none>
<none>	5CSEMA5	00000000	<none>

Select and delete SOCVHPS

You will now see two lines in the Programmer window as shown on the right. Since we are only configuring (i.e. sending a bit-stream to) the FPGA part of the Cyclone V chip, we need to **delete** the SOCVHPS (stands for System-on-Chip V High Performance System, which is the ARM processor) from the programmer set up.

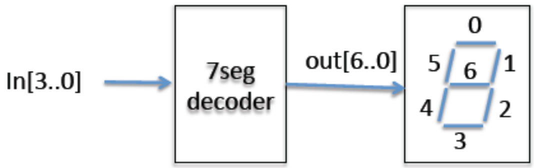
Next click the **AddFile** button. Navigate to the folder containing the **ex1sol.sof** file. Select this. Finally click the **Start** button.

The **ex1sol.sof** file contains the solution to Exercise 1 designed by me. It has the bit-stream to configure (or programme) the FPGA part of Cyclone V. Once the bit-stream is successfully sent to the FPGA chip, this design will take over the function of the chip. You should be able to change the least significant four switches and see a hexadecimal number displayed on rightmost 7-segment display.

You should leave the programmer utility running in the background for ease of sending another design to the FPGA later. Return to Quartus software by clicking its window.

Step 4: Paper Design

The overall block diagram for the decoder is shown below. The decoder outputs **out[6..0]** drive the seven segments on the display. Note that the LED segments are **low active**, meaning that the LED will light up (ON) if the corresponding digital signal is at 0V.



The truth-table for the decoder is shown here:

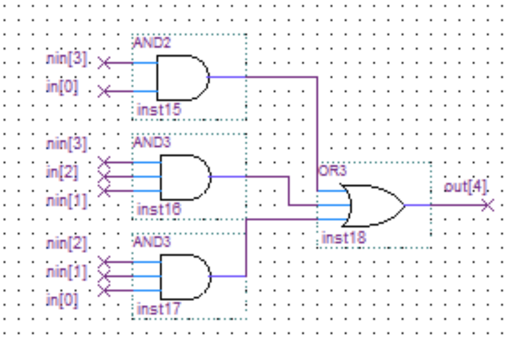
in[3..0]	out[6:0]	Digit
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

With what you have learned in the first year, you should be able to design the decoder in the form of seven Boolean equations, and then use K-map to minimise the logic. In order to save time, only derive the Boolean equation for out[4] as a Boolean function of in[3..0].

You also should **not** use K-map to perform any optimization. Quartus (and other modern CAD design software) will perform logic minimization for you and will do a much better job, taking into account the architecture of the FPGA chip.

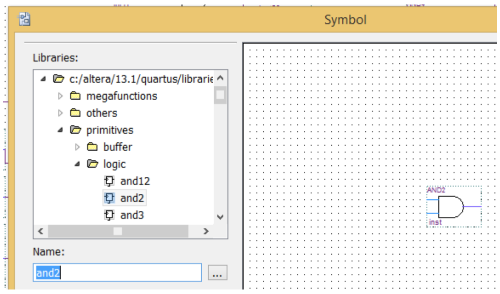
Step 5: Create the project "ex1"

- Create in your home directory the folder **../part_1/ex1**.
- Click **file>New Project Wizard**, complete the form. Use **ex1** as the project name and **ex1_top** as top-design name.
- Select the FPGA device as Cyclone V 5CSEMA5F31C6. Then click **Finish**.



Step 6: Specify the 7-segment decoder as schematic

- Download from the website the file: **My7Seg_incomplete.bdf.zip** and unzip in the folder **../part_1/ex1**. This is a ***partially completed*** schematic for the 7-segment decoder circuit with circuit for **out[4]** missing. You are now ready to enter the circuit to produce **out[4]** as gates using the schematic editor. This is shown on the right and it implements the equation:


Symbol

Libraries:

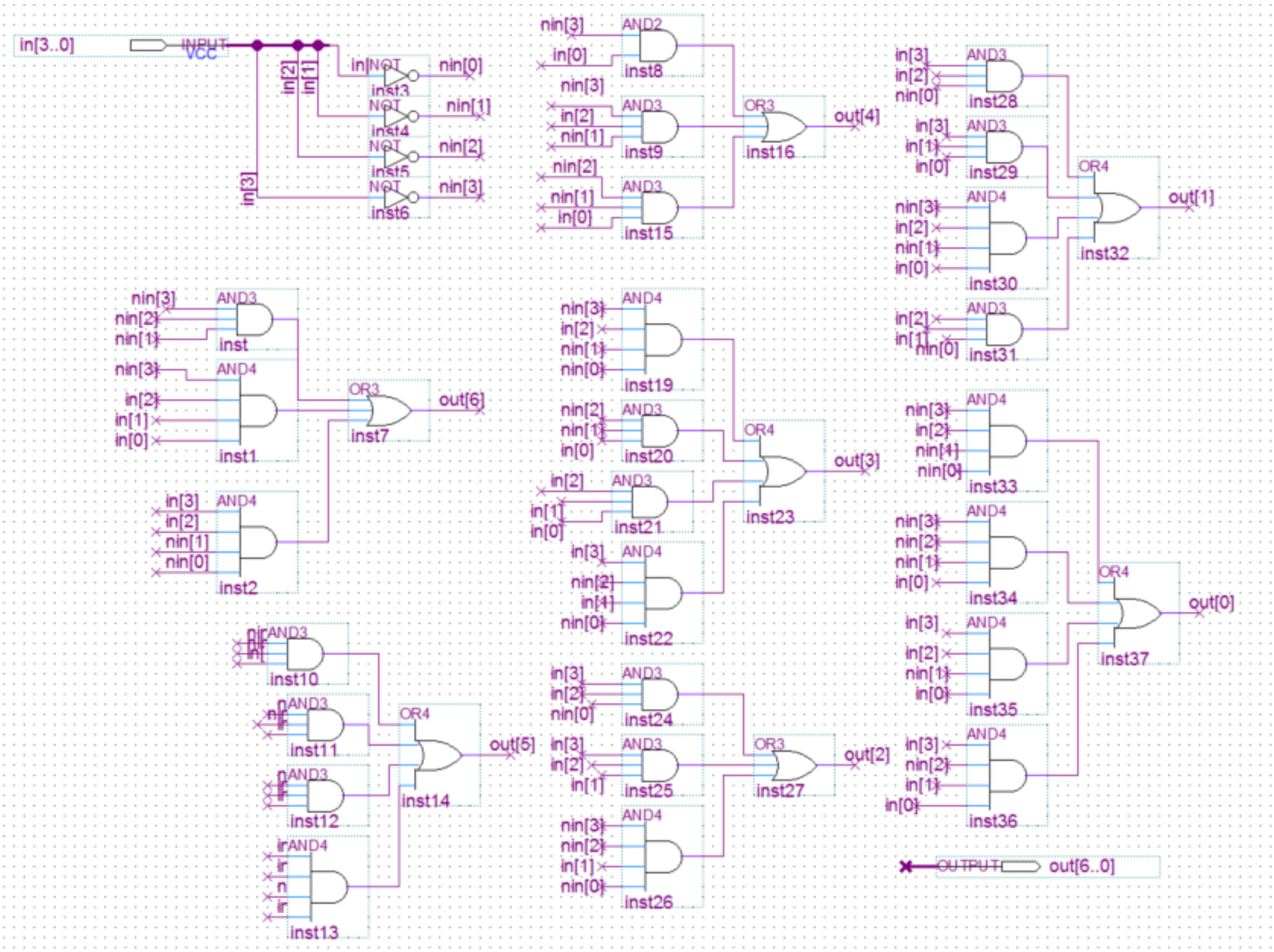
- c:/altera/13.1/quartus/librarie
  - megafunctions
  - others
  - primitives
    - buffer
    - logic
      - and12
      - and2
      - and3

Name: and2

$$\boxed{out4 = /in3*in0 + /in3*in2*/in1 + /in2*/in1*in0}$$

The Graphic Editor provides a number of libraries which include circuit elements that can be imported into a schematic. Double-click on the blank space in the Graphic Editor window, or click on the icon in the toolbar that looks like an AND gate. A pop-up box will appear. Expand the hierarchy in the Libraries box as shown in the figure. First expand libraries, then expand the library primitives, followed by expanding the library logic which comprises the logic gates. Select “and2”, which is a two-input AND gate, and click OK. Now, the AND symbol will appear in the Graphic Editor window. Using the mouse, move the symbol to a desirable location and click to place it there.

- Repeat and place two “and3” and one “or3” gates on the schematic. Change the names of all the input and output nodes accordingly. (It is quickest to put down all the gates first before wiring them up later.)
- Now wire up the gates by click and drag on the input nodes of the gates to extend a wire out, and then simply type the name of the node on the keyboard.
- When completed, you will see the entire schematic diagram for the decoder circuit as shown here:



Step 7: Include this file in project

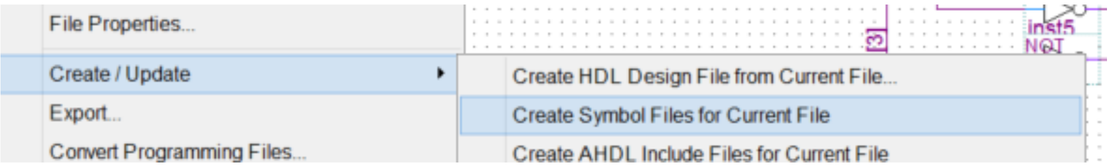
Every time you create a new entity or module as part of your design, you must include the file in the project.

- Click: **Project > Add Current Files to Project ….,**

Step 7: Make a symbol for the decoder

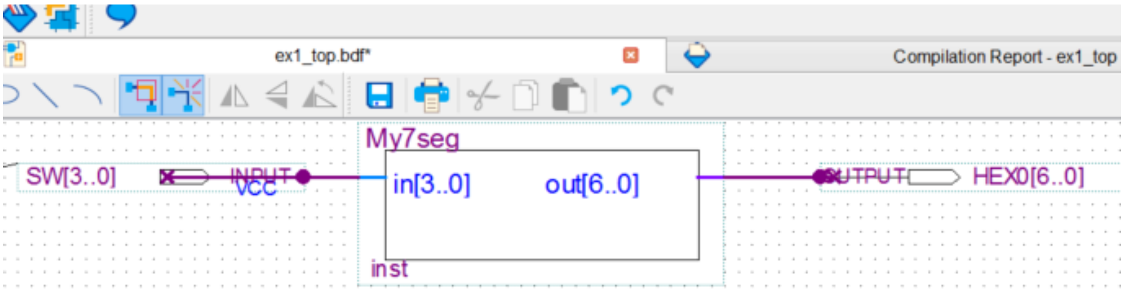
It is often convenient to encapsulate a circuit into a module (sometimes known as an “entity”), which is then used multiple times in a design. For us to do so, we need to create a symbol for **My7seg** decoder module.

Click **File > Creat/Update > Create Symbol …**


File Properties...
Create / Update
Export...
Convert Programming Files...
Create HDL Design File from Current File...
Create Symbol Files for Current File
Create AHDL Include Files for Current File
inst5
NOT

Step 8: Use this module at the top-level design schematic

- Now we will use the newly created entity **My7seg** in the top-level design.
- Click **File > New ….** and select Block Diagram /Schematic File as shown here:
- Use the button to select and place the **My7seg** module, input port and output port on the schematic.
- Double click the port symbol to edit the input and output pin names as **SW[3..0]** and **HEX0[6..0]** respectively.
- Use the bus wiring tool to wire up the ports to the module as two busses as shown below.
- Save this file.

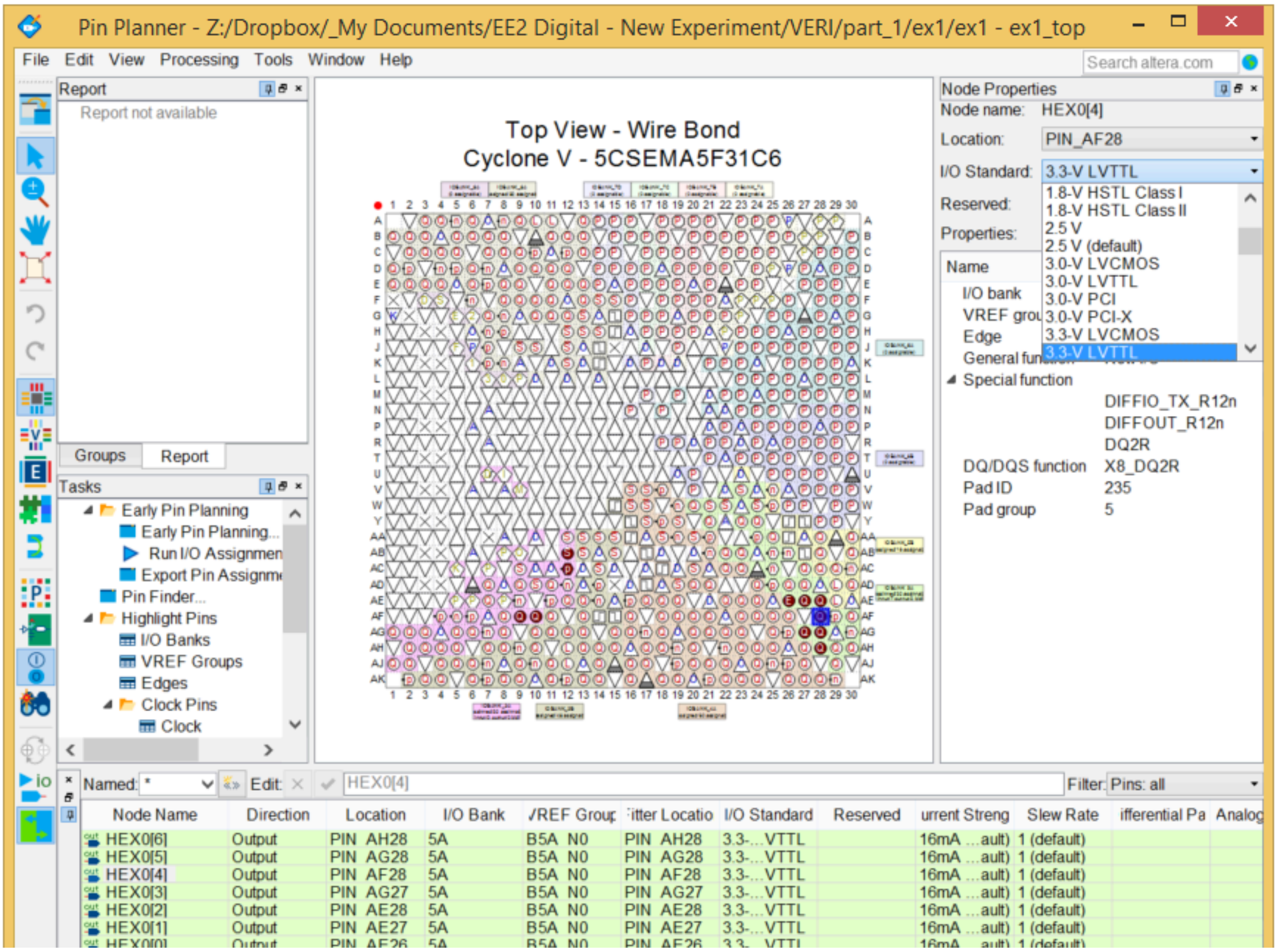


Step 9: Pin assignment & Compilation

You need to associate your design with the **physical pins** of the Cyclone V FPGA on the DE1-SOC board.

- Check that the device is corrected assigned as 5CSEMA5F31C6 using: **Assignments > Device …**
- Click: **Processing > Start > Start Analysis and Elaboration**. This will work out the input/output port names for your design. This should complete without error. Otherwise, fix all errors and re-analyse. (There will be many warnings – generally warnings are not important. But there MUST not be errors, which will be shown in RED.)
- Click **Assignment > Pin Planner** and a new window with the chip package diagram. You should also see the top-level input/output ports shown as a list.

Signal Name	Pin Location
HEX0[6]	PIN_AH28
HEX0[5]	PIN_AG28
HEX0[4]	PIN_AF28
HEX0[3]	PIN_AG27
HEX0[2]	PIN_AE28
HEX0[1]	PIN_AE27
HEX0[0]	PIN_AE26
SW[3]	PIN_AF10
SW[2]	PIN_AF9
SW[1]	PIN_AC12
SW[0]	PIN_AB12



Node name: HEX0[4]

Location: PIN_AF28

I/O Standard: 3.3-V LVTTL

Reserved:

Properties:

1.8-V HSTL Class I
1.8-V HSTL Class II
2.5 V
2.5 V (default)
3.0-V LVCMOS
3.0-V LVTTL
3.0-V PCI
3.0-V PCI-X
3.3-V LVCMOS
3.3-V LVTTL

Name	
I/O bank	
VREF grou	
Edge	
General fun	
Special function	DIFFIO_TX_R12n DIFFOUT_R12n DQ2R
DQ/DQS function	X8_DQ2R
Pad ID	235
Pad group	5

Named: * Edit: HEX0[4] Filter: Pins: all

Node Name	Direction	Location	I/O Bank	VREF Group	itter Locatio	I/O Standard	Reserved	urrent Streng	Slew Rate	ifferential Pa	Analog
HEX0[6]	Output	PIN_AH28	5A	B5A_N0	PIN_AH28	3.3-...VTTL		16mA ...ault)	1 (default)		
HEX0[5]	Output	PIN_AG28	5A	B5A_N0	PIN_AG28	3.3-...VTTL		16mA ...ault)	1 (default)		
HEX0[4]	Output	PIN_AF28	5A	B5A_N0	PIN_AF28	3.3-...VTTL		16mA ...ault)	1 (default)		
HEX0[3]	Output	PIN_AG27	5A	B5A_N0	PIN_AG27	3.3-...VTTL		16mA ...ault)	1 (default)		
HEX0[2]	Output	PIN_AE28	5A	B5A_N0	PIN_AE28	3.3-...VTTL		16mA ...ault)	1 (default)		
HEX0[1]	Output	PIN_AE27	5A	B5A_N0	PIN_AE27	3.3-...VTTL		16mA ...ault)	1 (default)		
HEX0[0]	Output	PIN_AE26	5A	B5A_N0	PIN_AE26	3.3-...VTTL		16mA ...ault)	1 (default)		

- Click on the appropriate pins one by one, and select the corresponding location from a dropdown list according to the list shown in the pin assignment table above. The I/O standard (i.e. interface voltages) should be "3.3V LVTTL".
- Click: **Processing > Start Compilation**, to build the entire design, and to generate all the necessary files. There should be NO error, but there will be many warnings.

Step 10: Program the FPGA on the DE1 Board

- You have now created in the **../part_1/ex1/output_files/** folder the file **ex1_top.sof**, which contain your design. (This should be the same design as the one I gave you to try out in Step 2 of this exercise.)
- Program the DE1 board with your version of **ex1_top.sof** and test that it is working properly.

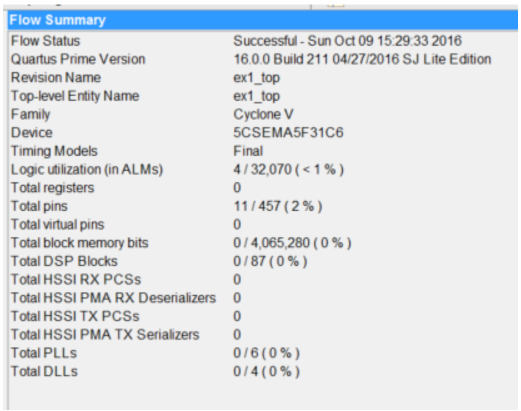
Step 11: Propagation Delay from inputs to outputs

- Click: **Tools > TimeQuest Timing Analyzer** to invoke the built in timing analyzer of Quartus. A new TimeQuest window will appear.
- Click: **Netlist > Create Timing Netlist**. Then select post-fit and slow-corner, then OK.
- In the "**Set Operating Conditions**" window, select "Slow 1100mV 0°C model".
- Now click: **Netlist > Update Timing Netlist** … This will use the specified timing model and condition to produce a set of timing data.
- Click: **Report > Datasheet > Report Datasheet**. This will produce a table showing the input-to-output propagation delay for various combination of rise and fall times (RR, RF, FR and FF). Make sure that you understanding what this table means.

- Repeat the procedcure again but for “Slow 1100mV 85°C Model”. What is the delay difference at these two temperature extremes? Why?

Step 12: Examine the resources used

- Now examine the Compilation Report. You should see something as shown here.
- It shows that this design used only 4 out of 32,070 ALMs (Adaptive Logic Modules), 11 of the 457 I/O pins and none of the other resources.



Flow Summary	
Flow Status	Successful - Sun Oct 09 15:29:33 2016
Quartus Prime Version	16.0.0 Build 211 04/27/2016 SJ Lite Edition
Revision Name	ex1_top
Top-level Entity Name	ex1_top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	4 / 32,070 (< 1 %)
Total registers	0
Total pins	11 / 457 (2 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Congratulations! You have now completed your first FPGA design!

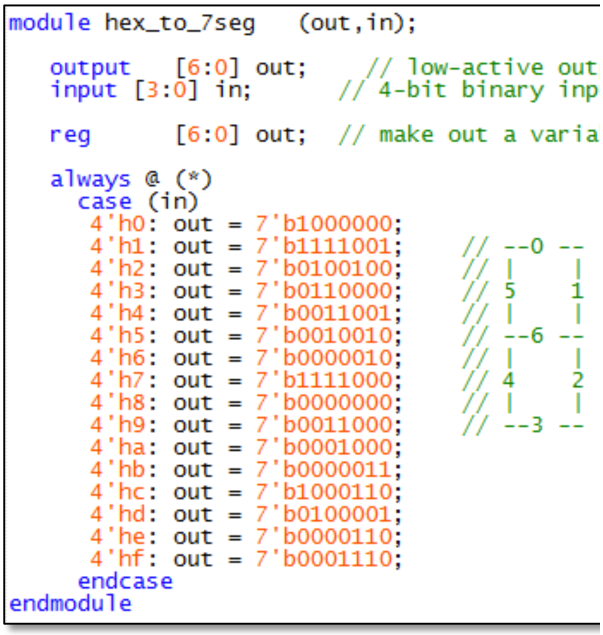
Experiment 2: 7-Segment decoder in Verilog HDL

I hope you now appreciate how limiting and slow it is to enter a design as a schematic diagram. Modern digital designers DO NOT USE schematic as a method of entry any more. Instead a designer would either use Verilog or VHDL hardware description language, or some high level language such as OpenCL or Vivaldo HLS to specify the design. In this experiment, you will design the Verilog version of what you have done in Experiment 1. Hopefully this will convince you never to use schematic capture for digital design again!§

Step 1: hex_to_7seg.v

- Create a new project **ex2** as before and a top-level module **ex2_top** as before in **ex2** folder.
- In Quartus, create a design file in Verilog HDL known as **hex_to_7seg.v** using:

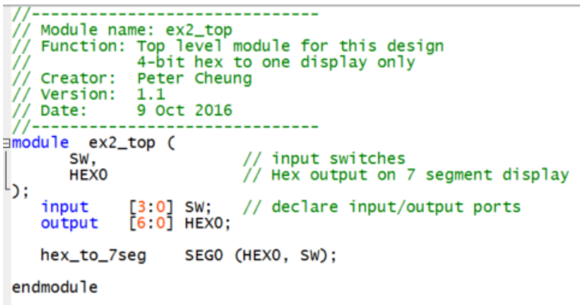
  File > New …. and select Verilog HDL from the list.
- Type the Verilog source file as shown below. (You should have seen this during one of the Lectures earlier). Make sure you pay attention to the syntax of Verilog. Save your file.
- A full compilation can take a long time. A far more efficient way to check the syntax of your code by clicking: **Process > Analyze current file.** You should get into a habit of ALWAYS perform this step to make sure that the new Verilog module you have created is as error free as possible. It will save you a lot of time.



```
module hex_to_7seg   (out,in);

   output     [6:0] out;     // low-active out
   input  [3:0] in;         // 4-bit binary inp

   reg        [6:0] out;     // make out a varia

   always @ (*)
     case (in)
      4'h0: out = 7'b1000000;
      4'h1: out = 7'b1111001;         // --0 --
      4'h2: out = 7'b0100100;         // |     |
      4'h3: out = 7'b0110000;         // 5     1
      4'h4: out = 7'b0011001;         // |     |
      4'h5: out = 7'b0010010;         // --6 --
      4'h6: out = 7'b0000010;         // |     |
      4'h7: out = 7'b1111000;         // 4     2
      4'h8: out = 7'b0000000;         // |     |
      4'h9: out = 7'b0011000;         // --3 --
      4'ha: out = 7'b0001000;
      4'hb: out = 7'b0000011;
      4'hc: out = 7'b1000110;
      4'hd: out = 7'b0100001;
      4'he: out = 7'b0000110;
      4'hf: out = 7'b0001110;
     endcase
endmodule
```

Step 2: Create Top-Level Specification in Verilog

- Instead of using schematic capture for the top-level module (that connects to physical pins on the FPGA), we will do this also in Verilog by creating the file: “**ex2_top.v**” as shown here. Set this as Top-Level entity.
- Click: **Project > Add/Remove Files**, and remove the .bdf file as part of this project.



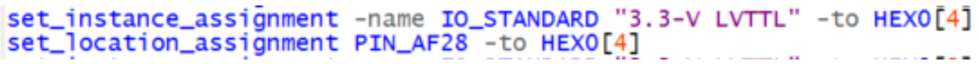
```
//--------------------------------
// Module name: ex2_top
// Function: Top level module for this design
//           4-bit hex to one display only
// Creator:  Peter Cheung
// Version:  1.1
// Date:     9 Oct 2016
//--------------------------------
module  ex2_top (
      SW,                    // input switches
      HEXO                   // Hex output on 7 segment display
);
   input    [3:0] SW;    // declare input/output ports
   output   [6:0] HEXO;

   hex_to_7seg    SEGO (HEXO, SW);

endmodule
```

- This allows you to remove the .bdf file and replace it with the .v file for the top-level specification.
- Verify that everything works properly with: **Process > Start > Start analysis & elaboration**. Make sure that there is no error. (Warnings often capture potential errors. However, the Quartus system generates many warnings, and nearly all of which are not important. Once you have gain confidence on the system, you may start ignoring the warning, but never ignore any error.)

You will save a lot of time if you ALWAYS use these two steps: analyze, and analysis & elaboration, and ensure that ALL errors are dealt with (and warning understood).

Step 3: Pin Assignment – the quick way

- Earlier you used the **pin assignment editor** to associate pins on the package to your signals. This is a tedious process. In ex1, if you have correctly completed the design, the pin assignment would have been stored in a file: "**ex1_top.qsf**" file.
- Open this file, either using Quartus' built-in editor by clicking: **File > Open file**… or use your own favourite edit on your PC.
- You will find lines of statement such as:



```
set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to HEX0[4]
set_location_assignment PIN_AF28 -to HEX0[4]
```

- The first line defines the voltage standard used by the HEX0[4] signal (3.3V logic).
- The second line defines the physical pin location of HEX0[4] is PIN_AF28.
- Now open the **ex2_top.qsf** file. You will see that there is no pin assignment for this design yet. Before full compilation, we need to tell Quartus which signal is connect to which physical pin on the FPGA.
- Instead of using the tedious **pin assignment** editor in **ex1**, we will modify the **ex2_top.qsf** file with our text editor to include the pin assignment information. To do this, first download from the experiment webpage the file: **pin_assignment.txt** to the VERI directory.
- Then use: **Edit > Insert File …** in Quartus to insert the whole of **pin_assignment.txt** in **ex2_top.qsf**.
- Note that we only use 7 pins in **ex2_top.v**, but **pin_assignment.txt** defines **all** pins used by the four parts of Experiment VERI. Quartus will generate lots of warnings which you may ignore about these unused pins not being driven. It will not create any error and the pin assignments for unused pins will be ignored.

Step 5: Test your design

- Recompile your design.
- Go to the Programmer window (assuming that you still have it opened). Delete the .**sof** file entry and add the current .**sof** file.
- Test your design on the board.

Step 6: Put module in mylib

Over the four weeks in the Lab, you will design and verify various Verilog modules which you will reuse. You should copy these to the "**mylib**" folder and include them in your new design as necessary.

Note: When you perform a compilation, there may be a popup window informing you that some "Chain_x.cdf" file has been modified, and ask if you wish to save it. Just click NO.

Experiment 3: Test yourself - 10-bit binary switch values on three 7-segment displays

Here is a "test yourself" exercise. Create your own design to display all 10-bit sliding switches as hexadecimal on three of the 7-segment LED displays.

Checkpoint: You should get to this point by the end of the 3-hour Lab Session or earlier.

Experiment 4 (optional): Displaying 10-bit binary as BCD digits on the 7-segment displays

In one of the lectures, you have been taught how to convert binary numbers to binary-code-decimal digits using the "**shift and add 3**" algorithm. You have been shown how to implement an 8-bit binary to BCD converter using Verilog. Furthermore in problem sheet 1, you have been asked to extend this to a 10-bit converter (**bin2bcd_10.v**).

For this optional exercise, you are required to display the 10-bit binary number as specified by the 10 sliding switches SW[9:0] as a decimal number using your 10-bit converter module and the 7-segment decoder. Record the resource usage of your design.

- Now download from the experiment website a 16-bit binary to BCD converter module provided (bin2bcd_16.v), and replace your 10-bit converter with this one.
- When instantiating the 16-bit converter, but only using 10 of the 16 bits, you should specify the input ports as: {6'b0, SW[9:0]}. (Remember that the {…} operator is for bit-concatenation.)
- Test your design on the DE1 Board.
- Compare the resource usage by this design (with **bin2bcd_16.v**) with that using the 10-bit version (**bin2bcd_10.v**). You will find that in fact the number of ALMs used will be the same.
- Basically Quartus optimizer removes unused resources. The module **bin2bcd_16.v** has six of its input connected to 0, and only 12 of its output connected to output pins. The CAD software will eliminate all the redundant logic. This should result in the same number of ALM being used as that with a 10-bit converter. In other words, for such combinational circuit, you only need to keep the 16-bit version for any numbers with 16 bits or lower.

Before you move onto Part II of VERI, you should copy the components (modules) you have designed to the "mylib" folder. In the following sessions, you will be using the various .v files from this repository of your own design. You will also be adding to it later.

Department of Electrical & Electronic Engineering
Imperial College London

2nd Year Laboratory

Experiment VERI: FPGA Design with Verilog (Part 2)

(webpage: www.ee.ic.ac.uk/pcheung/teaching/E2_Experiment /)

PART 2 – Counters and FSMs

1.0 Learning Outcomes

Part 2 of VERI teaches you:

- how to design different types of counters and timers;
- how to use the Modelsim simulator to verify the correct function of your design and the use of testbenches;
- how to predict the maximum operating clock frequency of your circuit sequential circuits;
- how to design some useful timing and counting components for later part of Experiment VERI.

1.1 Experiment 5: Designing a Counter

Step 1: Create the project for an 8-bit counter

- Create in your directory a folder named part_2.
- Click **file>New Project Wizard**, and create project **ex5** and top level file **ex5_top**. Then click **Finish**.
- Create the Verilog file: "**counter_8.v**" which contains your design in Verilog. I suggest you use convention of using **"_n"** to indicate the number of bits in a module.
- Click **File > New …** and select Verilog as the new file. An edit window will appear.

Step 2: Enter the Verilog specification of the 8-bit binary counter

- Enter the Verilog module as shown below (next page). Although you can miss out the comments, I recommend that you to retain them because the code is deliberately verbose in order to explain the meaning of the Verilog language.
- The line `**timescale 1ns / 100ps** tells the system to use 1 ns as the unit time step with a time resolution of 100ps.
- Make sure that you fully understand this Verilog code before proceeding to the next step. Save the file as **counter_8.v**. (I recommend that you use module name as the file name to avoid confusion.)

Step 3: Enter the Verilog specification of the 8-bit binary counter

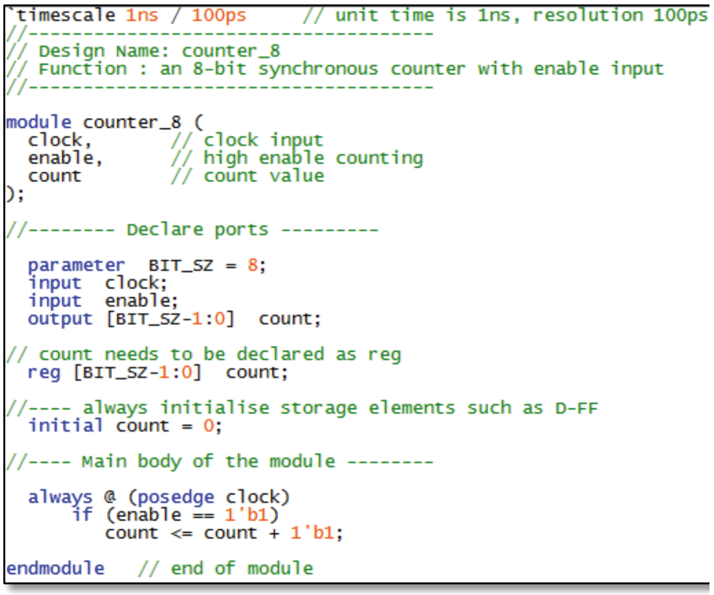
- While is opened in the Editor window, click **Project > Add Current File to** Project, then click **Project > Set as Top Level Entity**. This command tells Quartus that this module is the top-level of your design.

> Normally we use …**_top.v** as the top-level module, which connects to physical pins of the FPGA. However, for this experiment, the counter module is verified through simulation. So we don't need to create pin connects. The "Set as Top Level Entity" is very useful if you want to use the simulator to verify different modules in a large design. You can move up and down the module hierarchy and verify them from the lowest level up.

- Click **Processing > Analyze Current File**. This is the fastest way to check if this .v file has any syntax error.
- Then Click **Processing > Start > Start Analysis and Synthesis**. This takes the current Verilog module (and all other modules that it uses if any), and produce a register-level model of your design ready for register-transfer level (RTL) simulation. Unlike full compilation, this step does not require pin assignment and other device specific steps, but is sufficient for you to simulate the circuit as specified in Verilog.

Verilog code: 8-bit counter

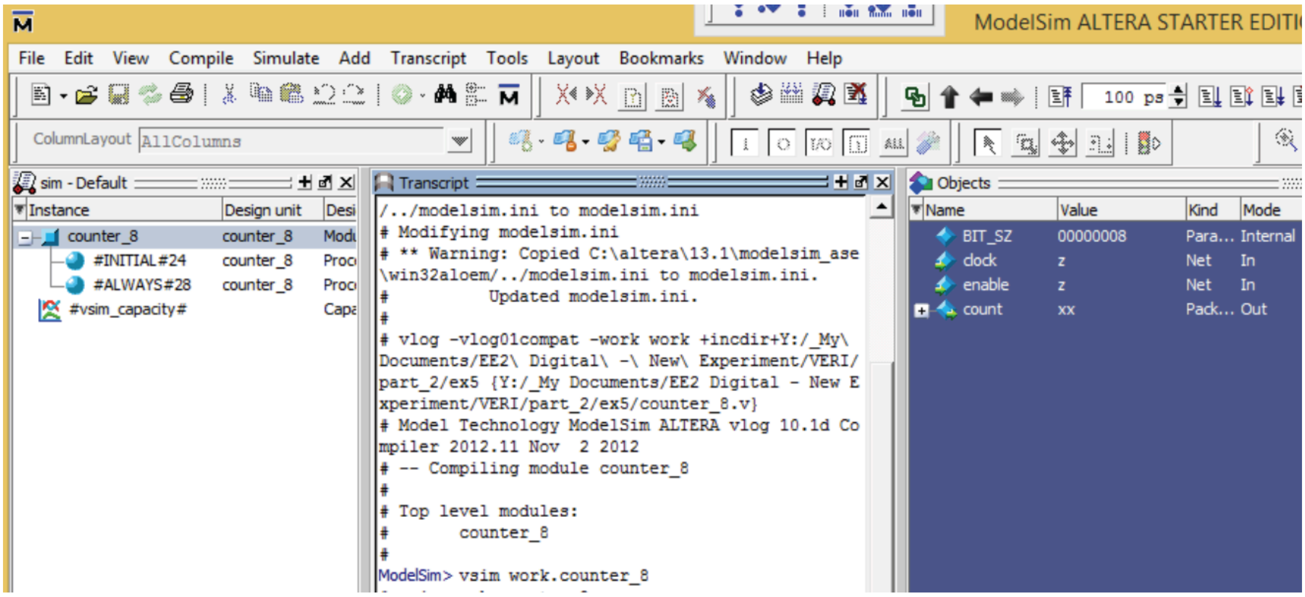
(Note that the first character on line 1 before 'timescale' is a backquote ` - not easy to find on many keyboards!)



```
`timescale 1ns / 100ps      // unit time is 1ns, resolution 100ps
//-------------------------------------
// Design Name: counter_8
// Function : an 8-bit synchronous counter with enable input
//-------------------------------------

module counter_8 (
  clock,        // clock input
  enable,       // high enable counting
  count         // count value
);

//-------- Declare ports ---------

  parameter  BIT_SZ = 8;
  input  clock;
  input  enable;
  output [BIT_SZ-1:0]  count;

// count needs to be declared as reg
  reg [BIT_SZ-1:0]  count;

//---- always initialise storage elements such as D-FF
  initial count = 0;

//---- Main body of the module --------

  always @ (posedge clock)
      if (enable == 1'b1)
          count <= count + 1'b1;

endmodule   // end of module
```

Step 4: Simulate the binary counter

- Click **Tools > Run Simulation Tools > RTL Simulation**. This command starts up Modelsim simulator programme as a separate process. Now you have entered the Modelsim environment.
- Click **Simulate > Start Simulation** …. Then select **work -> counter_8** from the popup window. This tells Modelsim to simulate this module.
- Note that Modelsim provides several windowpanes. The most important is the Transcript pane – this is where you enter commands[1] to drive the simulator. The wave pane is where results are displayed as waveforms. You are recommended to un-dock this pane as shown below so that it is in a separate window and spans the whole width of your monitor. Finally, there is the object pane, which shows all the signals (objects) of your design.



Instance	Design unit	Desi
counter_8	counter_8	Modu
#INITIAL#24	counter_8	Proc
#ALWAYS#28	counter_8	Proc
#vsim_capacity#		Capa

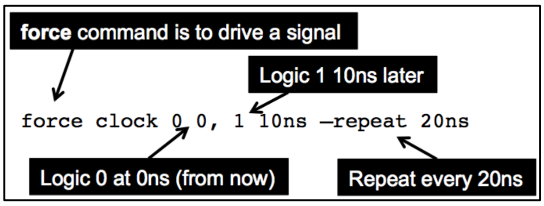
```
/../modelsim.ini to modelsim.ini
# Modifying modelsim.ini
# ** Warning: Copied C:\altera\13.1\modelsim_ase
\win32aloem/../modelsim.ini to modelsim.ini.
#          Updated modelsim.ini.
#
# vlog -vlog01compat -work work +incdir+Y:/_My\
Documents/EE2\ Digital\ -\ New\ Experiment/VERI/
part_2/ex5 {Y:/_My Documents/EE2 Digital - New E
xperiment/VERI/part_2/ex5/counter_8.v}
# Model Technology ModelSim ALTERA vlog 10.1d Co
mpiler 2012.11 Nov  2 2012
# -- Compiling module counter_8
#
# Top level modules:
#       counter_8
#
ModelSim> vsim work.counter_8
```

Name	Value	Kind	Mode
BIT_SZ	00000008	Para...	Internal
clock	z	Net	In
enable	z	Net	In
count	xx	Pack...	Out

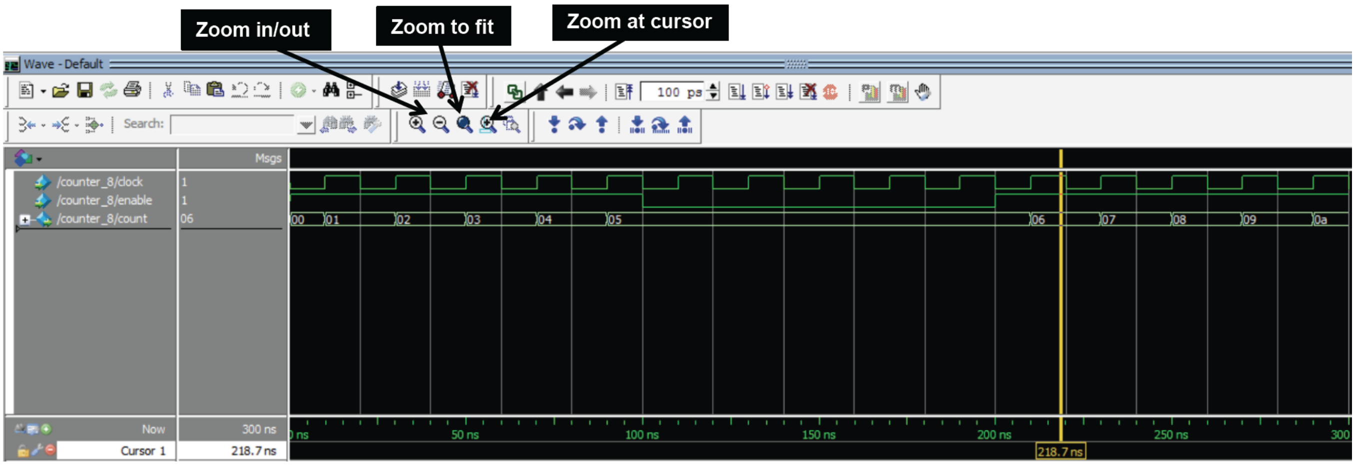
[1] Modelsim uses a scripting language known as **Tcl** to control how it is driven. You only need to learn Tcl if you want to do advance stuff with Modelsim for your personal interest.

Step 5: Add waveforms to the Wave window and drive signals

- In the transcript window, enter two commands: "**add wave clock enable**" and "**add wave –hexadecimal count**". This will add these signals as waveforms in the wave pane and show count values as hexadecimal.
- Now we want to drive **clock** with a 50MHz symmetrical signal. To do this, enter:
- Enter: "force enable 1" to enable the counter.
- Enter: "run 100ns" to run the simulator for 5 clock cycles (5 x 20ns = 100ns).
- You will see the waveform pane showing the counter counting from 0 to 5. Now force enable low and run for another 100ns. Then high again and run for 100ns.



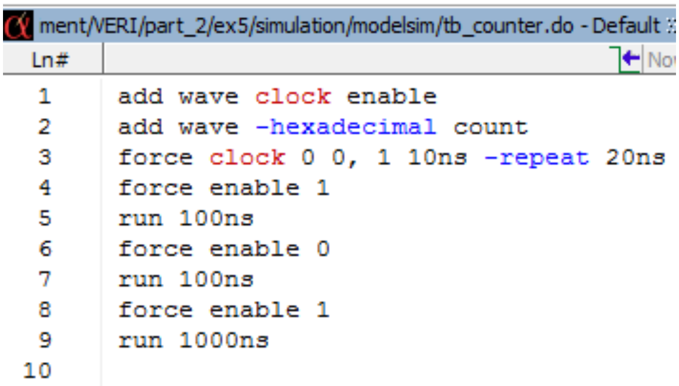
```
force clock 0 0, 1 10ns –repeat 20ns
```



- Click on the waveform put a cursor at a specific time for inspecting the signal values. The icons above the waveforms (as labeled) allow you to zoom in and out of the waveform. Try this yourself.

Step 6: Create a Testbench as a DO-file

- Interactively specifying the driving signals is very tedious and prone to error. Therefore the preferred method is to create a **"do"** file which is a text file containing a sequence of commands (as you have previously entered in the transcript window).
- Click **File > new > source** and select new "**do**" file. Then enter the command lines as shown on the right. Then save this as "**tb_counter.do**".


ment/VERI/part_2/ex5/simulation/modelsim/tb_counter.do - Default
Ln#
No

```
add wave clock enable
add wave -hexadecimal count
force clock 0 0, 1 10ns -repeat 20ns
force enable 1
run 100ns
force enable 0
run 100ns
force enable 1
run 1000ns
```

- Delete all signals from the wave window, and enter command

  vsim> restart

  vsim > do ./tb_counter.do

- This should provide exactly the same waveform results as in step 5. However, the **.do** file can be reused and modified far easier than typing them into the transcript window. It acts as a simple form of a **test-harness** (or **testbench**) for your design. Generally speaking, you must produce testbenches for all your designs instead of using interactive means to test your circuit. Not only because this saves time, it also allows you to change the code and verify its correctness in the same way for each version of your design.

Step 7: Single stepping

- Modelsim is very powerful. You can use it to debug your Verilog design almost like software. However, do remember that we are dealing with a hardware description that operates in parallel. In contrast, software codes are generally procedural and operate sequentially.
- Try the **vsim> step** command or click on the step-command pane to watch how you can step through your Verilog code. Signal values in the object and the wave windows are updated accordingly.
- Modelsim has many useful features to help you debug your design. Details of all the commands can be found in the Modelsim Reference Manual. This is easily available under **Help > PDF Documentations > Reference Manual**. Beware that this manual is very thick! DO NOT print this out.

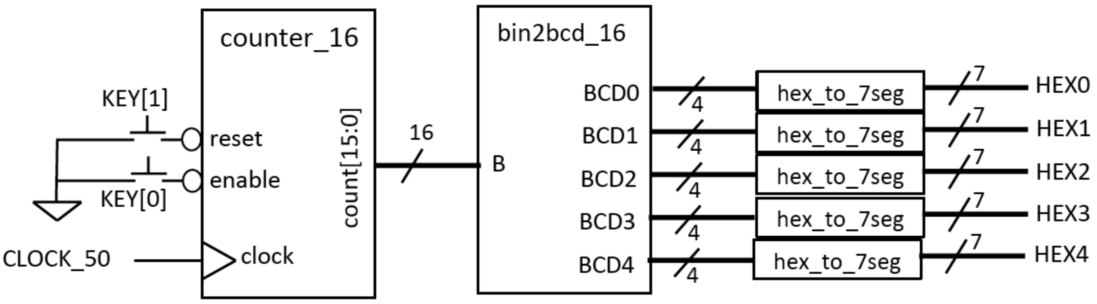
2.0 Experiment 6: Implementing a 16-bit counter on DE1

In this part of the experiment, you will test your counter design on the DE1 board. You will also learn how to find the maximum clock frequency that your design will work correctly.

Step 1: Create a new project ex6, and copy to this directly your files **counter_8.v**. Modify **counter_8.v** to **counter_16.v** and make it a 16-bit counter. Furthermore, add a reset input to reset the count value to zero synchronously to the clock. Download from the experiment webpage the component bin2bcd_16.v, a module I have designed to convert a 16-bit binary number to 5 BCD digits. You will also need the add3_ge5.v module. Put these module in the ../mylib folder, which should also contained the hex_to_7seg.v you designed in Part 1.

Step 2: Create a top-level module ex6_top.v in Verilog to specify the circuit shown below. Make sure that you have added all the relevant Verilog modules to the project using **Project**

> Add/Remove Files in Project: **counter_16.v**, **ex6_top.v** and finally add **hex_to_7seg.v**, **add3_ge5.v** and **bin2bcd_16.v** from your library folder **../mylib/**. Go to the **ex6_top.v** window and set this file as your top-level module.



Push button keys are low active, i.e. KEY[0] and KEY[1] are normally high and go low when pressed.

Step 3: Use **Processing > Analyze Current File** check your newly create Verilog files. This is the quickest way to find the basic syntax errors in your Verilog code. Once all the simple errors are fixed, use **Processing > Start Analysis and Elaboration** to perform fuller check of the "**ex6_top.v**" to make sure that files are consistent and correct. There is no need to simulate this circuit.

Step 4: Selecting the FPGA Device – Click **Assignments > Device**…. and select the correct Cyclone V FPGA: 5CSEMA5F31C6.

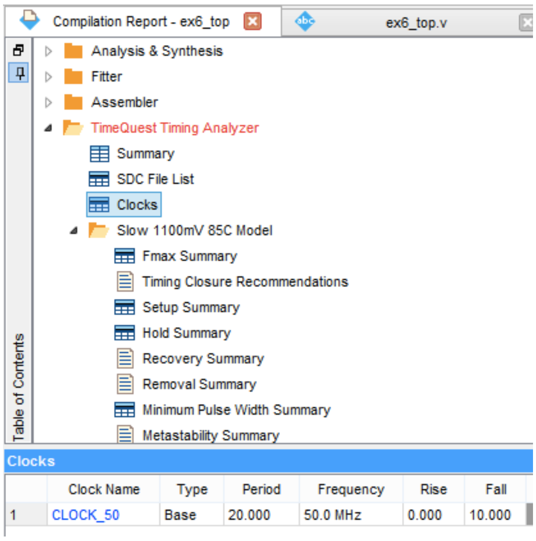
Step 5: Pin Assignment – Open the **ex6_top.qsf** file. Examine its content. You will find that no pins are being assigned yet. Insert into this file all the pin assignments. The easiest way to do this is click on: **Edit > Insert file ..** then select **../pin_assignment.txt** (you should have downloaded this file from the Experiment webpage). Note that you are currently not using all the pins assigned in the **pin_assignment.txt** file. Don't worry. This will only produce a few more warning messages. Full compilation can still go ahead without errors.

Step 6: Set clock frequency – Create a new file **"ex6_top.sdc"**[2] which should contain one single line:

```
create_clock -name "CLOCK_50" -period 20.000ns [get_ports {CLOCK_50}]
```

With this, Quartus will know that the signal CLOCK_50 is a 50 MHz clock.

Step 7: Full Compilation – Click: **Processing > Start Compilation**. This will go through the entire compilation process. Examine the Tasks window on the left and see all the steps being taken in order to generate the final bit-stream.



- Analysis & Synthesis
- Fitter
- Assembler
- TimeQuest Timing Analyzer
  - Summary
  - SDC File List
  - Clocks
  - Slow 1100mV 85C Model
    - Fmax Summary
    - Timing Closure Recommendations
    - Setup Summary
    - Hold Summary
    - Recovery Summary
    - Removal Summary
    - Minimum Pulse Width Summary
    - Metastability Summary

Clocks

	Clock Name	Type	Period	Frequency	Rise	Fall
1	CLOCK_50	Base	20.000	50.0 MHz	0.000	10.000

Step 8: Maximum clock frequency – As part of the compilation process, **TimeQuest** timing analyzer is used to predict various timing information. In the "Compilation Report" window, you should see a list of reports resulting from the compilation. Double-click **TimeQuest Timing Analyzer** entry, and you should see a list similar to the one

[2] Synopsis Delay Constraint (.sdc) files are standard formatted files introduced by Synopsis, a well-known company specializing on IC design CAD tools. With this, a designer can specify various timing constraints for the CAD tools the check against. Here we are only using this to define clock frequency.

shown here. Clicking on various entries under this will show the various timing specifications. Answer the following questions:

What are the predicted maximum frequencies for this circuit under the highest and lowest temperatures? What are the other interesting timing data that you can discover with these reports? Why is the TimeQuest entry red, indicating that there may be a problem?

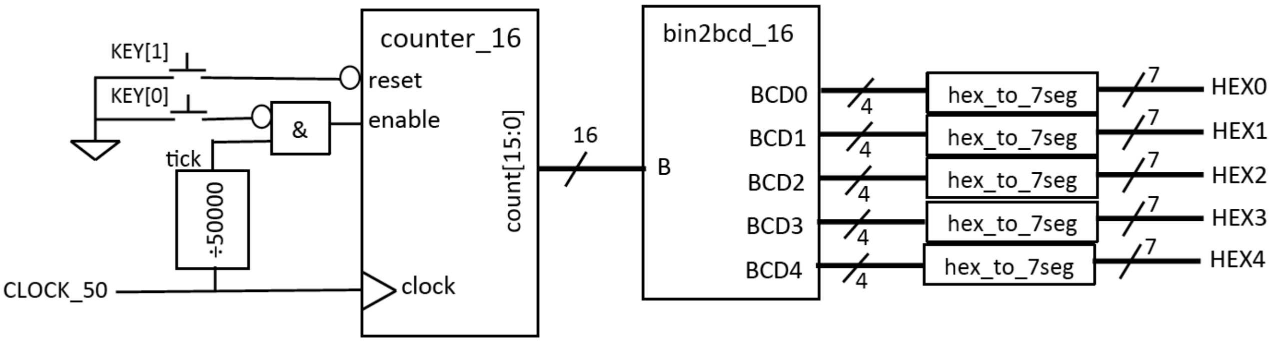
Step 9: Test your design on DE1 – program the DE1 and check that your design works.

Step 10: Examine the amount of FPGA resources being used by this 16-bit counter. Explain the results.

Test-yourself Task (compulsory) **– Cascade counter**

You are now required to create something yourself. In the previous exercise, the 16-bit counter is counting a 20MHz clock. This is much too fast for us to see the counter changing. This part of the experiment requires you use the counter to count the number of millisecond elapsed. You would need to do this by having two counters cascaded (i.e. connected in series) with each other. The overall block diagram is shown below.

The divide-by-50000 circuit generates a 1 cycle high pulse every 50,000 clock cycles. Therefore the output signal tick provides one enable pulse every millisecond. (See notes.)



Modify your circuit to implement this and test the new circuit on the DE1 board.

3.0 Experiment 7: Linear Feedback Shift Register (LFSR) and PRBS

You encountered a 4-bit LFSR in Lecture 5 slide 17, which implements the primitive polynomial: $1 + X^3 + X^4$. You are now required to implement a 7-bit LFSR implementing the polynomial: $1 + X + X^7$. Assuming that you initialize the shift register to 7'd1, work out manually the first 10 sequence values of the output sequence. (The output sequence should be 127 long without repetition, is known as a **pseudo-random binary sequence** or **PRBS**.)

Connect the shift register clock to KEY[3] and use the momentary key to cycle through the first ten values of the PRBS. The random output should be displayed as two hexadecimal digits.

Checkpoint: You should get to this point by the end of the second week.

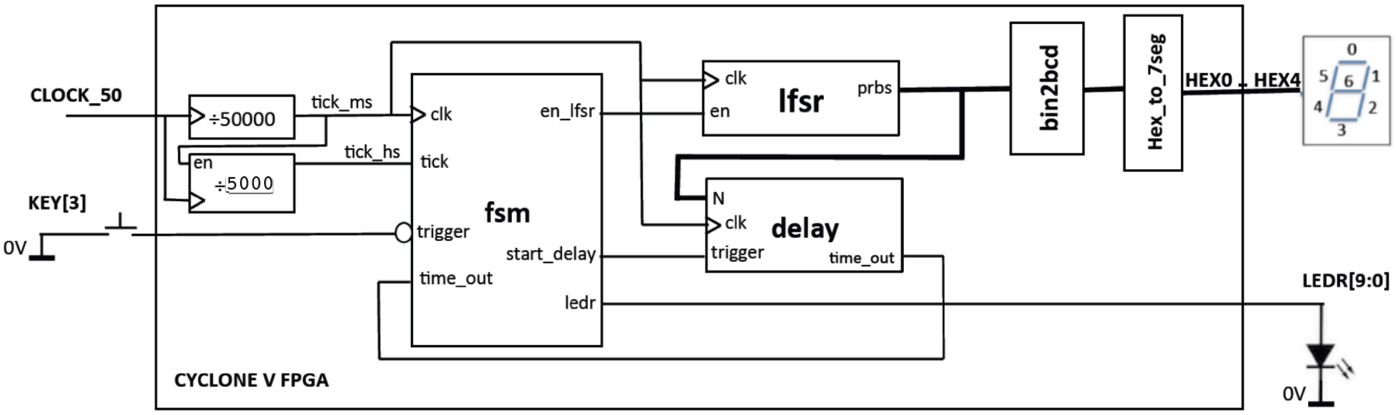
4.0 Experiment 8 (Optional challenge): Starting line delay circuit

The next two experiments are optional. They are designed to provide a challenge to those who finish early, or for those who want to learn more about digital design, Verilog and FPGAs. The two experiments are linked – what you designed in Experiment 8 will be used in Experiment 9.

The goal here is to design a Formula 1 style of race starting lights. The specification of your circuit is:

1. The circuit is triggered (or started) by pressing KEY[3] (don't forget KEY[3] is low active);
2. The 10 LEDs (below the 7-segment displays) will then start lighting up from left to right at 0.5 second interval, until all LEDs are ON;
3. The circuit then waits for a random period of time between 0.25 and 16 seconds before all LEDs turn OFF;
4. You should also display the random delay period in milliseconds on five 7-segment displays.

In order to assist you in designing this circuit without spending too much time, the following overall block diagram of the circuit is provided. You should also download the solution bit-stream for this experiment from the experiment webpage (**ex8sol.sof**) and try it out before attempt it yourself.



In the above diagram, all signals on the left of the block are inputs and the signals on the right are outputs.

The two clock divider circuits provide clock ticks once every 1ms and 0.5sec respectively. Each clock tick should be a positive pulse lasting one period of **CLOCK_50** (i.e. 20ns). The system then use the **tick_ms** signal as the clock of the remaining circuit.

The **LFSR** module produces a pseudo-random binary sequence (**PRBS**), which is used to determine the random delay required. The **enable** signal to the **LFSR** allows this to cycle through a number of clock cycles before it is stopped at a random value.

The **delay** module is triggered after all 10 LEDs are lid, and then provides a delay of **N** clock cycles (at 1ms period) before asserting the **time_out** signal (for 1ms).

The delay value N is fed to the binary to BCD converter, which then drives the 7-segment displays.

There are several design decisions to be made:

1. How many bits LFSR is required?
2. How many bits should you use in the delay module?

The FSM module is the key module to the entire system. You must decide what are the states that are required, draw the state diagram and then map that to Verilog.

5.0 Experiment 9 (Optional challenge): A Reaction Meter

Extend your circuit in Experiment 8 by adding a reaction counter. This should count the time between all the LEDs turning OFF and you pressing **KEY[0]**. The reaction time, instead of the random delay, should be displayed on the 7-segment displays in milliseconds.

Department of Electrical & Electronic Engineering

Imperial College London

2nd Year Laboratory

Experiment VERI: FPGA Design with Verilog (Part 3)

PART 3 – Analogue I/O and SPI serial Interface

1.0 The Add-on Card

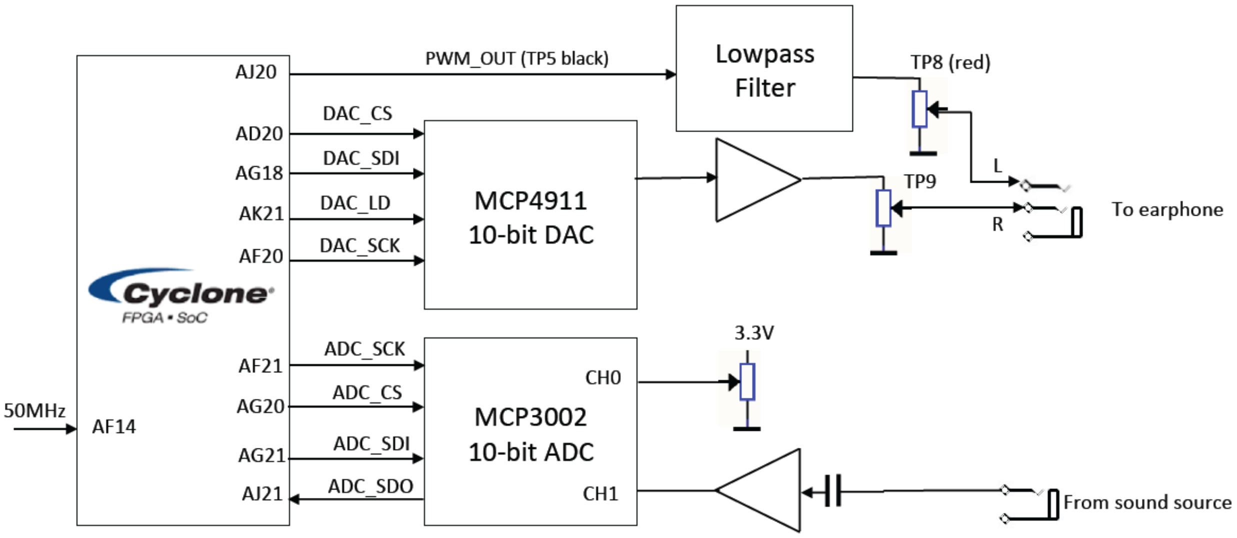
This part of the experiment introduces you to the add-on card to the DE1 board. The add-on card consists of a 10-bit ADC and a 10-bit DAC, a quad op-amp, sockets for earphone (analogue output) and sound source (analogue input), and a potentiometer. The overall block diagram of the add-on card is shown below. It should be plugged into the expansion socket furthest away from the edge of the DE1 board. Beware of the alignment between the plug and the socket. If the add-on board is inserted correctly, the green LED will light up when the DE1 board is turned ON.

You do not need to understand all the circuitry on this board in details. Nevertheless, the schematic diagram and a detail explanation on how this board works, together with all the datasheets of the components used, are provided on the Experiment webpage.

For this part of the experiment, you would need to bring your personal earphone, and from the Lab, get a 3.5mm lead and a digital voltmeter.

By the end of this part of the experiment, you will have:

- Understood and verified the operation of the **S**erial-to-**P**arallel **I**nterface (SPI) of the digital-to-analogue converter (DAC) MCP4911 using Modelsim;
- Tested the DAC and measured its output voltage range;
- Learned how to use a ROM (read-only memory) and a constant coefficient multiplier;
- Use the analogue-to-digital converter (ADC) MCP3002 to convert dc voltages;
- (Finally), designed a sinewave tone generator with variable frequency which is controlled with the slide switches, and the frequency value showed on the 7-segment displays in decimal format.



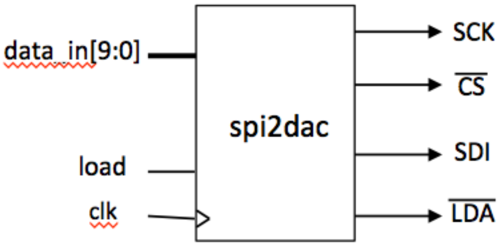
2.0 Experiment 10: Interface with the MCP4911 Digital-to-Analogue Converter

Step 1: Understanding Datasheet - Go to the Experiment website and download the datasheet for the MCP4911 DAC and the file **spi2dac.v**, which is a Verilog module that implements the SPI interface circuit to communicate with the DAC. Make sure that you understand from reading the datasheet:

- the purpose of each pin on the DAC (Section 3.0, page 17 of datasheet);
- how information is sent to the DAC through the serial data input (SDI) pin (Section 5.0, page 23-24);
- how to configure the DAC's internal function (page 25);
- DAC's timing specifications and timing diagram (pages 4 and 7).

There is no need for you to know how exactly the DAC works internally. However, you need to have sufficient appreciation of the serial interface in order to conduct this part of the experiment. Furthermore, don't worry if you don't fully understand the Verilog code in **spi2dac.v**. This will be explained in a Lecture.

Step 2: Timing diagram – The **spi2dac** module takes a 10-bit number in parallel (controlled through the load signal which must be high for at least 20ns) and generates the necessary serial signals to drive the MCP4911 DAC. Based on the information from the Datasheet, draw in your logbook the expected timing diagram of the SPI interface signals when a word 10'h23b is sent to the DAC.

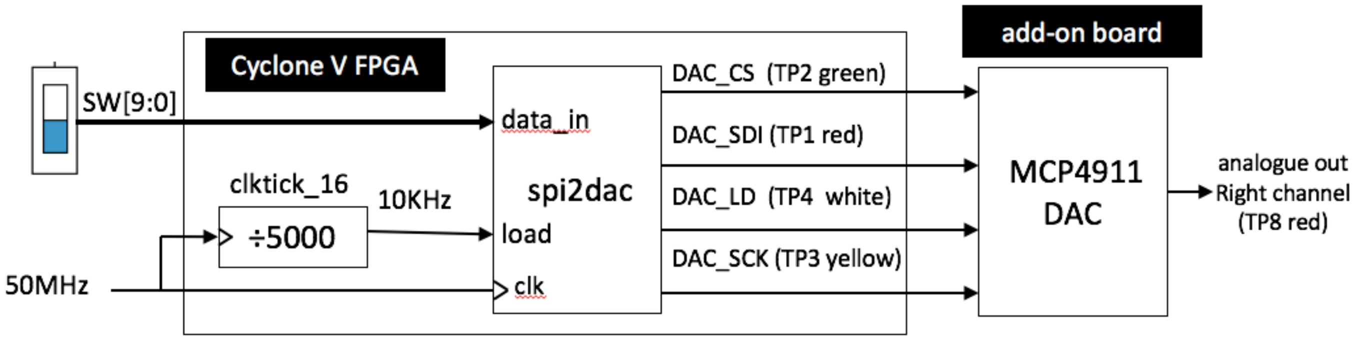


Step 3: Verify timing of spi2dac.v using Modelsim – The steps are:

1. Create a project ex10 and a top-level module ex10_top.v
2. Copy to the directory ex10 the file spi2dac.v downloaded from the webpage
3. Make this file top-level module (for now)
4. Click: … **> Process > Start > Analyze and Synthesise**
5. Start Modelsim (**Tools > Run Simulation Tools > RTL Simulation**)
6. Design a **do-file** as a testbench to exercise the input signals correctly
7. Run the do-file and match the waveform generated with your prediction

Step 4: Testing the DAC on DE1

In order to test the **spi2dac.v** module and verify that DAC works properly, create the top-level design **ex10_top.v** that implements the circuit shown in the following diagram.



The **data_in** value determined by the 10 switches (**SW[9:0]**) is loaded to the **spi2dac** module at a rate of 10k samples per second as governed by the **load** signal. The steps for this part are:

1. Download from the experiment website the file **spi2dac.v**.
2. Check that the **clktick_16.v** module that you used last week is in the **"mylib"** folder.
3. Create a top-level module **ex10_top**.v to connect all modules together as shown in the diagram.
4. Click: **Project > Add/Remove Files in Project** …, and select all the relevant files used here. This step is important – it allows you to select which modules to include in your design.
5. When **ex1__top**.v is the current file, click: **Project > Set as Top-Level Entity**. This is another useful step, which defines the top module, and all those module below this one, for compilation. With steps 4 and 5, you can move up or down the design hierarchy in a project for compilation.
6. Edit the **ex10_top.qsf** file to include **pin_assignment.txt**.
7. Compile and correct errors as necessary.

Once the design is compiled without error, download the bit-stream file to the DE1 board. Using the DVM feature of the scope, measure the DAC output voltage at TP8 for SW[9:0] = 0 and 10'h3ff. (The voltage range of the DAC output should be from 0V to 3.3V.)

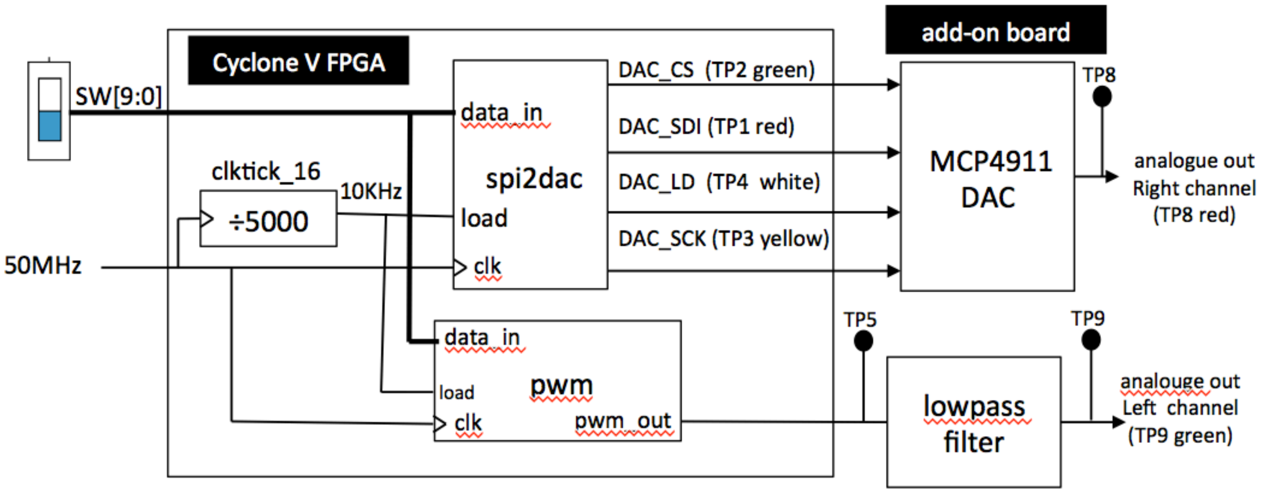
Step 5: Verify the signals on an oscilloscope

Confirm that the signals produced by the FPGA with the **spi2dac.v** module agree with those from Modelsim. Set SW[9:0] to 10'h23b and measure DAC_SCK (TP3) and DAC_SDI (TP1) using an oscilloscope. You may need to trigger the scope externally with the DAC_CS signal (TP2). Compare the waveforms to those predicted by Modelsim.

3.0 Experiment 11: D-to-A conversion using pulse-width modulation

Instead of using a DAC chip (and SPI serial interface to communicate with the chip), an alternative method to produce an analogue output from a digital number is to use pulse-width modulation (PWM). The Verilog code for a **pwm.v** module is given to you in Lecture 9 slide 15.

Create a design **ex11_top.v** according to the circuit shown below. Use the scope to examine the signals at TP5 and TP8. Compare the output voltage ranges at TP8 and TP9.



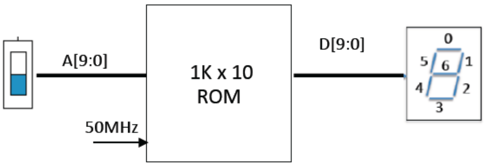
4.0 Experiment 12: Designing and testing a sinewave table in ROM

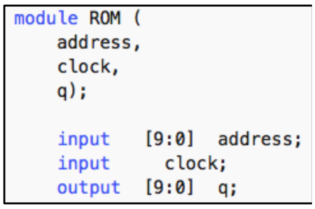
This part of the experiment leads you through the design of a 1K x 10 bit ROM, which stores a table of sine values suitable to drive our DAC. The relationship between the content of the ROM D[9:0] and its address A[9:0] is:

D[9:0] = int(511*sin(A[9:0]*2*pi/1024)+512) for 1023 ≥ A[9:0] ≥ 0

Since the DAC accepts an input range of 0 to 1023, we must add an offset of 512 in this equation. (This number representation is know as **off-set binary** code.)

Before generating the ROM in Quartus using the "**Memory Compiler**" tool, we need to first create a text file specifying the contents of the ROM. This can be done in different ways. Included on the [Experiment webpage]() are: 1) a Python script to do this; 2) a Matlab script to do the same thing; 3) a memory initialization file **rom_data.mif** created by either method. Download these files and examine them.





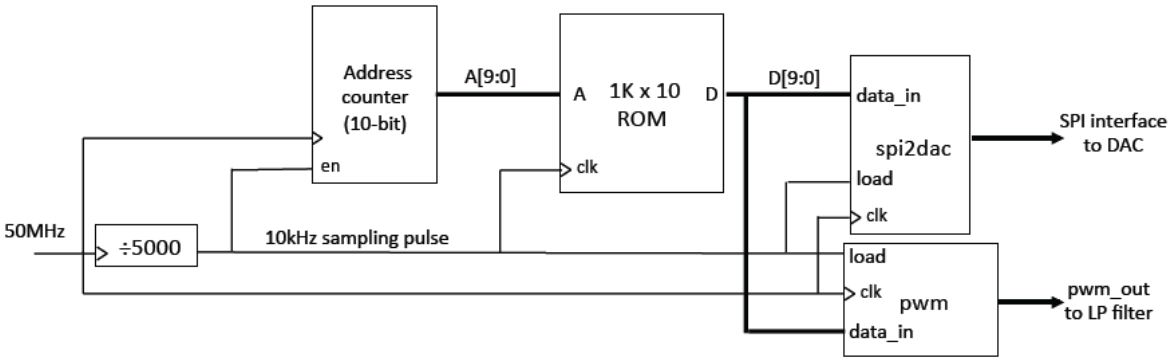
```
module ROM (
    address,
    clock,
    q);

    input   [9:0]  address;
    input      clock;
    output  [9:0]  q;
```

Click Tools > IP Catalog to bring up a tool which helps to create a 1-Port ROM. A catalog window will pop up. Select from the window **>Library >Basic Functions > Onchip Memory > ROM 1-Port.** Complete the on-screen form to create **ROM.v**.

To verify the ROM, create the design **ex12_top**.v, that uses the switches **SW[9:0]** to specify the address to the ROM, and display the contents stored at the specified location on the four 7-segment display. Once this is done and loaded onto the DE1, verify that the contents stored in the ROM matches those specified in the **rom_data.mif** file.

5.0 Experiment 13: A fixed frequency sinewave generator

Let us now replace the slide switches with a 10-bit binary counter and connect the ROM data output to **spi2dac** and **pwm** modules as shown in the figure below. Since the ROM contains one cycle of sinewave and the address to the ROM is incremented every cycle of the 10kHz clock, a perfect sinewave is produced at the left and right outputs of the 3.5mm jack socket.



Implement this circuit and verify that the signals produced by both the DAC and the PWM are as expected. What is the frequency of the sinewave?

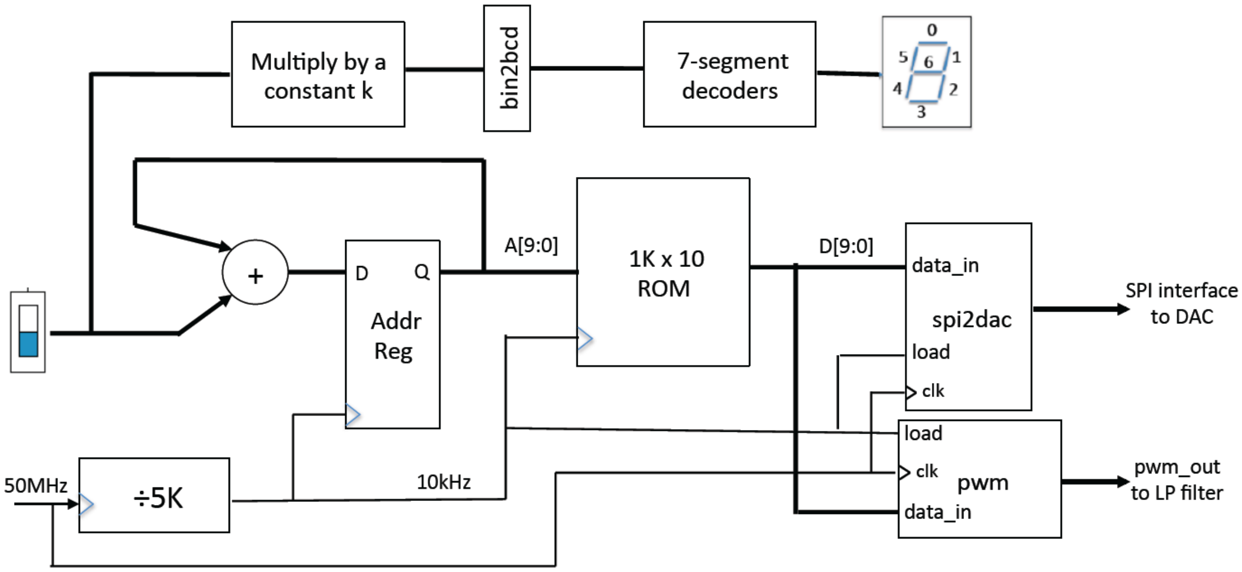
6.0 Experiment 14 (optional challenge): A variable sinewave generator

Combine everything together to produce a design **ex14_top.v**, which produces a variable frequency sinewave using table-lookup method. The sampling frequency is 10kHz, and the sine value is read from the ROM that is preloaded with one-cycle of a sinewave (i.e. the address of the ROM is the phase and the content is the sine value). On every sample period, advance the address (i.e. the phase) by an amount determined by SW[9:0]. Derive the relationship between the output signal frequency and the switch setting.

The overall block diagram is shown below. The switch setting is multiplied by a constant k to convert the phase increment SW[9:0] to frequency.

You can produce a 10-bit x 14-bit constant coefficient multiplier using the IP catalog tool. The 14-bit constant is 14'h2710 (which is 14'd10000). The product is a 24-bit number, and the frequency is the top 14-bits (Why?). The frequency can then be displayed on the 7-segment displays.

Produce a 439Hz sinewave, which is close to 440Hz, the frequency commonly found in tuning forks. Make sure that this is indeed correct (through listening or measuring with a frequency counter).



7. Experiment 15 (Optional Challenge): Using the A-to-D converter

In this experiment, you will learn to use A-to-D converter MCP3002 on the add-on board to convert analogue voltages to digital signals. Again, download from the webpage the **spi2adc.v** module, which is already written for you to use.

Instead of using the slide switches to control the frequency, use the A-to-D converter to convert the dc voltage of the potentiometer (which is between 0v and 3.3v) and use this converter value instead.

To help you know what you should aim for, the solutions for **ex14sol.sof** and **ex15sol.sof** are available to download.

Department of Electrical & Electronic Engineering

Imperial College London

2nd Year Laboratory

Experiment: FPGA Design with Verilog (Part 4)

PART 4 – Real-time Audio Signal Processing

1.0 Putting everything together

In this part of the experiment, you will learn to combine the ADC with the DAC on the Add-on card, and use the DE1 to perform some simple audio processing.

The goal of the final week's laboratory session is to implement a **speech echo effect synthesizer**. You need to bring your earphone to the lab in order to listen to the audio output.

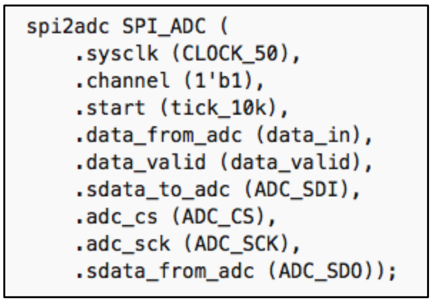
2.0 Experiment 16: An audio in-and-out (all pass) loop

Download from the Experiment webpage the file ex16_proto.zip, which contains the prototype folder for this experiment.

- Examine the contents within this folder. You should find the following Verilog files:

Module	Function
ex16_top.v	Top-level design; interface to pins
spi2dac.v	SPI interface circuit to DAC from Part 3
spi2adc.v	SPI interface circuit to ADC
pwm.v	Pulse-width modulation DAC from Part 3
clktick_16.v	Clock divider to generate sampling clock ticks at 10kHz from Part 2
pulse_gen.v	Generate a one-cycle pulse on rising edge of a trigger signal
hex_to_7seg.v	Hex to 7-segment decoder from Part 1
allpass.v	"**processor**" module – this performs processing, which simply passes input to output.

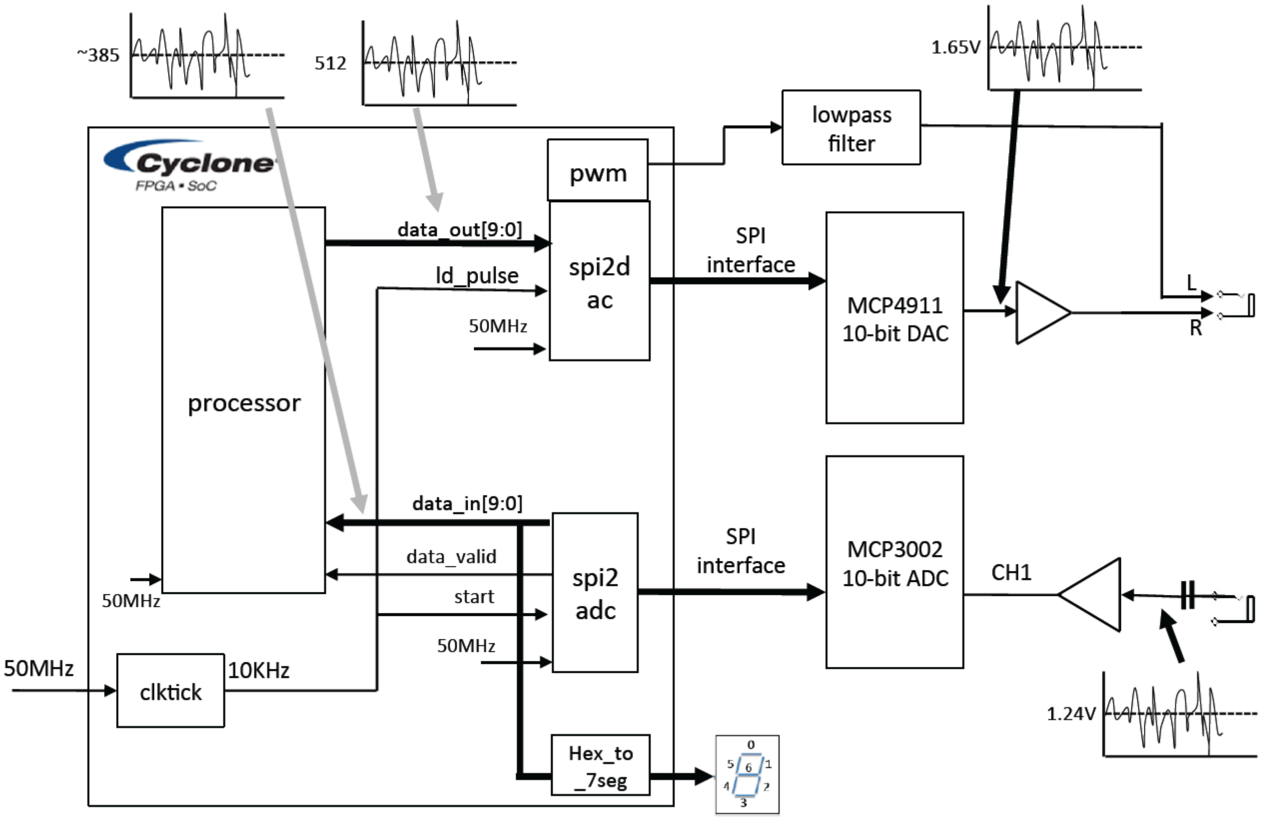
- Study **ex16_top.v**. This specifies a system as shown in the following diagram (the part inside the Cyclone V). Make sure you understand how this works.
- Note how the **spi2adc.v** module is used. Explicitly associating the signal names INSIDE the module (e.g. sysclk) to OUTSIDE (e.g. CLOCK_50) allow connections to be defined independent of the order. This is a more verbose but is a much safer way to make connections to modules.



```
spi2adc SPI_ADC (
      .sysclk (CLOCK_50),
      .channel (1'b1),
      .start (tick_10k),
      .data_from_adc (data_in),
      .data_valid (data_valid),
      .sdata_to_adc (ADC_SDI),
      .adc_cs (ADC_CS),
      .adc_sck (ADC_SCK),
      .sdata_from_adc (ADC_SDO));
```

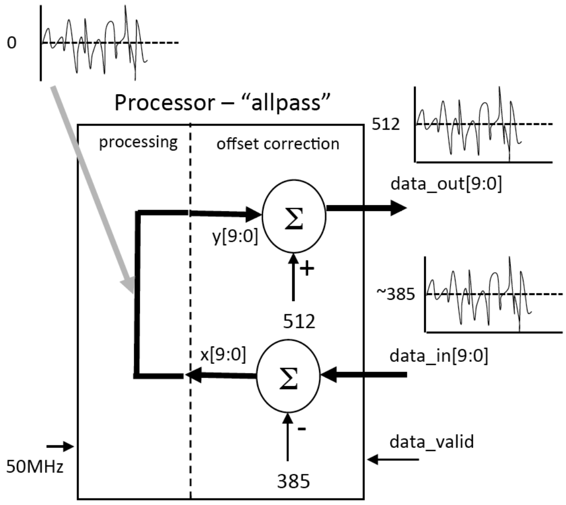
- The ADC has two analogue input channels: CH0 and CH1. They connected to the potentiometer and to the 3.5mm socket respectively. We only use CH1 for **ex16**.
- Now examine the module **allpass.v**. The name of this module is "**processor**" and is different from the name of the Verilog file. There is no need to use the same name except that normally it is more convenient to do so. However, in this case, we have deliberately used the filename "**allpass**" to describe its function, while using a more universal name for the module. You can choose "**allpass.v**" as the source of the module "**processor**" now. Later, you can have a different Verilog file to define a

different "**processor**". Which version of "**processor**" you use in your design is specified in **Project > Add/Remove File in Project**.



- Make sure that you understand fully what the Verilog file "**allpass.v**" does. It actually does very little. It:
  1. Corrects the ADC converter data (which uses offset binary with 0V represented by a value of ~385), but subtracting the offset from **data_out[9:0]** to obtain a 2's complement value **x[9:0]**.
  2. Connects X to Y, i.e. does nothing and hence "allpass".
  3. Converts the Y value from 2's complement to offset binary for the DAC. The offset now is at 512 as shown below.

- Build your design for testing on the DE1 Board. To do this, you should:
  1. Open each .v file, and use **Processing > Analyze Current File** on each of the Verilog file to ensure that there is not syntax error.
  2. Use **Project > Add/Remove** File in Project to include all the .v files you need. Here we select **allpass.v** to supply the "**processor**" module. In the future, you could substitute **allpass.v** with another file for a different processor.
  3. While **ex16_top.v** is the current file in the editor window, use **Project > Set as Top-level Entity** to define top is the top-level module.



4. Use **Project > Start > Analysis and Synthesize** … to check for errors and warnings without compiling everything.
5. Check that Device, Pin and **TimeQuest** clock period are all assigned correctly.
6. Compile the whole design and download the bit-stream file "**ex16_top.sof**" to DE1.
7. Test that it is working properly. You can use the PC to play some music or speech files (downloadable from Experiment webpage), and use an earphone to listen to the DAC output. When no signal is sent to the DE1 board, the display should show a hex value of 181 to 188.

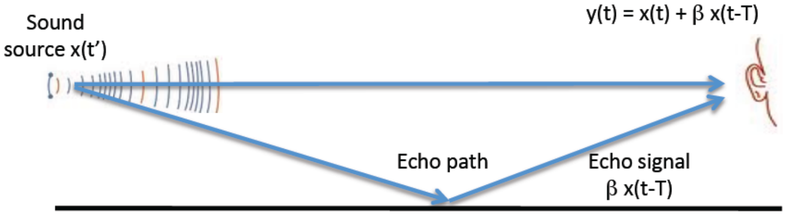
When you get to this part, the experiment framework is shown to be working. It takes audio samples at 10kHz from the ADC, passes it through a processor module and output the processed sample to the DAC.

Test yourself

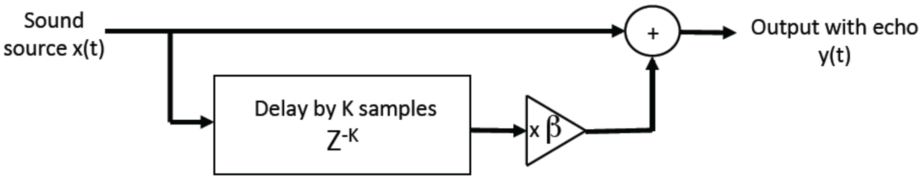
Now create a new Verilog file **mult4.v** which is a processor module (i.e. module name is still "processor"), that amplifies the input by a factor of four. Test that this is working (i.e. the signal to the earphone should be louder or distorted). The easiest way to multiple by 4 is to perform arithmetic left shift by 2 bits.

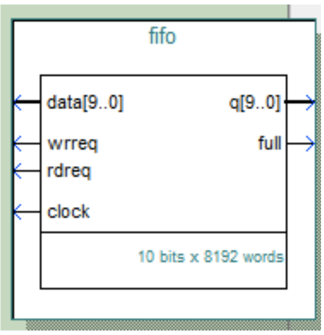
3.0 Experiment 17: Echo Synthesizer with fixed delay

In this part of the experiment, you will design, implement and test a circuit that simulates the effect of simple echo. The diagram below shows two components of a sound source reaching its listener: the direct path signal x(t) and the echo signal β x(t-T) which is a weaker version of x(t) attenuated by a factor β, bounced off the floor. The echo signal is also delayed by T relative to the direct-path signal x(t).



Such simple echo can be implemented as signal flow graph as shown below. This involves three components: a delay block that delays x(t) by K sample periods; a gain block which multiplies the delayed signal by the factor β; and the adder.





The delay block can be implemented with a first-in-first-out (FIFO) buffer. A FIFO is found in all forms of digital systems. The rule is simple: received data are stored in sequence in such a way that they can be retrieved in the order that they arrive. When a new data item arrives and the FIFO is not full, it is written to the FIFO. As a stored data item is retrieved, it is removed from the FIFO. This allows the send and retrieve rates to be different in the short term. If the send rate is higher than retrieve rate, eventually the buffer will get full. If the buffer is full, it should not receive any more data (otherwise existing store data would be corrupted). A “full” status signal is asserted to tell the sender not send any more data. Similarly if the buffer is empty, it cannot provide any data for retrieval. An “empty” status signal is used to indicate that the FIFO has no more data to provide.

Create a new project using the files from Experiment 16 as your prototype. With IP Catalog tool, generate a FIFO component of size 8192 x 10-bit as shown here. You only need to provide only the “full” status signal. This FIFO is used to store the most recent 8192 samples, hence providing a delay of 0.8192msec since the sampling frequency is 10KHz. Before the echo simulation circuit starts to provide the echo, the FIFO must first be completely filled (i.e. wait until the “full” signal is asserted). Thereafter, the writing of the ADC sample and DAC sample is synchronous, and the FIFO remains full. The read data is always the write data delayed by 8192 sample period.

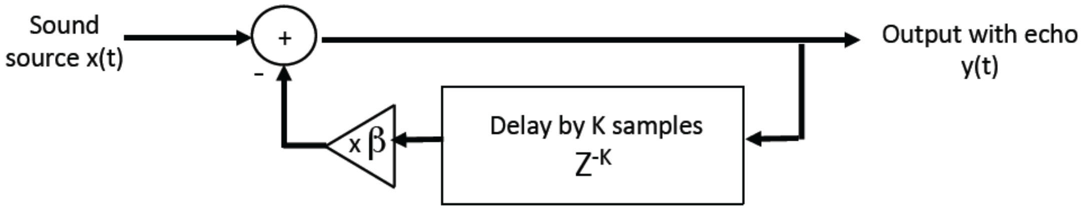
The attenuation factor β should be ½ or ¼, which can easily be implemented with a simply binary shift.

Deliverable

Implement the simple echo simulator and test that it works. For the purpose of test, download three different sound files: clapping.mp3, hello.mp3 and hitchhiker.mp3, and play them on the PC or phone in a loop. Use your earphone to listen to the effect of the echo synthesizer.

4.0 Experiment 18: Multiple echoes

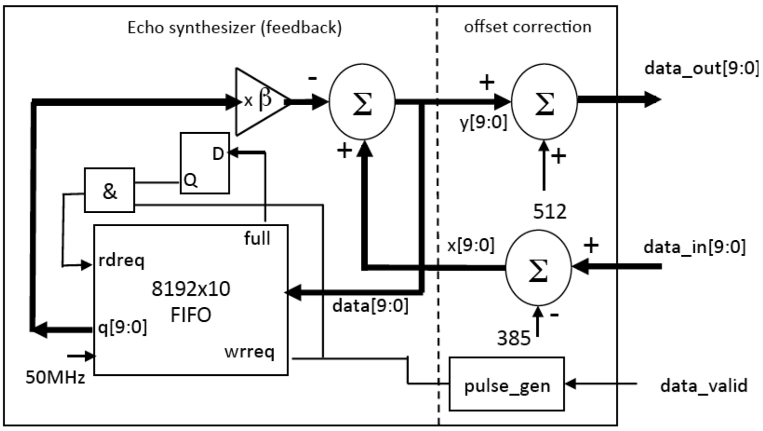
The design in Experiment 17 produces a single echo. The signal flow graph only has feedforward paths. Multiple echoes can be produce with a slight modification of the signal flow graph to the one shown below.



The delay block now stores the output sample y(t) instead of the input sample x(t). The attenuated and delayed y(t) is SUBTRACTED from x(t) to produce the next output. (Why must this be a subtract and not an add?)

Provide a design to implement this architecture and test it.

Processor – multiple echoes

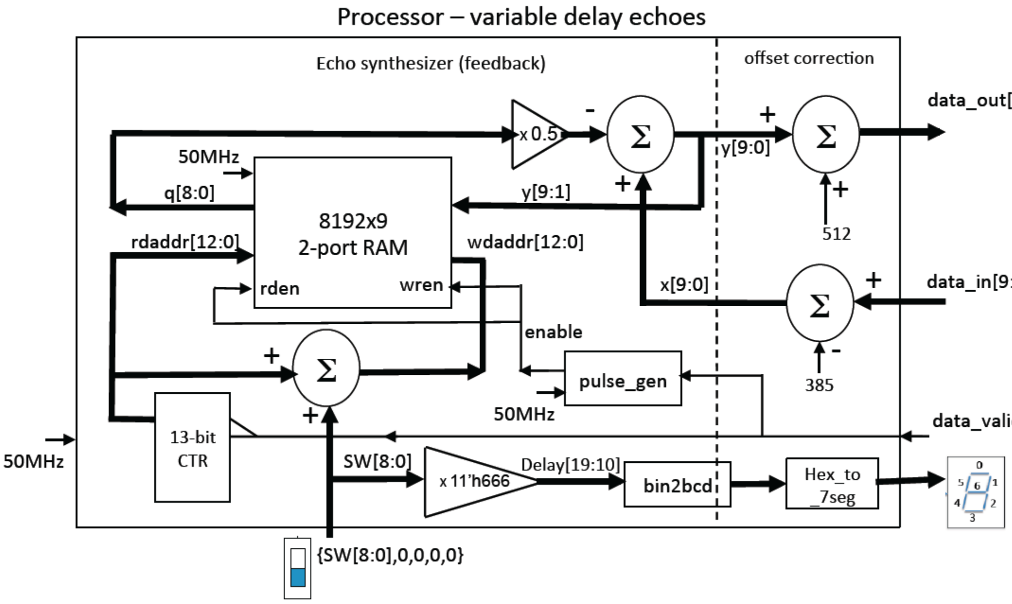


5.0 Experiment 19 (Optional challenge): Echo Synthesizer with Variable delay

In this experiment, you will design, implement and test a system with variable delay. A bit-stream (echo.sof) that implements a solution can be downloaded from the Experiment webpage. You also need to download the three MP3 test files. Connect the audio input to the speaker of the PC and play the audio files in a loop. Program DE1 with echo.sof and listen to the output with your earphone. Change the delay of the echo with SW[8:0]. The amount of delay in millisecond is displayed on the 7-segment displays as a decimal number.

The design of this experiment is shown in the block diagram below. It consists of a number of modules:

- RAM Delay Bock - In place of the FIFO to implement the delay block, it uses a 2-port RAM block (8192 x 9-bit) – one write port (to store the ADC samples) and one read port.
- Address Generator - A 13-bit counter is used to generate the read address to the RAM. (Why 13-bits?) The counter value is incremented on the negative edge of the data_valid signal at a frequency of 10KHz. In this way, the address generator computes the address used on the next read and write cycle. The write address is generated from the read address by adding the value taken from SW[8:0]. Since the address is 13-bits wide, the 9-bit delay value is zero-padded in its lower 4 bits. Therefore, the delay between the read and write samples is: SW[8:0] x 16 x 0.1 msec.
- The read and write enable signals are common, and it is generated from the data_valid signal with the pulse_gen module.
- The write data value y[9:1] is 9-bit instead of 10-bit wide. This is because the embedded memory in the Cyclone III FPGA is configurable as 9-bit in data width, but not 10-bit. Therefore the output data value is truncated to 9-bit before storing in the delay block.
- The read data value is of course also 9-bit wide. Therefore the x0.5 can easily be implemented by sign-extending the 9-bit value to 10-bit: {q[8],q[8:0]}.
- The implementation of the feedback loop to generate the echo effect is identical to that from the previous experiment.
- To display the delay value in milliseconds, the value of SW[8:0] is first multiplied by 1638 (why) with a constant multiplier. This gives a 20-bit product, the most significant 10-bits of which is the delay in milliseconds. (Why?) This is then converted from binary to BCD and decoded for display on the 7-segment displays.

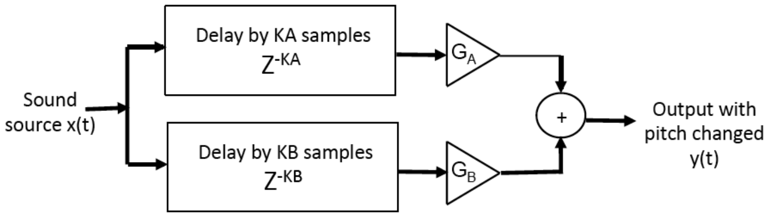


6.0 Experiment 20: Voice Corruptor (Not part of this Lab)

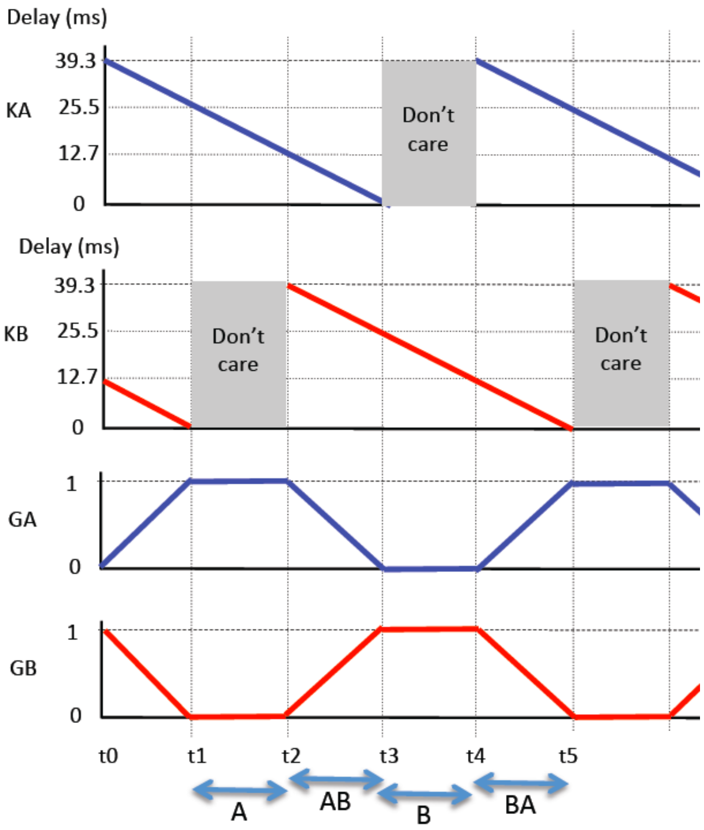
This part of the experiment is outside the scope of the experiment. It is designed to provide you with an open problem so that you can explore designing digital systems and implementing digital circuits using the DE1 and the add-on card at your own leisure. You need to check out a set of kits to take home from stores. For example, you might want to try this out over the Christmas break.

You are now equipped with all the tools and knowledge to design a reasonably complex audio processing system. The idea here is to design something that will take a human speech signal and then "corrupt" in a way that the identity of the speaker is masked while the speech remain intelligible.

One way to do this is to change the pitch of the speaker (e.g. make it sounds like Donald Duck). There are many ways to perform pitch change of speech. One method, which is linked to the previous experiments, is to employ a technique based on cross fading (i.e. combining) of two separately delayed version of the speech signal. The technique is depicted in the block diagram below.



The sound source is delayed through two separate blocks, providing KA and KB sample delays, which vary with time. The delayed signals are then attenuated by GA and GB, and combined with the adder. In order to minimize the artifacts and discontinuities in the output signal and to maintain a constant volume, the gain values GA and GB are designed to cross fade with each other – i.e. when one is ramping up (from 0 to 1), the other is ramping down. A plot of the four parameters, KA, KB, GA and GB, vs time is shown below.



There are four regions.

1. Region A (t1 to t2) - Only channel A is contributing to the output. The delay KA is gradually decreasing linearly from 25.5ms to 12.7ms (255 to 127 x 100μs). The gain GA is constant at 1.
2. Region AB (t2 to t3) – Both channels contribute to the output with A decreasing and B increasing their respective contributions. The two channels are cross faded before GA drops from 1 to 0 while GB increases in the other direction.
3. Region B (t3 to t4) – This is similar to Region A, but the behavior applies to channel B instead of channel A.
4. Regions BA (t4 to t5) – Similar to Region AB, but the two channels are reversed.

The pattern repeats itself indefinitely. Note that the "don't care" portion of KA and KB is due to the fact that during this period, the gain GA or GB is zero.

Hints:

- Initially, try the delay ramping gradient of 0.5, i.e. the delay is dropped by k over time 2*k.
- You can use a 9-bit down counter to define both the delay KA and the four regions.
- You can derive all other values: KB, GA and GB, from the counter values.
- You can design a four state synchronous state machine to control the corruptor circuit.
- Instead of delay varying ramping high to 0, you can reverse the direction of the ramping. Alternative you can design the delay to vary up and then down.
- In addition to pitch changes, you may explore other audio effects.

QUALIS™

Verilog HDL QUICK REFERENCE CARD

Revision 2.1

()	Grouping	[ ]	Optional
{}	Repeated	|	Alternative
bold	As is	CAPS	User Identifier

1. Module

module MODID[({PORTID,})];
  [**input** | **output** | **inout** [range] {PORTID,};]
  [{declaration}]
  [{parallel_statement}]
  [specify_block]
endmodule

range ::= [constexpr **:** constexpr]

2. Declarations

parameter {PARID = constexpr,};

wire | **wand** | **wor** [range] {WIRID,};

reg [range] {REGID [range],};

integer {INTID [range],};

time {TIMID [range],};

real {REALID,};

realtime {REALTIMID,};

event {EVTID,};

task TASKID;
  [{**input** | **output** | **inout** [range] {ARGID,};}]
  [{declaration}]
begin
  [{sequential_statement}]
end
endtask

function [range] FCTID;
  {**input** [range] {ARGID,};}
  [{declaration}]
begin
  [{sequential_statement}]
end
endfunction

3. Parallel Statements

assign [(strength1, strength0)] WIRID = expr;

initial sequential_statement

always sequential_statement

MODID [**#**({expr,})] INSTID
  ([{expr,} | {.PORTID(expr),}]);

GATEID [(strength1, strength0)] [#delay]
    [INSTID] ({expr,});

defparam {HIERID = constexpr,};

strength ::= **supply** | **strong** | **pull** | **weak** | **highz**

delay ::= number | PARID | (expr [, expr [, expr]])

4. Gate Primitives

and (out, in$_1$, ..., in$_N$);   **nand** (out, in$_1$, ..., in$_N$);
or (out, in$_1$, ..., in$_N$);   **nor** (out, in$_1$, ..., in$_N$);
xor (out, in$_1$, ..., in$_N$);   **xnor** (out, in$_1$, ..., in$_N$);

buf (out$_1$, ..., out$_N$, in);   **not** (out$_1$, ..., out$_N$, in);

bufif0 (out, in, ctl);   **bufif1** (out, in, ctl);
notif0 (out, in, ctl);   **notif1** (out, in, ctl);

pullup (out);   **pulldown** (out);

[r]**pmos** (out, in, ctl);
[r]**nmos** (out, in, ctl);
[r]**cmos** (out, in, nctl, pctl);

[r]**tran** (inout, inout);
[r]**tranif1** (inout, inout, ctl);
[r]**tranif0** (inout, inout, ctl);

5. Sequential Statements

;

begin[**:** BLKID
  [{declaration}]]
  [{sequential_statement}]
end

if (expr) sequential_statement
[**else** sequential_statement]

case | **casex** | **casez** (expr)
  [{{expr,}**:** sequential_statement}]
  [**default:** sequential_statement]
endcase

forever sequential_statement

repeat (expr) sequential_statement

while (expr) sequential_statement

for (lvalue = expr; expr; lvalue = expr)
  sequential_statement

#(number | (expr)) sequential_statement

@ (event [{**or** event}]) sequential_statement

lvalue [**<**]= [**#**(number | (expr))] expr;

lvalue [**<**]= [**@** (event [{**or** event}])] expr;

wait (expr) sequential_statement

-> EVENTID;

fork[**:** BLKID
  [{declaration}]]
  [{sequential_statement}]
join

TASKID[({expr,})];

disable BLKID | TASKID;

assign lvalue = expr;

deassign lvalue;

lvalue ::=
  ID[range] | ID[expr] | **{**{lvalue,}**}**

event ::= [**posedge** | **negedge**] expr

6. Specify Block

specify_block ::= **specify**
    {specify_statement}
  endspecify

6.1. Specify Block Statements

specparam {ID = constexpr,};

(terminal **=>** terminal) = path_delay;

((terminal,} ***>** {terminal,}) = path_delay;

if (expr) (terminal [+|-]**=>** terminal) = path_delay;

if (expr) ({terminal,} [+|-]***>** {terminal,}) =
    path_delay;

[**if** (expr)] ([**posedge**|**negedge**] terminal **=>**
    (terminal [+|-]**:** expr)) = path_delay;

[**if** (expr)] ([**posedge**|**negedge**] terminal ***>**
    ({terminal,} [+|-]**:** expr)) = path_delay;

$setup(tevent, tevent, expr [, ID]);

$hold(tevent, tevent, expr [, ID]);

$setuphold(tevent, tevent, expr, expr [, ID]);

$period(tevent, expr [, ID]);

$width(tevent, expr, constexpr [, ID]);

$skew(tevent, tevent, expr [, ID]);

$recovery(tevent, tevent, expr [, ID]);

tevent ::= [**posedge** | **negedge**] terminal
    [**&&&** scalar_expr]

path_delay ::=
  expr | (expr, expr [, expr [, expr, expr, expr]])

terminal ::= ID[range] | ID[expr]

7. Expressions

primary

unop primary

expr binop expr

expr **?** expr **:** expr

primary ::=
literal | lvalue | FCTID({expr,}) | **(** expr **)**

7.1. Unary Operators

+, -	Positive, Negative
!	Logical negation
~	Bitwise negation
&, ~&	Bitwise and, nand
|, ~|	Bitwise or, nor
^, ~^, ^~	Bitwise xor, xnor

7.2. Binary Operators

Increasing precedence:

?:	if/else
||	Logical or
&&	Logical and
|	Bitwise or
^, ^~	Bitwise xor, xnor
&	Bitwise and
==, != , ===, !==	Equality
<, <=, >, >=	Inequality
<<, >>	Logical shift
+, -	Addition, Subtraction
***, /, %**	Multiply, Divide, Modulo

7.3. Sizes of Expressions

unsized constant		32
sized constant		as specified
i *op* j	+,-,*,/,%,&,|,^,^~	max(L(i), L(j))
op i	+, -, ~	L(i)
i *op* j	===, !==, ==, !=	
	&&, ||, >, >=, <, <=	1
op i	&, ~&, |, ~|, ^, ~^	1
i *op* j	>>, <<	L(i)
i ? j : k		max(L(j), L(k))
{i,...,j}		L(i) + ... + L(j)
{i{j,...k}}		i * (L(j)+...+L(k))
i = j		L(i)

8. System Tasks

* indicates tasks not part of the IEEE standard
but mentioned in the informative appendix.

8.1. Input

$readmemb("fname", ID [, startadd [, stopadd]]);
$readmemh("fname", ID [, startadd [, stopadd]]);
***$sreadmemb(**ID, startadd, stopadd {, string});
***$sreadmemh(**ID, startadd, stopadd {, string});

8.2. Output

$display[defbase]([fmtstr,] {expr,});
$write[defbase] ([fmtstr,] {expr,});
$strobe[defbase] ([fmtstr,] {expr,});
$monitor[defbase] ([fmtstr,] {expr,});
$fdisplay[defbase] (fileno, [fmtstr,] {expr,});
$fwrite[defbase] (fileno, [fmtstr,] {expr,});
$fstrobe(fileno, [fmtstr,] {expr,});
$fmonitor(fileno, [fmtstr,] {expr,});
fileno = **$fopen("**filename**")**;
$fclose(fileno);

defbase ::= **h** | **b** | **o**

8.3. Time

$time	"now" as TIME
$stime	"now" as INTEGER
$realtime	"now" as REAL
$scale(hierid)	Scale "foreign" time value
$printtimescale[(path)**]**	Display time unit & precision
$timeformat(unit#, prec#, "unit", minwidth**)**	Set time %t display format

8.4. Simulation Control

$stop	Interrupt
$finish	Terminate
***$save("fn")**	Save current simulation
***$incsave("fn")**	Delta-save since last save
***$restart("fn")**	Restart with saved simulation
***$input("fn")**	Read commands from file
***$log[("fn")]**	Enable output logging to file
***$nolog**	Disable output logging
***$key[("fn")]**	Enable input logging to file
***$nokey**	Disable input logging
***$scope**(hiername)	Set scope to hierarchy
***$showscopes**	Scopes at current scope
***$showscopes(1)**	All scopes at & below scope
***$showvars**	Info on all variables in scope
***$showvars(**ID**)**	Info on specified variable
***$countdrivers(**net**)**	>1 driver predicate
***$list**[(ID)]	List source of [named] block
$monitoron	Enable $monitor task
$monitoroff	Disable $monitor task
$dumpon	Enable val change dumping
$dumpoff	Disable val change dumping
$dumpfile("fn")	Name of dump file
$dumplimit(size)	Max size of dump file
$dumpflush	Flush dump file buffer
$dumpvars(levels [{, MODID | VARID}]**)**	Variables to dump
$dumpall	Force a dump now
***$reset[(0)]**	Reset simulation to time 0
***$reset(1)**	Reset and run again
***$reset(0|1,** expr**)**	Reset with reset_value
***$reset_value**	Reset_value of last $reset
***$reset_count**	# of times $reset was used

8.5. Miscellaneous

$random[(ID)]	
***$getpattern**(mem)	Assign mem content
$rtoi(expr)	Convert real to integer
$itor(expr)	Convert integer to real
$realtobits(expr)	Convert real to 64-bit vector
$bitstoreal(expr)	Convert 64-bit vector to real

8.6. Escape Sequences in Format Strings

n, **t**, \\\\, \\"	newline, TAB, '\\', '"'
\\xxx	character as octal value
%%	character '%'
%[w.d]**e**, %[w.d]E	display real in scientific form
%[w.d]**f**, %[w.d]F	display real in decimal form
%[w.d]**g**, %[w.d]G	display real in shortest form
%[0]**h**, %[0]H	display in hexadecimal
%[0]**d**, %[0]D	display in decimal
%[0]**o**, %[0]O	display in octal
%[0]**b**, %[0]B	display in binary
%[0]**c**, %[0]C	display as ASCII character
%[0]**v**, %[0]V	display net signal strength
%[0]**s**, %[0]**S**	display as string
%[0]**t**, %[0]T	display in current time format
%[0]**m**, %[0]**M**	display hierarchical name

9. Lexical Elements

hierarchical identifier ::=	{INSTID **.**} identifier
identifier ::=	letter | _ { alphanumeric | **$** | _}
escaped identifer ::=	\\{nonwhite}
decimal literal ::=	[**+**|**-**]integer [**.** integer] [E|e[**+**|**-**] integer]
based literal ::=	integer **'** base {hexdigit | **x** | **z**}
base ::=	**b** | **o** | **d** | **h**
comment ::=	// comment newline
comment block ::=	/* comment */

Qualis Design Corporation
Elite Consulting and Training in High-Level Design

Phone: +1-503-670-7200 FAX: +1-503-670-0809
E-mail: info@qualis.com com
Web: http://www.qualis.com

Also available: VHDL Quick Reference Card
1164 Packages Quick Reference Card