ECE 4514 Fall 2019: Homework 2

Assignment posted on January 31
Assignment due on February 7 11:59 PM

Homework weight: 100 points

- Homework 2 is a simulation-based assignment on Finite State Machine Design. Refer to the following additional documentation: Lecture notes for Lecture 4 and the design example for Lecture 4 (bitxmit), the I2C standard specification, the DE1SoC schematics/ user manual, the WM 8732 datasheet.
- A grading rubric will be posted early next week.

The Audio Interface and Audio Codec on the DE1-SoC board

Your DE1SOC kit (which will be distributed next week, 2/4 – 2/9) contains an audio CODEC: the Wolfson WM8731. This is a chip that can sample analog waveforms from a microphone or a line input and that can produce a digital sample stream (in stereo, with a left channel and right channel). It can also convert a digital sample stream to audio for headphones.

A CODEC is the analog front-end for a digital processor. In this case, it's an audio CODEC that handles audio A/D and D/A conversion. The CODEC offers a selection of sample rates and precisions, including high-quality audio (24-bit 96KHz). Your mobile phone and your PC contain a similar chip, and it is attached to the MIC/LINE plugs of these devices.

The objective of the next few homework assignments (Homework 2 through 5) is to program this chip and do experiments with digital audio processing.

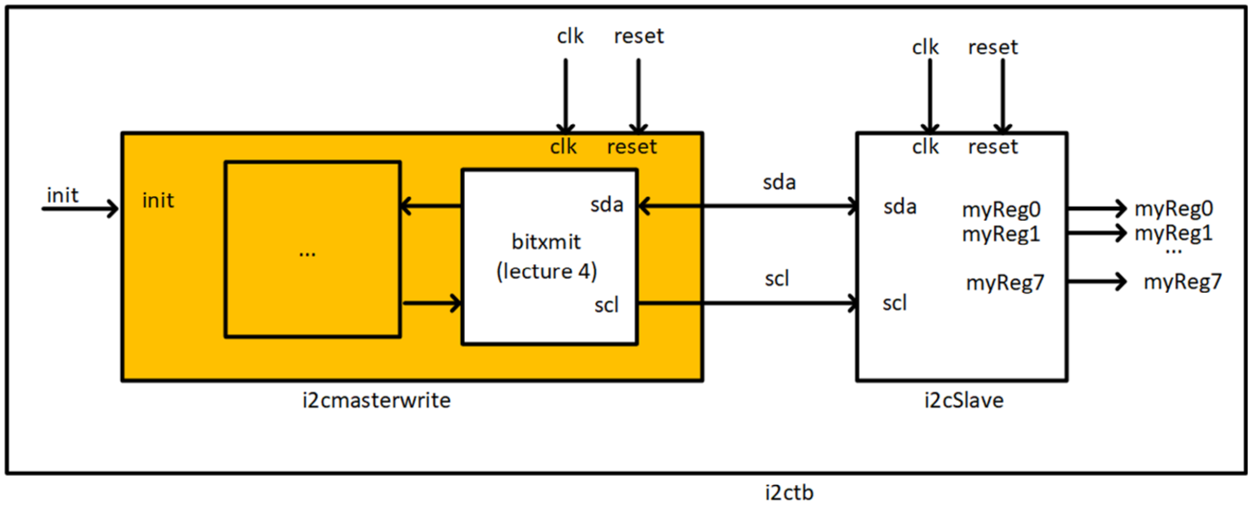
The interface between the FPGA and the audio codec consists of two different busses.

- First, an audio bus transports audio samples in a digital serial format. This audio bus is a real-time bus, meaning that it transports a left/right audio sample for every sample period.
- Second, an I2C bus steers the settings of the audio codec. The I2C bus enables control of settings such as the volume, sample rate and format, muting and audio routing. There are 11 control registers associated with the chip.

In homework 2, we will concentrate on the I2C bus. You will design a module which controls the audio codec. In this homework, you will design an I2C master interface according to the specifications below. You will have to simulate the I2C master interface in conjunction with an I2C slave interface, which you can use to verify the proper operation of your design. The operational details of the audio codec (such as the audio format or setting the sample rate) are not important at the moment: the present homework concentrates on the design of an I2C master. The following documentation is provided with the homework, which is useful as background material.

1. Philips I2C bus standard specification
2. Wolfson WM 8731 data sheet (page 45 discusses the I2C interface)
3. DE1-SoC schematic (page 18 shows the audio code chip and its integration onto the board).

The assignment

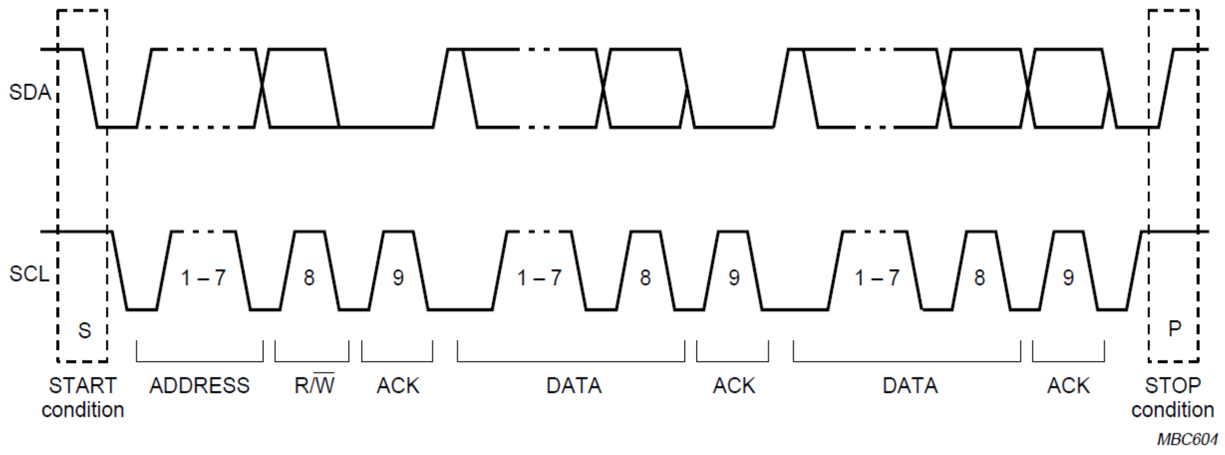


You are designing a module i2cmasterwrite, which will generate an I2C programming sequence upon receipt of an init pulse. I2cmasterwrite is a write-only master. You don't need to implement I2C read functionality.

We discussed the design of an I2C Master in lecture 4, and we also discussed that it's best to design this module as a hierarchical FSM, i.e., a chain of FSMs. You also received the code for the front-end FSM, bitxmit. You can find that in the example repository of lecture 4 (https://github.com/vt-ece4514-s19/bitxmit), and you are allowed to use any of those files to solve this homework.

Refer to the I2C standard for the programming sequence used by I2C. As a brief reminder, an i2c write sequence consists of the following.

1. The i2c master sends a START condition
2. The i2c master sends a 7-bit address, msb to lsb
3. The i2c master sends the R/Wbar bit (which will always be 0)
4. The i2c slave sends an ACK
5. The i2c master sends 8 databits
6. The i2c slave sends an ACK
7. Repeat from 5 if an additional byte needs to be sent
8. The i2c master sends a STOP condition.



The i2cslave module has 8 outputs, myReg0 to myReg7. To program these registers, the I2C slave will accept a control address, which selects the register (0 to 7) followed by a control data byte, which writes the data for that register. When the programming sequence is complete, the resulting value will appear at the output of the I2CSlave myReg outputs. The example with Lecture 4 (bitxmit testbench) illustrates such a sequence, programming the value AA in register 5.

The design needs to be made for an 50MHz system clock, and the I2C bus needs to operate at 100 Kbps.

The homework2 needs to demonstrate a synthesizable implementation for i2cmasterwrite, which generates the following programming sequence in response to an init pulse:

1. Set myReg05 to 0xAA
2. Set myReg01 to 0x12
3. Set myReg02 to 0x55
4. Set myReg00 to 0xDE

Bidirectional port access

The I2C data line is an open-collector line, which will appear as an inout signal in the Verilog code:

```
module i2cmasterwrite(input  clk,
                      input  reset,
                      input  init,
                      output sck,
                      inout  sda
                      );
```

Reading/writing to a bidirectional port in Verilog works as follows.

First, you create a control signal sdaread which will control the direction of the port (reading or writing). Next, you create a variable sdaout, which will contain the value that you would like to write to the port. The value which is assigned to sda, when the port is used for reading, is high-impedance (1’bz):

```
  reg sdaread;
  reg sdaout;
  assign sda = sdaread ? 1'bz : sdaout;
```

To read from sda, just read the sda signal in your code. When sdaread is 1, the signal will read back as the value driven into it from the I2C bus. To write into sda, set sdaread to 0. The signal sda will now reflect the value stored in sdaout.

What to turn in

You need to design the i2cmasterwrite module, making use of finite state machines. Even though the module will only be simulated, your code for i2cmaster must be synthesizable in Quartus.

Correct output (the timing does not have to be exact; only the proper values have to appear in the proper order).

```
Vsim -c i2ctb -do "run -all"
Reading C:/intelFPGA_lite/17.0/modelsim_ase/tcl/vsim/pref.tcl

# 10.5b

# vsim -c i2ctb -do "run -all"
# Start time: 16:05:16 on Jan 31,2019
# Loading work.i2ctb
# Loading work.i2cmasterwrite
# Loading work.i2cgenerator
# Loading work.bitxmit
# Loading work.i2cSlave
# Loading work.registerInterface
# Loading work.serialInterface
# run -all
# t        0 0xx 0xx 0xx 0xx 0xx 0xx 0xx 0xx
# t       25 000 000 000 000 000 000 000 000
# t   672225 000 000 000 000 000 0aa 000 000
# t  1401725 000 012 000 000 000 0aa 000 000
# t  2131225 000 012 055 000 000 0aa 000 000
# t  2860725 0de 012 055 000 000 0aa 000 000
# Break key hit
# Simulation stop requested.
VSIM 2>
```

Coding Guidelines

Please refer to Chapter "RTL Coding Guidelines " of the Reuse Methodology Manual (accessible for free on the Tech network: https://link.springer.com/book/10.1007%2Fb116360). These are generic guidelines for HDL coding; I expect you to aim to follow at least Chapter 5.2 and Chapter 5.5.

I value code clarity but I do not intend to penalize you if you don't follow these guidelines to the letter. In the other hand, if you turn in messy code with obvious violations against the principles and main directives in this guideline, you will receive a penalty of up to 25% of the points (even if everything works perfectly).