CPEN311 Winter 2020 Term 2
University of British Columbia

Lab 3: Add a strength meter to the simple iPod (Embedded Processors, Signal Processing/ Basic Filtering (Averaging))

In this laboratory, you will enhance your "simple iPod" of the previous lab in order to add an LED strength meter that will show the strength of the audio signal. You will do this using an embedded PicoBlaze processor. This will serve as an example of how embedded processors are used in practice and will close the gap between hardware and software, i.e. between this course and many other courses in your curriculum. This will also be an example of Digital Signal Processing using an embedded processor (we will do real-time averaging, which is a basic form of filtering). This lab represents the final step in our journey from transistors to software.

Start off from you solution to Lab 2, and from the PicoBlaze in-class activity. You will have to essentially merge these two. There is hence no template for this laboratory. You **MUST** use the Picoblaze processor in the manner described below, even though probably you could do this lab purely using logic, since the point of this lab is to practice the usage of embedded processors. You need to do the following:

Starting off from your "iPod" from Lab 2, instantiate a PicoBlaze processor to do the following things:

i. Your main program will toggle LEDR[0] every 1 second (trivial).

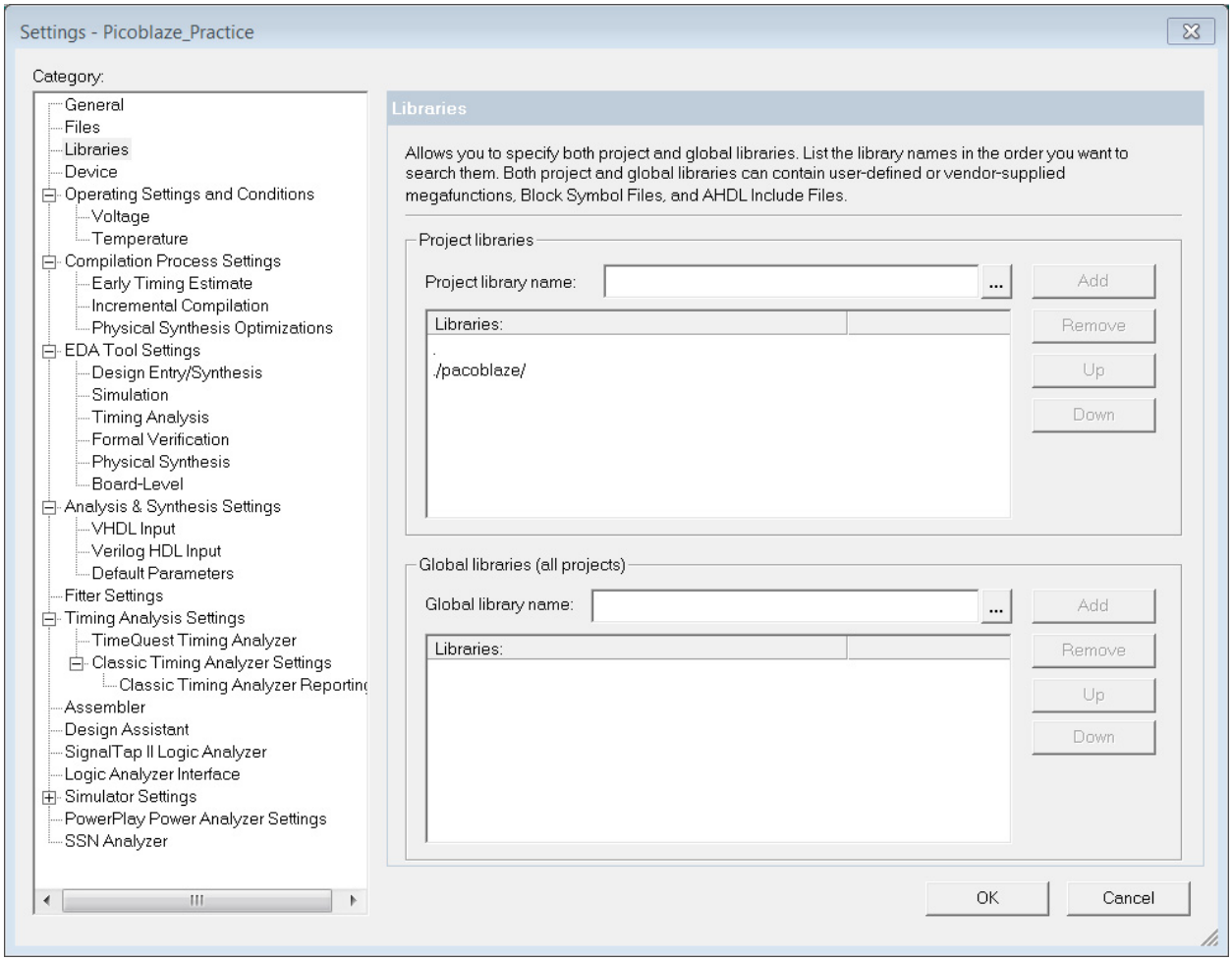
ii. The interrupt routine will be activated each time a new value is read from the Flash memory. Each value is a sound sample, each sample has its "intensity", or absolute value. The interrupt will accumulate (=sum) 256 of these absolute values (a value each time the interrupt is called), and the interrupt routine will divide this sum by 256 every 256-th interrupt. This is essentially an **averaging filter** operation, i.e. we are averaging every 256 absolute values of samples. Division by 256, because that is the power of 2, can be done very simply by discarding $\log_2(256)$ bits from the sum.

iii. Every time we do this division by 256, the PicoBlaze interrupt routine should output the average value to the LEDG[7:0]. (If you have a DE1-SoC, use LEDR[9:2]) Note that you have to "fill" the LEDs from left to

right, i.e. make the LEDs light up to the value of the most significant binary digit of the average. For example, if the average of the absolute values is, in binary, 00101101, then since the highest bit that is "1" is bit #5 (where bit #0 is the LSB), the LEDs should be XXXXXX00 (where "X" is on and "0" is off). As always, look at what the solution does if you have any doubts.

iv. After each averaged value is output to the appropriate LEDs, the accumulator is set to 0 to prepare to average the next 256 values, and so on.

Note: When merging the lab2 template with the in-class activity, do not add all the "pacoblaze" files to the project, rather make sure that the pacoblaze directory is in the project search path (see image below) and Quartus will automatically find the files. The reason for this is that the pacoblaze directory contains some testbench files that are not synthesizeable and will cause errors in Quartus if included in the project. Furthermore, make sure to merge the Picoblaze in-class activity into the Lab2 template, not the other way around. The Lab2 template contains pins and device assignments and compiler assignments that you must keep for Lab3 to work properly.



Specific grading requirements for this lab

In this lab, you do not need to show the TAs simulations. Functionality will be worth 50% for this lab, and simulations 0%. You should however, for your own benefit, simulate any modules that you write for this lab, as a matter of good practice. If you do not use the Picoblaze/Pacoblaze for this lab, you will receive 0% in the lab.

If in doubt about what to do, remember that you can always load the solution SOF and see what it does.

Do not forget the rules of good design:

- Always design while thinking about the **hardware** implementation
- Use **simple** structures
- **Incremental** design and test
- **Modular design**
- Use the **RTL viewer**
- Verification and test: Use **simulations**, LEDs, 7-segments, SignalTap, LCD scope (or SignalTap equivalent)
- Write **Clean**, **Neat**, and **Legible** code
- Give **meaningful** names to variables, use **comments**
- Go over and understand the warnings given by Quartus during compilation
- The circuit should be correct by **design**, not just by simulation.
- The code should be **verifiable** by **inspection**
- Design in a **modular** fashion, always thinking about future **reutilization** of the module.
- **Re-use** proven modules, instead of re-inventing the wheel all the time

Good luck and have fun!