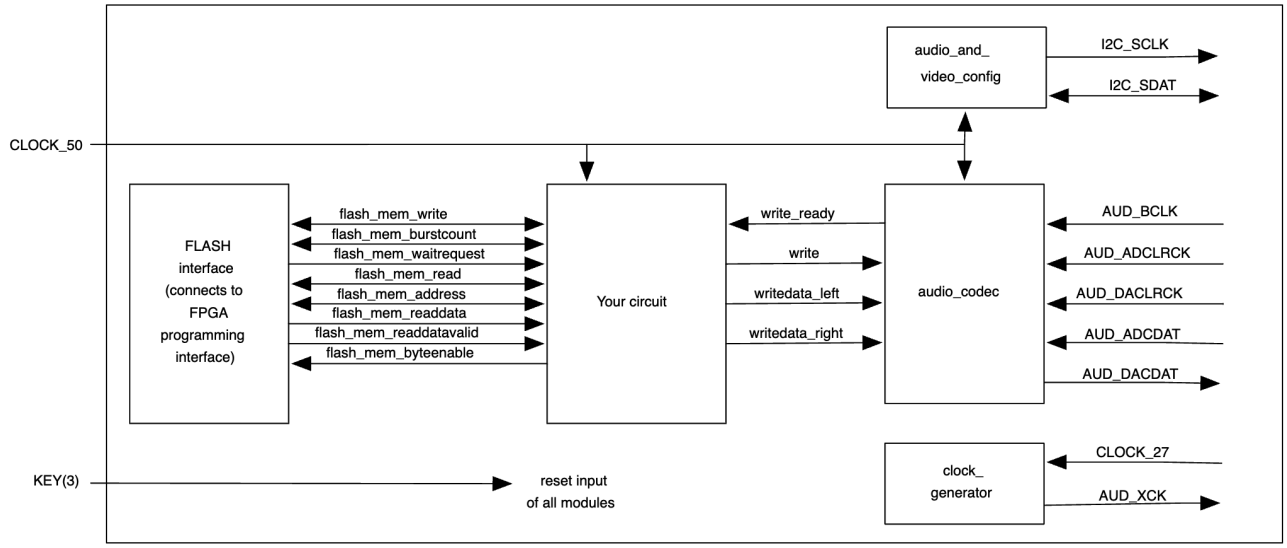
CPEN 311 Lab 5

1 Introduction

Many digital systems process "streaming data" such as audio or video. Rather than implementing complex processing on a small amount of data, these systems often perform relatively simple processing on a large amount of data. In this lab, you will create a system that reads sound samples from off-chip *flash* memory and sends them to your board's audio port. In doing this, you will learn how to read from off-chip memory, deal with audio samples, and interface to an audio core using a handshaking mechanism.

2 Background

The figure below shows the block diagram of the circuit you will build. The block labeled *your circuit* is what you will design. The *audio_codec*, the *audio_and_video_config*, and the *clock_generator* blocks will be given to you, as well as the *flash_memory_interface* block. The interface between your circuit and the audio codec is simple and will be described below. Your circuit will be clocked by the 50MHz clock and will receive an active-low synchronous reset input from KEY[3].



There are two parts to this lab. First, you will build a state machine that reads from the flash memory; you will test this separately. Once that works, you will add the part of the circuit that interfaces with the audio core.

3 Design

3.1 Task 1: Programming the Flash Memory

The flash chip on the DE1-SoC is a separate chip on the board (not part of the FPGA). Flash memory chips are non-volatile (i.e., maintain their contents even after they are powered down) but can still be reprogrammed. The first task is to download our test song to the flash memory chip on your board. Since flash memory is non-volatile, you only have to do this once. Once your chip is programmed, it will remain there as you complete the rest of the lab.

1. Launch the Programmer tool in Quartus Prime.
2. Click *Auto-Detect*. If you get a popup box titled *Select Device*, then choose *5CSEMA5*.

3. If you get a popup asking "Do you want to update the Programmer's device list, overwriting an existing settings?", choose *Yes*.
4. Right-click the *5CSEMA5* device and choose *Edit→Change File*, and select the `american_hero_song88.jic` file from the `data` folder.
5. A new entry will be added to the programmer list for the device *EPCS128*, which is the flash memory chip. For this entry, check the *Erase* box and click *Start*. Wait for it to complete.
6. Next, for the same device, check the *Program/Configure* box and click *Start*. The song will now be programmed into the flash memory.

3.2 Task 2: Build a Flash Memory Reader

In this task, you will build a state machine to read samples from the flash memory chip. The sound samples are each 16-bit signed numbers, and there are 0x200000 (i.e., 2097152 in base 10) samples in the flash memory. To aid your development process, in this task, you will create a state machine that reads the samples from the flash and stores them in on-chip memory. Then, using the in-system memory content editor, you can verify that your flash memory reader is working correctly.

3.2.1 The flash controller interface

You will be interfacing with an Altera flash controller, provided in the `components` folder. It uses the Avalon memory-mapped interface (in docs/avalon_spec.pdf of your Lab folder). The Flash controller is described in Section 39 of the embedded IP user guide (in docs/ug_embedded_ip.pdf of your Lab folder), under *Altera Generic Quad SPI Controller*. Look at Table 39-1, which describes the interface signals.

Aside from `clock` and `reset`, there are two sets of signals: one for accessing control and status registers (CSR) and one for memory access of the flash device. You **do not** need to access the CSR registers so that you can ignore these signals. You should read the descriptions about the `avl_mem_*` signals for retrieving data from the flash memory.

3.2.2 Building your state machine

You will implement your design inside `flash_reader.sv` and the comprehensive tests in `tb_flash_reader.sv`.

If you look inside `flash_reader.sv`, you will see that it instantiates the flash controller. You can include this in your project by including the file `flash.qxp` from the `components` folder.

You will also need to set the *Programming Mode* to *Active Serial* as follows:

- go to *Assignments→Device*, and click on *Device and Pin Options*;
- select the *Configuration* tab;
- in the Configuration Scheme drop-down menu, choose *Active Serial x1*;
- hit *OK* twice to exit the menus.

The state machine you will build will handle the various interface signals to the Flash controller. The inputs to the controller are `flash_mem_write`, `flash_mem_burstcount`, `flash_mem_read`, `flash_mem_address`, `flash_mem_writedata`, and `flash_mem_byteenable`. The outputs are `flash_mem_waitrequest`, hl`flash_mem_readdata`, and `flash_mem_readdatavalid`.

Since we will only be *reading* from the flash, you can connect the `flash_mem_write` port to constant 0 and leave `flash_mem_writedata` unconnected. Also, since we will not be using the burst capabilities of the flash controller, you can connect the `flash_mem_burstcount` to constant 1.

The address signal is a 32-bit word index in the memory, and the data signal is 32 bits wide; you will want all bytes in a word, so set all four bits in `flash_mem_byteenable` to 1. Since an audio sample is only 16 bits, there are two audio samples per word in the flash memory. So, every time you do a 32-bit read from the flash, you will get two samples. For each word, the first audio sample will be in the lower order of 16 bits, and the next audio sample will be in the upper order of 16 bits. So, the very first

sample is in the lower half of the flash at address 0, and the very last sample in the song will be in the upper half of the flash at address 0x7FFFF.

Be sure you respect the `flash_mem_waitrequest` and `flash_mem_readdatavalid` signals in your implementation.

3.2.3 Testing your Flash reader in simulation

As usual, you will need to have a comprehensive testbench in `tb_flash_reader.sv`.

But what about the Flash module? Unlike in the VGA lab, we are not providing a simulation-only Flash module, which is inconvenient since flash_reader.sv needs to instantiate one. Therefore, to successfully test your flash reader; you will have to write your own simulation-only version, based on the template in `tb_flash_reader.sv` and your understanding of the Altera Generic Quad SPI Controller module from the Embedded IP user guide.

We will not test your simulation-only fake flash module directly. That means you don't need to emulate the full functionality of the Flash. Just enough to enable thorough testing of your flash reader.

Make sure that your flash reader also works on the FPGA. Suppose your flash reader works in simulation but not on the FPGA. It may well be that your simulation-only version of the flash module is incorrect.

3.2.4 Testing your Flash reader on the FPGA

Although you will eventually want your flash reader to read all 0x200000 samples, in this task, you will test your reader by reading only **256 samples** and storing them in on-chip memory. The on-chip memory will not be part of the final design for this lab. It is just used here to test your Task 2 in isolation.

You should use the memory wizard to create an on-chip memory that is **16 bits** wide. Your memory module should be called `s_mem`, and the instance should be named `samples` (as in `flash_reader.sv`); its instance ID used in the memory content editor should be `S` (uppercase). You will store the **first 256 samples** in this memory and use the memory content editor to examine them. The result should be as follows:

```
000000  E3 64  C6 C8  C5 45  C3 C3  C7 04  CA 46  CD 88  D0 CA  D2 0C  .d...E.....F......
000009  D3 4E  D4 90  D5 D3  D5 D3  D5 D3  D5 D3  D5 D4  DA 96  DF 59  .N..............Y
000012  E4 1B  E8 DE  E8 A0  E8 63  E8 26  E7 E9  E7 69  E6 E9  E6 69  .......c.&...i...i
00001b  E5 E9  DE 24  D6 5F  CE 9A  C6 D6  C3 50  BF CB  BC 45  B8 C0  ...$._.....P...E..
000024  B7 7C  B6 38  B4 F4  B3 B0  B7 73  BB 36  BE F9  C2 BC  C3 FE  .|.8.....s.6......
00002d  C5 40  C6 82  C7 C5  C9 85  CB 46  CD 07  CE C8  D0 4A  D1 CD  .@.......F.....J..
000036  D3 4F  D4 D2  D6 93  D8 55  DA 16  DB D8  DC D8  DD D8  DE D8  .O.....U..........
00003f  DF D8  E9 1E  F2 65  FB AC  04 F3  09 F9  0F 00  14 07  19 0E  .....e............
000048  17 51  15 94  13 D7  12 1B  0C 15  06 0F  00 09  FA 04  F7 3F  .Q...............?
000051  F4 7B  F1 B7  EE F3  F1 32  F3 71  F5 B0  F7 F0  F9 B3  FB 76  .{.....2.q.......v
00005a  FD 39  FE FC  FF 7D  FF FE  00 7F  01 00  FE 3E  FB 7D  F8 BC  .9...}.......>.}..
000063  F5 FB  F4 38  F2 76  F0 B4  EE F2  EA EF  E6 EC  E2 E9  DE E7  ...8.v............
00006c  DA 22  D5 5E  D0 9A  CB D6  C7 90  C3 4B  BF 06  BA C1  B8 BE  ."."^.......K......
000075  B6 BB  B4 B8  B2 B5  B3 34  B3 B4  B4 33  B4 B3  B4 33  B3 B3  .......4...3...3..
00007e  B3 33  B2 B3  B2 B3  B2 B3  B2 B3  B2 B3  B4 73  B6 33  B7 F3  .3...........s.3..
000087  B9 B4  BD F7  C2 3B  C6 7E  CA C2  CF 86  D4 4A  D9 0E  DD D2  .....;.~.....J....
000090  E3 98  E9 5E  EF 24  F4 EB  F6 2E  F7 71  F8 B4  F9 F7  FA B8  ...^.$.....q......
000099  FB 7A  FC 3B  FC FD  F9 FB  F6 FA  F3 F8  F0 F7  ED F3  EA F0  .z.;..............
0000a2  E7 EC  E4 E9  E4 67  E3 E5  E3 63  E2 E1  E3 E1  E4 E2  E5 E3  .....g...c........
0000ab  E6 E4  E6 E4  E6 E5  E6 E6  E6 E7  E8 E6  EA E6  EC E6  EE E6  ..................
0000b4  F5 AB  FC 71  03 36  09 FC  0F 02  14 08  19 0E  1E 15  1F D8  ...q.6............
0000bd  21 9C  23 60  25 24  23 E3  22 A3  21 63  20 23  1F 61  1E A0  !.#`%$#.".!c #.a..
0000c6  1D DF  1D 1E  1E DE  20 9E  22 5E  24 1F  24 E1  25 A3  26 65  ......  ."^$.$.%.&e
0000cf  27 28  25 67  23 A6  21 E5  20 24  1C E1  19 9F  16 5C  13 1A  '(%g#.!. $.....\..
0000d8  10 16  0D 13  0A 10  07 0D  03 49  FF 86  FB C3  F8 00  F6 BC  .........I........
0000e1  F5 79  F4 35  F2 F2  F2 31  F1 71  F0 B1  EF F1  EF 70  EE EF  .y.5...1.q.....p..
0000ea  EE 6E  ED EE  F0 AE  F3 6F  F6 2F  F8 F0  FC 74  FF F9  03 7D  .n.....o./...t...}
0000f3  07 02  0A 04  0D 07  10 0A  13 0D  14 4F  15 92  16 D4  18 17  ...........O......
0000fc  18 57  18 98  18 D8  19 19                                      .W......
```

Note: In the output above, the tool has displayed each 16 quantity as two 8-bit quantities in big-endian order (MSB followed by LSB). So, the first sample is 0xE364, the second sample is 0xC6C8, etc. If the first word appears to be 0xFFFF, you need to push the reset button.

Common error #1: If each pair of samples is swapped (i.e., the first sample is C6C8 and the second sample is E364, etc), it is possible that you forgot that each 32-bit word in the flash corresponds to two samples: the sample in the lower order of 16 bits comes first, followed by the sample in the upper order of 16 bits.

Common error #2: If the numbers don't look anything like this, it is possible that you read more than 256 samples. In that case, later samples would overwrite earlier ones in the on-chip memory so that you could see a later part of the song. Make sure you are stopping at 256 samples in this task.

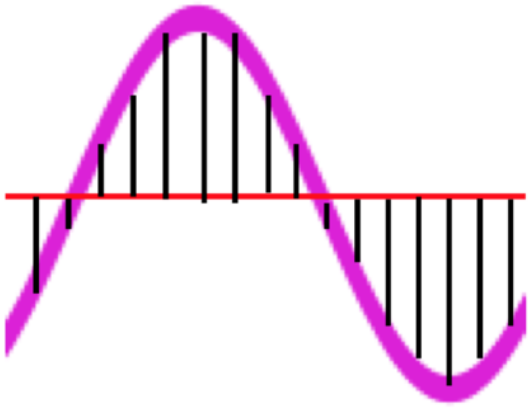
——————— Do not continue until you are sure this works both in simulation and on the FPGA. ———————

3.3 Task 3: Sound in the digital domain

Before continuing, you must understand how sound is represented in the digital domain. In the real world, a sound can be modeled as a continuous wave. This wave may be a simple sine wave (which is a pure tone) or a more complex wave (which can actually be represented as a sum of sine waves).

In the digital domain, this continuous wave is converted into discrete audio samples using a regular sample rate. The collection of samples represents an approximation of the original waveform and, hence, an approximation of the original sound. The amplitude of the continuous wave is an analog value even at these discrete sample points; the analog amplitude is quantized into a 16-bit digital value; any error here is roughly 1 part in 65,000, which is barely perceptible. The rate at which the wave is sampled (the "sampling rate") determines the fidelity of the sampled sound frequency range relative to the original continuous waveform. Typical sampling rates are in the order of 20KHz–80KHz; this results in sufficient samples to reconstruct the original waveform "well enough". We will assume an 88,000 Hz sampling rate — more than enough to make the most dedicated audiophile happy.

The figure below shows the sampling of a single tone (a single sine wave). The samples are a collection of values, some negative and some positive, that can be combined to "look like" the original waveform (in this case, a sine wave). Note that the frequency of the samples is much higher than the frequency of the sine wave.



The sampling rate, the number of samples, and the recording length are all related. In this lab, our song has 2097152 samples, which is less than a minute at our sampling rate. The amplitude of the overall waveform determines the volume. If you multiply all sample values in a song by a constant factor, the overall volume will change.

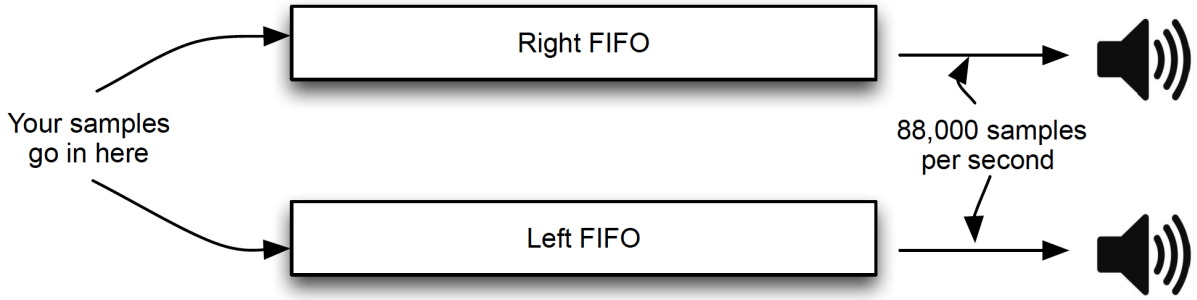
Remember that any complex waveform can represent a sum of sine waves. If the frequencies of these sine waves are altered, the tone will change. This means, for example, if you take a song that was sampled at a certain rate and play it back at a lower sampling rate, the wavelength of each constituent sine wave be *longer* in the reconstructed signal. This will result in a *lower* tone (in fact, for pure tones, halving the playback rate lowers the tone by exactly one octave).

There are no deliverables for this task.

3.4 Task 4: Understanding the CODEC and running the test program

We are supplying you with a pre-designed audio CODEC (coder-decoder). This circuit was designed and distributed by Intel/Altera. The CODEC actually consists of three blocks, and you will include all three blocks in your Verilog code (as shown in `sound.sv`).

The operation of the CODEC is shown below. Your circuit will insert samples into the head of two 128-entry first-in, first-out (FIFO) queues, one dedicated to the right audio output and one dedicated to the left audio output. Your circuit can insert samples into the FIFO if there is space (details on this timing are below). Every $\frac{1}{88000}^{th}$ of a second, the CODEC will remove the top element in each FIFO and send it to the two speakers. The purpose of the FIFOs is to "balance out" bursty activity in your circuit. Without the FIFOs, your circuit would have to produce new samples exactly once per $\frac{1}{88000}^{th}$ of a second, which may make timing in your circuit tricky. With the FIFOs, as long as the average rate of queue entry is the same as the average rate of queue exit (88000 samples per second), the system will correctly play notes. Note that the FIFOs are part of the CODEC core; you do not implement these FIFOs in your own circuit.



To insert samples into the two FIFOs, your circuit should monitor the `write_ready` signal (an output of the CODEC). When this signal goes high, it means there is space in each FIFO for a sample. When your circuit sees it go high, it can drive the `writedata_left` and `writedata_right` buses with the two samples (one for the left speaker and one for the right speaker) and assert the `write` line (these are all inputs to the CODEC). You should then wait for `write_ready` to go low (this may take one or more cycles before it happens), and when it does, you can lower `write` and start again. This handshaking scheme is important, and your circuit will have to implement it properly, or you will not hear any sound.

The `writedata_left` and `writedata_right` buses are each 16-bit wide in our CODEC (it can be configured differently, but you will not change that). The smallest sample value is -2^{15}, and the largest sample value is $2^{15}-1$. Therefore, if you want to make a maximum amplitude square wave of 242 Hz, insert 182 samples of value -2^{15} followed by 182 samples of value $2^{15}-1$ (assuming an 88,000 Hz sampling rate). Note that you can choose a smaller amplitude square wave; the amplitude determines the volume of the sound. For us, 2^{10} was plenty loud.

Note that the CODEC we supply can also operate in the other direction: taking samples from a microphone and supplying them to your circuit. You won't be using that feature in this lab. To turn this feature off, you should keep `read_s` at 0, and ignore outputs `read_ready`, `readdata_left` and `readdata_right`.

To understand the CODEC, we have provided a sample Verilog file that plays a single tone on the CODEC. It plays 91 samples of value -2^{15} followed by 91 samples of value $2^{15}-1$ (and then repeats). This is the sound for a single tone.

Create a new project using the provided Verilog files. You will notice that the audio core consists of a number of Verilog files; all of these need to be added to your project. The state machine that sends samples to the core is in `sound.sv`; you should add this to your project too. Import the pin assignments, compile, and download. To test the circuit, you will need earphones or speakers plugged into the line-out port of your Altera board (to avoid annoying others, please do not use speakers while debugging in the lab). Ensure you can hear a constant tone from earphones plugged into the earphone port of your DE1-SoC.

Examine `sound.sv` carefully and ensure you understand how it works. Pay particular attention to the handshaking interface to the sound core, as you will have to implement something similar in the next task. Do not worry about the code that makes up the audio core (the `*.v` files); you don't need to know what is inside those files for this lab.

There are no deliverables for this task, but be sure you do it — otherwise, it will make the next task significantly more difficult.

3.5 Task 5: Playing the audio samples

In this task, you will assemble all the pieces from the previous tasks. You will design a circuit that reads samples from the flash (as in Task 2) and sends them to the audio core (as in Task 4). Remember that, unlike Task 2, here you want to read **all 0x200000 samples** (not just the first 256). Your design should operate in a continuous loop; after sending the last sample, you should start again with the first.

The audio stream we have given you is **mono** (not stereo). You, therefore, need to send each sample to **both** the left and right speaker. A common error is that if the right and left FIFOs get out of sync (one has more elements than the other), then the `write_ready` signal does not operate as expected. This **Should Not Happen** given the problem statement, but if for some reason you have a bug where the FIFOs do get out of sync, then resetting isn't enough to empty the FIFOs. Power cycling the board and reprogramming the FPGA is necessary.

You will place your design in `task5/music.sv` and the comprehensive testbench in `task5/tb_music.sv`. As in task 2, you may have to build additional simulation-only modules to help you test your design; make sure you have a copy of each file in the folder in which it is used.

You will likely find that if you send samples directly to the audio core, it will be *really loud*. You will, therefore, **divide each sample by 64** before sending it to the CODEC. **Remember** that the samples are all **signed**, so be careful.

A common error will be to hear significant noise (perhaps as loud as the song). If you get this, it is possible that you are not remembering that the samples are **signed**. If you are manipulating signed numbers as unsigned numbers, then the negative samples may be incorrect, leading to an awful-sounding song. Think about how sign extension must work when bit widths do not match.

3.6 Task 6: Alvin and the Chipmunks

As described in Task 3, if the sampling rate and the playback rate are not the same, the song will either be played too fast or too slow. In this task, you will modify your circuit so that you can play back the song in one of three modes (use two slider switches to select the mode):

- SW[1:0] == 'b00 or 'b11: Normal mode: as in Task 5
- SW[1:0] == 'b01: "Alvin and the Chipmunks" mode: playback faster than the sampling rate (twice as fast)
- SW[1:0] == 'b10 "Laid back" mode: playback slower than the sampling rate (half the speed)

You need to be able to change modes *as the circuit is running* (so, for example, as it is playing back, you can switch to "Alvin and the Chipmunks" mode by changing a switch). This means you can't just modify the input files since the flash memory is not big enough to store three versions of the song.

The resulting design will be in `task6/chipmunks.sv`, with its comprehensive testbench in `task6/tb_chipmunks.sv`.

At first glance, it may seem that the best way to do this is to change the playback rate. However, this would mean going into the audio core and making extensive changes, which would be very difficult. Instead, it is much easier to do this by modifying only your code. If you think about this, you should be able to devise a fairly straightforward way to do it.

4 Deliverables and Evaluation

4.1 Using Canvas

Ensure you don't include any extra files that will cause your compiler to fail.

Any template files we give you should be directly modified and submitted using **the same filename**.

Do not share your code or look at other codes. We will run academic integrity checks in December for all the labs. If you are deemed to have cheated, we will follow the academic integrity procedures mandated by the department and UBC to the fullest extent.

4.2 Submission Rules for all students

Please follow these rules strictly.

1. You must not **rename any files** we have provided.
2. You must not **add** any files that contain unused Verilog code; this may cause compilation to fail.
3. Your testbench files must begin with `tb_` and **correspond to design file names** (e.g., `tb_rtl_foo.sv` and `tb_syn_foo.sv` for design `foo.sv`).
4. You must not have **multiple copies of the same module** in separate committed source files in the same task folder. This will cause the compiler to fail because of duplicate module definitions.
5. Your modules must not **rely on files from another folder**. In particular, this means that any memory images you read in your testbenches must be in the same folder. For this lab, you may rely on modules we provided in `components/` for your design.
6. You must not **alter the module declarations, port lists, etc.** in the provided skeleton files.
7. You must not **rename any modules, ports, or signals** in the provided skeleton files.
8. You must not **alter the width or polarity of any signal** in the skeleton files (e.g., everything depending on the clock is posedge-triggered, and `rst_n` must remain active-low).
9. Your sequential elements must be triggered **only on the positive edge of the clock** (and the negative edge of reset if you have an asynchronous active-low reset). No non-clock (or possibly reset) signal edges, no negative-edge clock signals, or other shenanigans.
10. You must not add logic to the clock and reset signals (e.g., invert them). When building digital hardware, the clock and reset must arrive simultaneously as all your FFs; otherwise, your circuit will be slow and, at worst, not working.

If your code does not compile, synthesize, and simulate under these conditions (e.g., because of syntax errors, misconnected ports, or missing files), you will receive **0 marks**. You must submit separate testbench files for the RTL and the post-synthesis netlist; see the naming convention in the deliverables section.

4.3 Submitting with a partner

You do not need a partner, but you may have one.

- If you have a partner, **both you and your partner must submit the same zip file** using your respective Canvas logins. If only one of you submits, you both will get 0 marks.
- **Both partners** must **write the name of their respective partner as a comment** on their submission. **Failing to do so will result in 0 marks**.
- Your partner must be **from the same lab section** and **must be present during the grading process**.
- **You and your partner must arrive at the lab right at the beginning of the grading lab** and write your names on a sheet the TA will provide. **If you or your partner arrive late, you and your partner will get 0 marks.**
- The TA has finite hours, and we cannot wait for team members to arrive at arbitrary times, regardless of whether it is your fault or not. **So please choose your partner (if you want one) very carefully**.

5 Marking: Total 12 Marks

The marks are assigned in the following manner. *Nearly* 70% of your grade will be from the in-person demo for each task. The remaining (*nearly*) 30% of your grade will be from answering the questions about your code by your TA.

5.1 Task 2 [4 marks]

Deliverables in folder `task2`:

- Completed `flash_reader.sv`
- Completed `tb_flash_reader.sv`
- Any other files you added to the design or testbench.

5.2 Task 5 [5 marks]

Deliverables in folder `task5`:

- Completed `music.sv`
- Completed `tb_music.sv`
- Any other files you added to the design or testbench.

5.3 Task 6 [3 marks]

Deliverables in folder `task6`:

- Completed `chipmunks.sv`
- Completed `tb_chipmunks.sv`
- Any other files you added to the design or testbench.