IL2450 - System Level Validation

Lab 1

Testing An FIR Filter

V. 3.0 – March 5, 2010

Purpose of the lab

In this lab you will write a verification plan and different testbenches to test the functionality of a provided FIR filter. An optional part of the lab consists in designing, dimensioning and testing your own FIR filter system.

Part 1 – Theory on FIR filters

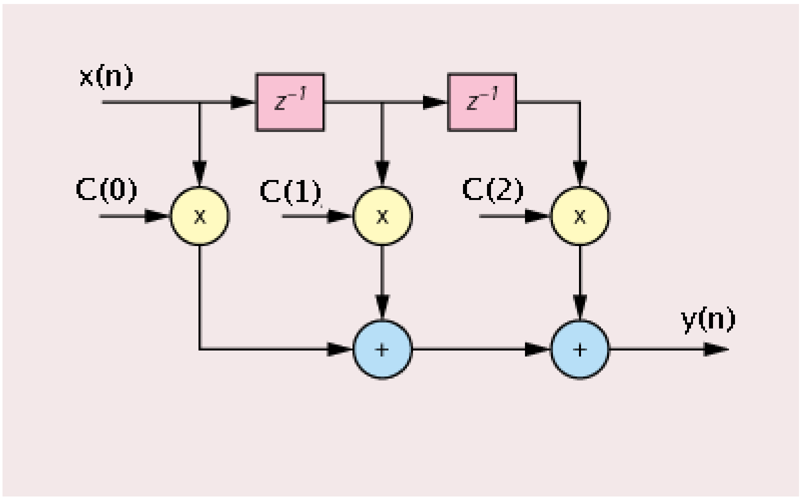
This part of the lab contains an introduction to FIR filter theory. Be sure to go through it carefully as understanding how FIR filters work is very important to be able to complete correctly the lab.

FIR filter stands for Finite Impulse Response filter. FIR filters are digital filters working on sampled data. They provide one output sample per input sample we provide it with. The output of an FIR filter at each cycle is determined only by a number N of input values, N-1 past input values and the current value. We refer to N as the number of taps of the filter or as the order of the filter. At the beginning of the operation of the filter, past values are usually assumed to be zero. FIRs are called Finite Impulse Response filters because, if they receive an impulse as input, the output will be different than zero only for a number of cycles equal to the order of the filter. This is in contrast with what happens with IIR filters, for which the output value in each cycle depends on all the input values of the filter in past cycles.

To calculate the output of the filter in the cycle k, the filter uses the following formula:

$$y_k = \sum_{i=0}^{i=N-1} C_i \times x_{k-i} = C_0 \times x_{k-0} + C_1 \times x_{k-1} + \ldots + C_N \times x_{k-N}$$

The structure of a third order FIR filter is shown in the following picture:



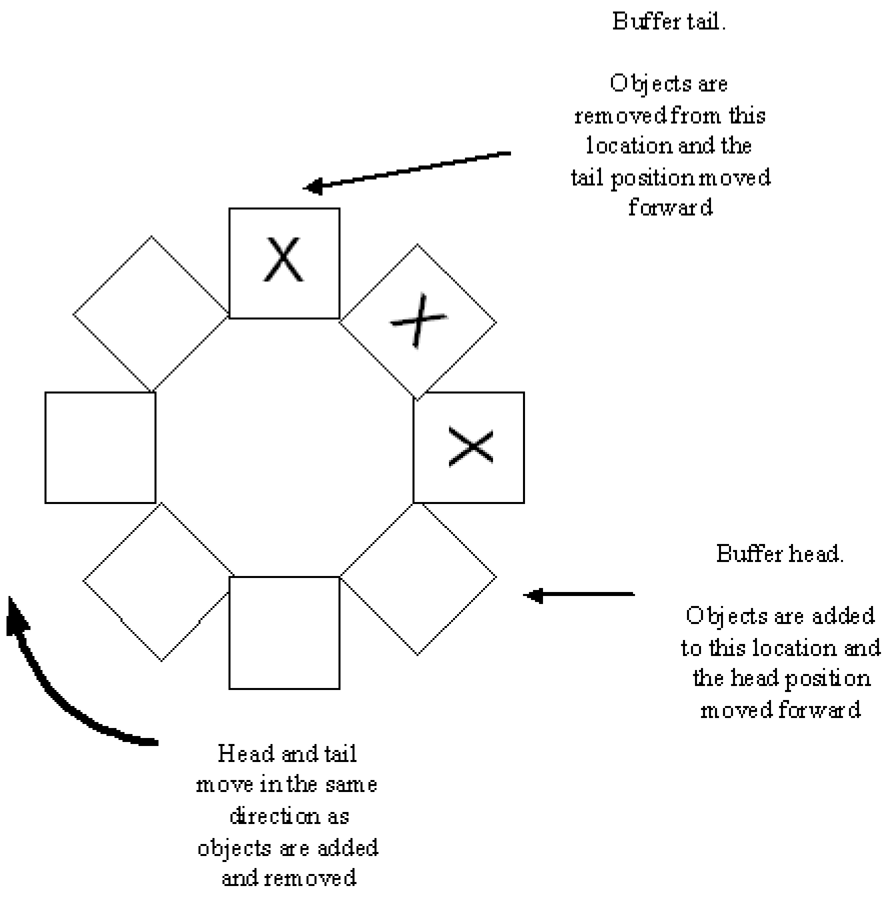
The Z^{-1} blocks are delay blocks, acting as synchronous registers and delaying the signal of one cycle.

FIR filters can be implemented in different ways. In order to calculate one output value of the FIR, it is necessary for the FIR filter to store the N-1 last input values. The simplest and most intuitive solution is to use a shift register. Shift registers allow all the samples to be available at the same time, and if a sufficiently big adder chain and enough multipliers are available, it is possible to do the whole calculation in only one cycle. Another solution consists in implementing a circular buffer. A circular buffer can be implemented in a memory by using pointers. Usually the memory provides only one or two data per cycle, and we need therefore to pipeline the process of calculating one output value. One single multiply/accumulate unit (MAC) is enough to perform the calculation of the output in N iterations.

In general, circular buffers are implemented with a certain number of memory locations logically organized in a circular structure with a favourite rotation sense. In memory, the locations are organized as a block of contiguous memory locations going from a starting block to and ending block. Logically, the locations are seen by the entities accessing them as a circular structure.

If the number of elements that the buffer can contain is variable, then the buffer is implemented with two different pointers, one that always points to the first free memory slot and the other that always points to the last full position. When an object must be added to the buffer, it is added at the location pointed by the buffer head pointer. The pointer is then incremented to point to the next location in the logic circular structure of the buffer. When an object must be removed form the buffer, it is read from the location pointed by the buffer tail pointer, which is then incremented to point to the next position. Incrementing a pointer consists in incrementing it and in resetting it to first position of the block if the end of the allocated memory block is reached.

See the picture for a schematic example of an 8-slots circular buffer containing three objects:


Buffer tail.
Objects are removed from this location and the tail position moved forward
X
X
X
Buffer head.
Objects are added to this location and the head position moved forward
Head and tail move in the same direction as objects are added and removed

Two possible errors can be generated by a circular buffer: buffer overflow happens when the buffer head pointer overtakes the buffer tail pointer. A buffer overflow means that there are no free locations in the buffer and no data can be stored. A buffer underrun happens when the buffer tail pointer overtakes the buffer head pointer: in this case the buffer is empty and no data can be read.

The structure of a circular buffer for a FIR filter can be simplified thanks to the fact that the number of data stored in the buffer is previously known and is fixed. Therefore, the buffer can be dimensioned exactly of the needed dimension to hold the necessary data, without supplementary free space allocated, the buffer tail pointer always coinciding with the buffer head pointer. Every time a new sample arrives, the oldest sample is automatically discarded. Buffer underrun and overflow never occur.

Part 2 – Description of the DUV

This part of the lab describes the DUV. Only by having a good understanding of the system it is possible to go through the lab. You must think of this part of the lab manual as the specifications that the designers gave you. Just as in real life verification, the focus of these specifications will mostly be on the interface of the block and its externally-observable behaviour. The block will mostly be seen as a black box. What you will have to check is if the Verilog model of the DUV satisfies the specifications and acts as it should.

1 – General system

The FIR filter block is implemented as the connection of three different blocks: the ROM, which stores the coefficients of the filter, the RAM, used for storing the incoming stream of data, and the FIR block itself, which contains the MAC unit and the control logic. The system has been optimized to take advantage of the fact that the most efficient FIR filters are symmetric, meaning that the coefficients C_i and C_{N-1-i} are the same. Optimization can be made to take advantage of this point since it is possible to use the following optimization when N is even:

$$y_k = \sum_{i=0}^{i=N/2} C_i \times (x_{k-i} + x_{k-(N-1)+i})$$

$$y_k = C_0 \times (x_{k-0} + x_{k-(N-1)}) + C_1 \times (x_{k-1} + x_{k-(N-1)+1}) + \ldots + C_{N/2} \times (x_{k-N/2+1} + x_{k-N/2})$$

A similar optimization exists when N is odd:

$$y_k = \sum_{i=0}^{i=(N-1)/2} C_i \times (x_{k-i} + x_{k-(N-1)+i}) + C_{(N+1)/2} \times x_{k-(N-1)/2}$$

$$y_k = C_0 \times (x_{k-0} + x_{k-(N-1)}) + C_1 \times (x_{k-1} + x_{k-(N-1)+1}) + \ldots + C_{(N-1)/2} \times x_{k-(N-1)/2+1}$$

The optimization leads to a reduction in the number of multiplications by a factor of two. When only one MAC unit is available, the number of cycles needed to perform the complete calculation of a value is reduced by half. Also, the number of coefficients needed is reduced by half, and so is the number of values that must be stored inside the coefficient lookup table.

To take advantage of the optimization, however, the filter needs to access in each cycle two different input values. For this reason, a dual ported RAM must be used.

The DUV of this lab session is using a dual ported RAM and is optimized to take advantage of the parallelism of the coefficients' table.

The block is synchronous to the system clock and gets input values from the input port *Sample*, when there is a rising edge of the signal *SampleClock*. The block then starts to perform its elaboration and, when it is done, outputs the result on the port *sum* and rises the line *Dav* to signal that the elaboration is concluded and a new output sample has just been sent to the output line.

2 – FIR block parameters and interface

The FIR block is configured by different parameters, as reported in the following table:

Parameter Name	**Function**	**Legal Range**
NrOfTaps	Number of taps	N (natural)
SampleWidth	Width of the sample input	M (natural)
CoeffWidth	Width of coefficients coming from the ROM	K (natural)
SumWidth	Width of internal sum of FIR filter	M + K + floor($\log_2$(N))
TruncatedMSBs	Number of MSBs discarded from the internal sum port.	SumWidth > TruncatedMSBs + TruncatedLSBs
TruncatedLSBs	Number of LSBs discarded from the internal sum port.	SumWidth > TruncatedMSBs + TruncatedLSBs
AddrsWidth	Width of RAM address	$\log_2$(NrOfWords)
CoeffAddrsWidth	Width of ROM address	$\log_2$(NrOfWords)

The FIR block has these input/output ports:

Port Name	**Type**	**Direction**	**Description**
ResetN	Bit	INPUT	Master reset
clk	Bit	INPUT	System clock
sampleClk	Bit	INPUT	FIR captures input on rising edge of SampleClk
sample	Signed[SampleWidth]	INPUT	Input sample dataline
sum	Signed[SumWidth-TruncatedMSBs-TruncatedLSBs]	OUTPUT	Output sample dataline
dav	Bit	OUTPUT	Signal rises when the output is ready
rwN1	Bit	OUTPUT	RAM read/write signal 1 (read when 1, write when 0)
rwN2	Bit	OUTPUT	RAM read/write signal 2 (read when 1, write when 0)
sampleaddrs1	Bit[RamAddrsWidth]	OUTPUT	RAM address line 1
sampleaddrs2	Bit[RamAddrsWidth]	OUTPUT	RAM address line 2
dataIn	Signed[SampleWidth]	OUTPUT	RAM write dataline
dataOut1	Signed[SampleWidth]	INPUT	RAM read dataline 1
dataOut2	Signed[SampleWidth]	INPUT	RAM read dataline 2

Port Name	Type	Direction	Description
read	Bit	OUTPUT	ROM read signal
coeffaddrs	Bit[CoeffAddrsWidth]	OUTPUT	ROM address line
coeff	Signed[CoeffWidth]	INPUT	ROM dataline

3 – RAM block parameters and interface

The RAM block describes a synchronous dual-port RAM. It has two address buses, two data buses, and two read/write signals. When a read action occurs, the RAM outputs the data on the rising edge of the clock; when a write action occurs, the input data is stored in the memory array on the rising edge of the clock.

The RAM block is configured by different parameters, as reported in the following table:

Parameter Name	Function	Legal Range
NrOfWords	Number of words stored in the RAM	
WordSize	Number of bits per word stored in the RAM	
AddrsSize	Number of address bits of the RAM	$\log_2$(NrOfWords) rounded up

The RAM block has these input/output ports:

Port Name	Type	Direction	Description
resetN	Bit	INPUT	Master Reset
clk	Bit	INPUT	System Clock
rwN1	Bit	INPUT	Read/Write signal 1
rwN2	Bit	INPUT	Read/Write signal 2
addrs1	Bit[RamAddrsWidth]	INPUT	Address signal 1 for RAM
addrs2	Bit[RamAddrsWidth]	INPUT	Address signal 2 for RAM
dataIn1	Signed[SampleWidth]	INPUT	Write dataline 1 for RAM
dataIn2	Signed[SampleWidth]	INPUT	Write dataline 2 for RAM
dataOut1	Signed[SampleWidth]	OUTPUT	Read dataline 1 for RAM
dataOut2	Signed[SampleWidth]	OUTPUT	Read dataline 2 for RAM

4 – ROM block parameters and interface

The ROM block describes a synchronous single-port ROM.

The ROM block is configured by different parameters, as reported in the following table:

Parameter Name	**Function**	**Legal Range**
NrOfWords	Number of words stored in the ROM	
WordSize	Number of bits per word stored in the ROM	
AddrsSize	Number of address bits of the ROM	$\log_2$(NrOfWords) rounded up
coeffs	Array of initial values to be stored in the ROM. The type of the value is bit[0:NrOfWords-1][WordSize-1:0]	

The ROM block has these input/output ports:

Port Name	Type	Direction	Description
resetN	Bit	INPUT	Master Reset
clk	Bit	INPUT	System Clock
read	Bit	INPUT	Read signal for the ROM
addrs	Bit[AddrsSize]	INPUT	Address line
dataOut	Signed[WordSize]	OUTPUT	Output dataline

5 – FIR system interface

The complete block has these following input/output ports:

Port Name	Type	Direction	Description
resetN	Bit	INPUT	Master reset
clk	Bit	INPUT	System clock
sampleClk	Bit	INPUT	FIR captures input on rising edge of SampleClk
sample	Signed[SampleWidth]	INPUT	Input sample dataline
sum	Signed[FirWidth]	OUTPUT	Output sample dataline
dav	Bit	OUTPUT	Signal rises when output is ready

6 – Rounding, internal and output widths

As the tables show, the FIR block contains parameters to set the width of the internal and of the external sum lines. The block performs operations on its inputs by using signal widths determined by the parameter *SumWidth*. *SumWidth* must be big enough to avoid clipping during the computation. To keep the system simple, the size of the complete MAC unit is equal to *SumWidth*, and no roundings are performed until the complete calculation of the output sample has been performed and the filtered sample is ready to be outputted. A window of bits is then selected in the internal sum signal by discarding a certain number of MSBs and of LSBs, performing rounding and saturation if needed.

Once the final sum result has been calculated, the result is clipped by discarding a certain number of MSBs as specified by the parameter *TruncatedMSBs* and a certain number of LSBs as specified by the parameter *TruncatedLSBs*.

The MSBs are discarded if both these conditions hold:

- they are all zeros or all ones
- the value of the bit following the least significant of the clipped bits is equal to that of the

bits preceding it.

If at least one of the conditions does not hold, the result is saturated to the maximum positive or the maximum negative value. We refer to this condition as positive or negative overflow: given the number of MSBs we chose to discard, the result is too big in magnitude to be represented and saturation has to be performed.

A certain number of LSBs, as specified by the parameter *TruncatedLSBs*, is discarded from the internal sum signal. To round to the nearest integer, if the MSB of the discarded LSBs is 1, then the result is rounded up, by adding one to the truncated signal. If the MSB of the discarded LSBs is 0, the result is rounded down, nothing is added to the truncated signal.

7 – Connection of the FIR block with the memories

The ROM memory is used only to store the coefficients. Its length must be enough to hold the entire table of the coefficients, while the width of its lines must be enough to hold the coefficients with sufficient precision. The port *coeff* must be connected to the dataline of the ROM, the port *coeffAddrs* to the ROM address line and the port read to the read signal of the ROM.

The RAM memory is used to hold the sample inputs. The ports *rwN1*, *sampleAddrs1*, *dataIn* and *dataOut1* must be connected to one of the two lines of the RAM, respectively to the ports *rwNx*, *addrsx*, *dataInx* and *dataOutx*. These ports are used both to read the samples from the RAM and to write a new sample to the RAM when a new input sample must be stored.

The ports *rwN2*, *sampleAddrs2* and *dataOut2* must be connected to the other line of the RAM, respectively to the ports *rwNy*, *addrsy*, *dataOuty*. These ports are used to read samples from the RAM.

Before starting with the tasks: setting up the environment and generating the golden model.

The DUV is composed of three Verilog modules, each contained in a different file:

- **rom.v** contains a behavioral model describing the ROM
- **ram.v** contains a simple behavioral model describing the RAM
- **fir.v** contains a model of a FIR block containing the MAC and the control logic necessary for the FIR filter itself to work.

Create a directory in your home folder to hold the files necessary for this lab and download the three files **fir.v**, **ram.v** and **rom.v** from the lab 1 page of the course website. Use the same directory for all of the tasks in this lab. Each task will contain instructions on what you will have to do in the different sections and what will be the deliverables.

You will test the testbench by using it to filter an audio sample file, which is provided by the lab assistants. Download the file **sample.wav** from the lab 1 page of the course website and put it in the directory in which you are working. It is a 3 seconds audio clip sampled at 8 bits at the rate of 11025 samples/sec. A low-quality clip has been chosen so that the simulation of the testbench in the will be short.

In industry, Matlab is used extensively to create reference "golden" models for testbenches. In this lab you are going to use Matlab to provide a reference output for our DUV. Matlab will also be used to create an input file for the testbench, in a format it can easily understand, by interpreting and translating a wave audio clip. Instead of installing Matlab, you can install an opensource version of matlab, called Octave, from your Ubuntu system, by typing:

```
sudo apt-get install octave
```

You should now be able to launch Octave by typing the command

```
octave
```

at your prompt.

To begin with, you will use Octave to read the file **sample.wav** and output its samples to an hex file.

```
[y,fs,nbits,opts]=wavread('sample.wav');
```

This command will read the wave file, save the samples to the vector y, the sample frequency to the variable *fs*, the number of sample bits to the variable *nbits*, and various information on the file to the variable *opts*. Check the sample frequency and the number of bits before proceeding.

Now, open a file for writing:

```
fwsample=fopen('sample.hex','wt');
```

Then, write to it the data in hexadecimal format:

```
fprintf(fwsample,'%x\n',(y*(2^7))+(y<0)*(2^8));
```

The file **sample.hex** will contain the samples of the wave files in hexadecimal format, one sample per line.

Now enter in a vector the coefficients used by the default FIR filter:

```
a=(2^-11)*[8,-25,-84,235,890,890,235,-84,-25,8];
```

Filter the samples of the wave file by issuing the command:

```
yf=filter(a,1,y);
```

The filtered samples will be floating point values. To map them on 8 bits, issue:

```
yf=round(yf*2^7);
```

The filtered samples will now be written to the file **sampleoutgolden.hex**, which will be used as a golden reference model and read by your testbench. To write the file, type the commands:

```
fwsampleoutgolden=fopen('sampleoutgolden.hex','wt');
fprintf(fwsampleoutgolden,'%x\n', yf+(yf<0)*(2^8));
```

We will also output a wave file for an "audio" testing of the filter. To create an audio file with the filtered samples, type:

```
wavwrite(yf/(2^7),fs,nbits,'sampleoutgolden.wav');
```

For the moment, you can close Octave. Check the file **sample.hex** to understand its structure.

Task 1 – Build a testbench for the system and generate the impulse response (15 pts)

For the second task you are asked to write a SystemVerilog testbench to test the system. The DUV needs to be assembled by connecting the FIR block to the two memories in the correct way. Use default values for the parameters of the filter, but set TruncatedMSBs and TruncatedLSBs to zero.

Use one global module connecting the three blocks. The module should also include a SystemVerilog program that you will use as the testbench. The program you create has to be fully parametrized, so that it would possible to reuse it for filtering systems with different parameters (line widths, number of taps...).

Your program must generate an impulse for the system. It must drive correctly the clock signal for the whole simulation; it must reset the system when the simulation is launched, then wait as many clock cycles as are necessary for the initialization of the RAM to complete, and then drive the *sample* and the *sampleClk* signals to introduce an impulse in the system. The impulse is obtained by outputting as the first sample the value 1. Then, the testbench must wait until the signal *dav* becomes high, which means that an output sample is ready, and print the sample value to the console. The testbench must then feed new null samples (samples of value zero) to the DUV, for a number of times equal to the number of taps of the filter. The output samples obtained from the DUV must be outputted to the console. Finally, the testbench must conclude the simulation after the last output sample has been read and printed to the console.

Task 2 – Check if the FIR output is correct (15 pts)

This task asks you to write a SystemVerilog testbench which gets its input from the Matlab-generated file **sample.hex**, feeds it to the FIR filter and outputs the results of the elaboration to a file called **sampleout.hex,** in hexadecimal format, one sample per line. This time use default parameters for all the parameters of the three blocks. Do not drive any parameter, not even TruncatedMSBs and TruncatedLSBs. The filenames and the number of samples in the files should be given as parameters for the testbench, and you should use the function **readmemh** to read the input files and store them into static vectors. The testbench must correctly drive the clock and reset signals for the DUV and, after the first sample has been elaborated, must react to the signal *dav* to know when a sample is ready, so that it can write it to the output file. New samples should be fed with a period long enough to allow the filter to complete its computations before the arrival of a new sample. The testbench must terminate the simulation when all the samples in the input file have been elaborated and the last valid sample outputted by the FIR system has been recorded on the output file.

It is also required that the testbench read the expected outputs from the Matlab-generated golden reference hex file **sampleoutgolden.hex**, and print an error message to the console whenever a discrepancy is found between the expected value and the one that was outputted by the FIR filter system. The message must contain information about the simulation time at which the error has been encountered and contain both the expected value and the value that has been obtained from the DUV.

As for the second task, you will need to deliver a SytemVerilog file containing a module enclosing the system and a testbench program.

Run the simulation, see if there are discrepancies between the golden model and the observed behavior of the FIR system. Why do you think there are differences between the results produced by Octave and the ones produced by the DUV? Read the documentation of the Matlab command **round** and try to run the command

```
round(-0.5)
```

to understand how rounding is done in Octave before answering. Do you think these discrepancies are potentially harmful? Should something be done about them? What could be done to solve this discrepancy?

Task 3 – Check timing (15 pts)

For this task you will create a SystemVerilog testbench to check the correct timing of the system and the addressing of the memories. To do so, you will create a new testbench by modifying the impulse response testbench you created for task one. You will create a new SystemVerilog file containing a testbench. The testbench will still output an impulse to the system but the first input value must be 7f (the maximum possible input value) instead of 01. The simulation should terminate once every sample from the scaled impulse response has been outputted. The testbench must use default values for all the parameters, even for *TruncatedMSBs* and *TrucatedLSBs*. However, this time the system does not need to print the output samples to the console. Instead, logic must be added to print an error message to the console each time there is a violation of one of the following timing properties:

- The signal *dav* is never high for two consecutive clock cycles.
- The signal *sum* changes only on the clock edges when the signal *dav* toggles from low to high.
- The coefficients in the ROM are accessed in the correct order: when the signal *SampleClk* is asserted, the value of the signal *CoeffAddrs* must be zero. Then, at each clock cycle, this

value must be incremented by one and finally reach zero one clock cycle before the signal *dav* is asserted.

- The number of clock cycles between the assertion of *SampleClk* and the assertion of *dav* is equal to the number of taps of the filter divided by two, rounded up for filters with odd number of taps.

As you have hopefully noticed, to spicy up your lab and to make it more fun, an error has been introduced artificially in the FIR filter and one of the above specifications should not be met. Identify it, then look at the code of the FIR filter block, find the location of the error and correct it. Correct the FIR filter file. If you decide to continue with the lab, use the corrected version of the FIR filter file.

Task 4 – Check if the internal arithmetic of the filter is correctly dimensioned (optional, 10 pts)

In this task you will check if the internal width of the MAC unit is large enough to allow the filter to process the input samples without clipping even in the worst case scenario. You will create two different testbenches. The first uses the same default parameters as the filters used in the previous task, but which sets the values for the filter coefficients to the maximum positive value given the number of bits on which they are codified. The testbench will provide as input samples a sequence of 10 worst-case samples, equal to the maximum positive value. Your testbench should monitor the signal *intSum* in the FIR block. This signal corresponds to the output value before the final clipping is performed. The testbench should output to the console the tenth sample produced, and you should check if the value is correct and if it corresponds to the expected value. You should then create a complementary testbench which uses as coefficients and as input samples the maximum negative values instead of the maximum positive.

Task 5 – Build your own filter and dimension it (optional, 20 pts)

For this task you will design and dimension your own low-pass FIR filter, by using Matlab. You can freely choose different parameters but it is required to use an odd number of taps for the filter. The filter should have an eight bits output. You can freely set these parameters:

- Number of Taps of the filter,
- Cut Frequency
- Number of bits in the coefficients

To generate in Matlab the coefficients for a low-pass FIR filter with a given cut frequency you can use the command:

```
a=fir1(NrOfTaps-1,CutFrequency/(SampleFrequency/2));
```

Notice that the first parameter of the **fir1** function is the number of taps of the filter diminished by one, and the second one is the cut frequency of the filter normalized by dividing it by half the sample frequency of the filter.

After running the command, the coefficients of the filter will be stored in the vector *a*.

To complete this task you are required to create a new SystemVerilog testbench for your filter which outputs a line on the simulator console each time a positive or negative overflow condition is encountered. Use the usual **sample.hex** input file to provide input samples to your testbench. Your testbench has to instantiate the different blocks composing the FIR system with the parameters relative to your own filter.

The testbench must probe the internal signals of the DUV to check for overflow condition since the condition is not detectable by only accessing the external interface of the filter.

Task 6 – Simulate and correct the DUV (optional, 15 pts)

As you did in the preparation tasks for this lab, use Matlab to generate a vector of golden expected values for the output of the filter you designed in task five. Remember to take into account the position of the bit window that you selected in the previous task (moving it introduces a scaling factor in the filter). Modify the testbench you wrote in the third task to instantiate the FIR system with the parameters corresponding to your own filter.

As for task two, the testbench must write the filtered samples to a file and must output a message to the console each time there is a discrepancy between the expected and the observed outputs of the filter.

Is there something wrong this time? Find out what the error is in the DUV and correct it.

Final task (just for the fun of it)

For your personal amusement, you will now convert the **sampleout.hex** and **sampleout2.hex** files to wave files to listen to them. This task has no real purpose and you won't need to submit any deliverables, but the assistants think that after all the hard work in the lab you need some personal gratification by seeing that the DUV actually works and by listening with your own hears to the low pass effect introduced by the FIR filter. To do that, open Octave and type the commands:

```
frsampleout=fopen('sampleout.hex','r');
yf=fscanf(frsampleout,'%x');
yf=(yf-(2^8)*(yf>2^7-1))*(2^-7);
wavwrite(yf,11025,8,'sampleout.wav');
frsampleout2=fopen('sampleout2.hex','r');
yf=fscanf(frsampleout2,'%x');
yf=(yf-(2^8)*(yf>2^7-1))*(2^-7);
wavwrite(yf,11025,8,'sampleout2.wav');
```

Listen to the original file **sample.wav** and compare it to the files your testbench has generated. See if you can hear the filtering effect. Congratulations, you have now completed this lab! Do you recognize from which movie the audio sample has been taken?