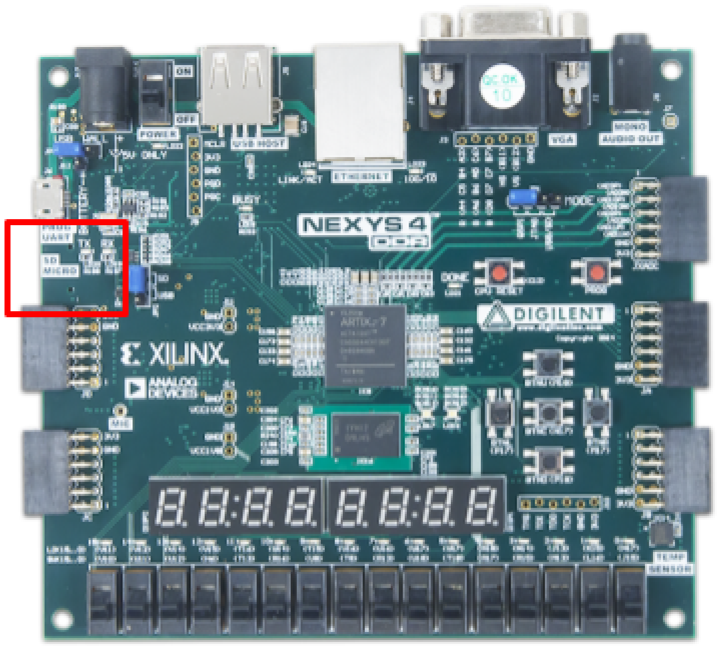
SD Card Tutorial

Why use an SD Card?

The microSD cards in lab have 2GB of storage, which is much more memory than what's available on the FPGA. So, it's a good option if you need to store large amounts of data (audio, images, etc).

- The Nexys 4 DDR has 4860 kBits of BRAM, which was used to store audio in Lab 5a
- Say you wanted to store a song with 8-bit samples
    - The board can store up to 607,500 samples on its BRAM. That's just 12.6 seconds of 48kHz audio.
    - You could store a 3 minute song if the audio was sampled at 3kHz, but that audio would sound terrible, and you would have no more BRAM left for other parts of your project. Lose-lose situation.
    - Our SD card could hold around 12 *hours* worth of 48kHz audio - one song barely makes a dent in it!


Top View

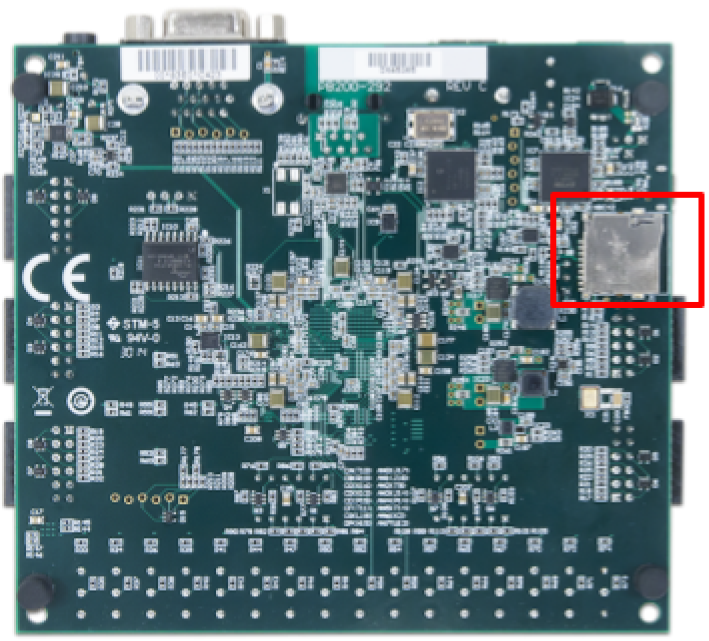

Bottom View

Table of Contents

1) Setup

There are different families of SD cards.

- SD cards have a capacity of up to 2GB,
- SDHC (SD High Capacity) cards can hold up to 32GB, and
- SDXC (SD eXtended Capacity) cards go up to 2TB.

Each of these families have different specifications, and the provided Verilog was designed to work with the SD family (up to 2GB). Ask the course staff for one!

1.1) Data and SD Card Formatting

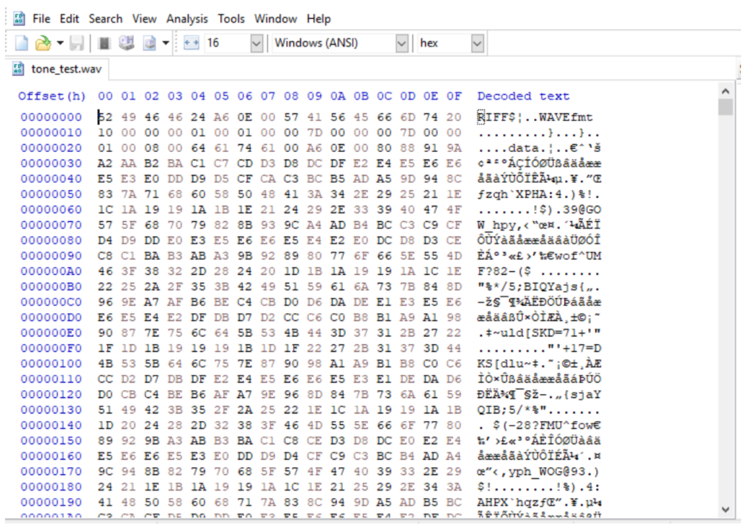
As written, the sd_controller module is compatible with unsigned 8-bit data.

- You can convert audio to this format using an audio editor like Audacity.
    - Here are instructions to do that.
- If you really can't keep the data to 8 bits, you can probably store the bytes of the data sample at adjacent addresses and develop your Verilog accordingly.

You can skip to section 1.2 if you don't need data to be stored on the SD card initially (so if you're just writing data to the SD card, or if you're writing data then reading that same data).

In order for data to be read correctly, it needs to be written *directly* to the SD card. This can be done using a hex editor, such as HxD. Download the portable version of HxD here. Other hex editors will work, too (HxD is only compatible with Windows, but alternatives for Linux/MacOS exist… if you know of a good one let me know!), but we'll go over how you'd do it in HxD.

1. Insert the SD card in your computer.
2. Launch the hex editor (run it as an administrator), and open the file containing your data. It will look something like this.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  52 49 46 46 24 A6 0E 00 57 41 56 45 66 6D 74 20  RIFF$¦..WAVEfmt 
00000010  10 00 00 00 01 00 01 00 00 7D 00 00 00 7D 00 00  .........}...}..
00000020  01 00 08 00 64 61 74 61 00 A6 0E 00 80 88 91 9A  ....data.¦..€ˆ'š
00000030  A2 AA B2 BA C1 C7 CD D3 D8 DC DF E2 E4 E5 E6 E6  ¢ª²ºÁÇÍÓØÜßâäåææ
00000040  E5 E3 E0 DD D9 D5 CF CA C3 BC B5 AD A5 9D 94 8C  åãàÝÙÕÏÊÃ¼µ.¥.”Œ
00000050  83 7A 71 68 60 58 50 48 41 3A 34 2E 29 25 21 1E  ƒzqh`XPHA:4.)%!.
00000060  1C 1A 19 19 1A 1B 1E 21 24 29 2E 33 39 40 47 4F  .......!$).39@GO
00000070  57 5F 68 70 79 82 8B 93 9C A4 AD B4 BC C3 C9 CF  W_hpy‚‹“œ¤.´¼ÃÉÏ
00000080  D4 D9 DD E0 E3 E5 E6 E6 E5 E4 E2 E0 DC D8 D3 CE  ÔÙÝàãåææåäâàÜØÓÎ
00000090  C8 C1 BA B3 AB A3 9B 92 89 80 77 6F 66 5E 55 4D  ÈÁº³«£›'‰€wof^UM
000000A0  46 3F 38 32 2D 28 24 20 1D 1B 1A 19 19 1A 1C 1E  F?82-($ ........
000000B0  22 25 2A 2F 35 3B 42 49 51 59 61 6A 73 7B 84 8D  "%*/5;BIQYajs{„.
000000C0  96 9E A7 AF B6 BE C4 CB D0 D6 DA DE E1 E3 E5 E6  –ž§¯¶¾ÄËÐÖÚÞáãåæ
000000D0  E6 E5 E4 E2 DF DB D7 D2 CC C6 C0 B8 B1 A9 A1 98  æåäâßÛ×ÒÌÆÀ¸±©¡˜
000000E0  90 87 7E 75 6C 64 5B 53 4B 44 3D 37 31 2B 27 22  .‡~uld[SKD=71+'"
000000F0  1F 1D 1B 19 19 19 1B 1D 1F 22 27 2B 31 37 3D 44  ........."'+17=D
00000100  4B 53 5B 64 6C 75 7E 87 90 98 A1 A9 B1 B8 C0 C6  KS[dlu~‡.˜¡©±¸ÀÆ
00000110  CC D2 D7 DB DF E2 E4 E5 E6 E6 E5 E3 E1 DE DA D6  ÌÒ×Ûßâäåææåãá ÞÚÖ
00000120  D0 CB C4 BE B6 AF A7 9E 96 8D 84 7B 73 6A 61 59  ÐËÄ¾¶¯§ž–.„{sjaY
00000130  51 49 42 3B 35 2F 2A 25 22 1E 1C 1A 19 19 1A 1B  QIB;5/*%".......
00000140  1D 20 24 28 2D 32 38 3F 46 4D 55 5E 66 6F 77 80  . $(-28?FMU^fow€
00000150  89 92 9B A3 AB B3 BA C1 C8 CE D3 D8 DC E0 E2 E4  ‰'›£«³ºÁÈÎÓØÜàâä
00000160  E5 E6 E6 E5 E3 E0 DD D9 D4 CF C9 C3 BC B4 AD A4  åææåãàÝÙÔÏÉÃ¼´.¤
00000170  9C 94 8B 82 79 70 68 5F 57 4F 47 40 39 33 2E 29  œ”‹‚yph_WOG@93.)
00000180  24 21 1E 1B 1A 19 19 1A 1C 1E 21 25 29 2E 34 3A  $!........!%).4:
00000190  41 48 50 58 60 68 71 7A 83 8C 94 9D A5 AD B5 BC  AHPX`hqzƒŒ”.¥.µ¼
```

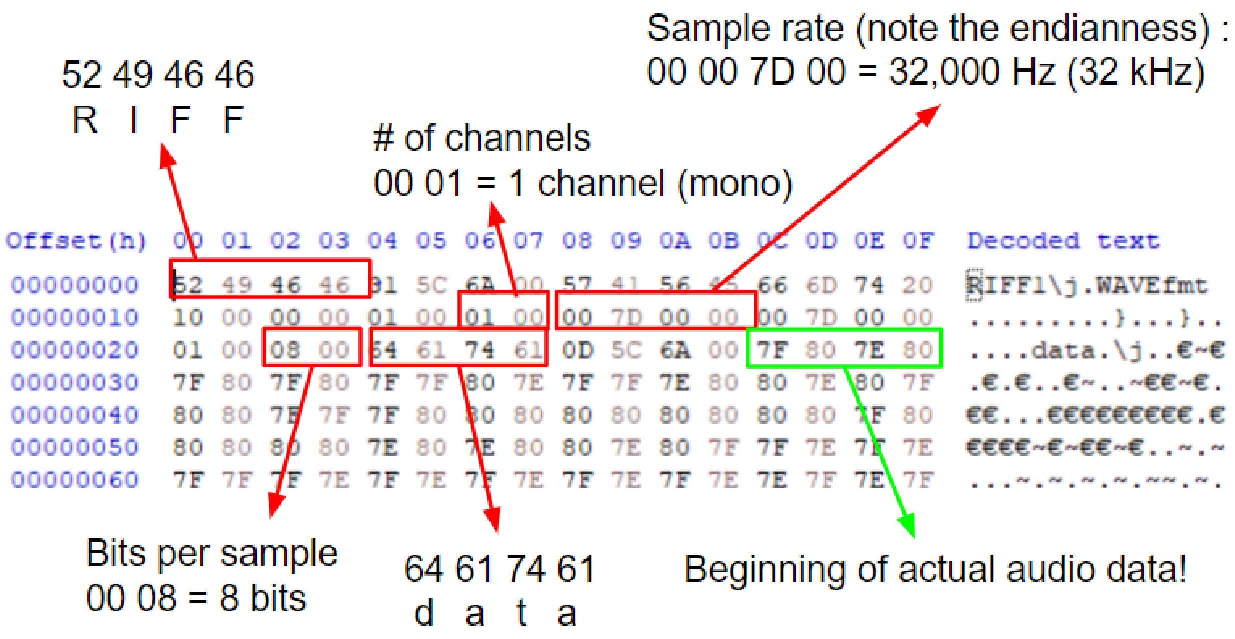
    a. Each byte here is a data sample from a .WAV file. There are 16 samples/row. The offset column keeps track of the number of rows. Adding the row number and column number gives you the address.

3. To copy the data to the SD card:
    a. *Select all* of the data (Edit > Select All) and *copy* it.
        i. Take note of the total number of samples in the file by looking at the final offset (all the way at the bottom… you should be routed here when you select all but you might have to scroll) and adding the number of samples in that row. This could be an important parameter in your Verilog code.
    b. Open the SD card in the hex editor.
        i. Tools > Open Disk
        ii. Choose the removable disk under "Physical disks" (your SD card, unless you have other removable disks in your computer) and uncheck "open as Readonly". - **please don't select your hard drive; make sure you have selected your SD card.**
        iii. Accept the warning about making your disk unreadable - just maybe double check that you chose the correct drive and more importantly not your hard drive. I don't want to know what happens if you write a song directly to your hard drive.

The SD card is split into "sectors," with each sector being 512 bytes. Always paste the data at the beginning of an empty sector. As you add more data files, it would be smart to keep track of the addresses where data is stored so you don't accidentally overwrite anything important.

    c. Find the first empty sector in the SD card (where all of the samples are "00"), and paste the selected samples (Edit > Paste Write).
        i. Take note of the beginning address and the ending address (can just add the number of samples to the beginning address to find this), as these could be important parameters in your Verilog.

An aside about .WAV files - If you're using a .WAV file, you're most likely going to want to start reading from starting at address 512 (the second SD card sector that holds the file). Check if the first 4 bytes are: 52 49 46 46 (or R I F F in ASCII). If they are, the first 44+ bytes of the .WAV file is a header that contains metadata - so data about the audio data in the file. More info about the header here - you can use it to find out things like the sampling rate and # bits/sample!



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  52 49 46 46 31 5C 6A 00 57 41 56 45 66 6D 74 20  RIFF1\j.WAVEfmt
00000010  10 00 00 00 01 00 01 00 00 7D 00 00 00 7D 00 00  .........}...}..
00000020  01 00 08 00 64 61 74 61 0D 5C 6A 00 7F 80 7E 80  ....data.\j..€~€
00000030  7F 80 7F 80 7F 7F 80 7E 7F 7F 7E 80 80 7E 80 7F  .€.€..€~..~€€~€.
00000040  80 80 7F 7F 7F 80 80 80 80 80 80 80 80 80 7F 80  €€...€€€€€€€€€.€
00000050  80 80 80 80 7E 80 7E 80 80 7E 80 7F 7F 7E 7F 7E  €€€€~€~€€~€..~.~
00000060  7F 7F 7F 7E 7F 7E 7F 7E 7F 7E 7F 7E 7E 7F 7E 7F  ...~.~.~.~.~~.~.
```

Some notes:

- The bytes in the header that are being interpreted as their values (rather than as ASCII) are in **big endian** format.
    - So, 32'h00_7D_00_00 in the .WAV file should be interpreted as 32'h00_00_7D_00, which corresponds to 32,000 (the sample rate of this audio file).
- "data," or 32'h64_61_74_61 indicates the start of the actual data.
    - **It's immediately followed by 4 bytes that indicate the size of the data section of the file**, *then* the audio data.
    - Some .WAV files have an INFO marker where a song's name/artist can be stored. Rather than assuming that the audio data would start 44 bytes in, look for this "data" marker to determine where the start of the actual audio data is.
- This explanation assumes that the .WAV file is in PCM format. In addition, to be compatible with the sd_controller module, you will want to convert the .WAV file to be an unsigned 8-bit PCM WAV file. The audio should also be mono (single channel), rather than stereo (two channels).
    - **Converting audio to this format is highly recommended.** See the appendix for instructions on how to convert the file to this format using Audacity.

1.2) Verilog

Download these two files:

- top_level_sd.sv (should have been provided to you with the link to this tutorial)
    - Module with an instance of sd_controller and all of its necessary input/output signals.
    - **Modify this module** to integrate the SD card controller with an existing project.
- sd_controller.v written by Jonathan Matthews
    - **Make sure you save it as a .sv file!**
    - SD card controller module with functionality for reading and writing through SPI (Serial Peripheral Interface)
    - **This module should not be modified!**

Also, check out Jonathan's project, a physical version of Mario Kart, here (he also talks about the SD card around the 4-minute mark). This video isn't necessary for the tutorial, but the project is pretty cool!

In your .xdc file, uncomment:

- `clk_100mhz`
- `sd_reset`
- `sd_cd`
- `sd_sck`
- `sd_cmd`
- `sd_dat[3:0]`

The sd_controller module must have a 25 MHz clock as an input. Create one using the Clock Wizard IP.

- The sd_controller module has *very* important timing loops, so the clock cycles must occur at this frequency.
- This also happens to be the same clock frequency required for VGA 640 X 480 graphics.

Familiarize yourself with the inputs and outputs of the sd_controller module.

```
module sd_controller(
    output reg cs, // Connect to SD_DAT[3].
    output mosi, // Connect to SD_CMD.
    input miso, // Connect to SD_DAT[0].
    output sclk, // Connect to SD_SCK.
                // For SPI mode, SD_DAT[2] and SD_DAT[1] should be held HIGH.
                // SD_RESET should be held LOW.

    input rd,   // Read-enable. When [ready] is HIGH, asseting [rd] will
                // begin a 512-byte READ operation at [address].
                // [byte_available] will transition HIGH as a new byte has been
                // read from the SD card. The byte is presented on [dout].
    output reg [7:0] dout, // Data output for READ operation.
    output reg byte_available, // A new byte has been presented on [dout].

    input wr,   // Write-enable. When [ready] is HIGH, asserting [wr] will
                // begin a 512-byte WRITE operation at [address].
                // [ready_for_next_byte] will transition HIGH to request that
                // the next byte to be written should be presentaed on [din].
    input [7:0] din, // Data input for WRITE operation.
    output reg ready_for_next_byte, // A new byte should be presented on [din].

    input reset, // Resets controller on assertion.
    output ready, // HIGH if the SD card is ready for a read or write operation.
    input [31:0] address,   // Memory address for read/write operation. This MUST
                            // be a multiple of 512 bytes, due to SD sectoring.
    input clk,  // 25 MHz clock.
    output [4:0] status // For debug purposes: Current state of controller.
);
```

SPI Signals - the Nexys 4 DDR board is the main device and the SD card is the secondary device

- `cs` - chip select.
- `mosi` - “main out secondary in,” data going from the board to the SD card
- `miso` - “main in secondary out,” data going from the SD card to the board
- `sclk` - serial clock

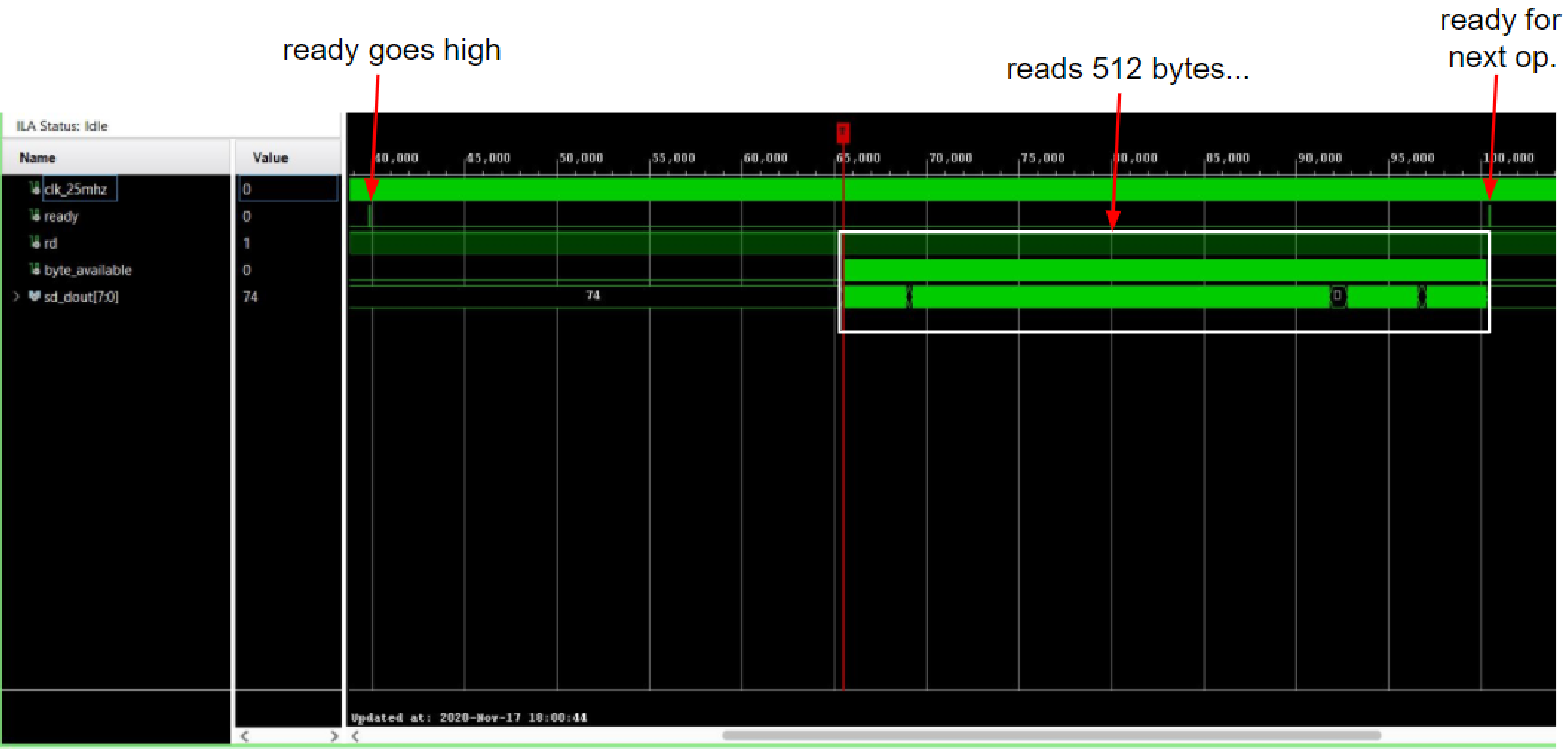
2) READING

The sd_controller module performs reads in 512-byte chunks.

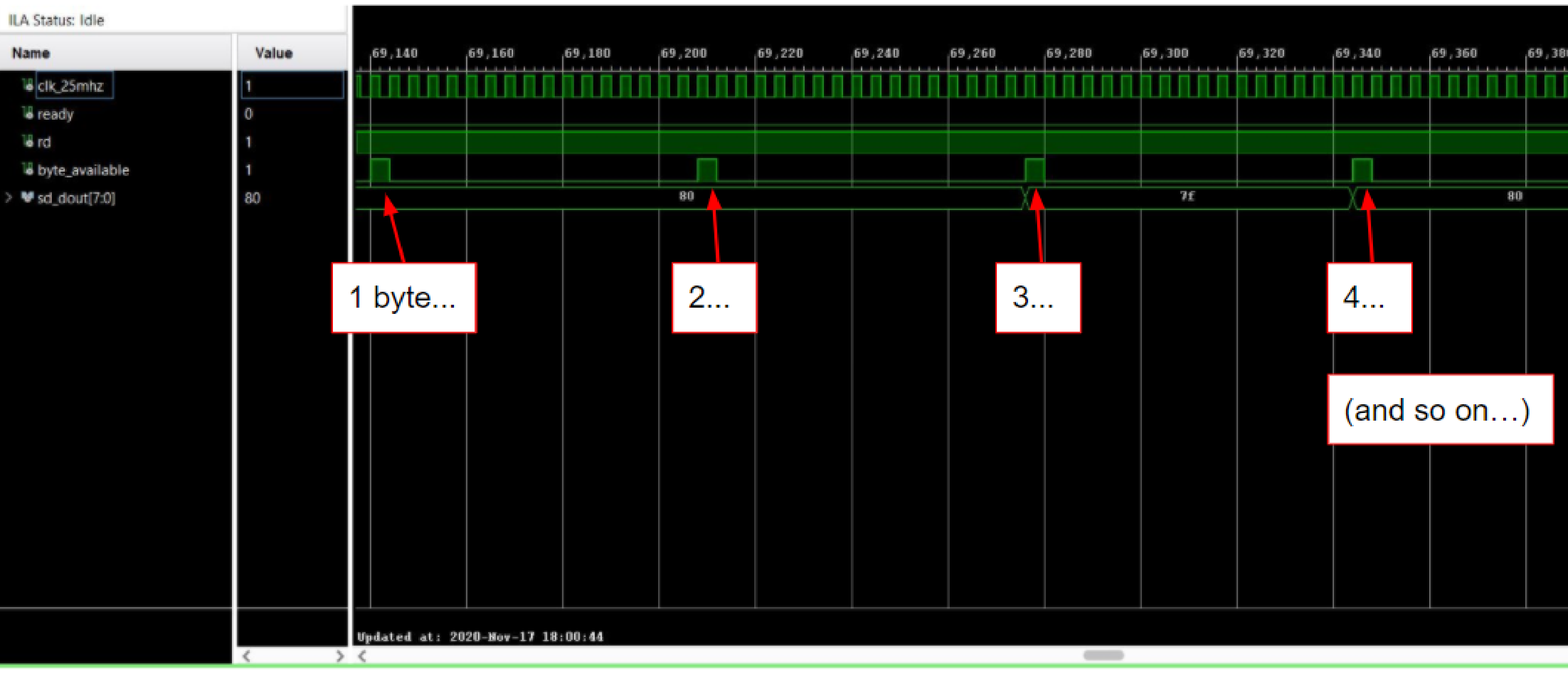
- When `rd` (the read enable) and `ready` (from sd_controller) are both asserted, a 512-byte read operation begins.
    - Once a read operation begins, the sd_controller module will present 512 bytes on `dout`, one at a time, beginning with the byte that is stored at `addr` (which must be a multiple of 512).
    - When a new byte is available, `byte_available` will be asserted.
        - Make sure to check for the rising edge of `byte_available` to prevent multiple reads from the same address.

- At the end of a read operation (once 512 bytes have been read)
    - If you want to keep reading data, increment the address (`addr`) by 512 or set it to whatever location on the SD card has the relevant data (as long as it's a multiple of 512).
    - sd_controller will assert `ready`, because it is ready for a new read operation.
        - If `rd` is still asserted, another read operation will automatically take place.

Read Operation Overview



zooming in a bit...

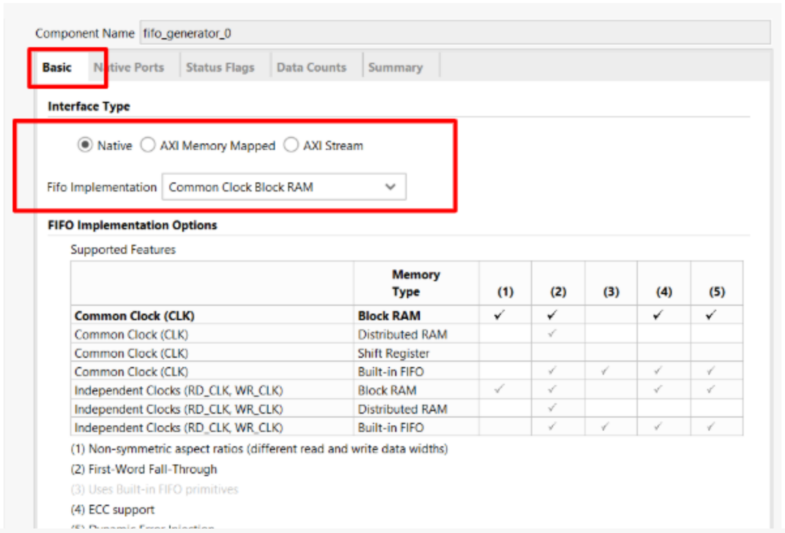


**ILA probing inputs/outputs of sd_controller.v (through an interface module written by Sara Nicholas)*

2.1) FIFO (First In, First Out)

For some applications, the sd_controller module reads data at a much faster rate than it needs to be used - use a FIFO to queue data until the system is ready to use it.

- There is a FIFO Generator IP in the IP Catalog (use the settings shown)



Component Name fifo_generator_0

Interface Type

Native [✓] AXI Memory Mapped AXI Stream

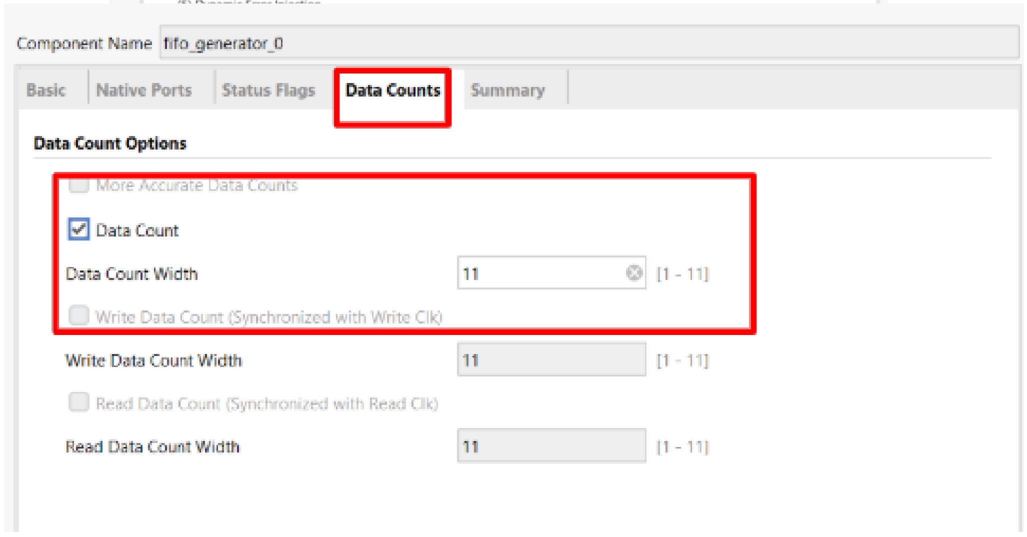
Fifo Implementation: Common Clock Block RAM

FIFO Implementation Options

Supported Features

	Memory Type	(1)	(2)	(3)	(4)	(5)
Common Clock (CLK)	**Block RAM**	✓	✓		✓	✓
Common Clock (CLK)	Distributed RAM		✓			
Common Clock (CLK)	Shift Register					
Common Clock (CLK)	Built-in FIFO		✓	✓	✓	✓
Independent Clocks (RD_CLK, WR_CLK)	Block RAM	✓	✓		✓	✓
Independent Clocks (RD_CLK, WR_CLK)	Distributed RAM		✓			
Independent Clocks (RD_CLK, WR_CLK)	Built-in FIFO		✓	✓	✓	✓

(1) Non-symmetric aspect ratios (different read and write data widths)
(2) First-Word Fall-Through
(3) Uses Built-in FIFO primitives
(4) ECC support



Component Name fifo_generator_0

Data Count Options

More Accurate Data Counts

[✓] Data Count

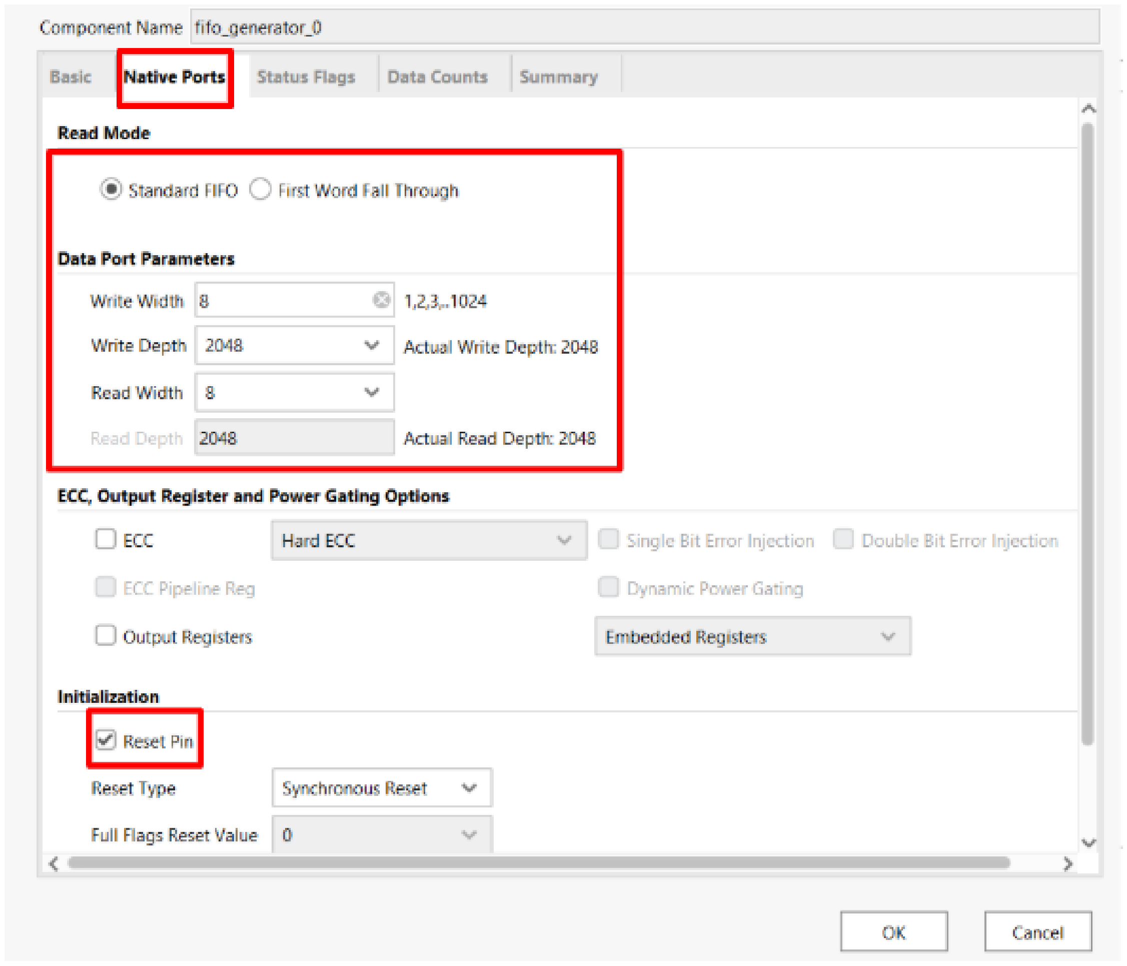
Data Count Width: 11 [1 - 11]

Write Data Count (Synchronized with Write Clk)

Write Data Count Width: 11 [1 - 11]

Read Data Count (Synchronized with Read Clk)

Read Data Count Width: 11 [1 - 11]


Component Name fifo_generator_0

Read Mode

(● Standard FIFO) ○ First Word Fall Through

Data Port Parameters

Write Width	8	1,2,3,..1024
Write Depth	2048	Actual Write Depth: 2048
Read Width	8	
Read Depth	2048	Actual Read Depth: 2048

ECC, Output Register and Power Gating Options

☐ ECC — Hard ECC — ☐ Single Bit Error Injection ☐ Double Bit Error Injection

☐ ECC Pipeline Reg — ☐ Dynamic Power Gating

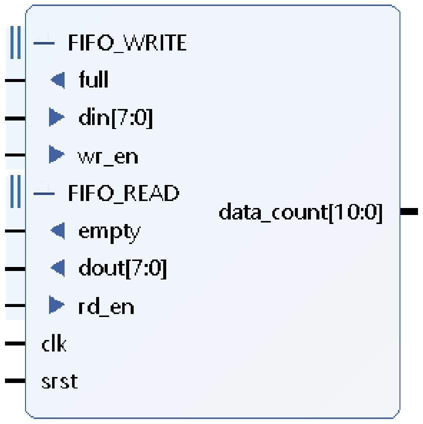
☐ Output Registers — Embedded Registers

Initialization

☑ Reset Pin

Reset Type	Synchronous Reset
Full Flags Reset Value	0

- Feel free to use a different write depth, just make sure it's not so small that it can't hold enough data for your purposes.
  - Use the default `data_count` width (it depends on the write depth)
  - Write width and read width depends on the number of bits in your data samples.



- `full/empty:` high when the fifo is full or empty.
  - When full, nothing can be written
  - When empty, nothing can be read
- `din:` data written to the FIFO
- `dout:` data read from the FIFO
- `wr_en:` FIFO write-enable
- `rd_en:` FIFO read-enable
- `clk:` use system clock
- `srst:` link to system reset
- `data_count:`keeps track of the number of data

samples stored in the FIFO at a given time

Overview

- On the rising edge of `byte_available` (from sd_controller)
  - Assert `wr_en` to write the byte from `dout` to the FIFO
    - `wr_en` should be a pulse -- you only want to write each byte to the FIFO once!
- Make sure the FIFO doesn't become full (as this will disable any reads) by adjusting `rd` (the read enable from the sd_controller module) to stop/start reading from the SD card.
  - One solution is to set `rd` low if `data_count` passes a certain threshold.
- Read from the FIFO at the rate at which you want to use the data.
  - Keep track of the number of cycles in order to determine when to assert the FIFO's `rd_en`
    - `rd_en` should be a pulse -- same reasoning as making wr_en a pulse.

3) Writing

The sd_controller module write operation writes 512 bytes to the SD card.

- When `wr` (the write enable) and `ready` (from sd_controller) are both asserted, a 512-byte write operation begins.
  - During a write operation, the sd_controller module will take in 512 bytes on `din` (one at a time), and write each one to the SD card.
    - Make sure whatever you have on `din` at a given time is the byte that should be written to the SD card.
    - Since sd_controller only takes in the address that corresponds to the beginning of the sector, it could be helpful to keep track of how many bytes have been written during the current operation (so the number of rising edges of `ready_for_next_byte`). This way, you'll know exactly what address a byte will be written to.
  - When sd_controller is ready to write a new byte to the SD card, it asserts `ready_for_next_byte`, and the value assigned to `din` is written to the SD card.
- At the end of a write operation (once 512 bytes have been written to the SD card from `addr` to `addr + 511`).
  - If you want to keep writing data, increment the address by 512, or set it to whatever address you want to start writing to (it just needs to be a multiple of 512).
  - sd_controller will assert `ready` when it's ready for a new write operation. If `wr` is still high, a new write operation will take place automatically.
    - Make sure wr is LOW when you don't want to write information to the SD card. Otherwise, whatever is on `din` could get written by mistake.

4) APPENDIX

4.1) USEFUL INFORMATION/SOURCES

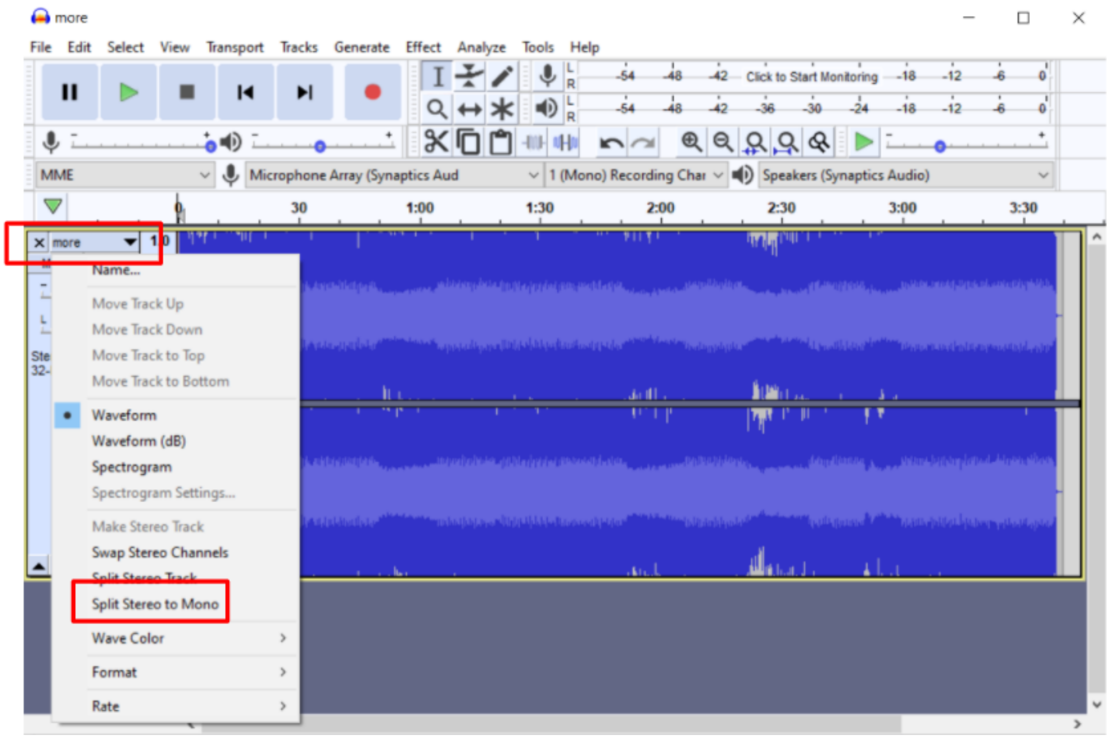
sd_controller Tutorial (Fall 2015) by Jonathan Matthews

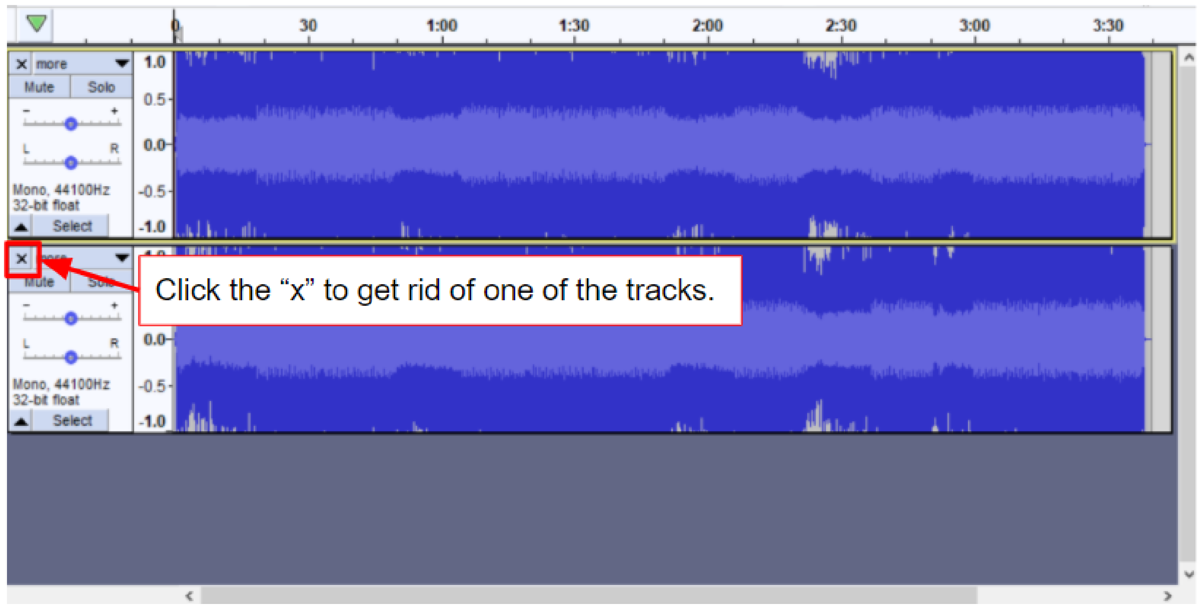
4.2) AUDIO FORMATTING IN AUDACITY

The given sd_controller module is compatible with unsigned 8-bit samples, and the Nexys 4 DDR outputs audio through a single channel (mono). To get the .WAV file into this format, I used Audacity, an audio-editing software, but this can probably be done with other software as well - regardless of software used, make the same changes to the audio file as outlined below.

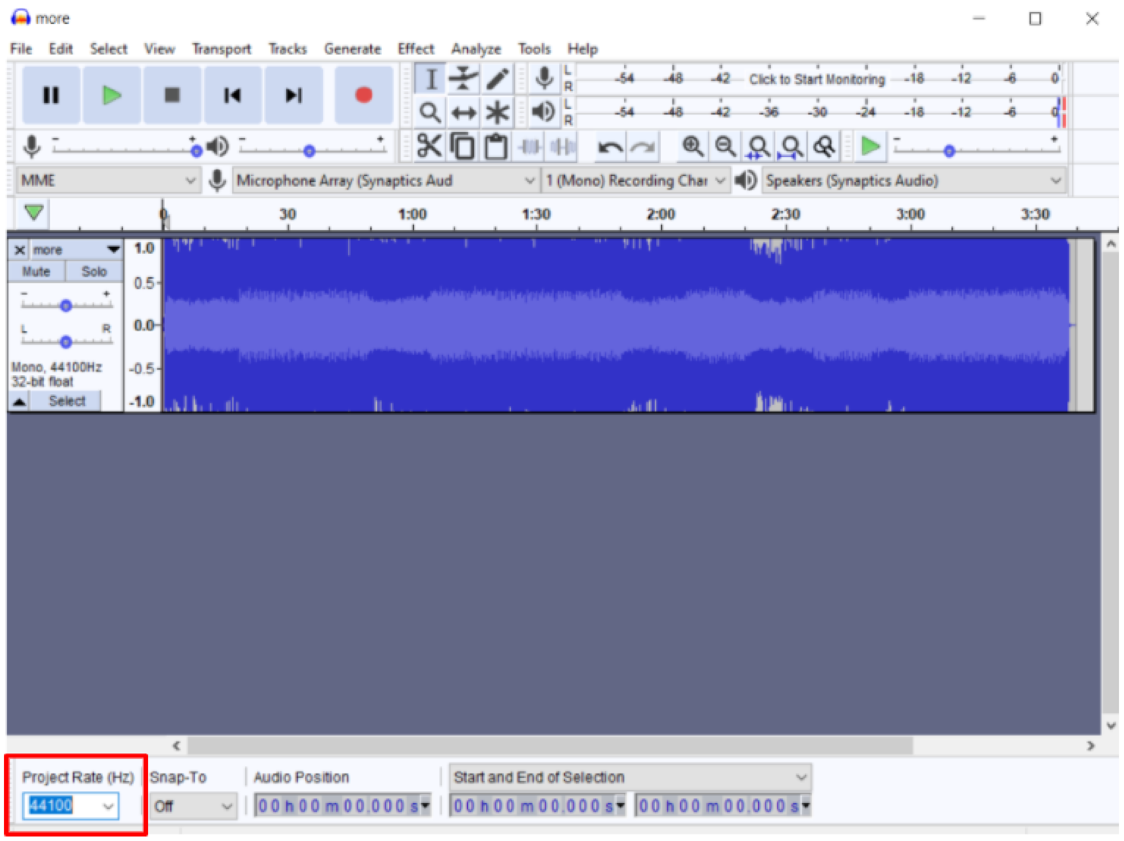
Given a file with the desired audio:

1. Launch Audacity, and open the audio file you want to convert.
    a. Select "Make a copy of the files before editing"
2. Convert the audio from mono to stereo.
    a. Clicking on the track's name will open a drop-down menu.
        i. Choose "Split Stereo to Mono," then remove one of the tracks.



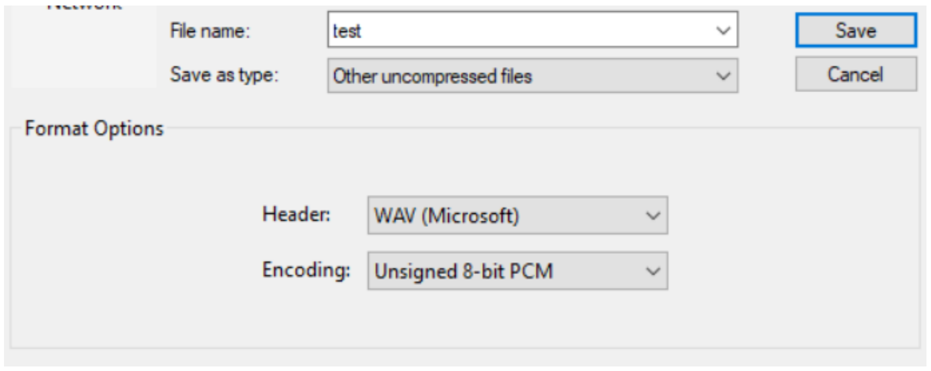


3. Change the sampling rate to whatever you want (not required, 44.1 kHz should be fine).
    a. At the bottom left corner of the screen, change the value under "Project Rate."



Project Rate (Hz) 44100
Snap-To Off
Audio Position 00 h 00 m 00.000 s
Start and End of Selection 00 h 00 m 00.000 s 00 h 00 m 00.000 s

4. File > Export > Export as WAV
    a. **Make sure the file is exported with these settings (WAV header, Unsigned 8-bit PCM encoding) for compatibility with the existing Verilog.**


File name: test
Save as type: Other uncompressed files
Format Options
Header: WAV (Microsoft)
Encoding: Unsigned 8-bit PCM

4.3) Audio Output from Nexys 4 DDR

- Integrate the given PWM module to your project.
    - `music_in` should be a byte of audio data.
- Uncomment `aud_sd` and `aud_pwm` in the .xdc file
    - `aud_sd` is an output that should always be high.
        - `assign aud_sd = 1’b1;`
    - `aud_pwm` is an output that should be assigned using this line of Verilog
        - `assign aud_pwm = output_from_pwm_module?1'bZ:1'b0;`
        - Rename “`output_from_pwm_module`” to whatever you named the output from the pwm module.