Imperial College
London

Lecture 10

Experiment VERI – an Overview

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/ee2_digital/
E-mail: p.cheung@imperial.ac.uk

Lecture Objectives

- Overall aim and objective of the Lab experiment VERI
- Why keep logbook? How will this be assessed?
- How to keep a good design structure and convention?
- Learning outcomes of each of the FOUR PARTS in VERI
- Steps required in producing a completed design
- Meaning of the various View Panes
- Modelsim – some details
- "do" file as testbench

This lecture is not the same as previous ones. I am not teaching you any new concept, architecture or circuit. Instead, I will go through the entire VERI Lab Experiment in order that you appreciate what I want you to learn in each of the four parts.

I will also point out the various pitfalls that students always make each year, and some of the useful "tricks of the trade".

Overview of the four parts in VERI

Parts	What you will learn?
Part 1	Why Verilog HDL is much better than schematic capture?
Part 2	How to design counters circuits and simple FSM? How to cascade multistage counters in the right way?
Part 3	How does SPI serial interface works? How to use ROM, multiplier and DAC to produce a sinewave of different frequencies? Compare the DAC output and PWM output.
Part 4	How to perfect ADC and DAC in an audio processing system? Create a real-time echo chamber effect.

- VERI is designed to teach you digital electronics in four steps.
- Each part is built upon the previous part.
- Each part has its own clearly defined learning outcomes.
- Each part has an optional section for those who go faster or want to do more.
- VERI is the ONLY WAY you learn how to design digital circuits using a hardware description language. You will not really learn through lectures and tutorials alone!

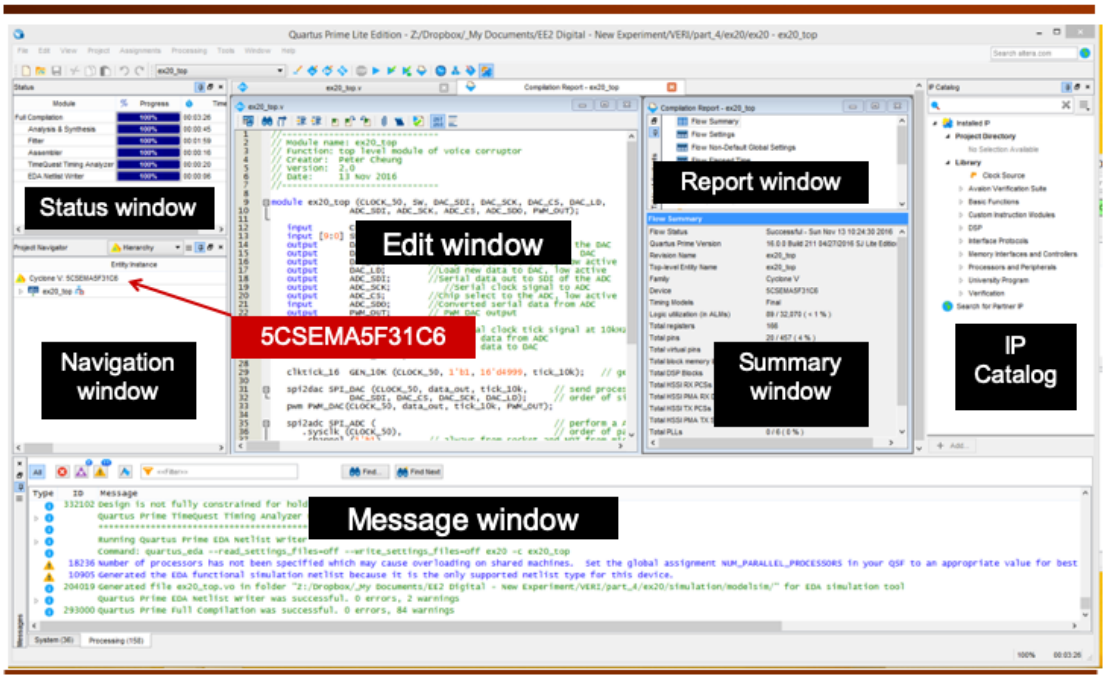
PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 3

VERI is organised in four sequential parts, each build upon the previous parts. Each part has been designed with very clear learning outcomes in mind, and is intended to take a 3-hour supervised laboratory session.

You need to download various files from the Experiment website in order to do this experiment. These files can be found on:

http://www.ee.ic.ac.uk/pcheung/teaching/E2_experiment/

Quartus Prime and window panes



PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 4

You MUST use Quartus 16 (known as Quartus Prime) standard edition and NOT Quartus 13 (known as Quartus II) because version 13 does not support Cyclone V FPGA chips. I recommend that you create a shortcut on your desktop for convenience.

Once you started Quartus software, you will eventually see many window panes appearing in the Quartus window:

Edit window – You should use this to edit all your source files. The editor is basic, but it is syntax sensitive, so it will highlight Verilog keywords for your.

Message window – This is where you find all the error and warning messages.

Navigation window – This shows the hierarchy of your design and provides a quick way of exploring various Verilog files.

Status window – This tells you the steps that the Quartus software is taking in order to produce the final design.

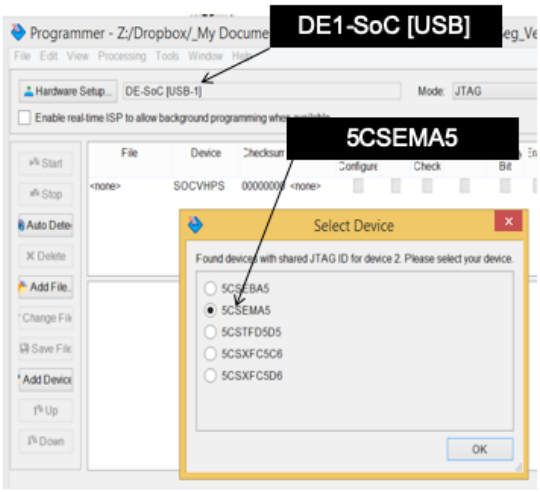
Summary window – This provides a quick summary of the resources being used by your design – useful to check for overall errors.

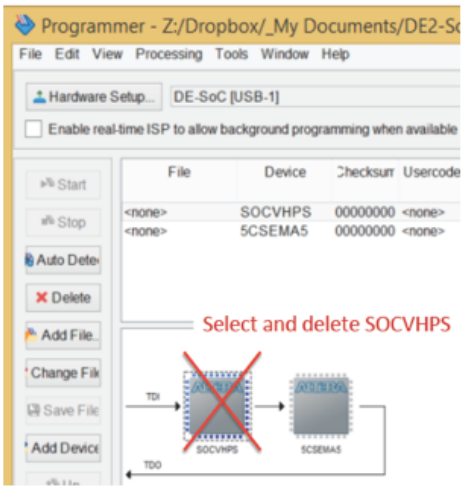
Report window – This is where you find out details of the compilation results such as timing and pin allocations.

IP catalog – This allows you to pick up modules in the component library provided by Altera, such as ROM, multiplier, FIFO etc.

You must also make sure that you have specified the exact FPGA device used on the DE1-SoC board. It is 5CSEMA5F31C6 and it is specified using **> Assignments > Device ….**

Programming the FPGA chip





- Each FPGA is programmed using something called "JTAG". Here we specify exactly which device family – 5CSEMA5.
- The chip has two parts, the FPGA part and the ARM processor part (known as HPS). You need to delete SOCVHPS in the list because we are not using this.

Once you have finished creating your design through hardware compilation, we need to send the bit-stream to the chip via the USB cable. The method of programming is specified in Hardware Setup, and we specify DE1-SoC [USB].

Next we use "Auto Detect" to find out what FPGA chip is connected, and specify that we expect to find the 5CSEMA5 chip family. This is one of many different variants of Cyclone V FPGAs. Each has a different protocol in programming the device.

Then, we have to delete the part of the chip that we are not using – this the ARM part, known as SOCVHPS. If you were to use the ARM processor (e.g. loading it with Linux), we would need this line in.

After that, you must select the "sof" file to send the chip.

Quartus file types

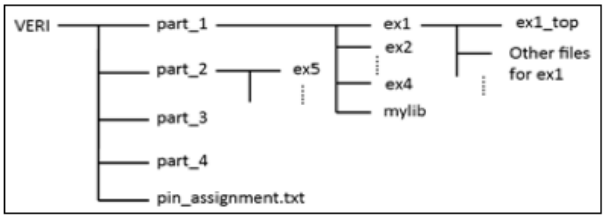
Extension	What is it?
.bdf	Block Design File - schematic diagram
.bsf	Block Symbol File - component symbol
.cdf	Chain Description File - ignore this
.do	DO file - Testbench in Modelsim
.mif	Memory Init File - Contents of ROM
.qpf	Quartus Project File - Specify project
.qsf	Quartus Setting File - Modules and pin assignments
.rpt	Report File - text file reporting on various things
.sof	SRAM Oject File - Bitstream file to program FPGA
.v	Verilog source file - your design

- Quartus uses and generates many different files associated with even then smallest design.
- Here are some of the more commonly used file types and what they mean. It is provided here for your reference and convenience.

You can find a full list of file types used by Quartus on the Experiment webpage.

House Keeping Issues

- Important to keep a tidy directory tree.
- Example shown here could be used.



- Recommended you use ex1, ex2, … etc for project names for the 20 separate experiments in VERI (some are optional). Create these in separate directories.
- Use ex1_top.v, ex2_top.v …. etc as the top-level module in each. Top-level modules is the one that you connect to the pins of the FPGA.
- Put those entities (modules) that you have verified into "mylib" folder. They are used to built up the various parts of VERI.
- Logbook keeping has TWO main purposes:
  1. To help you plan what you want to do and to reflect what you have done;
  2. To remind you what you have done and why in the future.
- You can use paper logbook or electronic logbook (e.g. Github).
- You will need to refer to the logbook when you have your oral in the last week of term.

Part 1 – Wean you from using schematics

- Ex 1 - create 7 segment decode module in schematic
- Ex 2 – create 7 segment decode in Verilog – see how much easier it is!
- Ex 3 – test yourself by extending this to display 10 slide switch value on three displays
- **Ex 4 – optional part to display the 10-bit value as decimal digits on four displays**

- Steps needed in creating a working design:
  1. Create a new project: **> File > New Project Wizard**
  2. Specify FPGA Devices: **> Assignments > Device ….**
  3. Create Verilog specification of various modules: **> New**
  4. Check for syntax errors: **> Processing > Analyze Current File**
  5. Specify which module is top-level: **> Project > Set as Top-Level Entity**
  6. Include all other modules used: **> Project > Add/Remove Files in Project**
  7. Specify pin assignment: **> Open > <top-level-file>.qsf**
     then: **> Edit > Insert File …..**
  8. Full compilation: **> Processing > Start Compilation**
  9. Program the device: **> Tools > Programming**

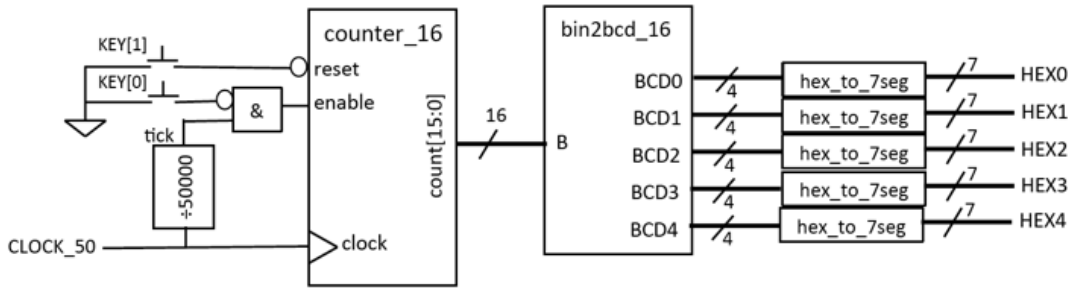
PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 8

The main goal of Part 1 is to let you get familiar with the entire design process.

Shown here is a summary of all the main steps that you have to go through to create a working design to send to the DE1-SoC.

Part 2 – Counters & FSMs

- Ex 5 – learning Modelsim and simulation of an 8-bit counter; how to create a testbench as a DO file in Modelsim
- Ex 6 – test yourself by creating your own 16-bit counter and display the counter output on five 7-segment displays as decimal number; check the maximum working frequency of the counter; cascading two counters to slow clock down.
- Ex 7 – create a linear feedback shift register (LFSR) to generate a pseudo random binary sequence (PRBS).
- **Ex 8 – optional part to simulate Formula 1 style starting light sequence**
- Ex 9 – optional part to test your reaction time in milliseconds



PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 9

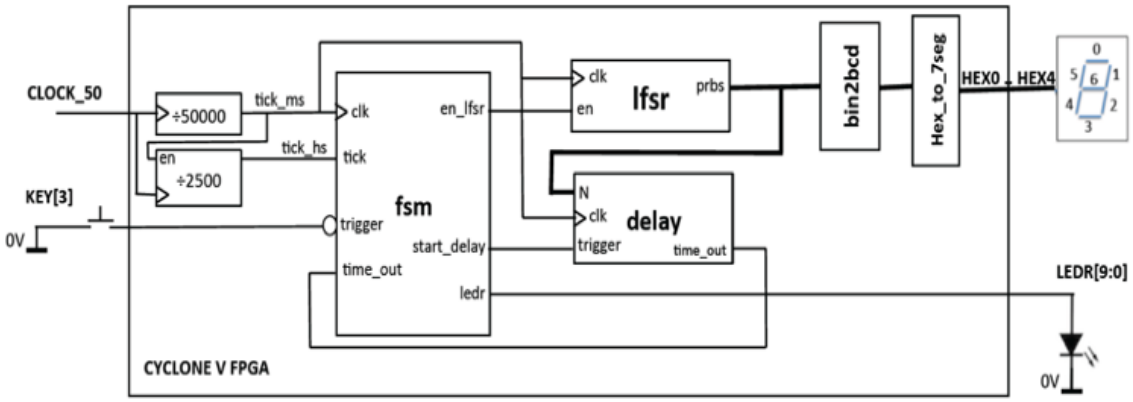
Part 2 of the Lab is starting to get harder. You will first learn to use Modelsim to simulate an 8-bit counter. On the way, you will also learn how to combine the interactive commands your type in the command window in Modelsim into a DO-file. You then will use this file as the testbench for your circuit.

Next you will extend the 8-bit counter to 16-bit with reset and enable input. You will also combine this with what you have already done in Part 1 to display the counter value as decimal number on the displays.

Next, you will find that the counter is counting too fast – you only see 88888 at the output. You then will add another circuit known as **prescaler**, which is another counter that produces a clock tick once every millisecond. This allows you to see what you see the count value changing.

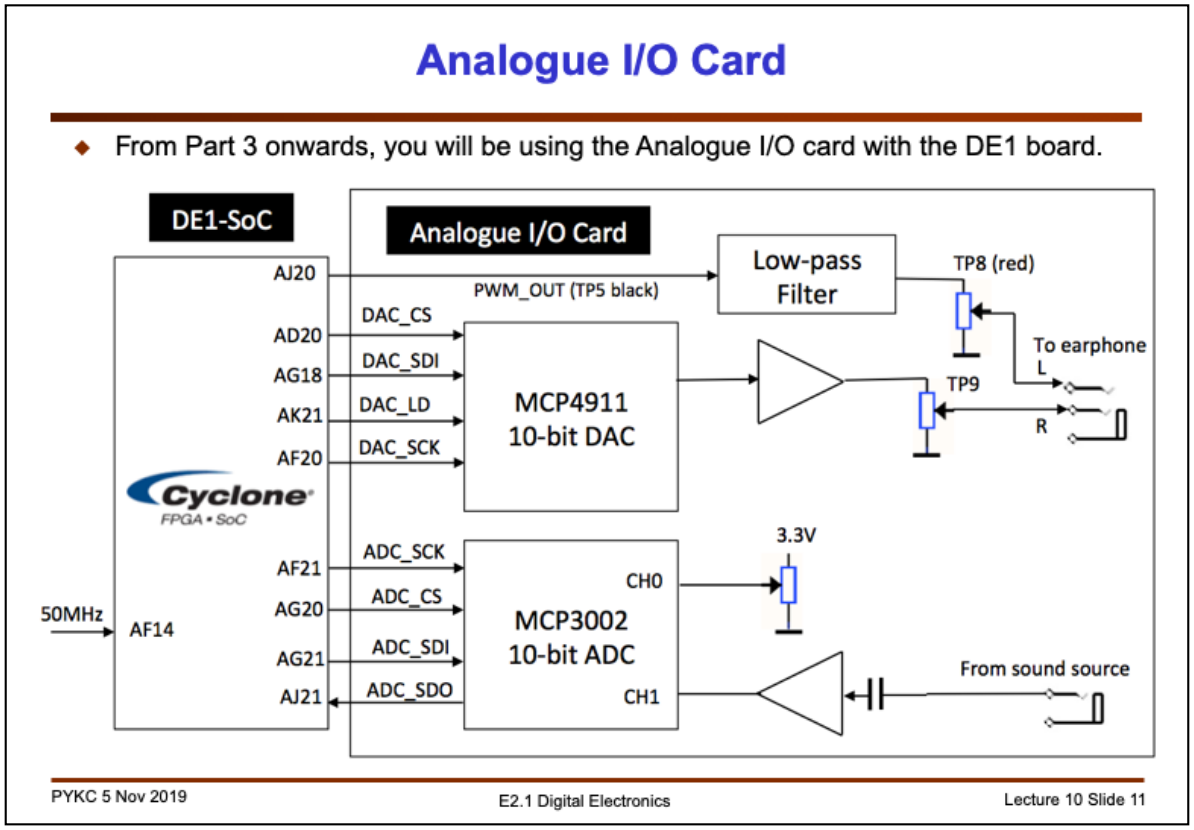
Then you will create a random number generator and test this on DE1.

Part 2 – Formula 1 starting light sequence



- This is challenging, but will put everything you have done together to create something quite fun to do.
- You can extend this further by adding a counter to count the number of millisecond after the light has gone out and you pressing KEY[0]. This reports your reaction time.

Optional challenge is quite hard, but should be very satisfying. Do this only if you have time.



Part 3 and Part 4 of VERI use the analogue I/O card with DE1. The I/O card contains a DAC and a ADC, both 10-bits. These produce an audio output on the right channel of a 3.5mm socket, and one channel of the analogue input from the other socket.

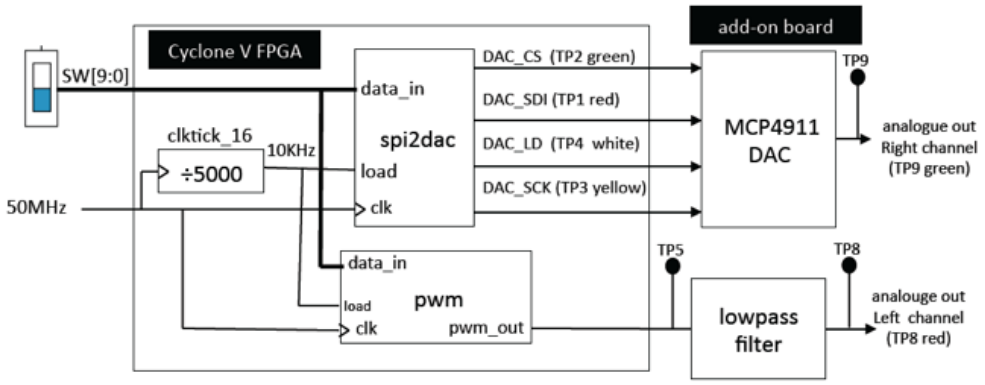
There is also a low-pass filter, which receives a PWM signal and produces an analogue output on the left channel of the audio socket. A 5k ohm potentiometer provides a dc voltage to the other channel of the ADC.

The I/O card is plugged into the 40-way socket on the DE1 board – the socket is the one that is furthest away from the board edge. Beware that you may not have aligned the pins correctly. This will no damage anything. If the I/O board is installed correctly, the GREEN LED will light up when the DE1 board is turned ON.

The communication between the DAC/DAC and the FPGA chip is through serial interface known as SPI (Serial Peripheral Interface). How exactly SPI works will be covered in another lecture later.

Part 3 – Serial Interface, DAC and signal generator

- Ex 10 – Testing the SPI interface to the DAC; examining timing waveforms.
- Ex 11 – Using pulse-width modulation (PWM) to produce a DAC.
- Ex 12 – Creating a ROM that contains one cycle of a sinewave.
- Ex 13 – Use the ROM, an address counter, the DAC and the PWM DAC to produce a fixed frequency sinewave signal on the right and left channels of the earphone socket.
- Ex 14 – Optional challenge: produce a variable frequency sinewave with frequency controlled by the slide switches and the frequency displayed on the 7-segment displays.
- **Ex 15 – Optional: Use the potentiometer to control the variable frequency generator.**



PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 12

Part 3 will introduce you to many different useful digital components. This includes: the SPI interface module spi2dac.v, the PWM module pwm.v, the ROM generator, the multiplier etc. In the end, you will be able to produce at least a fixed frequency sinewave on both channel of the output socket.

Displaying a binary number as decimal

- As the first part of the Lab Experiment VERI, you will be implementing the 7 segment decoder we designed in the last lecture. This will show every four binary bits as a hexadecimal digit on the display.
- Hex numbers are difficult to interpret. Often we would like to see the binary value displayed as decimal. For that we need to design a combinational circuit to converter from binary to binary-coded decimal. For example, the value 8'hff or 8'b11111111 is converted to 8'd255 in decimal.

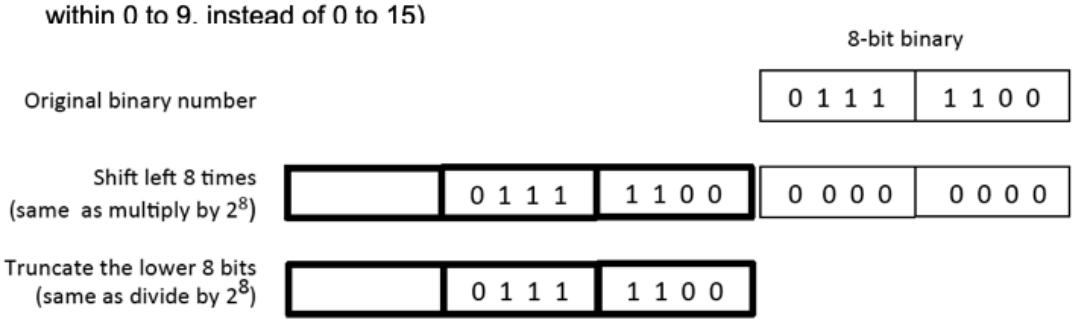
We now take another example of a relative complex combinational circuit, and see how we can specify our design in Verilog.

The goal is to design a circuit that converts an 8-bit binary number into three x 4-bit binary coded decimal values (i.e. 12 bit).

There is a well-known algorithm called "**shift-and-add-3**" algorithm to do this conversion. For example, if we take 8-bit hexadecimal number 8'hff (i.e. all 1's), it has two hex digits. Once converted to binary coded decimal (BCD) it becomes 255 (3 BCD digits).

Shift and Add 3 algorithm [1] – shifting operation

- Let us consider converting hexadecimal number 8'h7c (which is decimal 8'd124)
- Shift the 8-bit binary number left by 1 bit = multiply number by 2
- Shifting the number left 8 times = multiply number by 2^8
- Now truncate the number by dropping the bottom 8 bits = divide number by 2^8
- So far we have done nothing to the number – it has the same value
- The idea is that, as we shift the number left into the BCD digit "bins", we make the necessary conversion to the hex number so that it conforms to the BCD rule (i.e. falls within 0 to 9, instead of 0 to 15)



PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 14

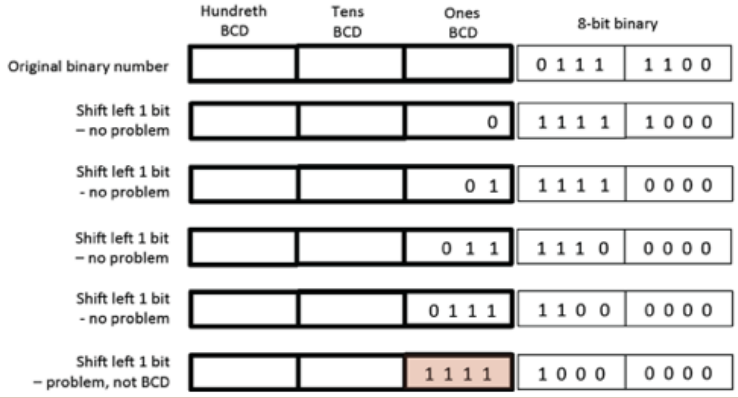
Before we examine this algorithm in detail, let us consider the arithmetic operation of shifting left by one bit. This is the same as a x 2 operation.

If we do it 8 times, then we have multiplied the original number by 256 or 2^8.

Now if you ignore the bottom 8-bit through a truncation process, you effectively divide the number by 256. In other words, we get back to the original number in binary (or in hexadecimal).

Shift and Add 3 algorithm [2] – shift left with problem

- If we take the original 8-bit binary number and shift this three times into the BCD digit positions. After 3 shifts we are still OK, because the **ones digit** has a value of 3 (which is OK as a BCD digit).
- If we shift again (4th time), the digit now has a value of 7. This is still OK. However, no matter what the next bit it, another shift will make this digit illegal (either as hexadecimal "e" or "f", both not BCD).
- In our case, this will be a "f"!



Our conversion algorithms works by shift the number left 8 times, but each time make an adjustment (or correction) if it is NOT a valid BCD digit.

Let us consider this example. We can shift the number four time left, and it will give a valid BCD digit of 7.

However, if we shift left again, then 7 becomes hex F, which is NOT valid. Therefore the algorithm demands that 3 is added to 7 (7 is larger or equal to 5) before we do the shift.

Shift and Add 3 algorithm [3] – shift and adjust

- So on the fourth shift, we detect that the value is > or = 5, then we adjust this number by adding 3 before the next shift.
- In that way, after the shift, we move a 1 into the tens BCD digit as shown here.

	Hundreth BCD	Tens BCD	Ones BCD	8-bit binary	
Original binary number				0 1 1 1	1 1 0 0
Shift left 1 bit – no problem			0	1 1 1 1	1 0 0 0
Shift left 1 bit - no problem			0 1	1 1 1 1	0 0 0 0
Shift left 1 bit – no problem			0 1 1	1 1 1 0	0 0 0 0
Shift left 1 bit - no problem			0 1 1 1	1 1 0 0	0 0 0 0
Perform adjustment Before shifting by adding 3			1 0 1 0	1 0 0 0	0 0 0 0
We perform adjustment (if >=5, add 3) before shift		1	0 1 0 1	1 1 0 0	0 0 0 0

PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 16

The rationale of this algorithm is the following. If the number is 5 or larger, after shift left, we will get 10 or larger, which cannot fit into a BCD digit. Therefore if the number 5 (or larger) we add 3 to it (after shifting is adding 6), which measure we carry forward a 1 to the next BCD digit.

Shift and Add 3 algorithm [4] – full conversion

- In summary, the basic idea is to shift the binary number left, one bit at a time, into locations reserved for the BCD results.
- Let us take the example of the binary number 8'h7C. This is being shifted into a 12-bit/3 digital BCD result of 12'd124 as shown below.

	Hundreth BCD	Tens BCD	Ones BCD	8-bit binary	
Original binary number				0 1 1 1	1 1 0 0
Shift left three times no adjust			0 1 1	1 1 1 0	0
Shift left Ones = 7, ≥5			0 1 1 1	1 1 0 0	
Add 3			1 0 1 0	1 1 0 0	
Shift left Ones = 5		1	0 1 0 1	1 0 0	
Add 3		1	1 0 0 0	1 0 0	
Shift left 2 times Tens = 6, ≥5		1 1 0	0 0 1 0	0	
Add 3		1 0 0 1	0 0 1 0	0	
Shift left BCD value is correct	1	0 0 1 0	0 1 0 0		

PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 17

To recap: the basic idea is to shift the binary number left, one bit at a time, into locations reserved for the BCD results. Let us take the example of the binary number 8'h7C. This is being shifted into a 12-bit/3 digital BCD result as shown above.

After 8 shift operations, the three BCD digits contain respectively: hundredth digit = 4'b0001, tens digit = 4'b0010 and ones digit = 4'b0100, thus representing the BCD value of 124.

The key idea behind the algorithm can be understood as follow (see the diagram in the slide):

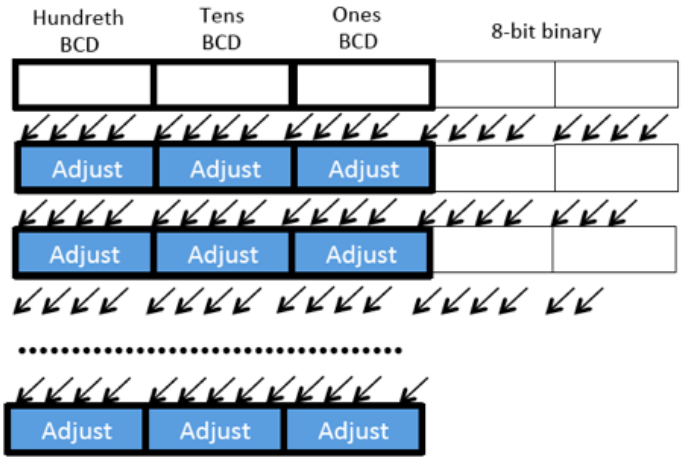
1.Each time the number is shifted left, it is multiplied by 2 as it is shifted to the BCD locations;

2.The values in the BCD digits are the same as as binary if its value is 9 or lower. However if it is 10 or above it is not correct because for BCD, this should carry over to the next digit. A correction must be made by adding 6 to this digit value.

3.The easiest way to do this is to detect if the value in the BCD digit locations are 5 or above BEFORE the shift (i.e. X2). If it is ≥5, then add 3 to the value (i.e. adjust by +6 after the shift).

Hardware implementation (1) – binary to BCD

- The hardware to perform binary to BCD conversion is shown below.
- Shifting is easy – just wiring all signals one position to the left.
- For each of the BCD locations, we need an "adjust" module which perform the follow operation: if the value is ≥5, then add 3.



PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 18

In order to understand how to we may implement this converter in hardware, you have to understand that shifting in hardware is easy. You just need to connect signals with one bit shift to the left. It DOES NOT need any gates, just wires!

Now we also need to do the adjust module, which simply performs the operation:

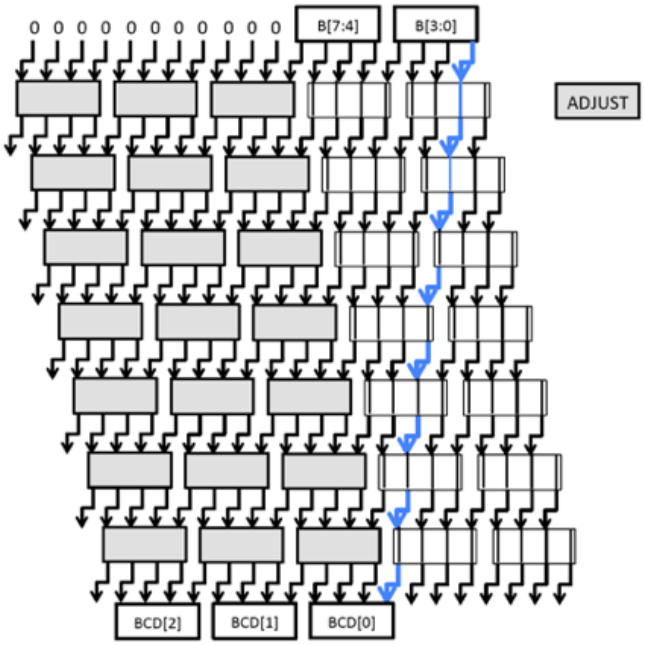
if (in >= 5) out = in + 3 else out = in

The easiest way to implement such a module is to use a **case statement**. This is set as a tutorial problem in Problem Sheet 1.

Hardware implementation (2) – array of gates

- Here is the full array of logic gates to do the conversion.
- After 8 shift and adjustment on the way, the result should be three BCD digits.
- Each ADJUST block perform the following operation:

```
if (input >= 5)
        output = input + 3
else
        output = input
```



PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 19

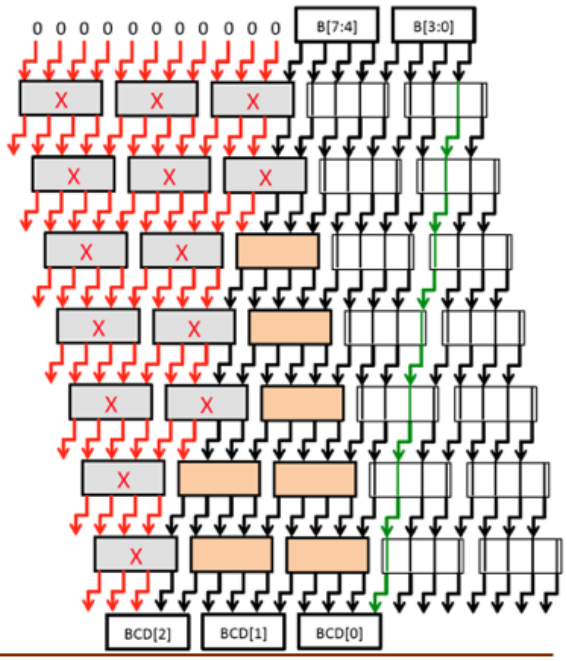
The entire full array is shown here. The shade module is the adjust module (which we call: **add3_ge5**).

As I said in the last slide, the easiest way to implement (specify) **add3_ge5** is using a case statement.

The BLUE signal path traces what happens to the least significant bit of the original number.

Hardware implementation (3) – propagate 0 to simplify

- If we now propagate forward all the 0s, we can eliminate all ADJUST modules except those in RED.
- All the others are just wires from input to output because the input values are GUARANTEED to be smaller than 5.



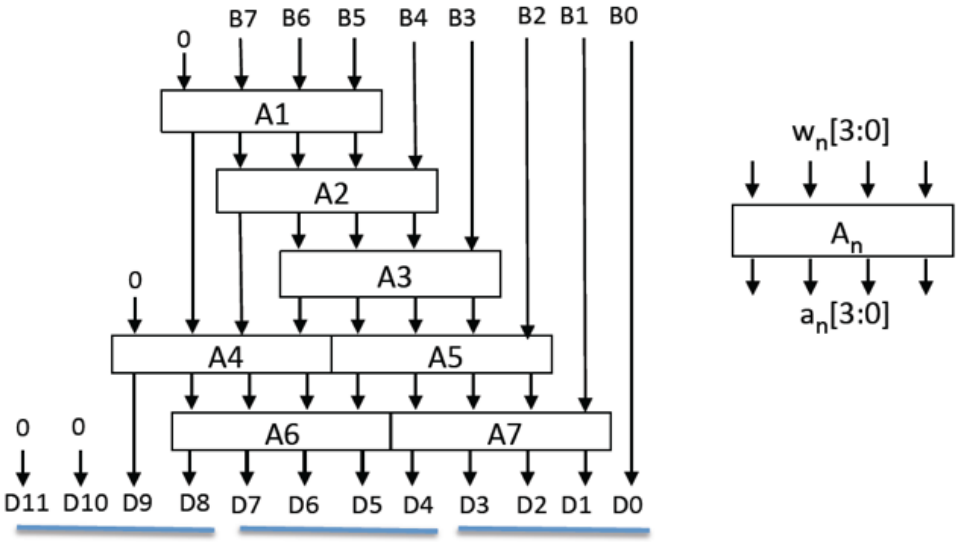
PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 20

The full array is more complicated than need be. If we propagate the '0's forward in the array of gates, you will find those marked with 'X' will always have its input less than 5. In which, **output = input** in these modules. THIS IS JUST A SET OF FOUR WIRES.

The only remaining **add3_ge5** modules are those shaped in orange.

Putting things together

- Once you have specified the **adjust module (A_n)** in Verilog, you can wire up the entire converter as shown here:



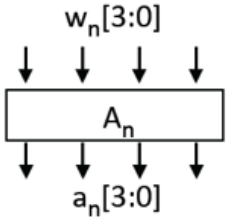
PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 21

After simplification, here are ALL the remaining **add3_ge5** modules for the 8-bit binary to BCD conversion (bin2bcd8). I have labeled the input ports to **add3_ge3 wn[3:0]** and the output parts **an[3:0]** where n is 1 to 7.

Binary to BCD conversion in Verilog

- Here is the Verilog code to perform the 8-bit binary to BCD conversion:

```
module bin2bcd8 (B, BCD_0, BCD_1, BCD_2);

   input [7:0] B;     // binary input number
   output [3:0]   BCD_0, BCD_1, BCD_2;    // BCD digit LSD to MSD

   wire [3:0]  w1,w2,w3,w4,w5,w6,w7;
   wire [3:0]  a1,a2,a3,a4,a5,a6,a7;

   // Instantiate a tree of add3-if-greater than or equal to 5 cells
   //   ... input is w_n, and output is a_n
   add3_ge5 A1 (w1,a1);
   add3_ge5 A2 (w2,a2);
   add3_ge5 A3 (w3,a3);
   add3_ge5 A4 (w4,a4);
   add3_ge5 A5 (w5,a5);
   add3_ge5 A6 (w6,a6);
   add3_ge5 A7 (w7,a7);

   // wire the tree of add3 modules together
   assign  w1 = {1'b0, B[7:5]};      // wn is the input port to module An
   assign  w2 = {a1[2:0], B[4]};
   assign  w3 = {a2[2:0], B[3]};
   assign  w4 = {1'b0, a1[3], a2[3], a3[3]};
   assign  w5 = {a3[2:0], B[2]};
   assign  w6 = {a4[2:0], a5[3]};
   assign  w7 = {a5[2:0], B[1]};

   // connect up to four BCD digit outputs
   assign BCD_0 = {a7[2:0],B[0]};
   assign BCD_1 = {a6[2:0],a7[3]};
   assign BCD_2 = {2'b0,a4[3],a6[3]};

endmodule
```



Assuming that we have designed a module "**add3_ge5**" to perform the adjustment as required, the converter can be implemented in Verilog by simply "WIRING UP" the various modules together.

The interconnections are specified in the wire statements.

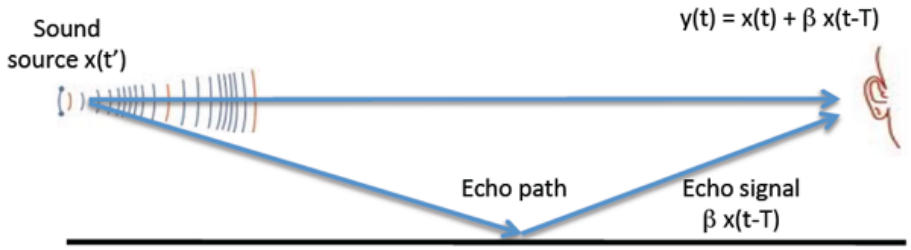
The next block is instantiating 7 **add3_ge5** modules.

The next block of code is to wire the modules together.

Finally the last statements are to connect up the signals from the modules to the output ports.

Part 4 – Echo Chamber Simulator

- Ex 16 – Audio In and Out without doing anything (allpass.v). This provides the overall framework to take an input sample and then output it.
- Ex 17 – Simple echo, just provide the simplest form of single echo. You will also learn about delay buffer of audio samples – a First-in-First-Out (FIFO) buffer.
- Ex 18 – A multiple echo with feedback path. This will the slide switches and the frequency displayed on the 7-segment displays.
- **Ex 19 – Optional challenge: A variable echo simulator, where you control the echo delay with the slide switches.**
- **Ex 20 – Really hard challenge: This is something for those who are really really keen to do over Christmas break. You will produce a voice corruptor – it changes the pitch of your voice without changing the speed of the signal.**



PYKC 5 Nov 2019 E2.1 Digital Electronics Lecture 10 Slide 23

The final part (Part 4) really brings everything together. The goal is to use the FPGA to implement a real-time speech processing system that perform echo simulation.

To start with, you will implement a DO-NOTHING block. This just tests out the system and takes an analogue input sample, then output it to the earphone. Nevertheless there are details that need to be taken care of. I will go into more details nearer the time in order to explain exactly what's happening.

Finally, the part includes something that CANNOT be done in four lab session, but if you are really keen in going further it is a real challenge.

Bringing everything you have learned in VERI, you can design a voice corruptor – some that that makes changes ones speech in a way that is unrecognizable, but it is still comply understandable.