University of Cape Town

EEE4120F

High Performance Embedded Systems

Practicals and Projects

17 April 2021

Introduction

Welcome to the practicals for EEE4120F. These instruction are applicable to all practicals so please take note.

Practicals take place on a Thursday, from 3PM to 6PM. You are responsible for attending these practicals and making use of the resources available. Many of these practicals will require access to specialist hardware that will only be available during the practical session.

It is assumed you have knowledge regarding tools such as git, LaTeX and running programs from the Linux command line.

It is suggested you have a dual boot system. The reason for this is that development tools are generally better on a Linux based system, whereas proprietary tools are often better supported on Windows. We recommend Ubuntu 18.04 LTS, though swapping between Ubuntu (or any Linux distro) and Windows is an effective strategy, particularly when using proprietary software.

The source code for all pracs is available on the EE-OCW GitHub Project Page: https://github.com/UCT-EE-OCW/EEE4120F-Pracs. You can download the source code for pracs from there.

Prac Overview

Please note that the due dates are provided in the Vula site for this course if you are taking this course for credit.

Table I: The Prac Overview for EEE4120F, 2021

Prac	**Dates**	**Tools**	**Objective**
Prac 1	see Vula	Octave	Performance measurement
Prac 2	see Vula	OpenCL	SIMD on GPUs
Prac 3	see Vula	MPI	Creating image filters on shared mem.
Prac 4	see Vula	Vivado	Xilinx Vivado development practices with simulation.
Prac 4a	Archived*	Vivado	Making a wall time clock (see Archived Prac4)
Prac 5a	Archived*	Vivado	Making a signal generator (see Archived Prac5)

* the pracs marked as Archived have been left for potential useful practice or training that may be of use to you in the class project.

1 Practical 1 - Testing methodologies

1.1 Introduction

The focus of this task is on using OCTAVE (the free sort-of MATLAB program) and doing some statistical operations. In later assignments you will make further use of these statistical functions, and perhaps reuse this code, in comparing and discussing results obtained in other practicals and projects (for example, correlation can be used in analysing gold standard results to higher-speed approximation result).

For installation and tips and tricks on using Octave, visit The EE Wiki (currently only accessible on the UCT Network - though you can use a VPN).

Note:
If you're doing a smaller installation of Octave, there are the libraries you require from Octave Forge: audio ; control ; data-smoothing ; fixed ; ga ; gnuplot ; image ; integration ; oct2mat; plot ; signal ; sockets ; specfun ; splines ; statistics ; strings

1.1.1 Correlation

Correlation is a useful statistical function for comparing two datasets to judge how similar or different they are. The correlation function returns a correlation coefficient, r, between -1 and 1. A value of 1 for r implies perfect positive correlation, i.e. the two datasets are the same. Correlation of 0 implies there is no correlation (the two datasets behave totally differently). A correlation of -1 indicates a total opposite - for example if you compare vectors x to - x you get a correlation of -1. Generally if $|r| >= 0.8$ there is strong correlation, between 0.5 and 0.8 moderate weak, less that 0.5 is weak (towards no) correlation.

Pearson's correlation (which you can read more about on Wikipedia) is implemented as follows:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{1}$$

1.1.2 Speed Up

$$Speedup = \frac{T_{p1}}{T_{p2}} \tag{2}$$

Where:
T_{p1} = Run-time of original / non-optimal program
T_{p2} = Run-time of optimised program

For obtaining a repeatable timing value, run each version (i.e. initial version and optimised version) of your programs more than once and discard the first measured time. You can, if

you want to be complete, indicate what the initial speed up was and then the average speed up.

1.1.3 Critical Section

When measuring performance, it's important to ensure that you are only measuring the section of your algorithm that you need to measure. For example, if you are measuring the execution time of an audio filter, you should not be recording the time taken to open and close the files you are applying this filter to.

1.2 Requirements

You are required to run the following experiments:

1.2.1 Measuring Execution Time of rand()

White noise is often generated with GNU Octave's random number generator `rand()`, which generates uniformly distributed random values in the interval [0,1). To create a sound wave of the white noise, the `wavwrite()` function in Octave is used. The `wavwrite()` function expects values in [-1.0, 1.0), so the `rand()` output must be multiplied by 2 and shifted down by 1. To generate 10 seconds white noise sampled at 48 kHz, the following instruction are called:

```
white = rand(48000*10,1)*2-1;
```

And to generate a wave file for this noise, we use the `wavwrite()` function as follows:

```
wavwrite(white, 48000, 16, 'white_noise_sound.wav');
```

NOTE:
`wavwrite` and `wavread` were deprecated in Octave 5.1. If you're using Octave 5.1 or greater, you need to use `audiowrite` and `audioread`.

```
audiowrite('w.wav', white, 48000, 'BitsPerSample', 16);
[y, fs] = audioread('w.wav', 16);
```

1.2.2 White Noise Generator Script

Next, write a function in a new file called `createwhiten.m` that implements a function with a **for loop** that generates a white noise signal, one sample at a time, comprising N duration in seconds. You can assume that N will always be positive and a multiple of 10. The white noise must be sampled at either 48 kHz or 8 kHz. Name your function `createwhiten(...)`. You

need to use the `rand()` function without arguments so that it will generate a single random value, and the main task is figuring out how to scale so that you create a suitable input to `wavwrite(...)` as explained above. Call the function and check output size as follows:

```
whiten = createwhiten(1000);
size(whiten);
```

should return:

```
ans = 48000000 1
```

Check that the resulting wave/sound file gives the same sound as the white noise sound.wav generated above by generating the new sound file named white noise sound2.wav and playing it back.

```
wavwrite(whiten, 48000, 16, 'white_noise_sound2.wav');
```

1.2.3 Visual Confirmation of Uniform Distribution

Confirm that you've created the sample correctly. Since it's a big signal, let's just look at the first 100 samples by plotting using a histogram function as follows:

```
hist(whiten, 100, 1);
```

This should give image shown in Figure 1:

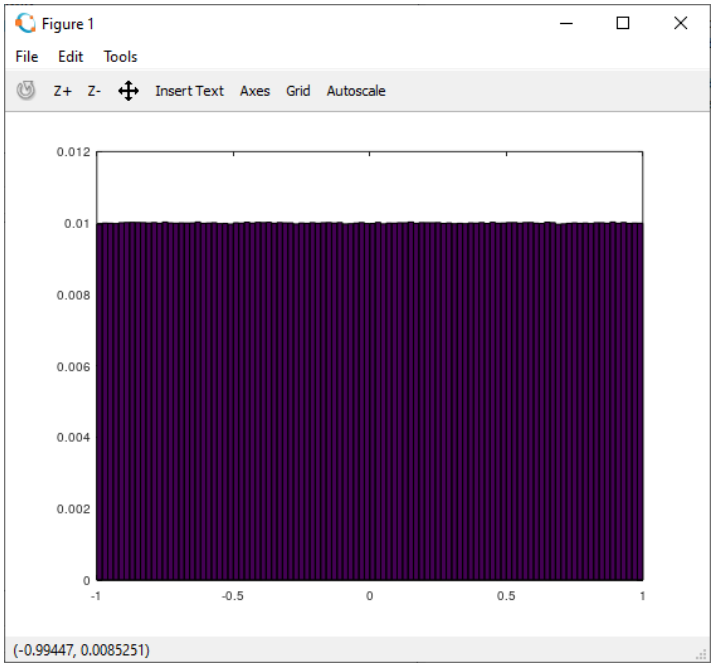

Figure 1: Output of hist(whiten, 100, 1);

1.2.4 Timing Execution

Now time how long it took to execute the script. Use the `tic()` and `toc()` functions as follows. Note that the call to tic is on the same line just before the function - this tends to have less delay between the start of the timer and starting the function. Of course, it is good practice to put it all in a script file.

```
tic; white = rand(48000*1000, 1)*2 - 1; runtime = toc();
disp(strcat("It took: ", num2str(runtime*1000), " ms to run"));
```

Call the `createwhiten(...)` function you created that does the same thing as the `white = rand(48000*1000,1)*2 - 1;` statement that generate the white noise. Measure the time it took for your `createwhiten(...)` function to run and show the timing difference (in milliseconds) and discuss the speed-up you have achieved (if any).

1.2.5 Implementing Pearson's Correlation

Implement the Pearson's correlation formula a new m file called it `mycorr.m`. Provide your code in your report. Marks will be awarded on elegance of your code.

1.2.6 Comparing Your Correlation Function to Octave

The `cor` function in OCTAVE performs a correlation operation. Compare your `mycorr` results to the build-in cor function. First read the noise wave file using `wavread(...)`, then you could, for example, use the following code to test your `mycorr(...)` function as compared to Octave's `cor(...)`:

```
x = wavread('white_noise_sound.wav');
y = x;
r1 = mycorr(x,y)
r2 = cor(x,y) # note that in some versions, this is called "corr"
disp(r2 - r1);
y(1) = 2; y(5) = -4; # i.e. fudge some of the value
r1 = mycorr(x,y)
r2 = cor(x,y)
disp(r2 - r1);
x = rand(1,10); y = rand(1,10);
r1 = mycorr(x,y)
r2 = cor(x,y)
disp(r2 - r1);
```

Generate sample sizes of varying sizes: for example 100, 1000 and 10000 samples. Do a table listing sample sizes vs. mycorr speed vs. corr speed. Indicate the average speed-up of corr to mycorr. Include a table of sample size vs time, etc. in your report.

1.2.7 Correlation of Shifted Signals

For the last experiment, we want to compare signals shifted in time. Generate sin curves of varying frequency and sampling sizes (again sample sizes 100, 1000 and 10000 samples). Compare samples of the same sizes that are shifted in time, e.g. if A uses x[1:100] then B might use x[11:110], in which case B is shifted in time by 10 samples.

For this step only use cor to save time. In your report, discuss what you expect the correlation of the identical but shifted signals would be. Run tests to confirm / verify your hypothesis. Provide screen shot plots of some signals you compared.

1.3 Submission

Hand in a report about 2-3 pages in length briefly describing your solutions for the tasks above. The format required is the IEEE conference format. Format your report as if it is an article (i.e. don't follow the same chronology as the prac-sheet – format it as "Introduction – Method – Results and Discussion – Conclusion"). In the "Method" section, theorise about what you expect and how you plan on testing said theory. In the "Results" section, confirm that you obtained what you expected (or explain why you obtained something unexpected). Hint: You are trying to answer two questions: 1) "Is my mycorr function better suited to run correlation tests, in comparison to the built-in equivalent?" and 2) "Are my theories relating to the correlation of time-shifted sinusoids correct?"

1.4 Marking

Table II: Prac 1 Marking Guide

Aspect	**Description**	**Mark Allocation**
Report	Intro	2
	Latex/Format	1
	Headings etc	1
	Captions	1
createwhiten	Code intro concepts	4
	Neatness	3
	Structure	3
Sz. v t	Table	2
	Graph	1
	Explanation	2
Shifted signals	Screenshots	2
	Code	2
	Correlation	2
	Explanation	3
	Expected vs Results	1
TOTAL		30

2 Prac 2 - OpenCL

2.1 Introduction

> OpenCL™ (Open Computing Language) is the open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers, mobile devices and embedded platforms. OpenCL greatly improves the speed and responsiveness of a wide spectrum of applications in numerous market categories including gaming and entertainment titles, scientific and medical software, professional creative tools, vision processing, and neural network training and inferencing. [1]

The focus of this practical is investigating the performance metrics of an OpenCL matrix multiplication implementation.

2.2 Requirements

The Blue Lab PCs have all the required packages installed, but if you wish to run this practical on your own machine, you will need to install OpenCL as per the instructions on the Wiki - http://wiki.ee.uct.ac.za/OpenCL.

2.3 The Programming Model

OpenCL uses a programming and memory model similar to OpenGL. The CPU must copy data to the GPU and then tell a kernel (OpenCL worker) to process the data. When the kernel is finished, the CPU can read back the result. Download and inspect the practical source code (available from Vula resources).

Familiarise yourself with the program and what the OpenCL wrapper is actually doing.

If you modify the code, be very careful with how you allocate and free memory. You need to allocate CPU and GPU memory separately, and then free them separately as well.

The local group size has to be an integer fraction of the global group size, in every dimension. Every time you change N, you have to recalculate the local group size.

2.4 Tasks

2.4.1 Speed-up

The default program multiplies two 3x3 matrices. This process takes much longer on the GPU than the CPU, mainly due to overhead. At what matrix size does it become worth

while to use the GPU instead of the CPU? Support your argument with measured results (presented in graphs andor tables). You may also, for instance, make assumptions regarding speed-up expected were the CPU version multi-threaded.

2.4.2 Data Transfer Overhead

The asynchronous nature of the OpenCL interface makes it difficult to obtain accurate timing information of the various steps of the process. Try to come up with a way to measure the data transfer overhead and processing time separately (hint: make use of the clFinish function[1] in the OpenCL WriteData and OpenCL ReadData functions).

Use this new information to comment on the sources of OpenCL processing delay. Also comment on the speed-up factor achieved when transfer overhead is not taken into account. Does this relate well to the number of threads that are running on the GPU? If not, provide an argument for why this is the case. Do this for large N (pick a value that takes long enough to dominate the transfer overhead, but does not let you finish a whole coffee between runs).

2.5 Submission

Compile your experiments and findings into an IEEE-style conference paper. Make sure you include ALL hardware details, including GPU and CPU clock rates, if available. Of particular importance is the local work group size and number of compute units.

The page limit is 3 pages. Submit your paper to the Vula Assignment for this practical.

2.6 Marks

Note that 33 marks are available, but you will still cannot score a mark above 100%.

[1] Which can be found in the Process_OpenCL() function in the OpenCL_Wrapper.c

Table III: Prac 2 Marking Guide

Aspect	**Description**	**Mark Allocation**
P1	Graphs/Tables *	6
	Comparison to CPU Speed up	3
P2	3 Metrics reported on	4
	Use of clFinish in Code	2
	Overhead discussion	3
	Discussion	4
General	Introduction	3
	Layout/Captions etc	3
	PC details	2
Bonus experiments		3
TOTAL		30

3 Prac 3 - MPI

3.1 Overview

The focus of this practical is on using message-passing in a distributed memory programming model. The processing algorithm that will be used is a median filter. You can find a tutorial about using of MPI at https://computing.llnl.gov/tutorials/mpi/, but this is much more detailed that what you need for this practical, although you may find some useful tips in that tutorial if you encounter problems in this prac.

3.2 The Programming Model

One thing to keep in mind, which is a characteristic of an MPI application, is that each process runs in a completely separate memory-space, possibly on a completed separate platform to where other processes are running. There are no shared memory structures, so any global memory you define is local to each instance. The only means of sharing data is through sending and receiving messages (although, of course, there are other methods that can still be used like changes to files in a folder that is shared on the networked). Typically MPI is only useful on distributed systems (systems where each node has its own memory, and the nodes are connected through some form of network – typically Ethernet). Setting up such a system is not trivial, and therefore not included in this practical. For this practical, all the processes will be running on the same CPU, with memory being separated by means of memory virtualisation (through use of the memory management unit).

3.2.1 MPI Commands

MPI programs are compiled with **mpic**, not gcc, and run with **mpirun**, not by calling the program binary directly. The makefile associated with this practical runs the program with 5 instances (one master and 4 slaves). Feel free to change this if you so wish.

Study the source code provided. It has been adapted from this example and uses simple blocking communication. You can implement this practical by means of blocking communication, so you don't need any more commands than those in the example code.

If you feel like being fancy, have a look at the MPI Iprobe command.

You can also have a look at this link to see how to tag your messages. Tags are typically used to determine the message type before actually receiving it, but it could be used for many other things as well.

There are many MPI datatypes, but for the purpose of this practical you can get away with using bytes (MPI BYTE) exclusively.

3.2.2 Using Doxygen

This practice include examples showing the use of Doxygen comments. While the purpose of this prac is not to learn Doxygen, I put that in as an example showing how this very useful code documentation framework can be used. Hence, the bonus learning for this prac is getting some insight into what coding that uses Doxygen looks like.

You are of course most welcome to try out use of Doxygen, and addition of Doxygen formatted comments while you are working on this prac. Note that you do not need to have Doxygen installed to compile and run this prac - that is just optional extras. If you do want to install Doxygen and see what the produced documentation looks like, than on Linux you can simply do an "apt-get" installation of Doxygen (if you haven't got it already installed) and to generate the documentation for this prac run the command make doxy. Note that by default, when you just run make on its own, it will not update the Doxygen documentation folder. The generated documentation will be in the folder Prac3
doxy
html (it should also update a Latex version of the documentation that is in Prac3
doxy
latex, but to generate the pdf from these Latex files you need to manually go into this sub-directory and run make).

You can find info about Doxygen and the comment types and formatting procedures at Doxygen Manual command. You might want to consider using Doxygen in the course project.

3.3 Problems

The file Prac3.cpp provides boilerplate code for an MPI application that incorporates the following operations:

1. Libraries you need to include.
2. Use of jpeglib to load a JPEG file that will be split up for processing.
3. Putting together a Master process (i.e., which will be called by the MPI process with rank 0) and a Slave process (called by MPI processes with rank 1 and above).
4. Reading and writing of a JPEG file (done by the Master).
5. Sending messages with data between Master and Slaves, and from Slaves to Master.
6. Showing that you can use printf from any node that sends the displayed text back to the Master (a useful hint for getting performance information!).

It is up to you though to decide how to partition up and distribute the data between nodes for processing (you want to mention your decision and reasoning for these aspects in your prac report).

3.3.1 Problem Partitioning

Choose a memory partitioning scheme. In your report, explain your partitioning scheme and why you chose it. Do not send the entire dataset to each process, as this is a waste of data. [**5 marks for describing the partitioning**]

You also need to have some means to indicate to the slave what size the data is. You can implement this any way you wish, but it is suggested that you send one message, with a known size and fixed format, that indicates data size and any other parameters you wish to send to the slave. The next message can then be the actual data to work on. [**5 marks for getting data to slaves**]

When the slave is finished, it must send the result back to the master. The master must then assemble the results and save the resulting filtered image to disk. [**5 marks sending back the data and reassembling**]

Provide snippets of your code in the report (it is important for your code snippets to be suitable commented, the use of comments in your main code repository is not as important for marking but is good practice). [**5 marks for code**]

3.3.2 Experimentation

Feel free to experiment with different number of processes (you can do this easily using just one machine with a multi-core CPU and telling mpirun to use more processes using the -np flag), although this is not required in 2021 (obviously if you are not using a cluster and limited to e.g. a single machine having 4 cores you would not be able to get performance improvements using beyond 4 processors because it would have to multitask the processes). It is sufficient to simply get your program working. [**5 marks for experiments**]

Your report should comment on the time performance though, so make sure you take time measurements. [**5 marks graphs and display of results**]

3.4 Submission

Please make sure that you have put the names and students numbers of your prac team into the Prac3.cpp file. This helps facilitate the marking process.

Compile your experiments and findings into an IEEE-style conference paper. The page limit is 3 pages. Submit your paper to the Vula Assignment for this practical.

3.5 Marking

Note that 33 marks are available, but you cannot score more than a total of 100%.

Table IV: Prac 3 Marking Guide

Aspect	**Description**	**Mark Allocation**
Partitioning	Listing (show data split)	2
	Explanation	3
To Slaves	Listing & ACK	2
	Explanation	3
On/From Slaves	Listing	3
Reassembling	Explanation	2
Code	Comments	2
	Use of MPI principles	3
Experiments	Experiment	3
	Good practice	2
Results	Graph/Table	2
	Discussion	3
Bonus		3
Total		**30**

4 Prac 4 - FPGA Introduction

4.1 Introduction

For this practical, it is important to complete the tutorial in order to get acquainted with the tools required to complete the practical. It is recommended that you work in groups of 3, and that all three members work through the tutorial individually.

It is also imperative that you read through the wiki. Operation of Vivado can be intimidating at first use. The Wiki has been carefully written to include all you need for the practicals.

4.2 Tutorial

The tutorial is not for marks but it is suggested you complete it, as it will give you the basics of Vivado, and allow you to upload a simple program to the FPGA. In the tutorial (as on the wiki) we use the Digilent Nexys A7 as an example board, but the process should be the same for the Nexys 4 DDR and Nexys 4. The only thing that will change between the three is the board you select when creating the project, and the constraints file you use. Note the constraints file does determine naming of some of the I/O, so double check that.

Most of the details on how to complete these steps are on the wiki under Xilinx Vivado.

1. Install Vivado
2. Install boards
   The only boards you might work with in this course are the Nexys 4, Nexys 4 DDR, and the Nexys A7. So you only need to worry about installing those.
3. Download and save the constraint files. That, or copy the simple example given on the wiki if you're using the A7.
4. Create a new project.
   - You can call it "Tutorial".
   - Set it as an RTL project and select "Do not add sources at this time". For the prac later, you can add the given source files here.
   - Select your board appropriately.
5. Add constraint source
   - Right click on constraints, select "Add sources". In the dialog box, make sure "Add or create constraints" is selected. Hit next.
   - Select "Create file" if you are copy pasting the constraints in, or "add files" if you have the constraints file downloaded locally. Press "Finish".

- If you were intending on copy-pasting constraints in, do so now. Ensure your constraints are correct for the board, and have the clock, two switches and two LEDs enabled.
- Right click on the constraints file, and select "Set as Target Constraint File"

6. Add the Verilog source
   - Right click on "Design Sources, select "Add sources". In the dialog box, make sure "add or create design sources" is selected. Hit next.
   - Select "Create file". Call it the same name as your project (good practice for the top level module). Press "Finish".
   - A dialog will open, with ports. You can just press "okay", as we'll define ports in the Verilog code.
   - Right click on the Verilog file, and select "Set as Top" (if it is not already set as top - indicated by being in bold).
   - In it, paste code to, each clock cycle, write switch[0] to LED[0] and the inverse of switch[1] to LED[1]. Example code can be found on the wiki.
7. Select "Run Synthesis"
8. Select "Run Implementation"
9. Select "Generate Bitstream"
10. Upload your bitstream to your target board (speak to a tutor to get a board)
    (a) Plug in the board
    (b) Select "Open Hardware Manager"
    (c) Select "Open Target" and then "Auto Connect"
    (d) Vivado should find the board. Select "Program Device" and select the board you've plugged in (The A7 shows as "xc7a100t_0". Press the "Program" button.
11. Hooray! You've created your first FPGA circuit. Toggle the switches to see if it operates as expected. Now let's move on to the fun stuff.

4.3 Practical

4.3.1 Introduction

In this practical, you will create a digital clock on an FPGA. There is no report for the practical, but you will need to submit screenshots of your testbenches and demonstrate your implementation to a tutor.

Source files are available on the EEE4120F OCW GitHub: https://github.com/UCT-EE-OCW/EEE4120F-Pracs.

4.3.2 Given Modules

1. TLM
   The top level module, called "Clock.v" in the source files on GitHub, contains the primary logic for your wall clock and allows you to implement I/O and other modules

2. Delay_Reset
   It's also useful as many components require a set up time. So by using a delayed reset signal, we can cater for reset times of peripherals.

3. Seven-Segment Driver
   This module takes 4 BCD values and displays them on the seven segment display.

4. Decoder
   Used by the Seven-Segment Driver to decode decimal to the appropriate cathode pins.

5. Debounce
   A debounce module you'll need to implement in order to debounce button presses.

6. PWM
   A module you'll need to implement in order to give the seven segment displays changing brightness. This can be tricky, it's suggested you leave it for last.

4.3.3 Requirements

The following outcomes are required to pass the demonstration:

1. Implement a simple state-machine to display the real time (hours and minutes) on the 7-segments display. You can start your clock at 00:00 upon reset. Make your clock faster in order to test that the time overflows correctly. Use a 24-hour time format.

   The easiest way to do this is with deeply nested if statements. Run a counter that overflows every second and increment the seconds counter on every overflow. Every time this seconds counter equals 59 (i.e. it will overflow on this clock-cycle), increment a minutes units counter. Every time this minutes units counter is about to overflow, increment a minutes tens counter, etc.

   Only use non-blocking assignments (<=). Blocking assignments (=) inside clocked structures are much more difficult to debug. Remember that the entire always block is evaluated at once: the statements are not evaluated sequentially.

2. Display the seconds on the LEDs, in binary format. This is done with a simple assignment outside the always block.

3. Use one of the buttons, properly debounced, to set the minutes. It must increment time by one minute every time it is pressed. Make sure that your time overflows correctly. You do not need to increment the hours when changing the minutes.

   It is recommended to write a Debounce module for this. On every clock cycle of the system clock (the fast one), check the state of the button. If it is not the same as the

current module output, change the output and start a dead time counter. While this counter is counting, do not change the output of the module, no matter what the input is doing. Use a deadtime of between 20 ms and 40 ms. To prevent unstable states, register the button before use.

In the clock state machine, you can use a register to store the button's 'previous' state. If the current state is high, and the previous state is low, the button signal went through a rising-edge. Do not use always @(posedge Button) – keep everything in the same clock domain.

4. Use another one of the buttons, properly debounced, to set the hours. It must increment time by one hour every time it is pressed. Make sure that your time overflows correctly.

5. Use the slide-switches to represent a binary "brightness" word. Make use of pulse-width modulation (PWM) to dim the brightness of the LED display.

   Ensure that the phasing between the driver signals and the PWM signals is correct. The easiest way to do this is to select a PWM frequency such that the PWM signal goes through exactly one period between driver signal state changes. You can implement this within the SS Driver module.

4.4 Mark Allocations

Table V: Prac 4 mark allocation

		Marks
Testbench		12
	Minute seconds reaching 60 and increasing minutes	
	Minutes reaching 59 and increasing hours	
	Time reaching 23:59 and wrapping back to 00:00	
	(3 each +3 for neatness)	
Demo		
	Time flows correctly (minutes overflow to an increase in hours, and 23:59 flows to 00:00) [6, subtracting 2 for each missed objective]	6
	Time can be scaled through a variable (i.e. the count to increase seconds isn't fixed)	2
	Minute button increases minutes and doesn't increase hours	2
	Hour button increases hours	1
	Debounce module implemented	2
	PWM module	5
	TOTAL	30

For the 5 marks on PWM: 1 mark for attempted, 2 marks for reading switches and adjusting, 3 marks for "flashing" implementation, 4 marks for some mix between flashing and decent PWM, 5 marks for correctly implemented.

5 Prac 5 - Vivado IP and Resource Usage

FPGAs are often used in DSP applications due to their highly parallelizable nature and ability to process data at high speeds. Usually FPGAs are used to sample signals and process them, but in this application we're going to generate waveforms and output them to a speaker!

If you aren't all that well acquainted with the physics of music, this video by Vihart[2] is a really great introduction! She speaks quite quickly, so you may need to watch it twice.

These videos don't have much to do with this prac, but are fun to learn about. If you'd like to learn more about modular synthesis (which is super cool!) check out this video by Andrew Huang![3] LookMumNoComputer also does a great bunch of custom modular synthesis projects - just check out his Furby Organ![4] He also has guides on how to build your own modular synth.[5]

We'll be implementing a sine wave - which you don't really hear much in music. But as Composerly shows us[6], you can create some greate tunes with nothing but sine waves and enough processing.

5.1 Tutorial

The tutorial starts with guidance on getting familiar with the Xilinx Vivado IP facilities, which leads in to the requirements for the prac.

5.1.1 Resource usage

Please read the Implement section in the Xilinx Vivado page on the UCT EE Wiki.

5.1.2 Vivado IP

For general information on the Vivado IP, please read the "Xilinx Vivado IP" section of the Xilinx Vivado wiki page.

Of course, when using an IP core, you need to have some knowledge of the technology. You should know how to use the IP cores - and, since you are university students and not necessarily going to just be users - you should also have some sense of what is happening behind the scenes. [7] In the tutorial, you will be guided through the relevant settings and

[2] https://youtu.be/i_0DXxNeaQ0
[3] https://youtu.be/cWslSTTkiFU
[4] https://youtu.be/GYLBjScgb7o
[5] https://youtu.be/4Kz8YopLTCQ
[6] https://www.youtube.com/watch?v=xLtTMkMr2Wg
[7] You'll get a better sense of the theory behind these concepts from the lectures building on the basics of the PLDs and FPGAs covered in ES1 and ES2.

what they mean, but when using other IP, you should do your research as to what you're adjusting.

Once you've created your project for your board, you can do the following to instantiate memory to hold the lookup table. You can repeat this process for different tables or waves (for example sawtooth, triangle, or perhaps another periodic waveform - maybe the waveform for a specific instrument).

1. Select "IP Catalogue" on the left hand side of the IDE
2. Search "BRAM".
3. Under "RAMs & ROMs & BRAM", select the "Block Memory Generator"
4. Select "Port A Settings"
5. Note the write and read width. These relate to the bitwidth of the data in the memory. Change this to 11 for both read and write. We use 11 because the audio module expects a signal 11 bits big.
6. The write and read depth relate to how many samples can be stored. Change this to 256, as out example sine table has 256 samples, and this we need 256 addresses.
7. Select "Other Options"
8. Select "Load Init File"
9. Browse to the Prac 5 sources, and load LUT_sinefull.coe
10. Click "Ok" at the bottom of the dialog
11. Click "Generate" on the next dialog. Press "Ok" when the information dialog shows up.
12. In sources view, there will be a template available to instantiate the IP.
13. Congrats! You've added a 11-bit 256 sample Block RAM IP!

It would be a good idea to set up a test bench to ensure you're reading the correct values from the BRAM as you expect. Here's some hints which might make setting up the test bench easier:

- You don't need to do any writing to the BRAM (write enable can be left low)
- You can leave the read enable signal high
- You can just increase the address by 1 each time - it will auto wrap around to 0

5.2 Practical

We're going to start by making a simple waveform generator to output middle C at f=261.625565Hz (or as close as you can get to it!). Then we're going to try free up some resources in our first implementation. From there, we're going to create an arpeggiator. If you're familiar with synthwave - it's usually an arpeggiator (or *arp*) that creates the repetitive notes in the background.

5.2.1 Provided files

You are provided with:

1. top.v
   The top level module for the project. A template to get you started.
2. pwm.v
   A PWM module to convert the BRAM samples to a PWM signal for the audio jack.
3. LUT_sinefull.coe
   A full sinewave table, 11 bits wide with 256 samples

5.2.2 Creating a waveform generator

In this section you need to create a simple output waveform generator at as close to 261.625565Hz as you can. The hardware operates at 100MHz, so 100Mhz/(261.62556Hz*256) gives us 382225.643736. But! We have 256 samples in our look up table. So we need to operate 256 times faster to complete one wave in the expected time. 382225.643736 divided by 256 is about 1493. So count to that value to produce a tone around middle C.

1. Start by creating a new project, and adding the full sine wave table to BRAM as per the tutorial above. A reminder to use the correct board settings when creating a new project and calling it fullsine.
2. Create a new test bench, showing loading a sample from the BRAM
3. Record resource usage for the full sine table (we're looking for total power, LUT, FF, BRAM)
4. Output this wave to the audio port, and record video of it playing. To do so, you need to tie in the AUD_SD and AUD_PWM signals from your constraints file. AUD_SD can be written high (as an enable). AUD_PWM needs to be a PWM signal. To generate this, pass the data from the BRAM to an instantiation of pwm.v, and pass the output of that PWM module to AUD_PWM.

Create a new project called quartersine and implement a quarter sine wave table. Record resource usage for this project too, as you will need to compare the implementations later. This resource will be helpful in implementation: https://zipcpu.com/dsp/2017/08/26/quarterwave.html. Make sure to remember that you only need 256/4 = 64 samples when you instantiate the BRAM on this project.

5.3 Create a simple major arpeggiator

A major scale in music sounds "happy". Minor progressions sound "sad". We're going to create a major arpeggio generator. A major arpeggio consists of a base note, a major third and a fifth[8]. As we saw in the Vihart video, this is just maths. If we have a frequency *f*, then the major third is just *f*1.25*, and the fifth is just *f*1.5*. We will finish off the arpeggio by completing the octave, which will be *f*2*.

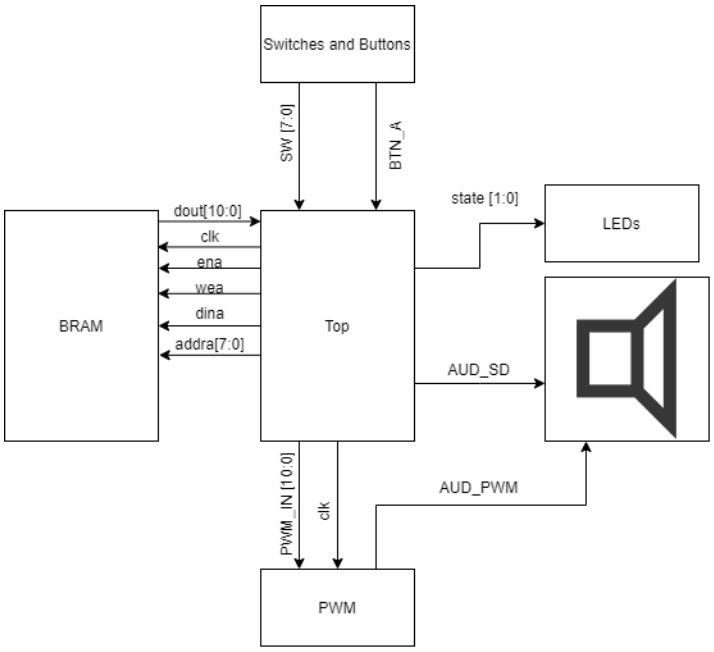

Figure 2: The overview block diagram

So let's think about what we need to do to implement this:

- We need a counter to change the note every 500ms
- We will need a case statement, which, depending on the note, will change the rate at which we switch through the BRAM address
- We will need to read in switches to add to the base note we've chosen (middle C)

[8] Super simple arpeggio explanation for guitarists

- We will need a state to hold whether we are sending out an arpeggio, or just out base note.

With these items in mind, we know our always block might look something like this. In the code I wrote, I made f_base the highest frequency (lowest count). The example is very simple, and doesn't cater for switching between f_base output and arpeggio output. It is just meant to serve as a guide.

Listing 1: Example of "Top" for switching between notes

```
always @(posedge CLK100MHZ) begin
    PWM <= douta; // tie memory to the PWM out

    f_base[8:0] = 746 + SW[7:0]; // get our base frequency

    note_switch = note_switch + 1; // keep track of when to change notes
    if (note_switch == 50000000) begin
        note = note +1;
        note_switch = 0;
    end

    // Output divider to control frequency
    clkdiv <= clkdiv + 1;

    case(note)
        0: begin // base note
            if (clkdiv >= f_base*2) begin
                clkdiv[12:0] <= 0;
                addra <= addra +1;
            end
        end
        1: begin //1.5 times faster
            if (clkdiv >= f_base*3/2) begin
                clkdiv[12:0] <= 0;
                addra <= addra +1;
            end
        end
        2: begin // 1.25 faster
            if (clkdiv >= f_base*5/4) begin
                clkdiv[12:0] <= 0;
                addra <= addra +1;
            end
        end
        3: begin //2 times faster
            if (clkdiv >= f_base) begin
                clkdiv[12:0] <= 0;
                addra <= addra +1;
            end
```

```
        end
        default: begin // Don't know what's happening, just output middle C
            if (clkdiv >= 1493) begin
                clkdiv[12:0] <= 0;
                addra <= addra +1;
            end
        end
    endcase;

end
```

5.4 Hand In

Submit the following as a single PDF:

1. Show evidence that the two implementations produce the same results. To do this:
    - Show screen captures of the test benches at t=0, t= 1/2 pi and t = pi (we're looking to see the changes in values in the output sine value at these points.) [6 marks]
    - To further elaborate on the above - we want to see that the value passed to the PWM module on either side of those values of t follow the same progression.
2. List the resource and power usage of the both implementations [5 marks]
3. Write a paragraph or two on resource consumption of the FPGA. Talk about resource availability, resources used, effort in terms of implementation [5 marks]

Submit a video on YouTube showing the arpeggiator working. Make sure to have the volume loud enough. Please describe what you are showing in the video. [10 marks] The video should show the following:

- In base_f mode, toggle the switches to show that the frequency changes.
- Enable arpeggio mode and show that works by recording the output from a speaker
- Change the output frequency while in arpeggiator mode

References

[1] “Opencl - the open standard for parallel programming of heterogeneous systems,” Jul 2013. [Online]. Available: https://www.khronos.org/opencl/