ECE 554 FPGA Tutorial

Tutorial on DE1-SoC Board (Audio)

Tutorial Description

- This tutorial will use the following peripherals of the FPGA board:

1. ADC Converter
2. Audio CODEC
3. VGA Output
4. PLL

- One push buttons on the board is used as active low reset.

Create a project

- DE1-SoC evaluation board provides a software that can help us generate all the project files. Normally, we have to create the project in Quartus software by Altera and set device and pin information. Using DE1-SoC system builder, we simply generate the project files with all the settings configured for us.

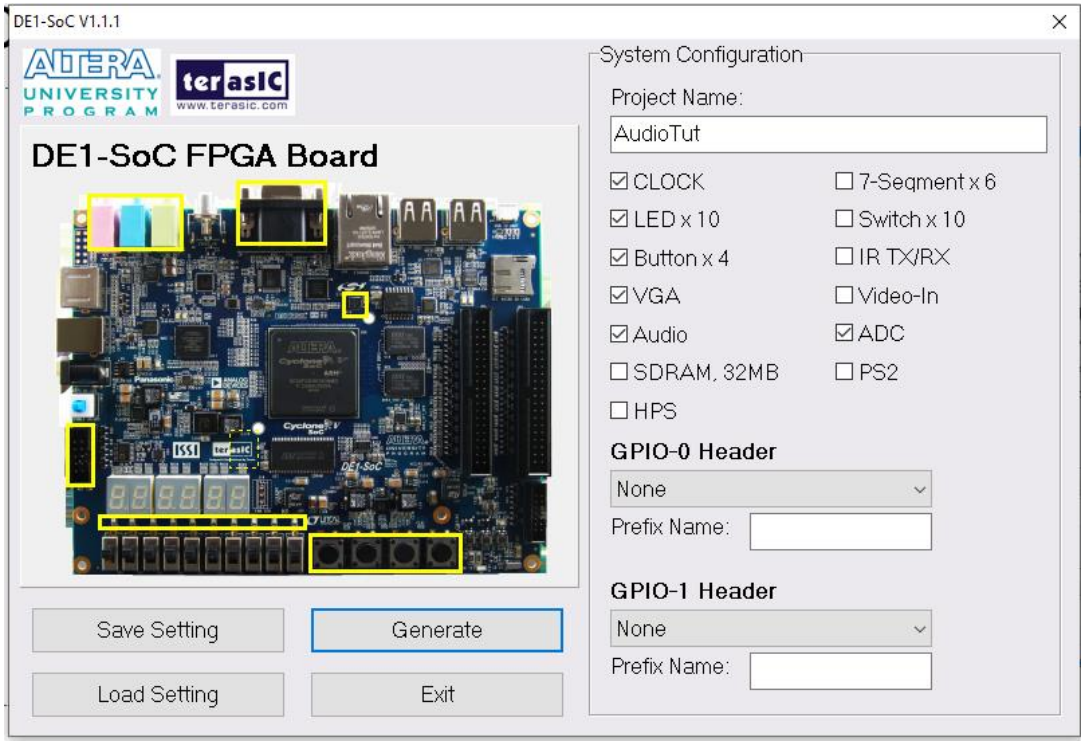
- Download Altera DE1-SoC SystemCD from the course website

- Open

DE1-SoC_v.5.1.1_HWrevF_SystemCD\Tools\SystemBuilder
DE1SoC_SystemBuilder.exe

Create a project

- Set “Project Name” to “AudioTut”
- Select
  - CLOCK
  - ButtonX4
  - VGA
  - Audio
- Set GPIO-0 Header to “None”
- Click “Generate”, save as “AudioTut.qpf”
- The generated files will be placed under the “CodeGenerated” folder under SystemBuilder



File description

- AudioTut.qpf : Quartus project file
- AudioTut.qsf : Quartus setting file, including device, pin assignments, etc.
- AudioTut.sdc : synopsys design constraint file used for compiling the design
- AudioTut.v : top level Verilog HDL template file

The use of system builder is very handy as it ensures the pin mappings in the .qsf file are correct.

In most cases you may wish to edit the generated .qsf file to use more “standard” signal names.

Add sources to the project

- Normally you would copy the generated files to your work area and edit both the .qsf and the top level .v file.

- In this tutorial you will use the provided **.qsf, .qpf**, and **AudioTut.v** files which have been edited to have more meaningful signal names for some ports.

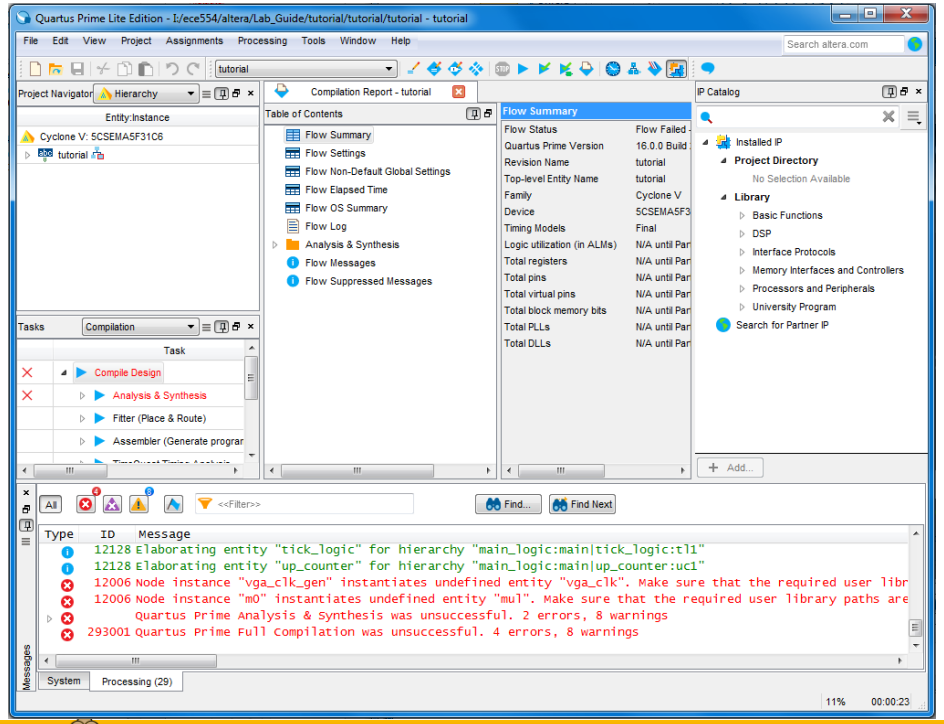
- In the tutorial zipfile, we have provided some Verilog files for you. Copy all the **.v** & **.sv** files to the **AudioTut** folder.

Open the project

- From **AudioTut** folder, click **AudioTut.qpf** to open the project in Quartus
- This may take a few minutes
- Click “project”-> “add/remove file in project”
- Add all the source files into your project

Compile

- Double click “compile design” in the bottom-left window to see what happens
- You should get error messages as shown in the picture



Add IP core into your project

- The error is because we haven't generated the IP core used in the project
- In this tutorial, we will use the following IP core

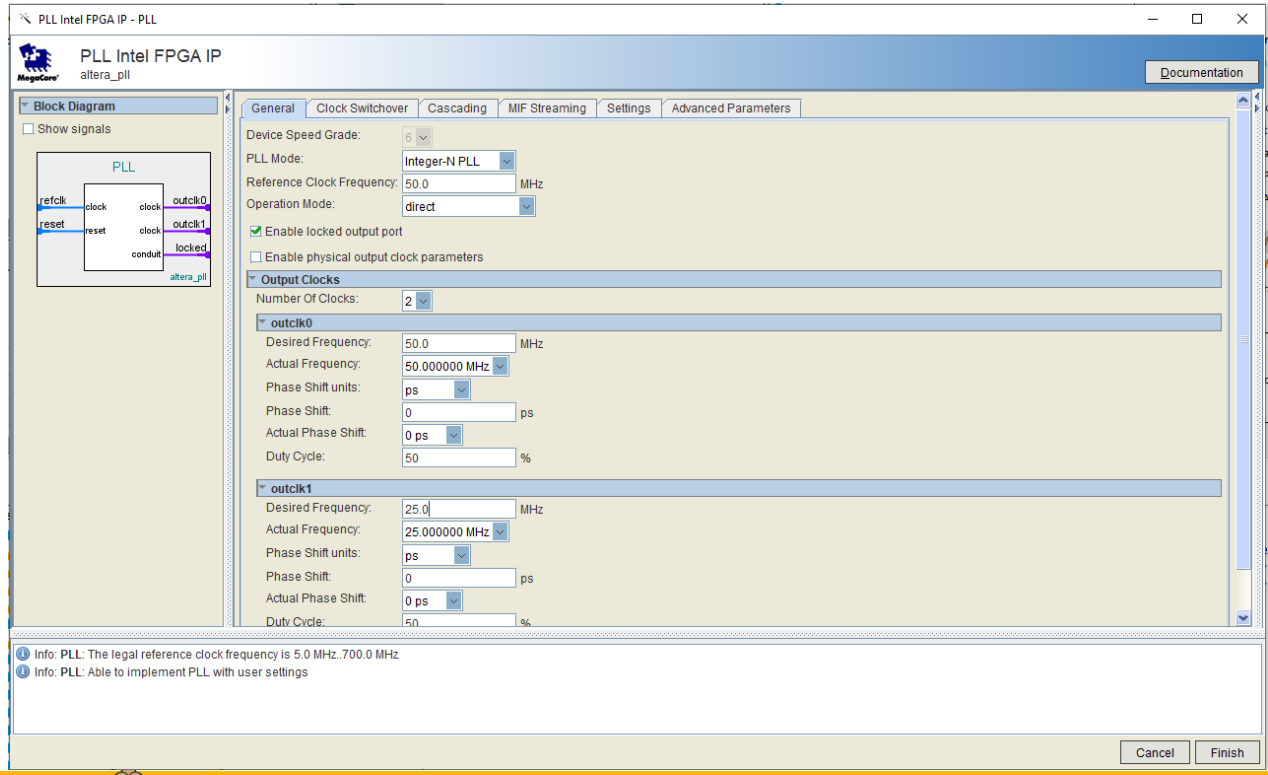
  Clock Generator (PLL)

Add Clock Generator IP core into your project

- From the top-right window, click “Tools”-> “IP Catalog”, the IP Catalog window will show on the right of the screen
- Search for “PLL” and select “Intel PLL”
- Name the file as “PLL” on the pop-up window, then click OK
- You may have to wait a few minutes for a new window (for MegaWizard) to open

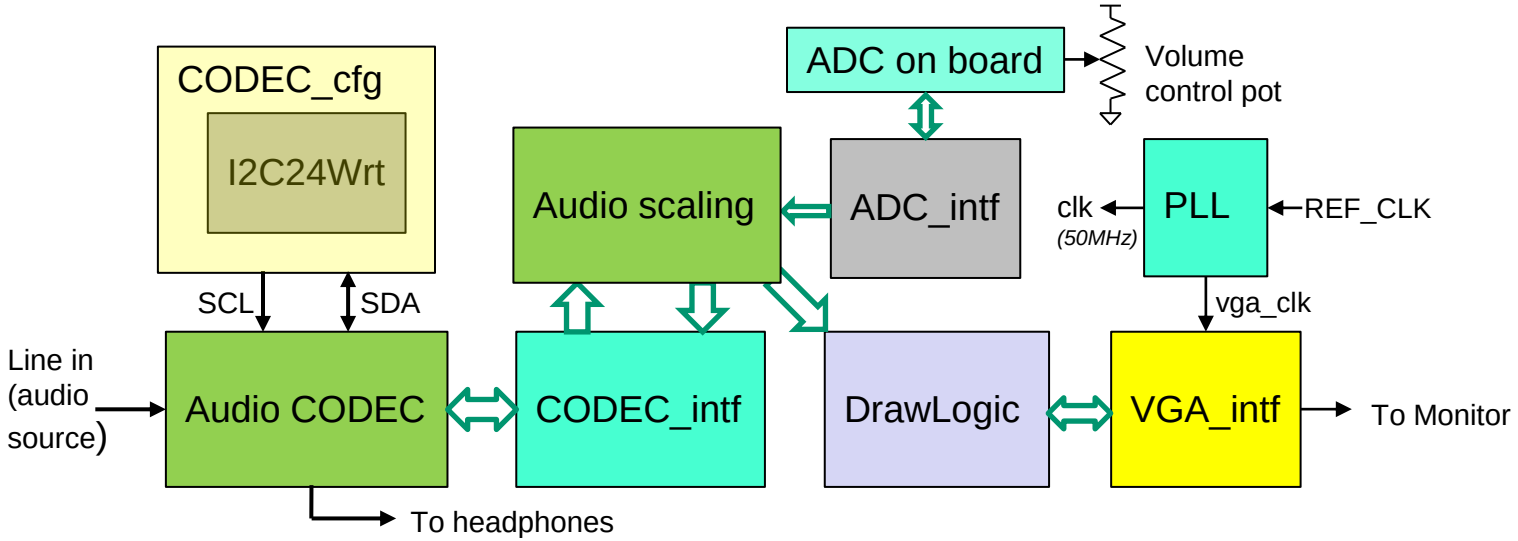
Add Clock Generator IP core into your project

- Set “Reference Clock Frequency” to 50MHz
- Set drop down for number of Clocks to 2
- Set “Desired Frequency” of first clock to 50MHz.
- Set Desired Frequency of 2nd clock to 25MHz



Add Clock Generator IP core into your project

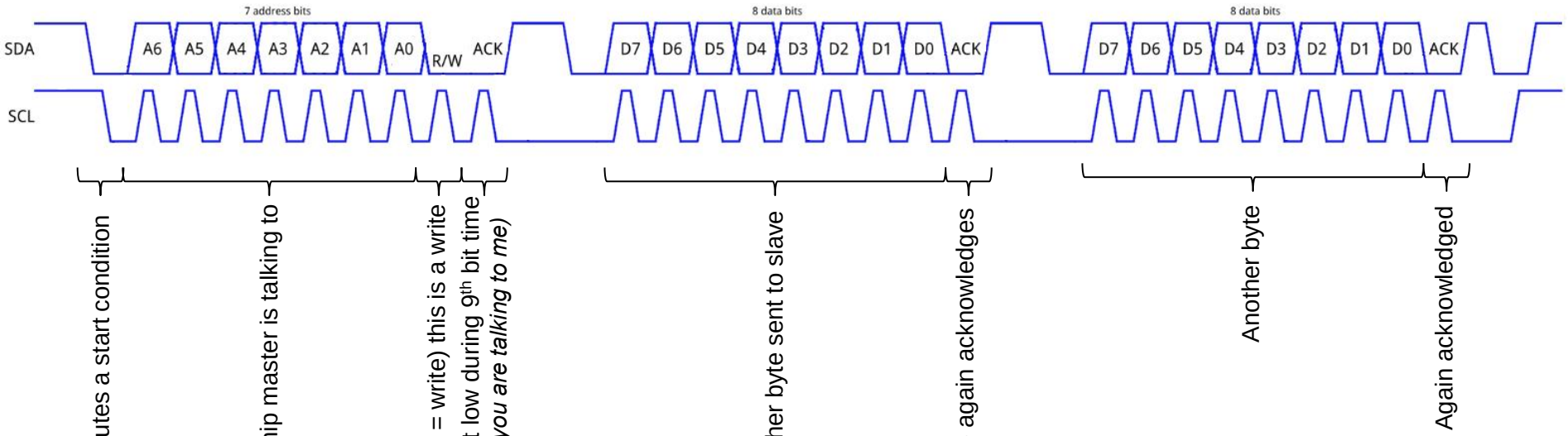
- A window will pop up asking you whether to add this file into your project
- In that window, check the box to automatically add the IP files to all projects
- Then click “Yes”



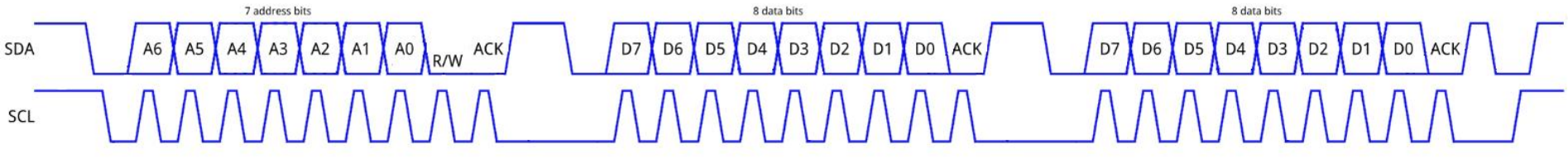
The audio is digitized by the ADC half of a CODEC. The digital data is scaled by a factor, and written to the DAC half of a CODEC which drives a headphone jack. The left/right audio signals are also sent to "DrawLogic" that creates a running graph of audio amplitude on a VGA monitor. The factor the digital audio is scaled by is controlled by a potentiometer being read by and analog to digital converter. A PLL is needed to generate a 25MHz **vga_clk** that is in phase with the 50MHz clock that runs the rest of the system. The Audio CODEC chip requires a lot of configuration to run in the mode we wish. This configuration occurs over an I2C bus. The I2C peripheral is a child of CODEC_cfg. The code for CODEC_cfg is incomplete, and you have to complete it.

I2C24Wrt is an I2C master that writes 24-bit packets *(8-bit slave address followed by a 16-bit data)* over an I2C bus. You apply the 16-bit data to write on **data16** and assert **wrt**. When the transaction is done it asserts **done** *(but we won't actually use that signal).* You need to complete the code to configure the CODEC **(see the next slide).**

Background on I2C



I2C Specific to the CODEC



address hardcode into the provided I2C24Wrt block

How does slave ack?

High byte sent to CODEC

tb does this

Low byte sent to CODEC

tb does this

Address of the CODEC chip itself. Yes, we only have one chip on the I2C bus, but there could be many slaves hanging off the SDA/SCL lines.

7-bit register address

9-bit data

Address of register within CODEC chip that we are writing

Data to be written to that register…yes…the registers are 9-bits wide. Weird but true.

Audio CODEC Configuration:

7-bit Reg Addr:	9-bit data:	16-bit data	Description:
0x00	0x105	0x0105	Write to lft or right is write to both, left not muted, cut down dB from default
0x01	0x105	0x0305	Write to lft or right is write to both, right not muted, cut down dB from default
0x04	0x012	0x0812	Not bypassed, Use line in not microphone.
0x05	0x006	0x0A06	48kHz deemphasis, soft mute disabled, high pass enabled
0x06	0x062	0x0C62	Most stuff powered up (MIC and OSC and CLKOUT powered down)
0x07	0x001	0x0E01	16-bits, Left Justified, no inversions of controls
0x09	0x001	0x1201	Turn it on

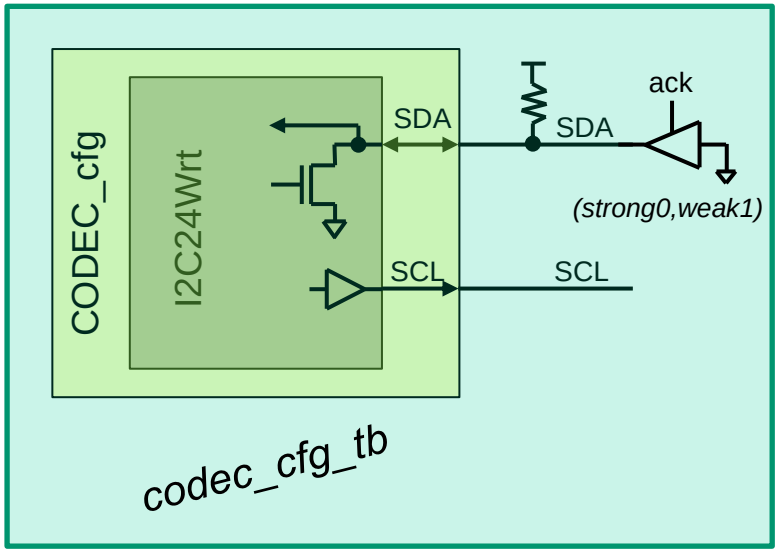
The CODEC is configured via an I2C bus. The I2C unit provided (***I2C24Wrt***) is not a general purpose I2C master but rather one that is hardcoded for 24-bit transactions, and the transactions must be writes.

See the 16-bit data column in the table above for the values that need to be sent via I2C to configure the audio CODEC as we wish.

Implement an 18-bit timer in this unit. Do not write to the first register until this 18-bit timer has expired *(this ensures CODEC has had time to get through its own reset sequence).* Instead of using the **done** signal from ***I2C24Wrt*** wait 2048 clocks between register writes to the CODEC. Do not prove to me that you forgot everything you learned in 551. It will break my heart.

You should also produce a testbench for **CODEC_cfg** *(call it **CODEC_cfg_tb.sv**) (see next page)*

codec_cfg_tb



The testbench will just be used for you to ensure the waveforms for SDA and SCL look approximately correct for the 7 writes that will occur to the CODEC.

We don't have a good model of the CODEC to make a self-checking testbench.

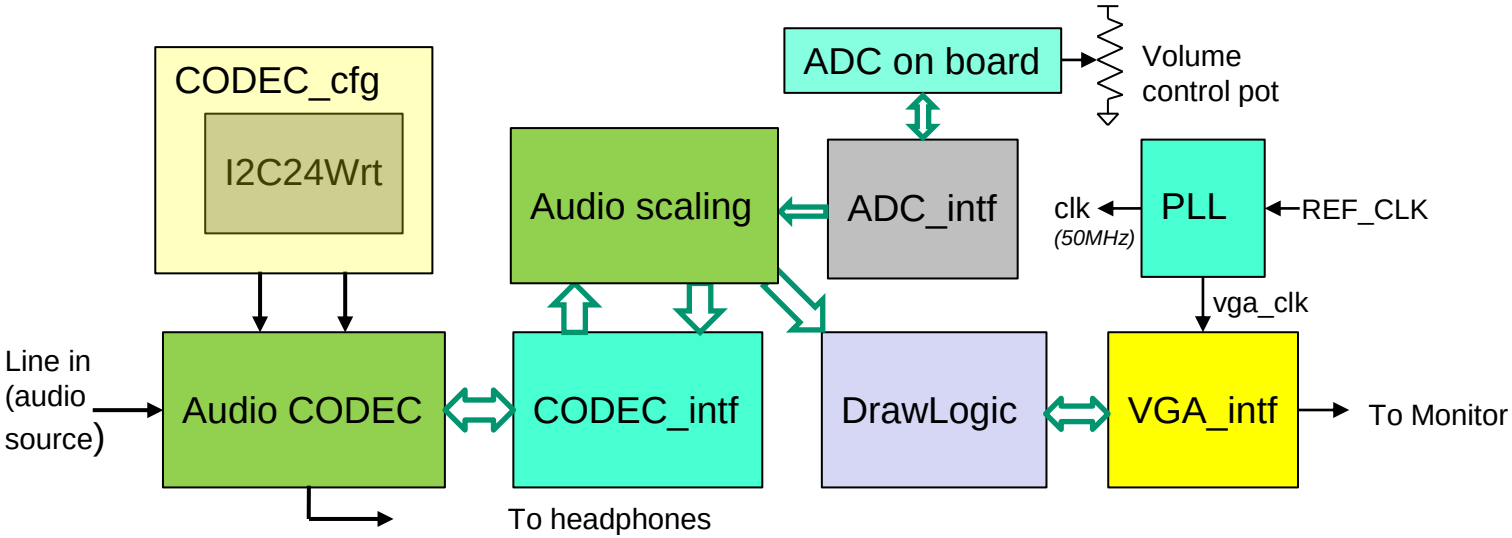
Your testbench does, however, have to at a minimum pull SDA low to properly ACK the bytes.

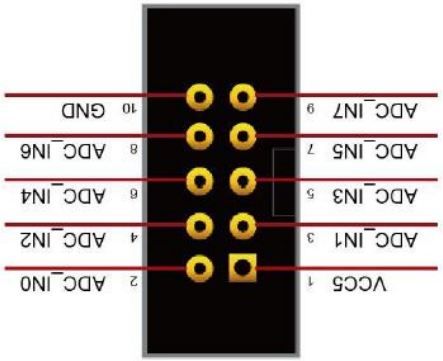
The SDA line is a drive type called "open drain". There is only a pull down transistor, not a full push-pull driver. On the system level there is a pull-up resistor on SDA. So if the master or the slave is not pulling SDA low it is by default pulled high by the resistor.

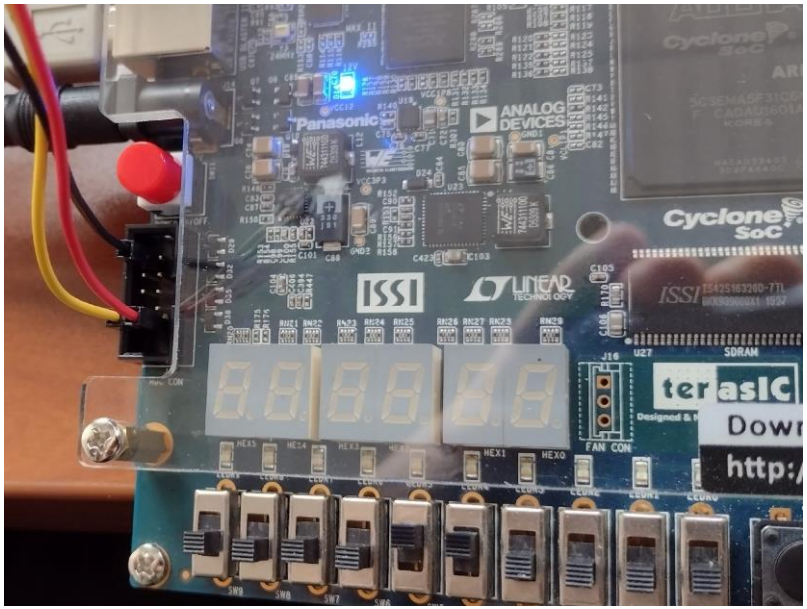
We don't have a good way of modeling the resistive pull-up in our testbench. What we can do, however, is to have a tri-state that is strong at driving a zero, and weak at driving a 1. When the slave is ACKing it drives a strong0, and when it is not a weak 1 is driven. This combination acts like an open drain driver with a pull-up resistor.

Compile and download your design into FPGA

- Now compile the project, and observe the progress of the compilation steps from the bottom-left. Full compilation may take a few minutes.
- You will then find the generated “AudioTut.sof” file under your project (AudioTut) folder
- From the DE1-SoC files, locate the following file *DE1-SoC_v.5.1.1_HWrevF_SystemCD\UserManual\My_First_Fpga.pdf* and follow chapter 4.2 to download the sof file into the FPGA.
- Plug the VGA cable to a monitor. Use SW0 switch as reset.
- Hook and audio source (like your phone) to the line in jack
- Hook either headphones or the small speaker to the headphone jack of the board. Connect the pot to the A2D interface pins of the board.
- Play music and be amazed. Change volume using pot.







Songs that begin with <u>good stereo separation:</u>

TNT (AC/DC)

Reckoner (Radiohead)

Handlebars (Flobots)

Soul Bossa Nova (Quincy Jones)

Just From Chevron (Dirty Projectors)

Submit & Demo:

- Submit both **CODEC_cfg.sv** & **CODEC_cfg_tb.sv** to the dropbox on Canvas
- You must demo your working tutorial to either Eric or Tananun by the end of lab period on Tues Jan 31st (15% penalty for each day late after that)
- The following slides are from 551 and a reminder of how to code a SM.

SM Coding Guidlines

1) Keep state assignment in separate **always_ff** block using non-blocking "<=" assignment
2) Code state transition logic and output logic together in a **always_comb** block using blocking assignments
3) Assign default values to all outputs, and the ***nxt_state*** registers. This helps avoid unintended latches
4) Remember to have a default to the case statement.

   Default should be (if possible) a state that transitions to the same state as reset would take the SM to.

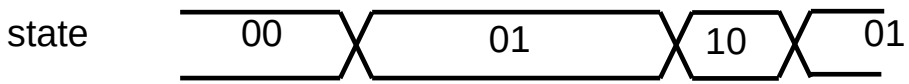
   Avoids latches

   Makes design more robust to spurious electrical/cosmic events.

The Beauty of the Enumerated type

- One construct old school verilog was missing was an enumerated type.
- For state definition we use **localparam** was used for code readability

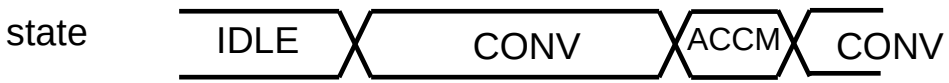
```
localparam IDLE = 2'b00;
localparam CONV = 2'b01;
localparam ACCM = 2'b10;
```



However, in waveform viewing it still shows up as digits. One has to always refer back to the encoding while debugging.

- System verilog corrected this by adding an enumerated type.

```
typedef enum reg [1:0] { IDLE, CONV, ACCM } state_t;
state_t state, nxt_state;                    // declare state and nxt_state signals
```



Makes debug much easier

SM in System Verilog

```
module UART_wrapper_sm(input clk, rst_n, …

typedef enum reg [1:0] {HIGH,MID,LOW} state_t;
state_t state, nxt_state;

logic capture_high,capture_mid,set_cmd_rdy,
        clr_cmd_rdy,clr_rdy;

////////////// infer state flops ///////////////
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    state <= HIGH;
  else
    state <= nxt_state;

always_comb  begin
  ///// default outputs //////
  capture_high   = 0;
  capture_mid  = 0
  set_cmd_rdy  = 0;
  clr_cmd_rdy = 0
  clr_rdy  = 0;
  nxt_state = HIGH;
```

```
  case (state)
    MID : if (rdy) begin
       capture_mid = 1;
       clr_rdy = 1;
       nxt_state = LOW;
    end else
      nxt_state = MID;
    LOW : if (rx_rdy) begin
       set_cmd_rdy = 1;
       clr_rdy = 1;
       nxt_state = HIGH;
     end  else
       nxt_state = LOW;
     ///// default case = HIGH /////
     default : if (rdy) begin
       capture_high = 1;
       clr_rdy = 1;
       clr_cmd_rdy = 1;
       nxt_state = MID;
     end
   endcase
end
```

Default nxt_state to state

Allows less coding if a state is to remain in same state

```
module UART_wrapper_sm(input clk, rst_n, …

typedef enum reg [1:0] {HIGH,MID,LOW} state_t;
state_t state, nxt_state;

logic capture_high,capture_mid,set_cmd_rdy,
      clr_cmd_rdy,clr_rdy;

////////// infer state flops ///////////////
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    state <= HIGH;
  else
    state <= nxt_state;

always_comb  begin
  ///// default outputs //////
  capture_high   = 0;
  capture_mid  = 0
  set_cmd_rdy  = 0;
  clr_cmd_rdy = 0
  clr_rdy  = 0;
  nxt_state = state;
```

```
  case (state)
    MID : if (rdy) begin
      capture_mid = 1;
      clr_rdy = 1;
      nxt_state = LOW;
    end else
      nxt_state = MID;
    LOW : if (rx_rdy) begin
      set_cmd_rdy = 1;
      clr_rdy = 1;
      nxt_state = HIGH;
    end  else
      nxt_state = LOW;
    ///// default case = HIGH /////
    default : if (rdy) begin
      capture_high = 1;
      clr_rdy = 1;
      clr_cmd_rdy = 1;
      nxt_state = MID;
    end
  endcase
end
```