**Journal**

**Exercise 3 HW/SW Co-design**

Date: **03/11-10**

Author: **Anders Hvidgaard Poder**

Stud.nr.: 19951439

# Introduction

In this journal I will answer the different questions posted in the text as well as my experiences with implementing an Architectural model in SystemC. However, before any of this can be done it is important to understand the design of the LAVMU model.

# LAVMU model breakdown (reverse engineering)

The first thing that springs to mind when looking at this design is that the IIRFilter is not in the data-path. Looking further into the code you notice that the Algo class implements the IIR Filter. Then what is the purpose of the IIRFilter class? We dive a little deeper into the PView class which uses the IIRFilter class to find out. It turns out that even though the IIRFilter class has the ability to filter samples, it is only used to generate the IIRFilter coefficients. Therefore the name IIRFilter is quite poorly chosen, and the name IIRFilterFactory would have been better, but as we only use the coefficients and not the filter an even better name would be IIRFilterCoefficentsFactory. Also the IIRFilter is purely composite under PView, so the class diagram could beneficially be updated to reflect this

The Top class creates the binding between the Algo and the PView, and also between the Algo and the Source and Sink Codec. This allows the data sent by the SourceCodec to be received by Algo and the data sent by Algo to be received by the SingCodec. It also allows the PView to transfer its coefficients to the Algo class. All this communication is done using abstract busses.

The Algo class accepts values from the SourceCodec and filter them according to the IIR filter implemented in the class with the coefficients received from the PView (who got them from the IIRFilter). The result of the filtering is then sent to the SinkCodec. The filter implementation in the Algo class does not include timing. This means that a filtration of a sample takes 0 time and can be done in one clock. Naturally it would be possible to determine the required computation clock cycles for the filter and thereby make the filter up to CCAM accurate, but this has not been done and the Algo implementation (model) is therefore untimed TLM. The Algo class has two clocks, the AudioClock, which is use by the SourceCodec and SinkCodec (12MHz) and a main clock (50MHz), which it shares with the PView. The Algo class has its sensitivity set to the AudioClock, and does as far as I can see not use the main clock at all, which is a little odd, as the filtration algorithm would be expected to run on the main clock and not the AudioClock. It would naturally only run when new data was available, but the instructions might as well have the granularity of the main clock, if timing was to be introduced, but as timing is not, it really do not matter.

Communication with the Source- and SinkCodec is done via channels. Here it may be seen that the SourceCodec uses an internal AudioClock of 12MHz, which it divides down by 125 to reach the 96kHz sampling frequency. Its main loop is executed with audio clock frequency (12MHz) and for every 125 loops a sample is read from the file and written to the Algo class channel. Furthermore synchronization signals are set to indicate to the Algo class that a value is now ready. In a real life example there is naturally no file and the sample is taken from an A/D Converter. As no extra wait is executed it is simulated that the sampling of the A/D and the writing to the Algo filter takes 0 time units, but the sampling is Cycle accurate, as it is done on a divided clock, just like the actual implementation would be. As the Algo filter has no computation delay it would also make very little sense to include the delay, as it is negligible compared to the filter computation. However if correct waits were added to Algo to make it computation cycle accurate, then it would be important to include the sampling and communication delay to see the total Source to Sink delay.

# Assignment 3.12.1

TLM is a system level model and is in SystemC a matter of augmenting the MoC (in SystemC) with timing constraints. The TLM may be made timed. This may be done by augmenting the code with wait statements for the different blocks. At the lowest level of implementation it should be for every Basic Block, but it may be done by simply estimating how much an entire transaction will take and then place the wait after that. It is possible to extend the TLM with more detail, like “implementing” the communication busses in SystemC and then augmenting this implementation with timing.

CAM is a model that is accurate down to the individual logic level changes, and is therefore accurate to the lowest timing level. A CAM can be simulated and be fully accurate compared to the actual execution an FPGA or ASIC.

In the SystemC class diagram the IIR algorithm is implemented in the Algo class and that no consideration is given to execution time of the algorithm, making it an untimed TLM. The interface to the IIR algorithm is implemented in the Codec Source and Codec Sink. Here it may be seen that the sampling of the input signal, and the reception of output from the Algo class is clock accurate, meaning that it is done on the AudioClock, sampled down to 96kHz, making it BCAM accurate on the A/D converter sampling. The CodecSink awaits data from the Alog at the correct Audio clock ticks, making its communication BCAM accurate. As Algo takes no time to process the data then the CodecSource will deliver the data on an Audio clock tick. The Algo will process it on the next master clock tick deliver it to the Sink in the same master clock tick. Depending on how the the actual SystemC task simulation of the code inside the loop is done, the Sink may receive the data in the same Audio clock cycle as the CodecSource sent it in, or the next one.

Later in the design process the Algo implementation might be augmented with timing, first perhaps just from estimation or from testing the algorithm on the actual HW and then adding this number, and even later a CCAM version may be created making the Algo model fully accurate. Further adding delays for communication channels would make the CodecSource to CodecSink part CAM accurate.

The PView module is an abstraction of the HW/SW boundary. It is used to communicate from the SW to the HW (setting the coefficients and gain) and form the HW to the SW (getting the peak value). It is the API for the filter.

# Assignment 3.12.2

After reading the assignment I decided that since it called for a replacement of the IIR filter the easiest approach was to simply duplicate the LAVMU\_Model project and then work from that.

However first we need to complete the design. The design is created by using the design from Assignment 3.12.1 and modifying it accordingly:

1. We realize that the LMS filter does not require coefficients, so we remove these from the PView.
2. We add setting the convergence factor to the PView and create an appropriate channel. No requirement that the length can be changed exists so we leave it hard coded.
3. We update Algo with the LMS filter algorithm, remove the coefficients and add convergence factor.
4. We update Algo to only deliver 1 channel as output
5. We update Top to make the correct mapping between PView and Algo and Algo and CodecSink.
6. We update CodecSource to read in the correct files and do a little renaming of channels.

The LAVMU\_Model already reads it input from two files, representing most likely the left and right channels of the stereo signal respectively. The LMS filter takes a noise and noise+signal input, but as there is no requirement that the filtered signal should be a stereo signal, so we can simply assume two mono microphones. Alternatively we would need two CodecSources and two LMS filters and then a mixer, but that is not what we are going to do ☺

I also decided to remove the reset code and simply have the Algo code automatically reset when the convergence factor changes.

As there are no extern “C”in the LMSFilter.h the methods are name mangled to allow c++ overloading, so I had to add this. Alternatively we could have changed to LMSFilter.c to LMSFilter.cpp, but that is not a nice solution. As the LMSFilter is only included once we decided to encase the include statement in extern “C”, thereby keeping the LMSFilter files intact.

When working on the CodecSource I discovered that the channels right and left, which I expected to simply change to Signal and Signal+Noise are processind in separate Audio clock cycles. This means that the right and left channels are not processed with 96 kHz each, but rather that they alternate with 96 kHz. They are in other words 1/96000 of a second of from each other. This is not a very nice way to sample a stereo signal, but then nowhere does it say that this is a stereo signal. For the LMS algorithm we want the samples to be taken “at the same time”. Naturally this is an impossible, but we should certainly strive to take them as close to each other as possible. This might be in two separate Audio clock ticks, giving a gap of 1/12000000 of a second between samples. I will however assume that we can sample the ADC’s “simultaneous”, perhaps by simply changing a logic level on an input of the ADC indicating it should take a sample, allowing the sample to be read from the ADCs at different times, without loosing the “same time”. This logic level change can be done in at most 2 main clock cycles, i.e. with a gap of 1/50000000 of a second between samples. In the implementation I will simply add a second audio channel and write to both channels at 96 kHz.

We keep the AudioSync to allow the Algo loop to run faster than the sampling, but it no longer switch between channels; it is only used to indicate that a new sample is ready.

The CodecSinc is updated to simply write the AudioOut to the file and not switch between left and right channel.

Putting it all together gives us the design below:



Naturally to do this design we should draw a lot more diagrams. A Block diagram and a few inner block diagrams would be an absolute minimum to show ownership, dependencies and communication channels. Furthermore a sequence/activity diagram or FSM to show the sampling and finally naturally a deployment diagram to show the HW/SW layout. All of these things will however be left as an exercise for the reader ☺

For the code please refer to the subversion repository at:

# Conclusion

The exercise has given me a good understanding of the SystemC API, and simulating channels and HW/SW interfaces. There is one aspect of interest and that is how the AudioSync is set and reset. Assuming a standard multi-threaded application checking a message queue and/or semaphore for data it is not possible (or at least extremely complicated) to know on what tick the scheduler is going to assign execution to the given thread. In this example where we have a CodecSource “thread” supplying data at 96 kHz on a 12MHz clock to an algorithm “thread” receiving data on the same 12MHz clock (slightly odd, as mentioned earlier), and finally sending the data to a CodecSink thread receiving data on the same 12MHz clock as the others. As the algorithm does all the processing in 0 time technically the whole thing could be over on the same tick, but it depends on how the scheduling is, as is shown on the below diagram. It is very likely that SystemC has a deterministic answer (I have not fully read the SystemC book yet), and one can certainly design a solution that do not suffer from this question, but it is one question that I was left with after this exercise.

