ENGINEERING COLLEGE OF AARHUS

Modeling of a smart pitch detector for a HW/SW Co-design

Multidisciplinary Project

Kim Bjerge (20097553)

Q1+Q2 2010

Abstract

This document describes a methodology for HW/SW Co-design applied for signal processing algorithms mapping to a SoPC design using FPGA. It is a system level design methodology that starts with an executable specification model in Matlab. This functional model is graduated refined producing intermediate UML/SysML and SystemC models with focus on designing the architecture with performance metrics in consideration. The method presented is based on a manually model transformation coming from the functional and architectural SystemC model to the final implementation in hardware and software. The methodology explained is exemplified with implementation of the smart pitch detecting algorithm on the DE2 board and the Nios II softcore processor from Altera.

Contents

1 Introduction

Hardware/software co-design is a relevant topic for the increasing capacity of the field-programmable gate arrays (FPGAs) that today is on the market. The introduction of FPGAs from Xilinx[15] and Altera [16] that supports building system-on-chip (SoC) designs, with softcore 32bits processors as Microblaze and NIOS II, enables the designer to choose between implementing the functionality in hardware or software. Wayne Wolf [2] already in 2003 describes this challenge in defining a methodology that enables designers to perform design and simulation at an appropriated level of abstraction. The designer needs to be able to perform co-simulation at mixed abstraction levels to execute enough situations to validate the design and decide on the partitioning functionality to hardware IP Blocks or software computation.

Recently there has been introduced a number of different methodologies based on UML-modeling, like the commercial tool FalconML [17], that is a design automation tool which covers the entire development process based on UML for combined HW/SW systems targeting FPGA designs. Another methodology was presented at the DATE conference in 2010 in the paper “Closing the Gap between UML-based Modeling, Simulation and Synthesis of Combined HW/SW Systems” [3]. This methodology supports design automation at higher levels of abstraction. For system specification and simulation it uses the system level design language (SLDL) SystemC. The approach of this paper is to bridge the gap between UML-based modeling and SystemC-based simulation. It presents a customized graphical SysML design environment based on Artisan Studio [18]. The automation supports generation of transaction level modeling (TLM) code for the QEMU [19] processor emulator connected to the memory-mapped SystemC TLM bus. The purpose is to make a simulator that support design exploration of a multiple set of architectures like ARM and PowerPC. The SystemC part is finally synthesized to VHDL for FPGA configuration.

The above system level design methodologies are the inspiration for this project. Would it be possible to use a combination of Matlab, UML and SystemC to define a development process for a HW/SW Co-design targeting the FPGA SoC platform? I would like to define a method that in contrary to automation is manually based. Would it be possible to used technologies like Matlab, UML and SystemC in a process for signal processing algorithms in design and implementation for FPGA based SoC designs? Many companies that used FPGA in their designs typically only use the FPGA design tools supplied by the FPGA vendors like “Quartus II” from Altera or the “ISE Design Suite” from Xilinx. I would like to present a method that can be used together with existing tools without the need of investment in an additional expensive EDA tools (Electronic Design and Automation).

Matlab will be used to develop a functional model of the signal processing algorithm. Simulink-to-FPGA [20] synthesis technology is another alternative that is provide by MathWorks together with FPGA vendors on the market, but here you have to learn Simulink and invest in expensive tool extension.

SysML block diagrams will be used to create an architectural model of the functionality and mapping to the SoPC architecture. The purpose is to give a graphical overview of the design structure. For partitioning, timing estimation and architecture refinement an executable model in SystemC will be used as a golden reference for final hardware and software implementation and verification. The methodology will be exemplified by modeling and implementation of a smart pitch detector targeting the Altera DE2 board including the Nios II softcore processor.

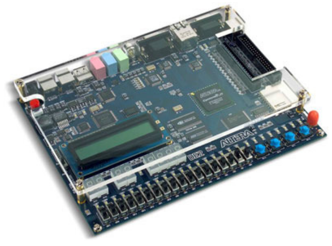
The learning objectives by doing this project is to combine the contents of a number of different course at the master study provided by the Engineering Colleague of Aarhus covering Embedded Real-Time Systems, DSP in applied digital signal processing, Hardware/Software Co-design and Advanced VHDL programming.

1.1 Report Structure

This report is split into four major sections: Introduction, Project Description, Project Execution and Conclusion. The Project Description is a copy of the original problem formulation with prioritized project goals and deliveries. The Project Execution is the theoretical and practical section where the execution of the project is described together with the presented methodology and project results. The UML part of the methodology is based is inspired based on papers [3], [4] and [5]. The architectural model of the system design is inspired by the book “System Design with SystemC” [6]. The effort of work performing the project and a conclusion summaries the finding, learning and experience gained the project.

2 Project Description

Pitch detection is important in many sound engineering applications like handheld “tuners”, pitch correction, speech recognition and sound effects. The paper “A Smarter Way To Find Pitch” [1] defines the “McLeod Pitch Method” (MPM) an improved method of finding the fundamental pitch in a sound.



This project is mostly about modeling of the MPM algorithm using a HW/SW Co-design design methodology based on Matlab, UML and SystemC with purpose of transforming the validated MPM algorithm to an embedded FPGA platform. The design must have focus on software and hardware architecture and partitioning. The implementation target will be the Altera DE2 board including the Nios II processor. The required sampling rate is 48 kHz with a bit resolution of 24 bits.

Prioritized Goals:

The main goals of the project are prioritize as listed below: *(Point 4 and 5 are less important)*

1. create a Matlab model of the MPM algorithm and validate the algorithm with sound examples
2. create and UML model of the overall design for the HW/SW architecture
3. create and executable model of the HW/SW architecture in fixed-point with SystemC
4. transformation of the SystemC model for implementation in VHDL and C++ on the Altera DE2 board including the Nios II processor
5. compare the performance of a software vs. hardware optimized version of MPM algorithm

Project Deliveries:

- **A validated Matlab model with sound examples**
- **A validated executable HW/SW architecture model in SystemC**
- **A project rapport describing the UML architecture, process and results**
- **A prototype on the Altera DE2 board with Nios II processor**

3 Project execution

The project is performed using a top-down methodology with focus on HW/SW Co-design. It starts with a functional behavioral model in Matlab and SystemC with the focus on finding a good high level abstraction on a solution for implementation, of the smart pitch detector in fixed point arithmetic prepared to be allocated either to software or hardware. UML class diagrams with ports are used to design the overall structure of the SystemC models.

The purpose of the architectural model in the following design step is to prepare a golden reference model for hardware and software implementation by partitioning to either hardware blocks or software threads and adding signal and timing to the SystemC model. The partitioning is base on analyzing the SystemC functional model, in consideration of number of operations and the required sampling rate. A first component based prototype is created, using the Altera SOPC builder for the DE2 board, in demonstrating a simple audio loop back functionality.

The following step is to implement and verify the hardware IP Block (NSDF computation) using Altera FPGA tools Quartus II and ModelSim. The HW IP Block is first designed and verified without using the internal RAM blocks of the FPGA. Quartus II is used to back annotate the SystemC architectural model with timing based on the Quartus analysis tool. Finally the HW IP Block is redesigned using the internal RAM blocks to store samples and temporary buffers. The design is refined and verified in a test bench using ModelSim.

The final system is implemented by writing the software part and adding the HW IP block to the SoPC design, based on the Matlab and SystemC functional and architectural models,. The final pitch detector is tested on the DE2 board writing the calculated pitch frequency in the LCD display. The audio is generated using the sound card in a connected PC running a tone generator program.

3.1 Functional model

This model view contains the highest abstraction in the design flow. It focuses on algorithmic design without considering implementation details. The aim is to create an executable specification which is untimed and focus on validating the signal processing algorithm. It is hardware and software neutral with focus on computation and algorithm functionality. The output is a functionally verified executable model which could be developed by the “Algorithm Developer”.

In this project we start with a Matlab model of the pitch detector consisting of computational model of the McLeod Pitch Method (MPM) including generating test signals as stimuli to the model. The second version of the functional model is a SystemC executable model where the design is separating the system into communicational and computational blocks based on transaction level modeling (TLM). The structural design is illustrated using UML object and class diagrams with ports. The algorithm is refined adding fixed-point arithmetic, sample rate and precision. The purpose is to prepare the algorithm for implementation on an embedded system. At this point the platform is not taken into consideration.

3.1.1 Pitch algorithm model in Matlab

The fast, accurate and robust method of finding a pitch in monophonic musical sounds is described in "A smarter way to find pitch" [1]. There are probably other methods even in finding the pitch in polytonal musical sounds, but the method described in the above paper has a good level of complexity for the purpose of this project. The algorithm includes a heavy number of multiplications, additions and even divisions that must be computed in very few milliseconds. The method defines the Normalized Square Difference Function (NSDF) as follows:

$$n'_t\ (\tau) = \frac{2r'_t\ (\tau)}{m'_t(\tau)} \qquad (9)$$

The greatest possible magnitude puts $n'_t(\tau)$ in the range of -1 to 1, where 1 means perfect correlation, 0 means no correlation and -1 means perfect negative correlation, irrespective to the waveform's amplitude. The computation of the autocorrelation function (ACF type II) is defined as:

$$r'_t(\tau) = \sum_{j=t}^{t+W-1-\tau} x_j x_{j+\tau} \quad (2)$$

In this definition the window size decreases with increasing τ reducing the number of computations. The discrete signal Square difference function (SDF type II) can be defined as:

$$d'_t\ (\tau) = m'_t(\tau) - 2r'_t(\tau) \quad (7)$$

Where m'_t is defined as:

$$m'_t(\tau) = \sum_{j=t}^{t+w-\tau-1} (x_j^2 + x_{j+\tau}^2) \quad (6)$$

Calculation of the pitch period will be to choose the best maximum peak given by the function computed by equation (9).

Combining equation (9), (2) and (6) in a Matlab model is listed below. The model is stimulated with a periodic sine wave signal modulated with the first and second harmonic sine frequency and noise.

```
%% Parameters for test of the pitch detector algorithm
fs = 48000;
f1 = 200;
f2 = 2*f1;
f3 = 4*f1;
A1 = 0.5;
A2 = 0.1;
N = 2000;
W = 2000;
tstart = 2000;
k = 0.80; % Ratio for determination of local maximums

%% Noisy test signal that contains a sine wave
tst = A1*sin(2*pi*(f1/fs)*(1:N)) + A2*sin(2*pi*(f2/fs)*(1:N))+
A2*sin(2*pi*(f2/fs)*(1:N));
x = [0.1*randn(1,N) (tst + 0.1*randn(1,N)) 0.1*randn(1,N)];

figure(1);
subplot(3,1,1);
plot(x);
title('Input signal with noise');

%% Normalized Squared difference function
% From article "A Smarter way to find pitch"
r_tau = zeros(1,W); % Autocorrelation ACF type II
m_tau = zeros(1,W); % Squared difference function modified SDF
n_tau = zeros(1,W); % Normalized squared difference function

t = tstart; % Start at time when periodic signal is avaiable
for tau = 1:W
    for j = t:t+W-1-tau
        r_tau(tau) = r_tau(tau) + x(j)*x(j+tau); % ACF type II (2)
        m_tau(tau) = m_tau(tau) + x(j)^2 + x(j+tau)^2; % SDF modified (6)
        n_tau(tau) = 2*r_tau(tau)/m_tau(tau); % Normalized NSDF (-1 to 1) (9)
    end
end
```

Calculation of the NSDF takes approximately O (W * W/2) times in calculating r_tau and m_tau. In the below figure is illustrated the result of the Matlab simulation.

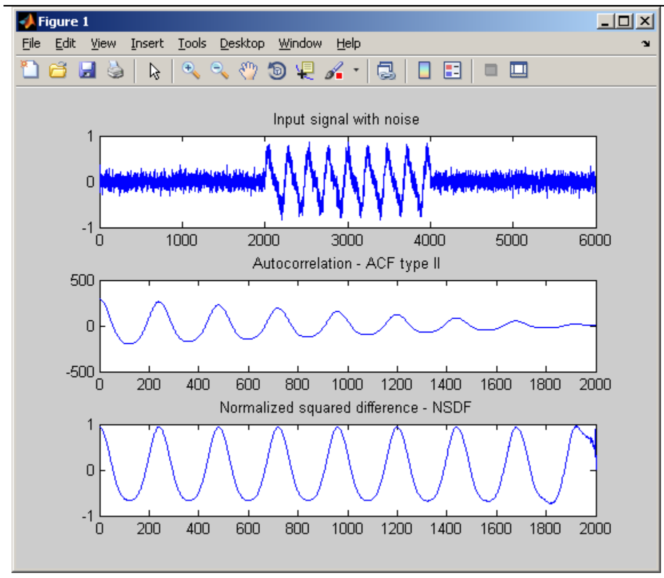


Figure 1 Matlab plot of NSDF with 200 Hz input sine signal and noise

The algorithm gives us correlation coefficients at integer τ. The first "major" peak represents the pitch period in the figure above at τ = 239. To find this maximum first an algorithm is developed to find the entire local maximums above zero crossing.

```
%% Find the local maxima
tau = n_tau;
n_1 = tau(1); % Treshold for detection of first maximum - parameter
n_2 = n_1;
treshold = n_1*k;
% Initialization
local_max_samples = zeros(1, 1);
local_max_values = zeros(1, 1);
idx = 1;
rising = 0;
n_1 = 0;

for n=1:W
    if (tau(n) > 0)
        if (tau(n) > n_1) rising = 1;
        end
        if (tau(n) < n_1)
            if (rising == 1)
                local_max_samples(idx) = n - 1;
                local_max_values(idx) = n_1;
                disp(['Local max: ' num2str(local_max_values(idx))
                      ' counts : ' num2str(local_max_samples(idx))]);
                idx = idx + 1;
            end
            rising = 0;
        end

    end
    n_1 = tau(n);
end
```

The above algorithm is looking at the slope of the computed NSDF curve for all values above zero. Whenever a descending slope is found and the previous slope was rising the local maximum is added to the vectors that contains information about the maximum. The local maximum vectors contain information about number of samples (τ) from start of the NSDF curve and the local maximum values.

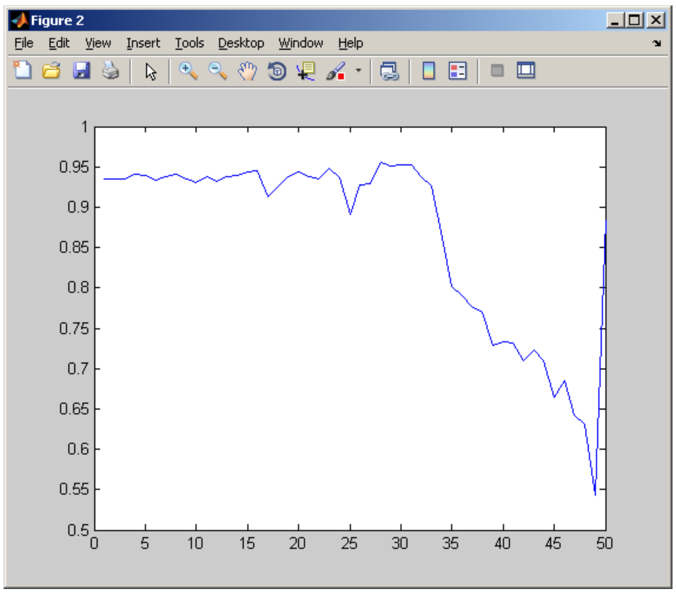


Figure 2 Plot of local maximum values

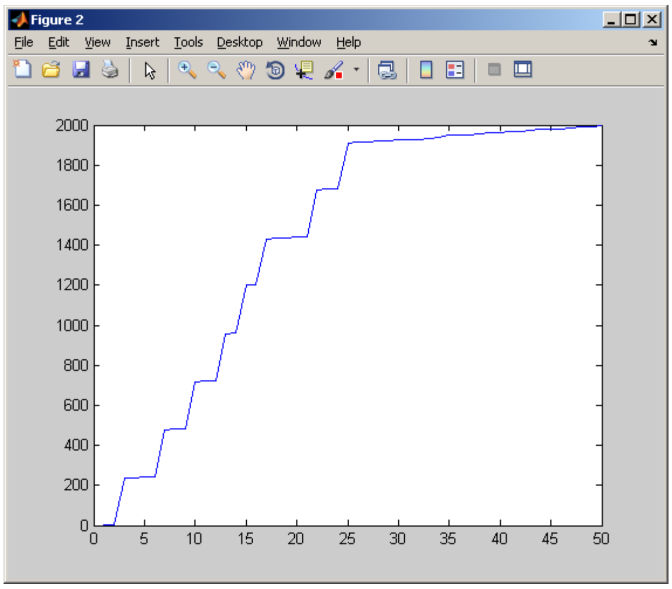


Figure 3 Plot of local maximum samples τ

The plotted function above indicates a pattern where the local peak maximums are found. By searching the maximum in the above vectors using the same algorithm again we can find the "major" peaks we are looking for.

The Matlab model below shows how to find the "major" peaks and calculate the pitch frequency and note. The slope of the local maximum curve is again use to find the "major" maximum peaks.

```
%% Find the maxima
max_samples = zeros(1, 1);
max_values = zeros(1, 1);
idx = 1;
rising = 0;
n_1 = local_max_values(1);
treshold = 0.80;

for n=2:length(local_max_samples)
    if (local_max_values(n) > n_1) rising = 1;
    end
    if (local_max_values(n) < n_1)
        if (rising == 1) && (n_1 > treshold)
           max_samples(idx) = local_max_samples(n - 1);
           max_values(idx) = n_1;
           disp(['Maximum: ' num2str(max_values(idx))
                 ' counts : ' num2str(max_samples(idx))]);
           idx = idx + 1;
        end
        rising = 0;
    end
    n_1 = local_max_values(n);
end

figure(2);
plot(max_values);

%% Use first maxima to find pitch
tau_max = max_samples(1);
maximum = max_values(1);
disp(['Tau max: ' num2str(tau_max) ' value ' num2str(maximum) ]);

fc = fs/tau_max;
note = log10(fc/27.5)/log10(sqrt(2));
disp(['Pitch frequence detected : ' num2str(fc)]);
```

Figure 4 plots the maximum peaks that is found based on the first iteration in where we have computed all the local maximums. The desired maximum is the first in this plot that we will use to calculate the final pitch. Parabolic interpolation could be used to improve the position of this maximum to a higher accuracy, but this has not been done in this project. The maximum we have found is based on discrete values and the real maximum will be in between the sample just before and after the maximum we have found.

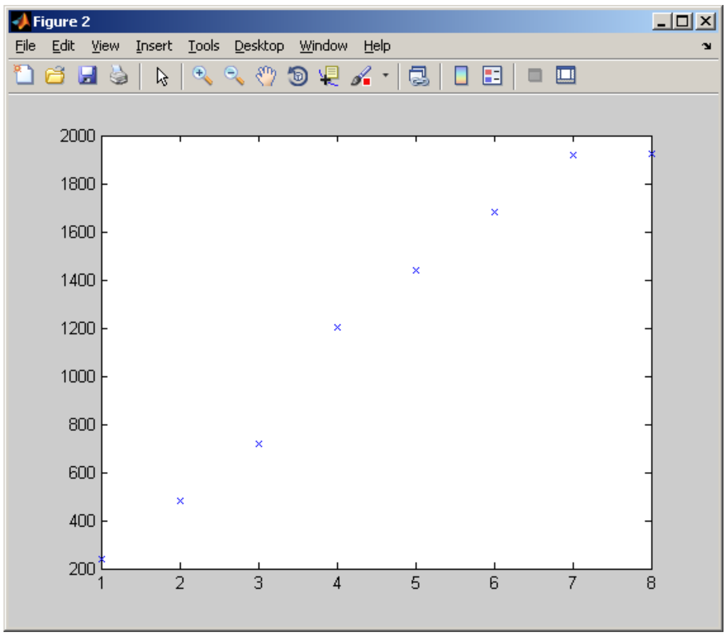


Figure 4 Maximums found in NSDF with 200 Hz input sine signal with noise

Major maximums found at τ: **238**, 480, 720, 1202, 1440, 1680, 1920, 1926

Final result printed from Matlab:

```
Maximum: 0.9403 counts : 239
Maximum: 0.94097 counts : 480
Maximum: 0.93847 counts : 720
Maximum: 0.94507 counts : 1202
Maximum: 0.94394 counts : 1440
Maximum: 0.94786 counts : 1680
Maximum: 0.95483 counts : 1920
Maximum: 0.95296 counts : 1926
Tau max: 239 value 0.9403
Pitch frequence detected : 200.8368
```

With a window size of 1024 the minimum pitch frequency we can detect will be:

fs/W = 48000/1024 = **46.875 Hz.**

3.1.2 Pitch algorithm model in SystemC

SystemC is an extended library based on C++ for creating models of electronic systems covering hardware and software elements. The following list shows the major components of SystemC.

- Simulation Kernel
- Data types covering logic, integers and fixed point
- Modules and hierarchy
- Channels and interfaces
- Events, sensitivity and notifications
- Threads and methods
- Predefined primitive channels as mutex's, FIFO's and signals

SystemC addresses the modelling of both software and hardware using C++. Since C++ already covers software, it should come as no surprise that SystemC focuses primarily on non-software areas. The primary application area of SystemC is design of electronic systems. The major hardware-oriented features implemented within SystemC include:

- Time model
- Hardware data types
- Module hierarchy to manage structure and connectivity
- Communications management between concurrent units of execution
- Concurrency model

SystemC contains an ultra light-weight cycle-based kernel for high-speed simulation. It makes it possible to have multiple levels of abstractions ranging from high-level functional models to detailed clock cycle accurate RTL (Register Transfer Level) models. It supports an iterative refinement of high-level models into lower levels of abstraction. The high-level part of a model can be reused for creation of test bench for verifying design and hardware implementation.

Electronic systems development using SystemC makes it possible to create an executable specification which is executed with the same behaviour as the final system. The most important activity in preparing the design for either hardware or software implementation is to use a model of computation (MoC) where computational modules are separated in processes communicating with each others through channels. This MoC abstracts busses and network in terms of channels and processing SW tasks or HW blocks in terms of computational modules.

A minor set of SystemC modules has been defined which all connects together by ports, interfaces and the sc_fifo channels. This MoC approach is funded on the Kahn Process Network (KPN) see page 53 in [9], processes are only allowed to communicate via uni-directional and point-to-point asynchronous message passing channels. Channels are not unbounded when using the sc_fifo channels but receivers will always block until a complete message (Sample in our case) is available.

All modules are parameterized with the template functionality of C++. This approach makes it possible to decide on the data type the classes operate on when they are instantiated. It would be easy to create other types of modules which can be added to the model. At the functional SystemC level, the type double is used for the data processing. Later the same model has been used for fixed point simulation just by changing the type globally for all classes.

The monitor stores the digital monitored signal in text files. The mixer takes more sc_fifo channels and mixes them based on a gain setting. The fork module is able to split a signal input to more sc_fifo channel outputs.

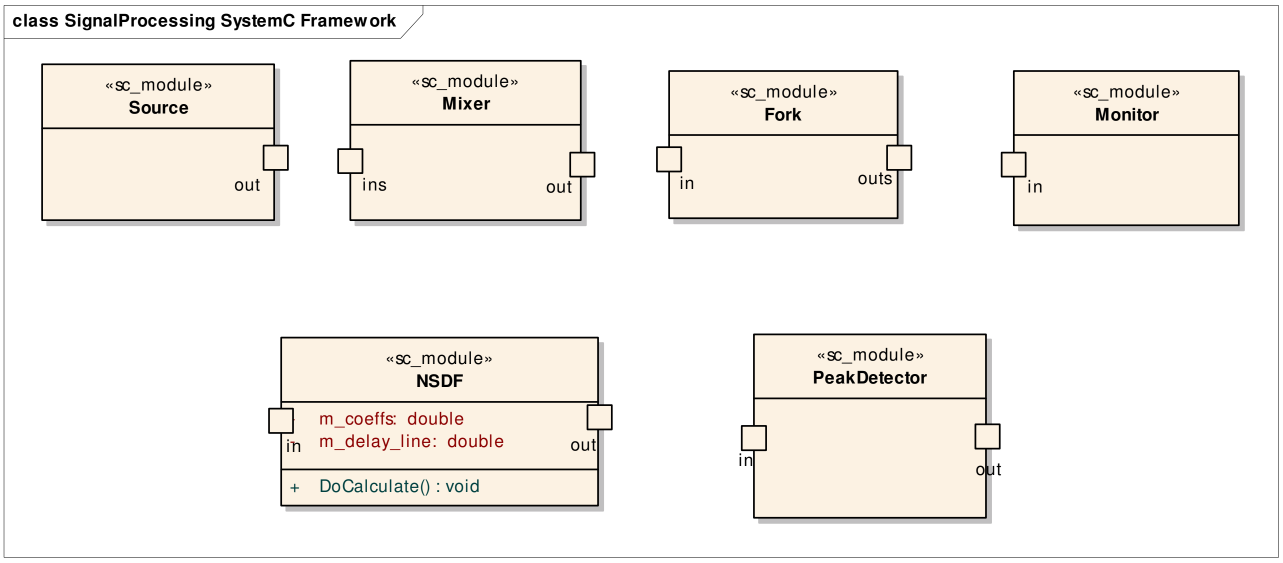


Figure 5 UML classes for SystemC modules with ports

In this example, 3 instance of the sine signal source are instantiated and connected to the mixer module. The source produces the same frequencies as we had in the Matlab model. The design is set up to make it easy to change the parameters for the model concerning sample rate, length of window for the NSDF computation and bit resolution of the sampling and algorithm.

```
#define WBUS        32      // Bus width
#define ALGO_BITS   32      // Number of bits in sample
#define WINDOW      1024    // Window size for normalized squared difference function
//#define WINDOW        16  // Window size for normalized squared difference function
#define ADC_BITS    24      // Number of bits from ADC
#define DIFF_BITS   (ALGO_BITS - ADC_BITS)

// Configurations parameters for the pitch detector model
const int samples = 1024;       // Number of samples
//const int samples = 16;       // Number of samples
const unsigned fs = 48000;      // Sample frequence
```

Figure 6 File that contains global configuration parameters (Defs.h)

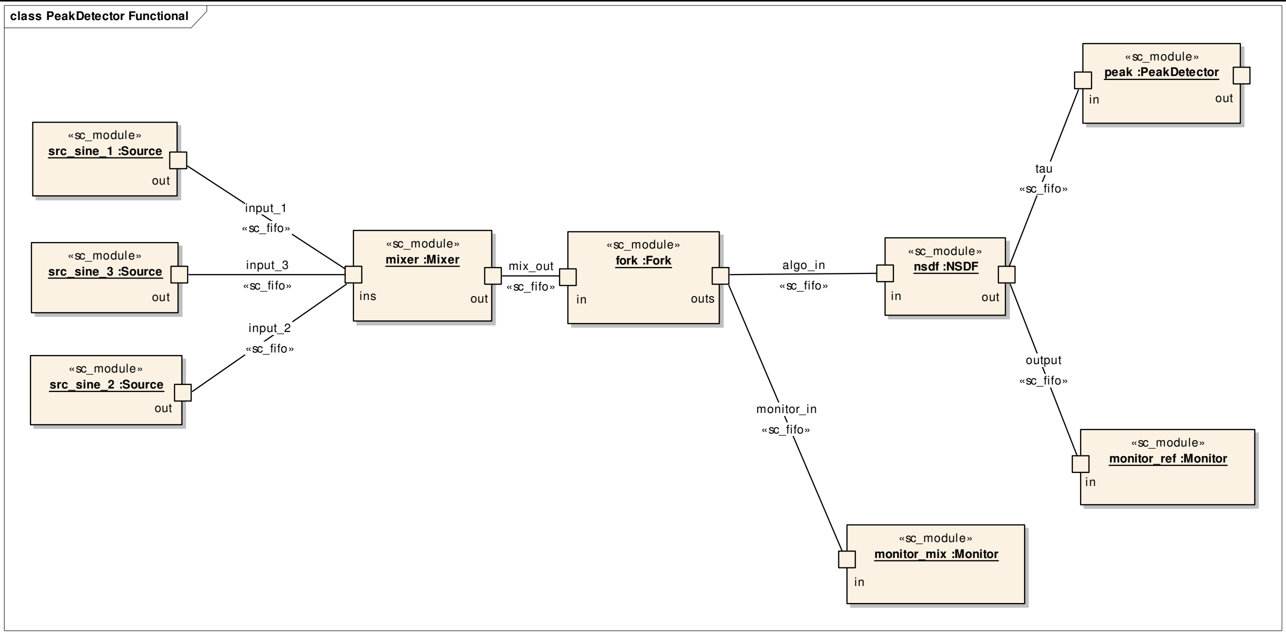

class PeakDetector Functional
«sc_module» src_sine_1 :Source out
«sc_module» src_sine_3 :Source out
«sc_module» src_sine_2 :Source out
input_1 «sc_fifo»
input_3 «sc_fifo»
input_2 «sc_fifo»
«sc_module» mixer :Mixer ins out
mix_out «sc_fifo»
«sc_module» fork :Fork in outs
algo_in «sc_fifo»
«sc_module» nsdf :NSDF in out
tau «sc_fifo»
«sc_module» peak :PeakDetector in out
output «sc_fifo»
«sc_module» monitor_ref :Monitor in
monitor_in «sc_fifo»
«sc_module» monitor_mix :Monitor in

Figure 7 UML object diagram for SystemC modules and connections using sc_fifo channels for communication

The Fork is splitting the output signal from the mixer to a monitor that saves the test stimuli signal values in a file and sends the same signal to the NSDF module. The output result from the NSDF module is stored in another file with the NSDF window of samples by the ref monitor. Finally the NSDF computation will be processed by the peak detector algorithm as described in the Matlab model.

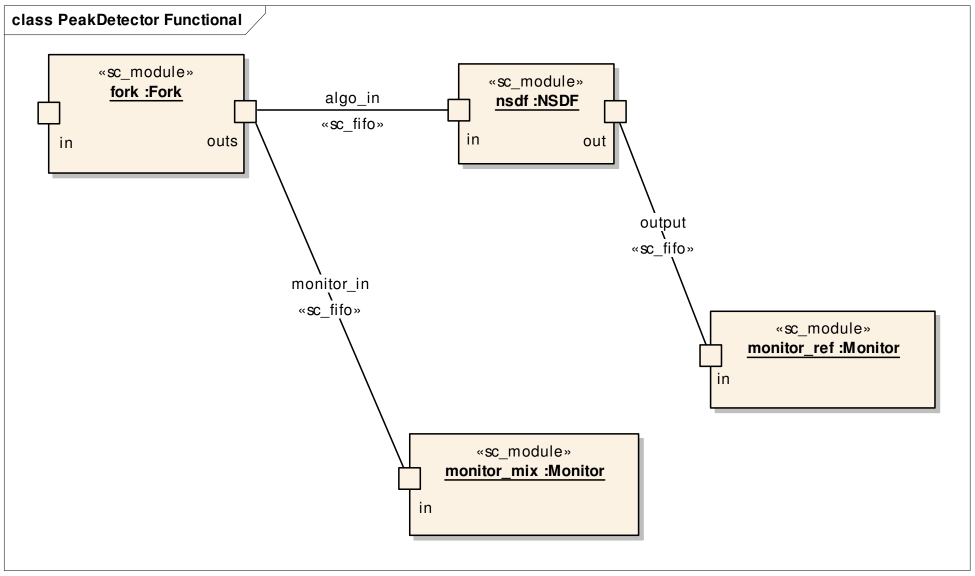
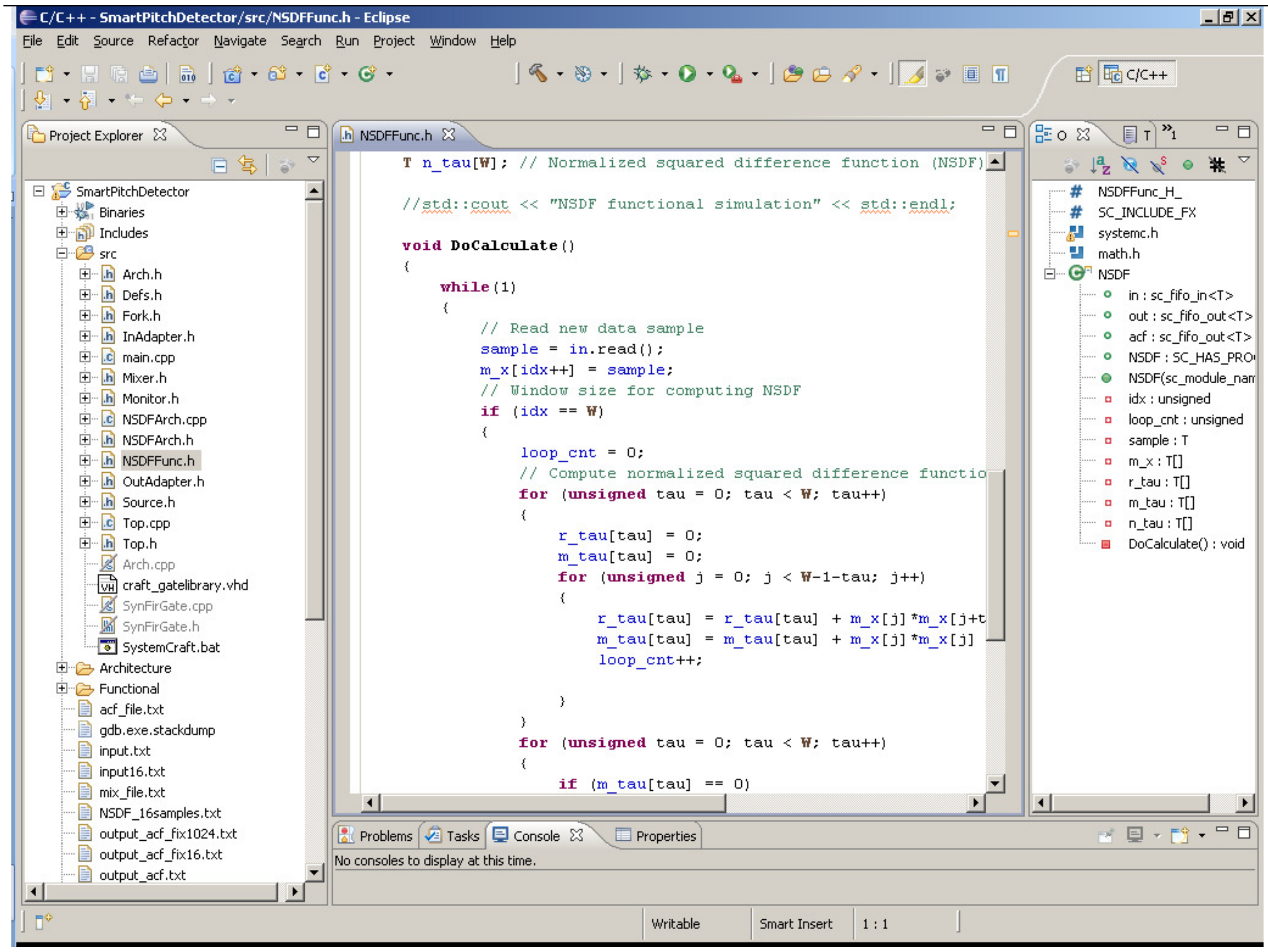

class PeakDetector Functional
«sc_module» fork :Fork in outs
algo_in «sc_fifo»
«sc_module» nsdf :NSDF in out
output «sc_fifo»
«sc_module» monitor_ref :Monitor in
monitor_in «sc_fifo»
«sc_module» monitor_mix :Monitor in

Figure 8 Monitoring NSDF result



```
T n_tau[W]; // Normalized squared difference function (NSDF)

//std::cout << "NSDF functional simulation" << std::endl;

void DoCalculate()
{
    while(1)
    {
        // Read new data sample
        sample = in.read();
        m_x[idx++] = sample;
        // Window size for computing NSDF
        if (idx == W)
        {
            loop_cnt = 0;
            // Compute normalized squared difference functio
            for (unsigned tau = 0; tau < W; tau++)
            {
                r_tau[tau] = 0;
                m_tau[tau] = 0;
                for (unsigned j = 0; j < W-1-tau; j++)
                {
                    r_tau[tau] = r_tau[tau] + m_x[j]*m_x[j+t
                    m_tau[tau] = m_tau[tau] + m_x[j]*m_x[j]
                    loop_cnt++;

                }
            }
            for (unsigned tau = 0; tau < W; tau++)
            {
                if (m_tau[tau] == 0)
```

No consoles to display at this time.

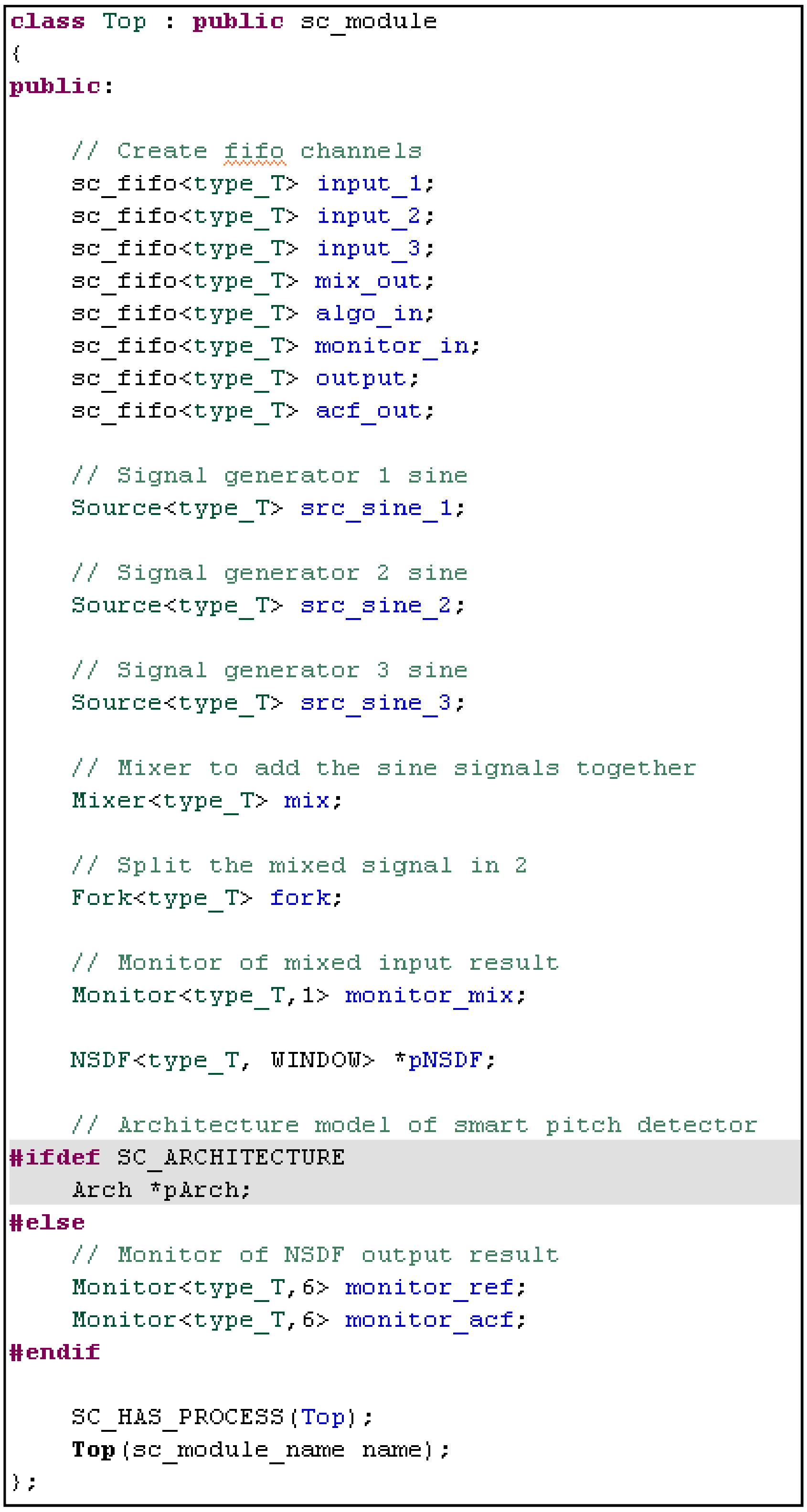
Figure 9 Eclipse IDE integrated with Cygwin and the SystemC library

The SystemC model is programmed using Eclipse and Cygwin running on Windows. SystemC version 2.2 is used. The model is programmed in the same project covering the functional and the architecture model. In Eclipse the build configuration Functional or Architecture can be set active deciding on the configuration to be compiled and executed.

The complete functional model is implemented in a top SystemC module which creates the instances of the modules, fifo channels and connections. On the next page, you can see the SystemC top module. All modules use the type_T type definition that can be substituted with the type double or sc_fix in exploring different fixed point implementations.

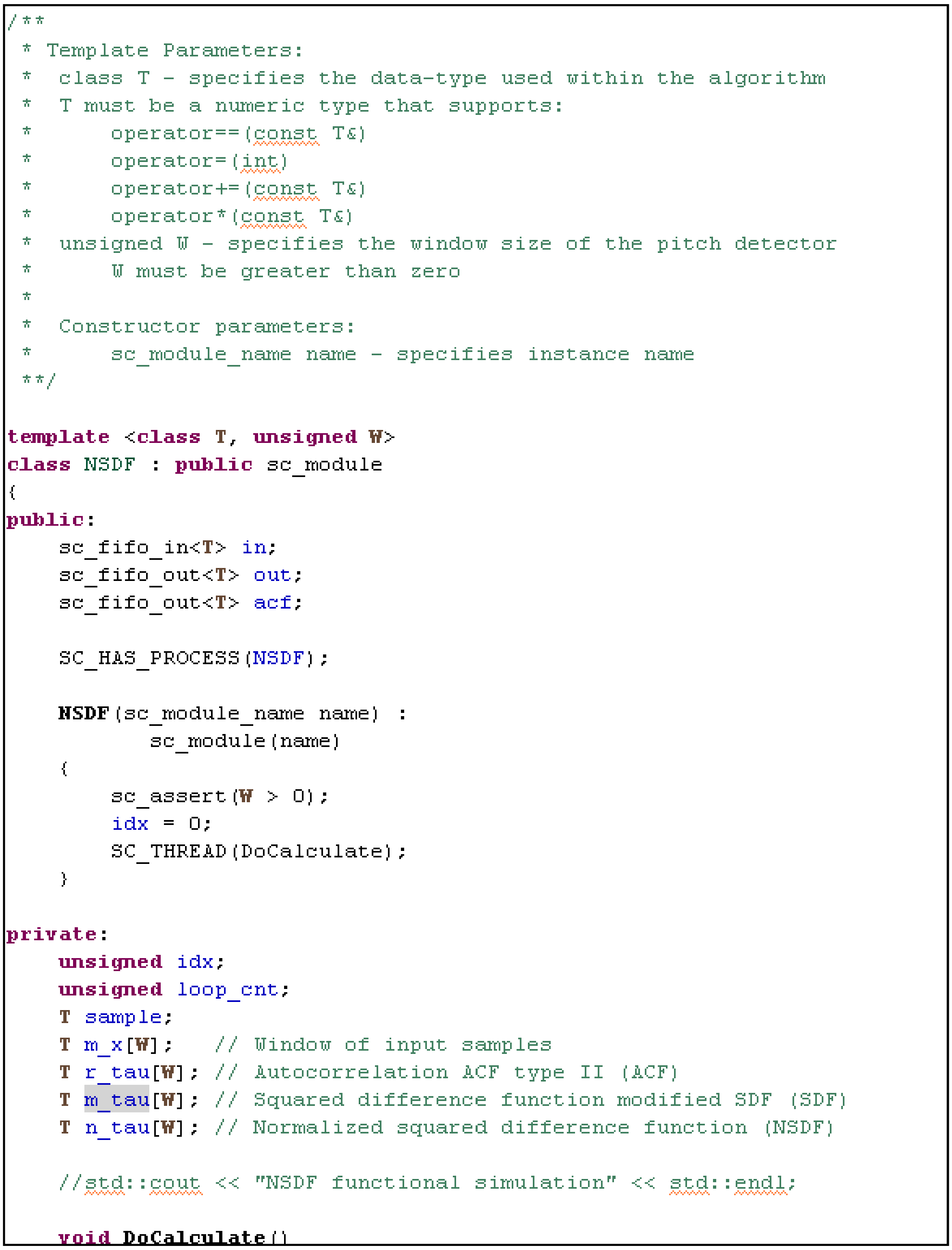
```
#if 1
typedef sc_fixed_fast<32,16> type_T;
const std::string ref_file("output_fixed.txt");
#else
typedef double type_T;
const std::string ref_file("output_double.txt");
#endif
```

Figure 10 Globally setting floating or fixed point simulation



```
class Top : public sc_module
{
public:

    // Create fifo channels
    sc_fifo<type_T> input_1;
    sc_fifo<type_T> input_2;
    sc_fifo<type_T> input_3;
    sc_fifo<type_T> mix_out;
    sc_fifo<type_T> algo_in;
    sc_fifo<type_T> monitor_in;
    sc_fifo<type_T> output;
    sc_fifo<type_T> acf_out;

    // Signal generator 1 sine
    Source<type_T> src_sine_1;

    // Signal generator 2 sine
    Source<type_T> src_sine_2;

    // Signal generator 3 sine
    Source<type_T> src_sine_3;

    // Mixer to add the sine signals together
    Mixer<type_T> mix;

    // Split the mixed signal in 2
    Fork<type_T> fork;

    // Monitor of mixed input result
    Monitor<type_T,1> monitor_mix;

    NSDF<type_T, WINDOW> *pNSDF;

    // Architecture model of smart pitch detector
#ifdef SC_ARCHITECTURE
    Arch *pArch;
#else
    // Monitor of NSDF output result
    Monitor<type_T,6> monitor_ref;
    Monitor<type_T,6> monitor_acf;
#endif

    SC_HAS_PROCESS(Top);
    Top(sc_module_name name);
};
```

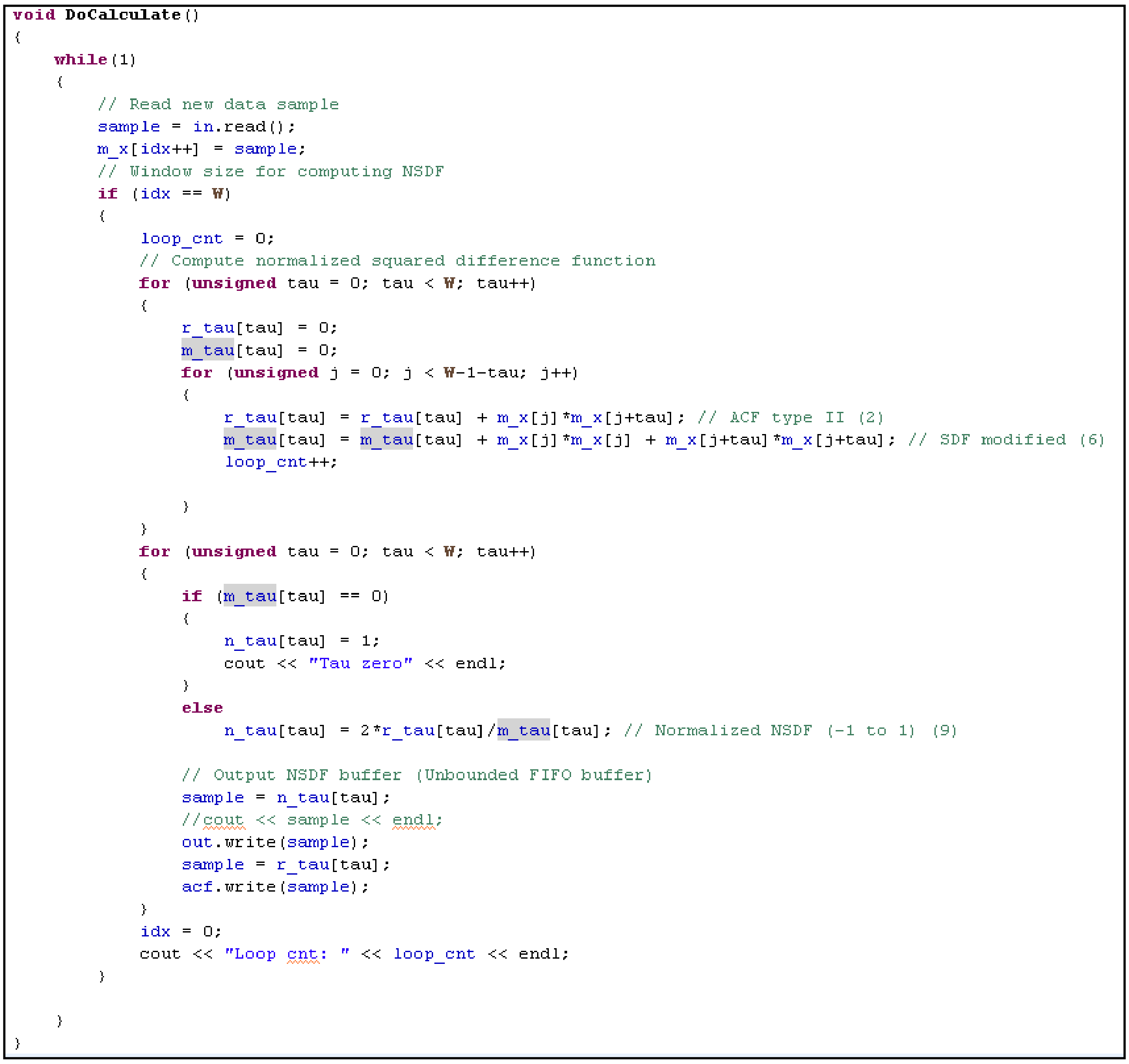
Figure 11 Top level module of the SystemC model



```
/**
 * Template Parameters:
 *  class T - specifies the data-type used within the algorithm
 *  T must be a numeric type that supports:
 *      operator==(const T&)
 *      operator=(int)
 *      operator+=(const T&)
 *      operator*(const T&)
 *  unsigned W - specifies the window size of the pitch detector
 *      W must be greater than zero
 *
 *  Constructor parameters:
 *      sc_module_name name - specifies instance name
 **/

template <class T, unsigned W>
class NSDF : public sc_module
{
public:
    sc_fifo_in<T> in;
    sc_fifo_out<T> out;
    sc_fifo_out<T> acf;

    SC_HAS_PROCESS(NSDF);

    NSDF(sc_module_name name)  :
           sc_module(name)
    {
        sc_assert(W > 0);
        idx = 0;
        SC_THREAD(DoCalculate);
    }

private:
    unsigned idx;
    unsigned loop_cnt;
    T sample;
    T m_x[W];   // Window of input samples
    T r_tau[W]; // Autocorrelation ACF type II (ACF)
    T m_tau[W]; // Squared difference function modified SDF (SDF)
    T n_tau[W]; // Normalized squared difference function (NSDF)

    //std::cout << "NSDF functional simulation" << std::endl;

    void DoCalculate()
```

Figure 12 NSDF SystemC module of the functional model

The normal squared difference function (NSDF) is modeled in SystemC module running in the thread DoCalculate. This thread reads input samples from the in port it collects a number of samples equal to the size of the window parameter **W** before start calculation of the NSDF.



```
void DoCalculate()
{
    while(1)
    {
        // Read new data sample
        sample = in.read();
        m_x[idx++] = sample;
        // Window size for computing NSDF
        if (idx == W)
        {
            loop_cnt = 0;
            // Compute normalized squared difference function
            for (unsigned tau = 0; tau < W; tau++)
            {
                r_tau[tau] = 0;
                m_tau[tau] = 0;
                for (unsigned j = 0; j < W-1-tau; j++)
                {
                    r_tau[tau] = r_tau[tau] + m_x[j]*m_x[j+tau]; // ACF type II (2)
                    m_tau[tau] = m_tau[tau] + m_x[j]*m_x[j] + m_x[j+tau]*m_x[j+tau]; // SDF modified (6)
                    loop_cnt++;

                }
            }
            for (unsigned tau = 0; tau < W; tau++)
            {
                if (m_tau[tau] == 0)
                {
                    n_tau[tau] = 1;
                    cout << "Tau zero" << endl;
                }
                else
                    n_tau[tau] = 2*r_tau[tau]/m_tau[tau]; // Normalized NSDF (-1 to 1) (9)

                // Output NSDF buffer (Unbounded FIFO buffer)
                sample = n_tau[tau];
                //cout << sample << endl;
                out.write(sample);
                sample = r_tau[tau];
                acf.write(sample);
            }
            idx = 0;
            cout << "Loop cnt: " << loop_cnt << endl;
        }

    }
}
```

Figure 13 NSDF SystemC DoCalculate computation

The NSDF calculation is performed in 2 loops. The first double loop is performed in O (W*W/2) times computing the m'_τ (m_tau) and r'_τ (r_tau). The NSDF (n'_τ) is computed in the second loop in O (W) times.

The reference monitor is storing the result to a file from where the result is plotted with Matlab and compares the computed NDSF in double format with the fixed point format (16.16).

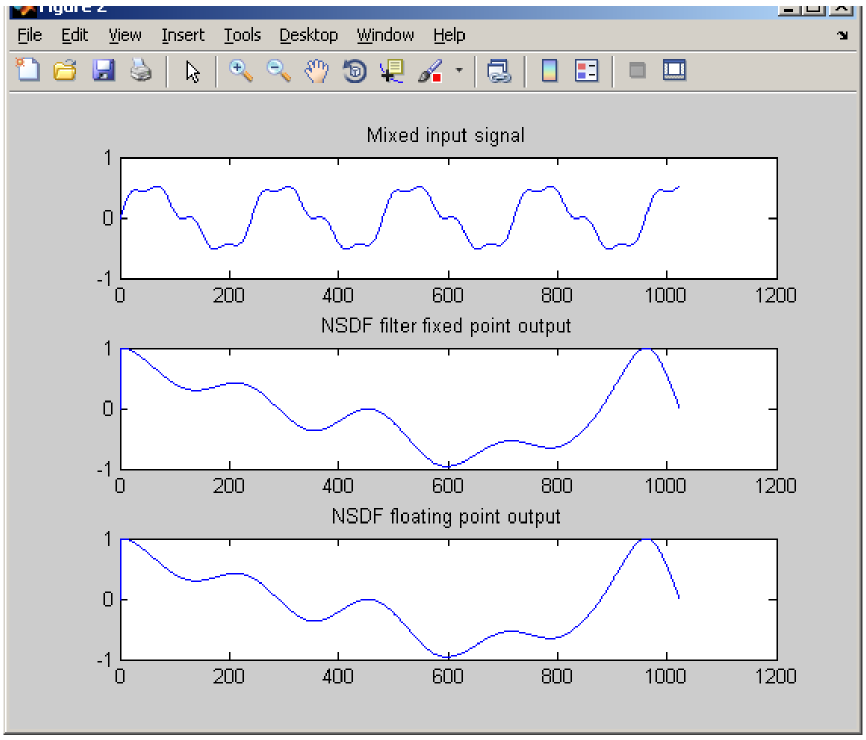


Figure 14 NSDF fixed vs. floating point with at fc = 50 Hz, Window size = 1024

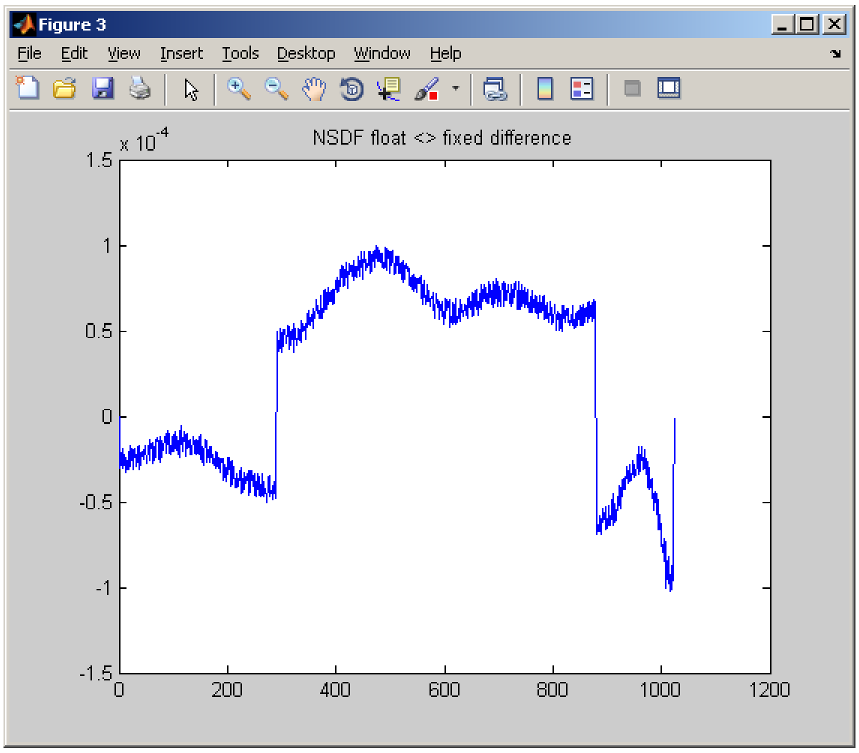


Figure 15 Difference between fixed and floating point calculation

The error difference between using float and fixed point is:

20*log (max signal / max error) ~ 20*log(1/1*10^-4) = **80 dB**

With an absolute error of approximately +/-0.0001 obtained from Figure 15.

With a resolution of 32 bit the theoretical signal-to-quantization noise is:

20*log(1/2^-23) = **192 dB**

That means the NSDF computation adds a computation error of = **112 dB**. With a window size of 1024 we have 524288 (O(W*W/2)) multiplications and additions in the double loop this is where the error sums up in truncating the result to format 16.16 in every loop iteration. This error will have an impact on where the maximum peak is found, but it will be less than the impact of the sampling resolution of 48 kHz.

In our case we are sampling with the rate of 48 kHz, this means the maximum peak we find will be in the area of +/- 21 us of the real peak. With a window size of 1024 the error will be:

20*log(max time for NSDF window/max sample time error) =
20*log((W/fs)/(1/fs)) = **20*log(W)** = 20*log(1024) = **60 dB**.

This is the error at 46.875 Hz (fs/W). This error will be even greater for peaks found at high frequencies.

3.2 Architectural model

The architectural view is a refinement of the functional model. It is a system prototype with hardware and software components. At this level the partitioning of SystemC modules in hardware and software is defined manually. For the hardware components, cycle time is added and signals between the hardware modules are defined. The architectural platform is used for design space exploration concerning performance, communication and resource handling.

The architectural view can be used for early software validation and is a streamlined model used by software developers to execute software before the final hardware is available. Performance estimation of the complete system response can be evaluated in terms of cycle time approximation. This is performed by adding hardware signals to the model including clock, reset, handshake and bus signals. The estimate of computation time is added to the model.

The significant cost parameter in deciding on partitioning to hardware or software will in the case of the NSDF computation be the execution time. We need to be able to compute the NSDF and finding the maximum peaks in the time it takes to collect NSDF window size (1024) of samples. That means we have in total **21.33 ms to compute the NSDF and find the peak.**

If we look at the functional model of the NSDF computation we have 3 multiplications and 8 additions in the first double loop that should be performed O(W*W/2) times. If we estimate with loop overhead and loading values from memory we have an estimate of approximately 25 operations in the inner loop. Mapping this algorithm to the Nios II processor running 50 Mhz gives an estimate of 25*524288/50000000 = **0.262 sec**. This is a very optimistic estimate since the 25 operations cannot be computed in 25 CPU cycles. An exact time will be measured later in a final software implementation on target. The 0.262 sec is too long therefore the NSDF will be allocated to a hardware IP block.

In the following UML diagram the model is refined adding reset, sample synchronization clock, CPU clock and a parallel bus to transfer samples directly from an audio codec to the NSDF IP block.

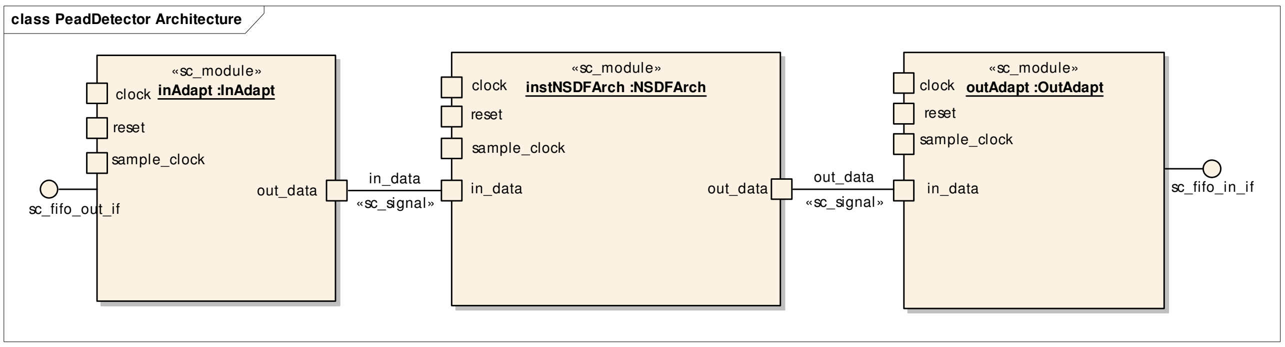


Figure 16 Architectural model of the NSDF IP block

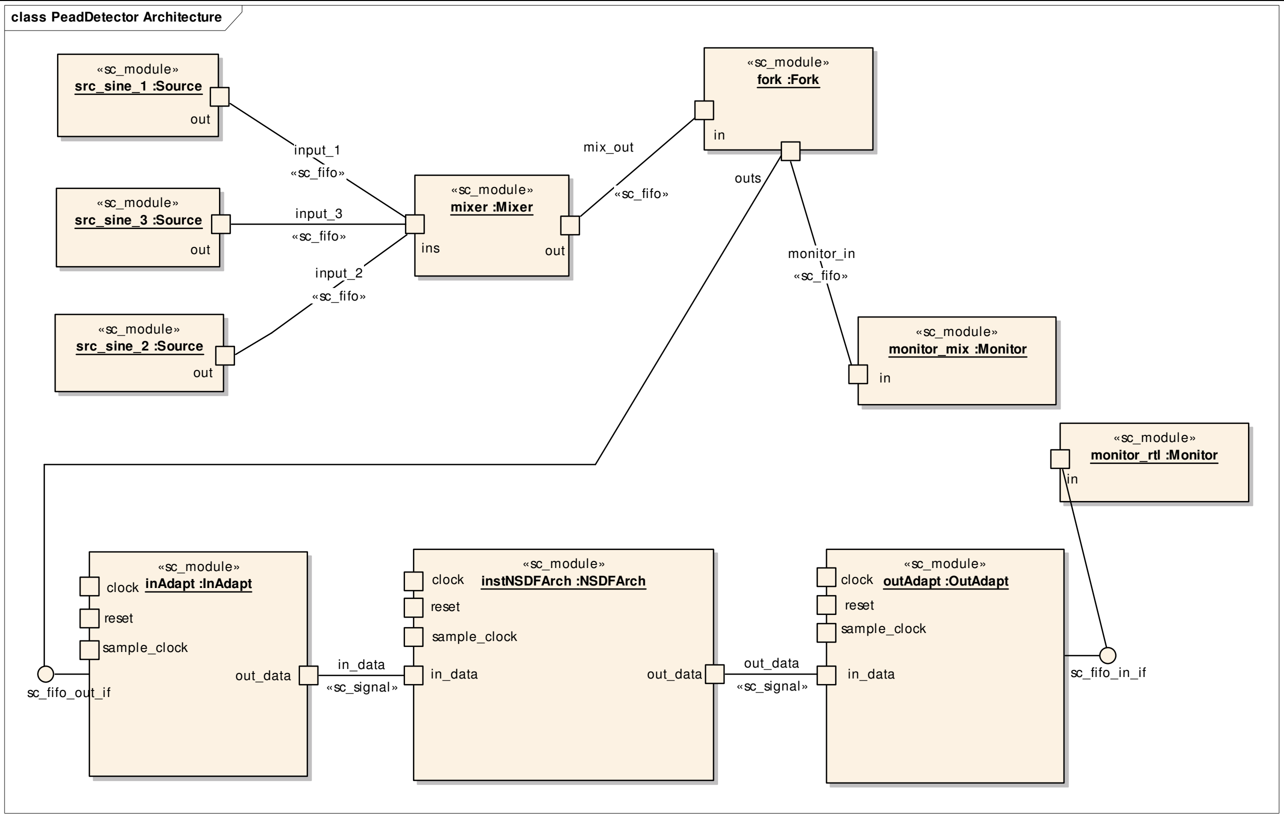


Figure 17 Complete NSDF architecture model with hardware signals

The modules from the functional model are reused and adapters are added to transform the sample stream from the fifo buffers to the physical simulated signals. This idea is inspired from the chapter 7. "Interface and Channel Design" in [6].

The peak detecting algorithm as described in the functional Matlab model will be partitioned to software. The major reason for this decision is that finding the peak mostly involves comparing and iterating O(W) times of the NSDF buffer. The initial guess would be to perform the peak detecting algorithm in less than 10ms. Most of the algorithm is using conditional statements more suited for software than hardware. It could also be that this part of the algorithm will be changed later which is easier in software. Implementation in hardware is more time consuming than software and therefore in this case software will be the best choice.

In the following pages is shown the code snippets of the architectural SystemC model for the NSDF module. A number of ASCII text files with stimuli inputs and results for computation of ACF, SDF and NSDF is generated and used later as a golden reference for verification of the hardware implementation.

```
template <class T, unsigned N>
class InAdapter: public sc_fifo_out_if <T >, public sc_module
{

  public:

    sc_in<bool> clock; // Clock
    sc_in<bool> reset; // Reset
    sc_in<bool> sample_clock; // Sample clock
    sc_out<sc_int<N> >  out_data; // Output port

    void write (const T & data)
    {
        double ddata;
        int idata;
        if (reset == false)
        {
            // Output sample data on negative edge of clock
            wait(sample_clock.negedge_event());
            wait(clock.posedge_event());
            ddata = (double)data;
            //cout << "Double: "<< ddata << endl;
            idata = float2fixed(ddata);
            idata = idata >> DIFF_BITS; // Size to 24 bits
            out_data.write(idata);
        }
        else wait(clock.posedge_event());
    }
```

Figure 18 Input adapter module that converts the fifo write operation to bus signals

```
class NSDFArch : public sc_module
{
public:
    sc_in<bool> clock; // Clock
    sc_in<bool> reset; // Reset
    sc_in<bool> sample_clock; // Sample clock
    sc_in<sc_int<ALGO_BITS> >  in_data; // Input port
    sc_out<sc_int<ALGO_BITS> > out_data; // Output port

public:

    SC_CTOR(NSDFArch)
    {
        SC_THREAD(DoCalculate);
        sensitive_pos << sample_clock;
        idx = 0;
    }

private:

    unsigned idx;
    sc_int<ALGO_BITS> m_x[WINDOW];  // Window of input samples
    sc_int<ALGO_BITS> r_tau[WINDOW]; // Autocorrelation ACF type II (ACF)
    sc_int<ALGO_BITS> m_tau[WINDOW]; // Squared difference function modified SDF (SDF)
    sc_int<ALGO_BITS> n_tau[WINDOW]; // Normalized squared difference function (NSDF)
    sc_int<ALGO_BITS> sample;

    void DoCalculate();
    void PrintBinary(sc_int<ALGO_BITS> sample, ofstream* file);
};
```

Figure 19 Architecture model of the NSDF module

```
// Name of test files
const std::string nsdf_file("output_nsdf.txt");
const std::string nsdf_fixed_file("output_nsdf_fix.txt");
const std::string acf_file("output_acf.txt");
const std::string acf_fixed_file("output_acf_fix.txt");
const std::string sdf_file("output_sdf.txt");
const std::string sdf_fixed_file("output_sdf_fix.txt");
const std::string mix_file("input.txt");

void NSDFArch::PrintBinary(sc_int<ALGO_BITS> sample, ofstream* file)
{
    int bit;
    // Uses only ADC_BITS
    for (int i = ADC_BITS-1; i >= 0; i--)
    {
        bit = sample[i];
        *file << bit;
    }
    *file << endl;
}

void NSDFArch::DoCalculate()
{
    sc_int<ALGO_BITS> temp1;
    sc_int<ALGO_BITS> temp2;
    sc_int<ALGO_BITS> temp3;
    sc_int<ALGO_BITS> temp4;

    sc_int<ALGO_BITS*2> temp2d;
    sc_int<ALGO_BITS*2> temp3d;

    cout << "NSDFArch Started" << endl;

    ofstream *output_nsdf = new ofstream(nsdf_file.c_str());
    ofstream *output_nsdf_modelSim = new ofstream(nsdf_fixed_file.c_str());
    ofstream *output_acf = new ofstream(acf_file.c_str());
    ofstream *output_acf_modelSim = new ofstream(acf_fixed_file.c_str());
    ofstream *output_sdf = new ofstream(sdf_file.c_str());
    ofstream *output_sdf_modelSim = new ofstream(sdf_fixed_file.c_str());
    ofstream *input_modelSim = new ofstream(mix_file.c_str());

    if (!output_nsdf)
    {
```

Figure 20 Test files generated by the NSDF module

```
if (idx == WINDOW)
{
    cout << " Calculating NSDF..." << endl;

    // Compute normalized squared difference function
    for (unsigned tau = 0; tau < WINDOW; tau++)
    {
        n_tau[tau] = m_x[tau];

        for (unsigned j = 0; j < WINDOW-1-tau; j++)
        {
            temp1 = r_tau[tau];
            temp2d = m_x[j]*m_x[j+tau];

            //cout << "m_xj[" << j << "] = " << m_x[j] << endl;
            //cout << "m_xj_tau[" << j+tau << "] = " << m_x[j+tau] << endl;

            temp2 = temp2d >> ALGO_BITS;
            r_tau[tau] = temp1 + temp2; // ACF type II (2)

            //cout << "r_tau[" << tau << "] = " << r_tau[tau] << endl;

            temp1 = m_tau[tau];
            temp2d = m_x[j]*m_x[j];
            temp2 = temp2d >> ALGO_BITS;
            temp3d = m_x[j+tau]*m_x[j+tau];
            temp3 = temp3d >> ALGO_BITS;

            temp4 = temp1 + temp2;
            m_tau[tau] = temp4 + temp3; // SDF modified (6)

            //cout << "m_tau[" << tau << "] = " << m_tau[tau] << endl << endl;

            // Back annotation from RTL timing analysis (Quartus)
            //wait(29, SC_NS); // Combinatorial
            wait(20, SC_NS); // Pipelined optimized
        }
    }
```

Figure 21 Refined computation of NSDF using integer arithmetic

Wait operations is inserted with back annotations of times measured later in design of hardware IP block.

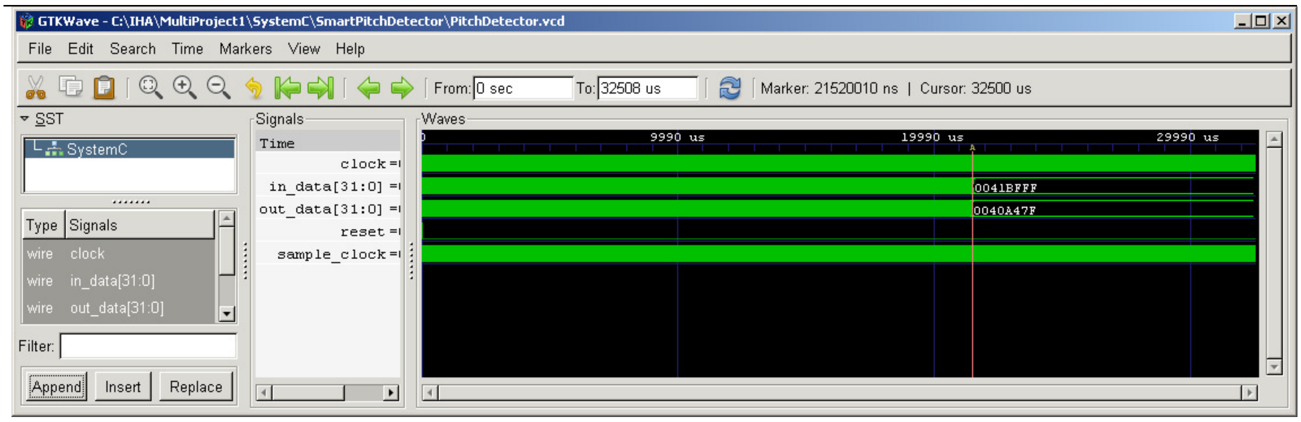


Figure 22 Wave view of the interface signals to the NDSF module

It is possible with SystemC to monitor a number of selected signals by generating wave files. The [1]GTKWave viewer can be used to measure the timing of the signals in the generated wave files. In the above example we can see that the collection of the 1024 samples takes 21.5 ms and the NSDF computation takes 32.5 – 21.5 = **11 ms**. The NSDF computation is based on inserting wait operations into the model based on the back annotated estimates found later during the hardware design phase.

The signals to be logged in the wave file are specified in the architectural top level module (Arch.cpp) see code below.

```
// Create tacefile
tracefile = sc_create_vcd_trace_file("PitchDetector");
if (!tracefile) cout << "Could not create trace file." << endl;
else cout << "Created PitchDetector.vcd" << endl;

// Set resolution of trace file to be in 1 NS
tracefile->set_time_unit(1, SC_NS);

sc_trace(tracefile, clock.signal(), "clock");
sc_trace(tracefile, sample_clock.signal(), "sample_clock");
sc_trace(tracefile, reset, "reset");
sc_trace(tracefile, in_data, "in_data");
sc_trace(tracefile, out_data, "out_data");
```

[1] GTKWave viewer is an open source project that can be found here: http://gtkwave.sourceforge.net/

3.3 Platform design and FPGA prototype

It is specified to use the Altera DE2 Development and Education board. This board features all the peripherals needed including the Cyclone II EP2C35F672C6 FPGA. A soft-core processor is a collection of component, written in a hardware description language (HDL), which can be synthesized for different programmable FPGA devices. The commonly used soft-core on the Altera platforms are the Nios II processor which can be customized to fit the purpose at hand by using the SOPC[2] Builder software. The components on the DE2 Board can be included in the SOPC Builder design. This *component-base design* approach is described in [21] where three different system level design approaches is defined: HW/SW Co-design, Platform-based design and component-based design.

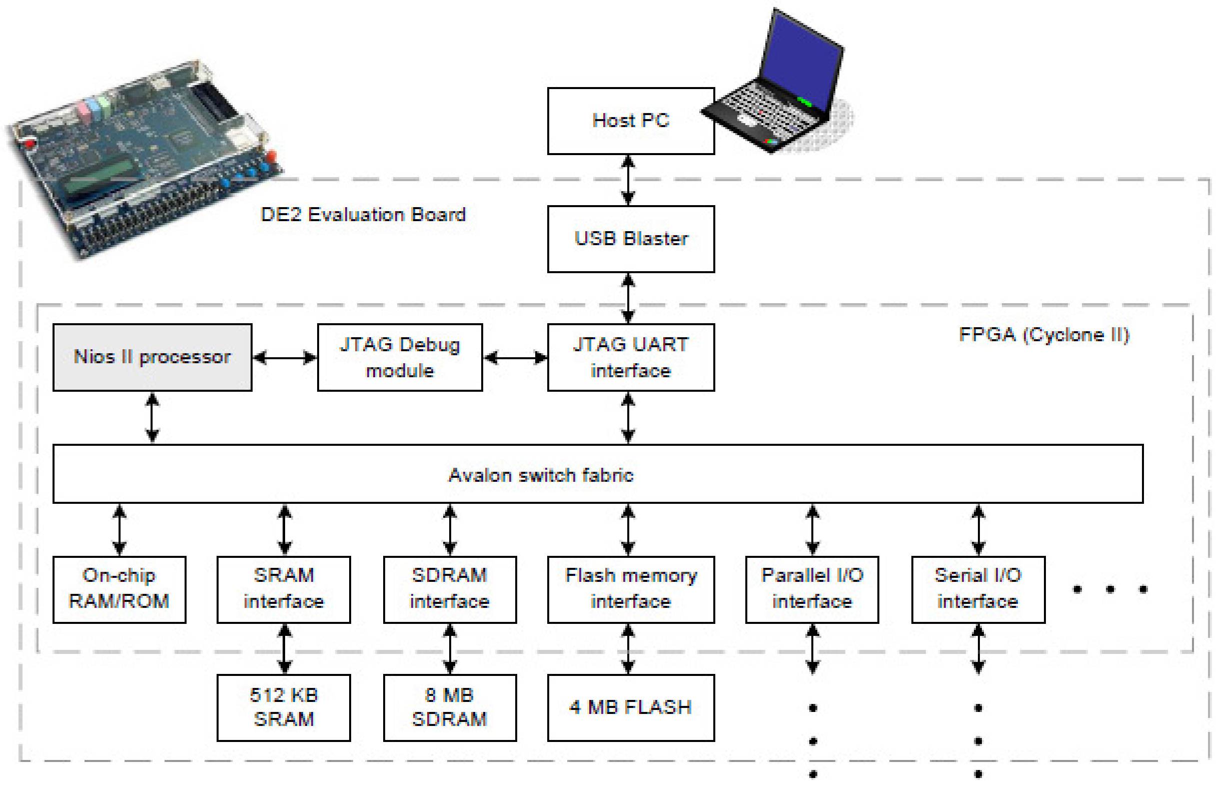


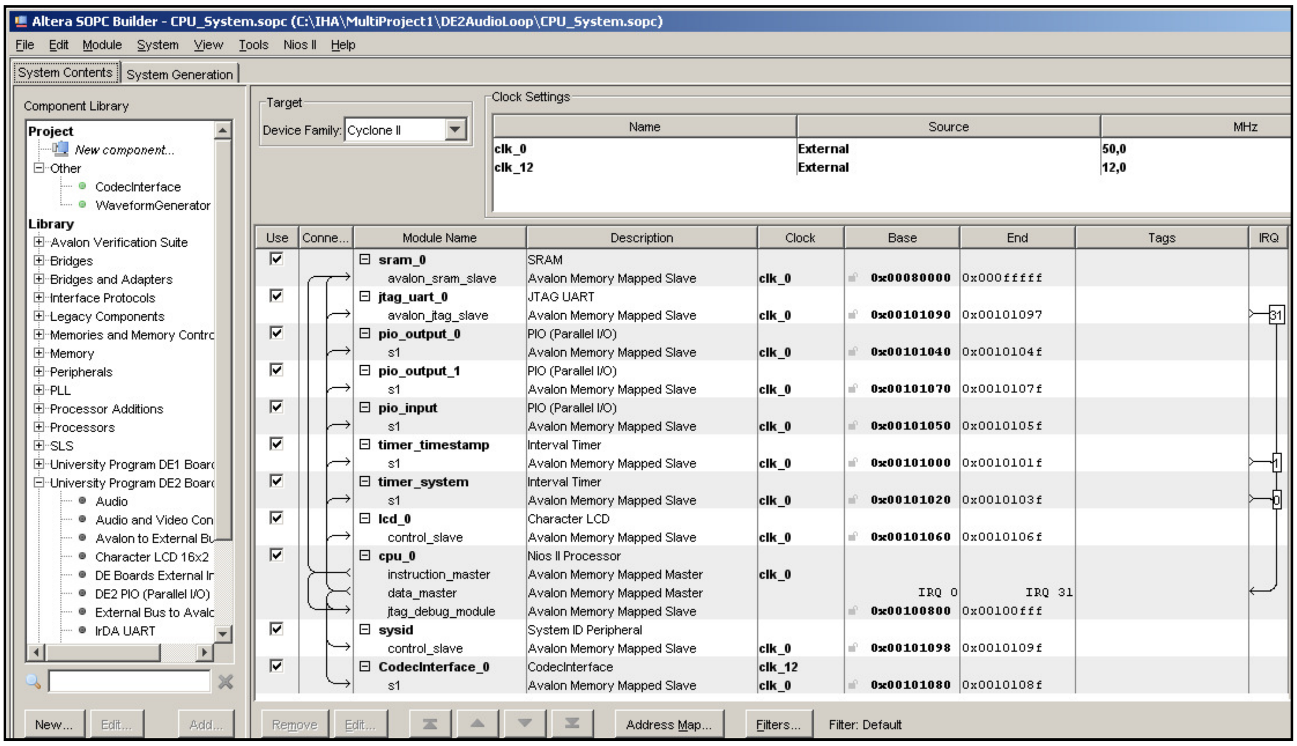
Figure 23 Peripherals integration on the DE2-Board as IP Cores using the Altera SOPC Builder software

The first prototype design includes the below IP Core components.

- Nios II processor - Core type: Nios II/s
- 512 Kbyte SRAM
- JTAG UART
- System timer
- Time stamp timer

[2] SOPC – System On Programmel Chip

- LCD controller
- Audio Codec Interface
- 2*PIO input (2*8 LEDs)
- PIO output (8 Switch inputs)


Altera SOPC Builder - CPU_System.sopc (C:\IHA\MultiProject1\DE2AudioLoop\CPU_System.sopc)

Target — Device Family: Cyclone II

Clock Settings

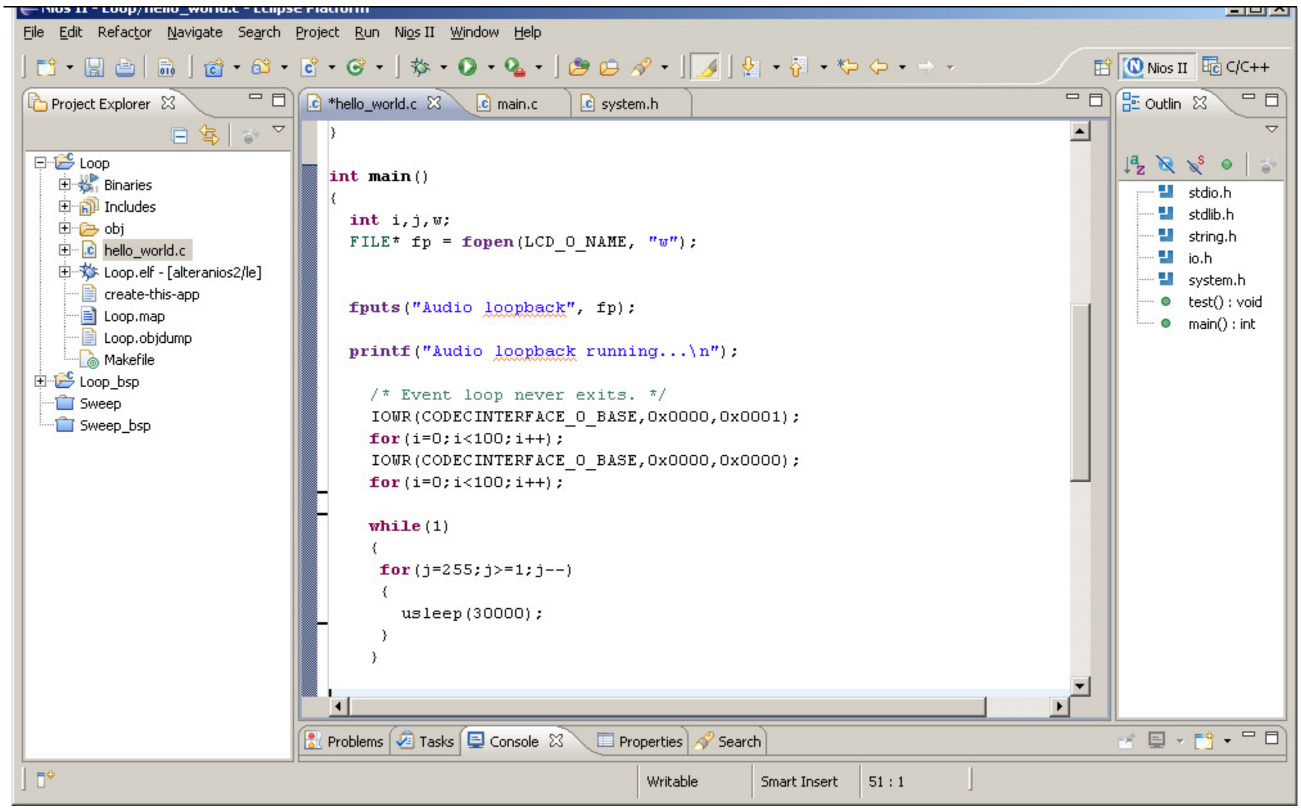
Name	Source	MHz
clk_0	External	50,0
clk_12	External	12,0

Use	Module Name	Description	Clock	Base	End	IRQ
✓	**sram_0**	SRAM				
	avalon_sram_slave	Avalon Memory Mapped Slave	clk_0	0x00080000	0x000fffff	
✓	**jtag_uart_0**	JTAG UART				
	avalon_jtag_slave	Avalon Memory Mapped Slave	clk_0	0x00101090	0x00101097	31
✓	**pio_output_0**	PIO (Parallel I/O)				
	s1	Avalon Memory Mapped Slave	clk_0	0x00101040	0x0010104f	
✓	**pio_output_1**	PIO (Parallel I/O)				
	s1	Avalon Memory Mapped Slave	clk_0	0x00101070	0x0010107f	
✓	**pio_input**	PIO (Parallel I/O)				
	s1	Avalon Memory Mapped Slave	clk_0	0x00101050	0x0010105f	
✓	**timer_timestamp**	Interval Timer				
	s1	Avalon Memory Mapped Slave	clk_0	0x00101000	0x0010101f	1
✓	**timer_system**	Interval Timer				
	s1	Avalon Memory Mapped Slave	clk_0	0x00101020	0x0010103f	0
✓	**lcd_0**	Character LCD				
	control_slave	Avalon Memory Mapped Slave	clk_0	0x00101060	0x0010106f	
✓	**cpu_0**	Nios II Processor				
	instruction_master	Avalon Memory Mapped Master	clk_0			
	data_master	Avalon Memory Mapped Master		IRQ 0	IRQ 31	
	jtag_debug_module	Avalon Memory Mapped Slave		0x00100800	0x00100fff	
✓	**sysid**	System ID Peripheral				
	control_slave	Avalon Memory Mapped Slave	clk_0	0x00101098	0x0010109f	
✓	**CodecInterface_0**	CodecInterface	clk_12			
	s1	Avalon Memory Mapped Slave	clk_0	0x00101080	0x0010108f	

Filter: Default

Figure 24 SOPC Builder of the audio loop back prototype

The SoPC design is used to test looping audio from input to output controlled by a small main program that initializes the codec interface. This C program is compiled and send to the Nios II/s using the Altera Nios II IDE Eclipse based program see Figure 25.



```c
}

int main()
{
  int i,j,w;
  FILE* fp = fopen(LCD_0_NAME, "w");


  fputs("Audio loopback", fp);

  printf("Audio loopback running...\n");

    /* Event loop never exits. */
    IOWR(CODECINTERFACE_0_BASE,0x0000,0x0001);
    for(i=0;i<100;i++);
    IOWR(CODECINTERFACE_0_BASE,0x0000,0x0000);
    for(i=0;i<100;i++);

    while(1)
    {
     for(j=255;j>=1;j--)
     {
       usleep(30000);
     }
    }
```

Figure 25 C main program that initialize the coded interface

The main program starts the audio streaming by writing to the memory map of this codec interface. Hereafter it just loops without doing anything. All audio streaming is done in hardware.

3.4 Hardware design and verification

In designing the architecture for the SoPC platform for the peak detector we will use the block diagrams [22] from SysML to describe the platform architecture. SysML blocks are modular units that can be used to define system components or items that flows through the system. In this project the block definition diagram is used to define the block characteristics in terms of their structure and the hierarchical relationship. The blocks relate to the IP Core components available with the SOPC Builder of Altera and new blocks that will be developed in this project.

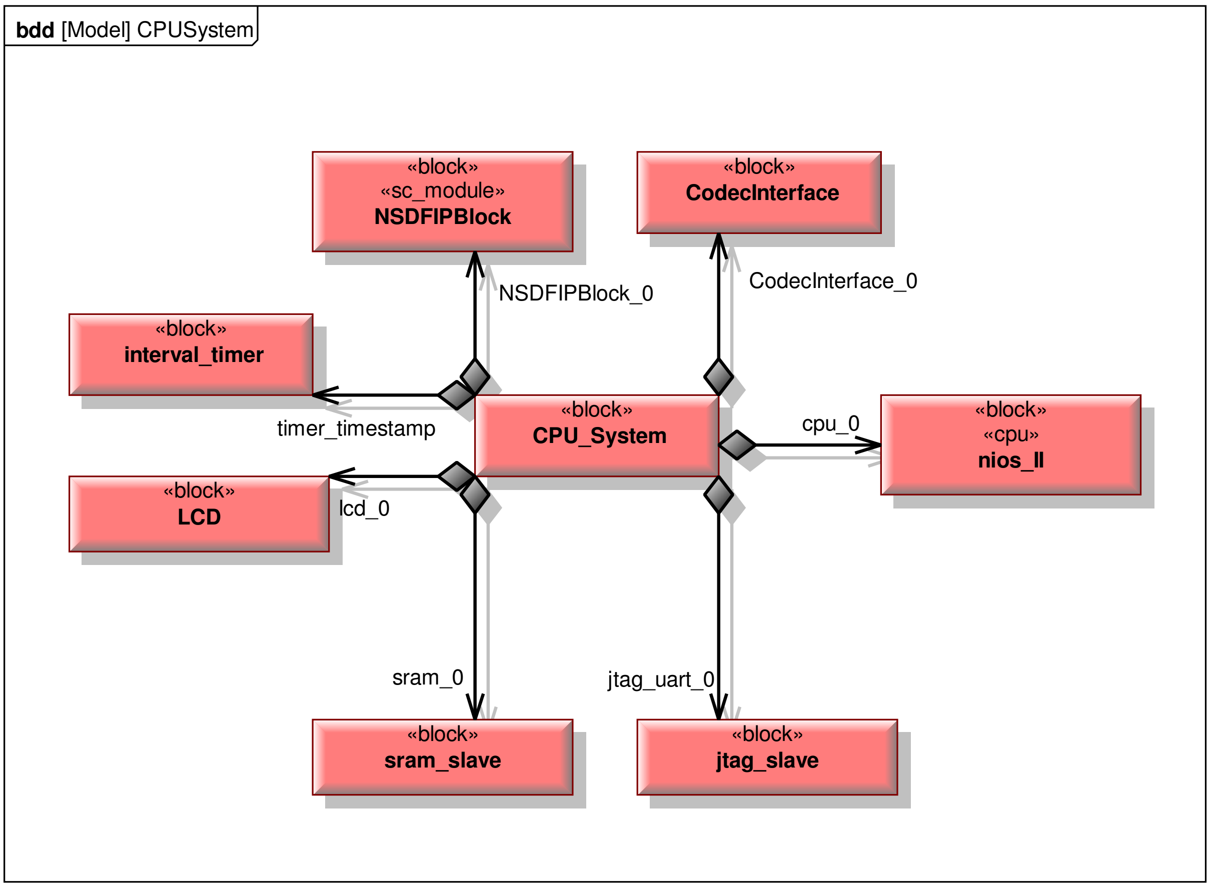


Figure 26 SysML block definition diagram for the SoPC design of the pitch detector

The block definition diagram (BDD) illustrated above defines the CPU_System that consist of a number of IP Core blocks. The CodecInterface[3] and the NSDFIPBlock are the new blocks to be used for the pitch detector design. The NSDFIPBlock is the module described in the SystemC architectural model. The CodecInterface block handles the interface to the audio codec from Wolfson Microelectronics (WM8731) part of the DE2 board. The block handles the control interface I2C and audio interface I2S to the audio codec. It configures the Audio Codec to 48 kHz/24 bit and converts the I2S audio interface to a 2 channel parallel interface. This parallel

[3] Open source IP block is used

interface between the CodeInterface and NSDFIPBLock is described in the internal block diagram (IBD) in Figure 27.

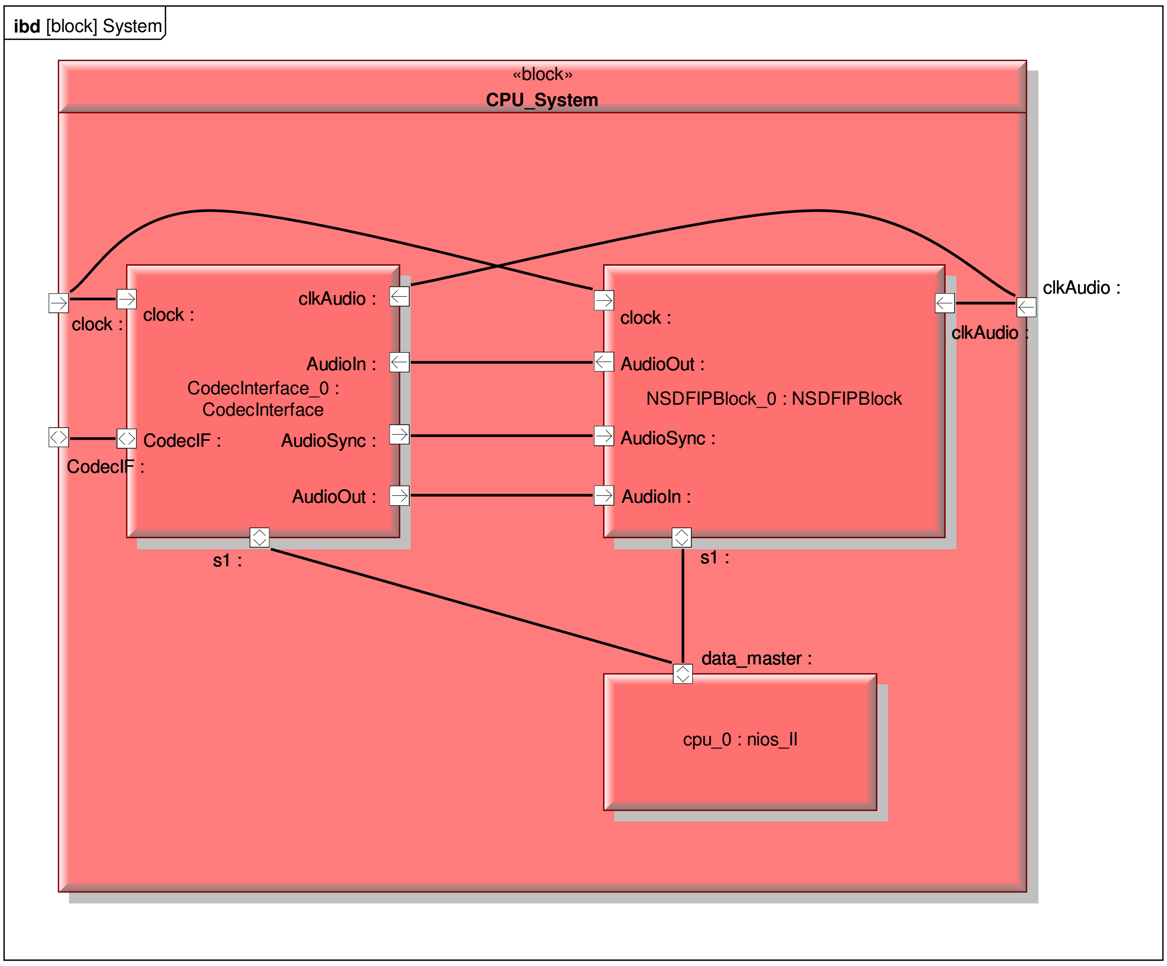


Figure 27 SysML internal block diagram of the parallel interface between CodecInterface and the NSDFIPBlock

The CodecInterface and NSDFIPBlock are memory mapped as slaves to the Nios II processor using the Avalon processor bus. The software will be able to read and write to the IP Core blocks. The CPU clock running 50 Mhz and a 12 Mhz audio clock is used to drive the sequential hardware design in sync with the Nios II processor. Flow ports from SysML are used to abstract the hardware signals between the IP Core blocks. The parallel audio interface is refined using atomic flow ports to specify the direction of the audio sample flow and the synchronization of the left and right audio samples. Nonatomic flow ports are used to define the Avelon slave (s1) interface and codec interface (CodecIF).

The codec interface will handling receiving and sending samples to and from the audio codec. The NSDF IP block receives audio samples and buffers them before computing the NSDF function. It signals the Nios II processer every time a new block is ready with the rate of window size/sample rate (1024/48000 = 21 ms).

3.4.1 NSDFIPBlock detailed design

This section presents a method in transforming the SystemC architectural model of the NSDFIPBlock into a hardware accelerator written in VHDL. The NSDF accelerator will be inserted into the SoPC design described in the previous chapter.

The architectural SystemC model is used as a reference in creating the design of the NSDF IP block. The computation of the NSDF is done in two loops see Figure 13. The first loop will be called the "inner loop1" computing m_tau and r_tau. The second "loop2" computes n_tau based on the m_tau and r_tau temporary buffers.

The inner loop1 of calculating the m_tau and r_tau takes O(W*w) times.

For W = 1024 where w = (W-1)/2 we have:

O(W*w) = 523264

If we choose the solution using the audio clock (fclka = 12 Mhz) at the sample speed of fs (48 kHz) we have the following number of clock cycles to calculate r_tau and m_tau with a window of W = 1024:

Capacity = fclka/fs * W = 12 Mhz / 48 kHz * 1024 = 256000

In this solution we will not have sufficient cycles to compute the inner loop1. We could unroll the inner loop1 and make 4 calculations of r_tau and m_tau in parallel this would reduce the **O(W*w)/4 = 130816**. The solution will increase the area usage of the FPGA resources. The Cyclone II FPGA used in the DE2 board has 70 of 9 bits multipliers. Using a resolution of 32 bits in the NSDF calculations makes usage of 4 multipliers for every multiplication. In calculation of the inner loop1 it will require 4*4=16 multipliers see the data flow graph (DFG) in Figure 29 and Figure 28 for calculation of the inner loop1.

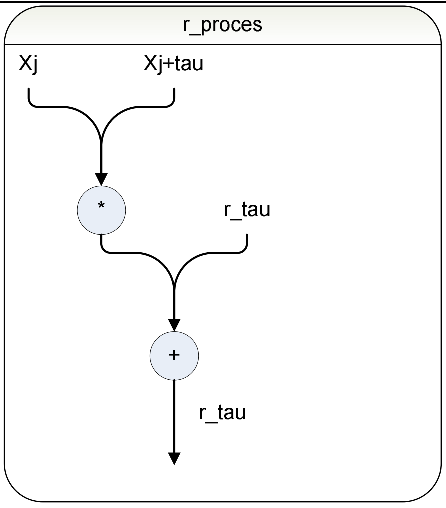


Figure 28 Data flow graph computation of r_tau

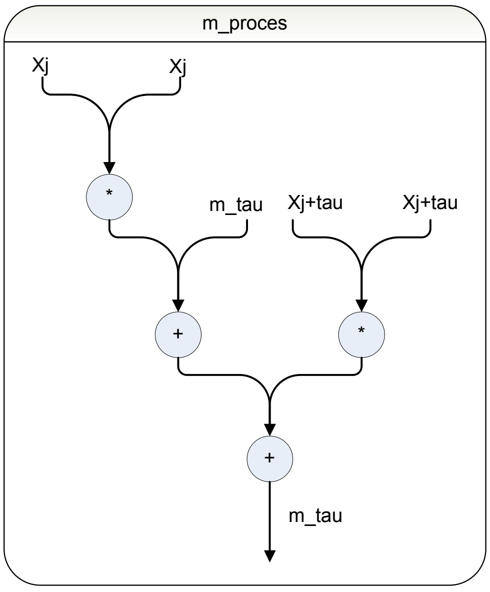


Figure 29 Data flow graph computation of m_tau

Running four r_tau and m_tau in parallel will be possible using 64 or nearly all 70 multipliers. It will also demand a very complex control logic to control four parallel data path's for computation of the inner loop1. The system clock (fclks = 50 Mhz) could be used in calculation of the NSDF instead of the 12 Mhz audio clock. In this case we will have sufficient capacity:

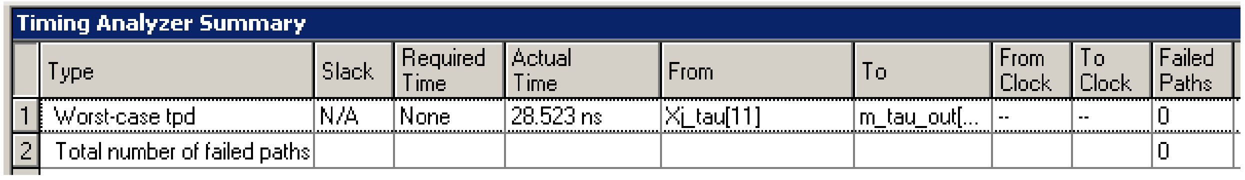
Capacity = fclks/fs * W = 50 Mhz / 48 kHz * 1024 = 1066667

Now we have plenty of room in calculating the NSDF. The challenge using the 50 Mhz clock is to ensure that the propagation delay for computation of the data path do not exceed the cycle period of 20 ns. The second challenge is the use of different clock domains. The 50 Mhz CPU clock and 12 Mhz audio clock could lead into problems if not designed correctly.

The first step is to implement the VHDL component for the inner loop1 calculating m_tau and r_tau see VHDL code snippet below:

```vhdl
architecture behaviour of ACF_SDF is

  -- Constant Declarations
  constant W1  : integer := 32;
  constant W2  : integer := W1*2;

  -- Type Declarations
  subtype typeW1 is signed(W1-1 downto 0);
  subtype typeW2 is signed(W2-1 downto 0);

  -- Internal signals
  signal r_tau : typeW1;
  signal m_tau : typeW1;

begin  -- behaviour

-- purpose: ACF_SDF Loop calculation

  r_pro : process (Xj, Xj_tau, r_tau_in)
  variable tmp : typeW2;
  begin
      tmp := signed(Xj) * signed(Xj_tau);
      r_tau <= tmp(W2-1 downto W1) + signed(r_tau_in);
  end process r_pro;

  m_pro : process (Xj, Xj_tau, m_tau_in)
  variable tmp1 : typeW2;
  variable tmp2 : typeW2;
  begin
      tmp1 := signed(Xj) * signed(Xj);
      tmp2 := signed(Xj_tau) * signed(Xj_tau);
      m_tau <= tmp1(W2-1 downto W1) + tmp2(W2-1 downto W1) + signed(m_tau_in);
  end process m_pro;

  r_tau_out <= std_logic_vector(r_tau);
  m_tau_out <= std_logic_vector(m_tau);

end behaviour;
```

Analyzing the synthesized VHDL code using the RTL viewer and the timing information after the post mapping we find that the inner loop1 computation of the r_tau and m_tau takes **28.523 ns.**



Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
1	Worst-case tpd	N/A	None	28.523 ns	Xj_tau[11]	m_tau_out[...	--	--	0
2	Total number of failed paths								0

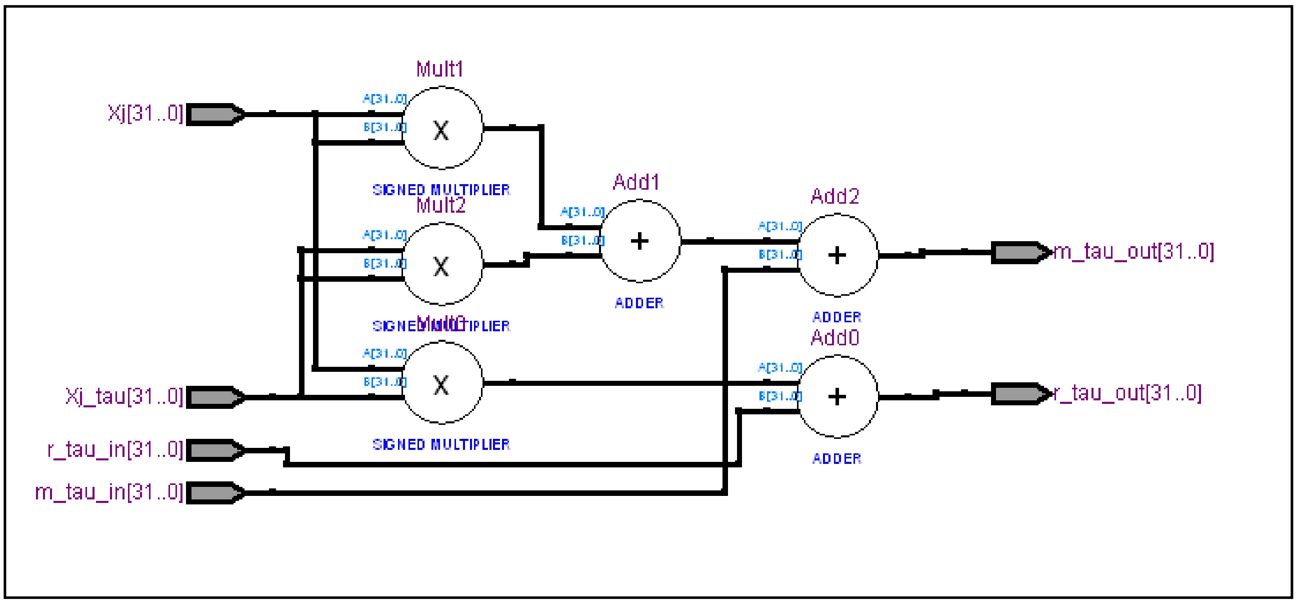


Figure 30 Combinatorial design, computation of m_tau and r_tau

We must maximum have a propagation delay of 20 ns in to compute m_tau and r_tau therefore a sequential pipelined version of the ACF_SDF block is needed. See VHDL code snippet below:

```
  -- Internal signals
  signal r_tau : typeW1;
  signal m_tau : typeW1;
  signal tmp   : typeW2;
  signal tmp1  : typeW2;
  signal tmp2  : typeW2;


begin  -- behaviour

-- purpose: ACF_SDF Loop calculation

  r_pro : process (clock, clken)
  variable acf : typeW1;
  begin
    if clken = '0' then                -- asynchronous reset (active low)
      r_tau <= (others => '0');
      tmp <= (others => '0');
    elsif rising_edge(clock) then  -- rising clock edge
      if clear = '1' then
         tmp <= (others => '0');
         r_tau <= (others => '0');
      else
         acf := r_tau;
         tmp <= signed(Xj) * signed(Xj_tau);
         r_tau <= tmp(W2-1 downto W1) + acf;
      end if;
    end if;
  end process r_pro;
```

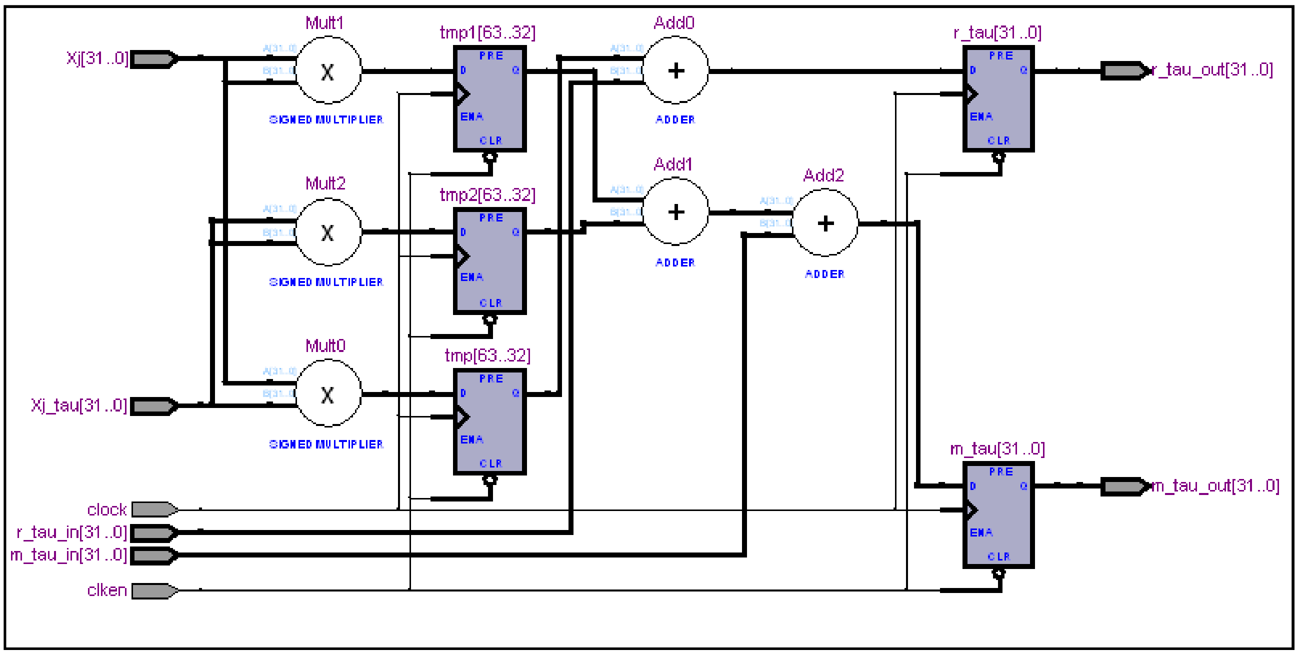


Figure 31 Sequential pipelined design, computation of r_tau and m_tau

We can see from the output of the RTL viewer that we now have inserted flip-flops to store the temporary results in tmp1, tmp2 and tmp. The computation of r_tau and m_tau has been pipelined using 2 CPU clock cycles to compute.

Computation of the n_tau in loop2 uses the results from the inner loop1 stored in the r_tau and m_tau buffers see data flow graph below:

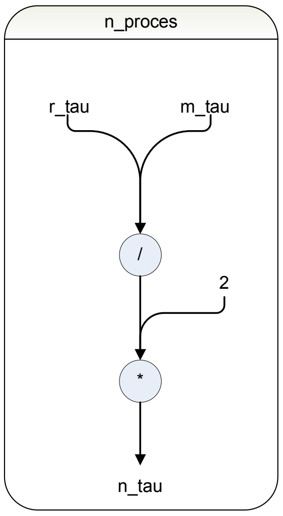
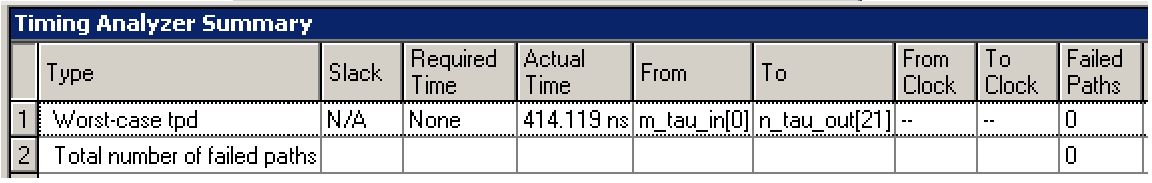


Figure 32 Data flow graph computation of n_tau

Implementation of this part in VHDL takes 414 ns, because division is not directly supported in hardware the VHDL synthesizes automatically inserts a divider IP Core component.

```
architecture behaviour of NSDF is

 -- Constant Declarations
 constant W1  : integer := 32;
 constant W2  : integer := W1*2;

 signal div : signed (W2-1 downto 0);
begin  -- behaviour

  -- purpose: NSDF Second loop calculation
  div <= shift_left(resize(signed(r_tau_in), W2), W1-1) / resize(signed(m_tau_in), W2);
  n_tau_out <= std_logic_vector( shift_left(div(W1-1 downto 0), 1) );

end behaviour;
```


Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
1	Worst-case tpd	N/A	None	414.119 ns	m_tau_in[0]	n_tau_out[21]	--	--	0
2	Total number of failed paths								0

The above solution will not work with a clock speed of 50 Mhz. Instead a customized pipelined version of a division must be used see VHDL code snippet below:

```
begin  -- behaviour

  -- purpose: NSDF Second loop calculation

  r_tau_in_w2s <= shift_left(resize(signed(r_tau_in), W2), W1-1);
  -- Divide by '1' if m_tau is zero
  m_tau_in_w1s <= (0 =>'1', others => '0') when signed(m_tau_in) = 0 else signed(m_tau_in);
  n_tau_out <= std_logic_vector(shift_left(signed(n_tau_out_w2s(W1-1 downto 0)), 1))
               when clken = '1' else (others => '0');

  -----------------------------------------------------------------------------
  -- Component computing NSDF (Takes approx 500 ns - 25 pipelined)
  -----------------------------------------------------------------------------
  div64 : entity work.Divide64 port map(
              clken => clken,
              clock => clock,
              denom => std_logic_vector(m_tau_in_w1s),
              numer => std_logic_vector(r_tau_in_w2s),
              quotient => n_tau_out_w2s,
              remain => remain);

end behaviour;
```

This pipelined version of the 64 bits divide operation is computed in 25 cycles. A control state machine will be designed using the 50 Mhz clock and computes the inner loop1 as a pipelined control. The final computation of the NSDF in loop2 will be done using 25 wait states calculating the final NSDF which means we have:

Inner loop1 compute time = 1/fclks * W*W/2 = 1 / 50 Mhz * 1024*1024/2 = **10.5 ms**

Loop2 compute time = 1/fclks * 25 * W = 1 / 50 Mhz * 25 * **1024** = **512 us**

Total estimated NSDF compute time = 11.0 ms

The typical design of a sequential HW IP Block is to have finite state machine with data ([4]FSMD) that controls the data path of the computation. Beside the FSMD and the computation of r_tau, m_tau and n_tau we need temporary memory to store the input sample data and buffers. The block diagram below shows the composition of the NSDFIPBlock and the HW blocks used to handle the data path of the design.

[4] FSMD is described by Gajski in [9]

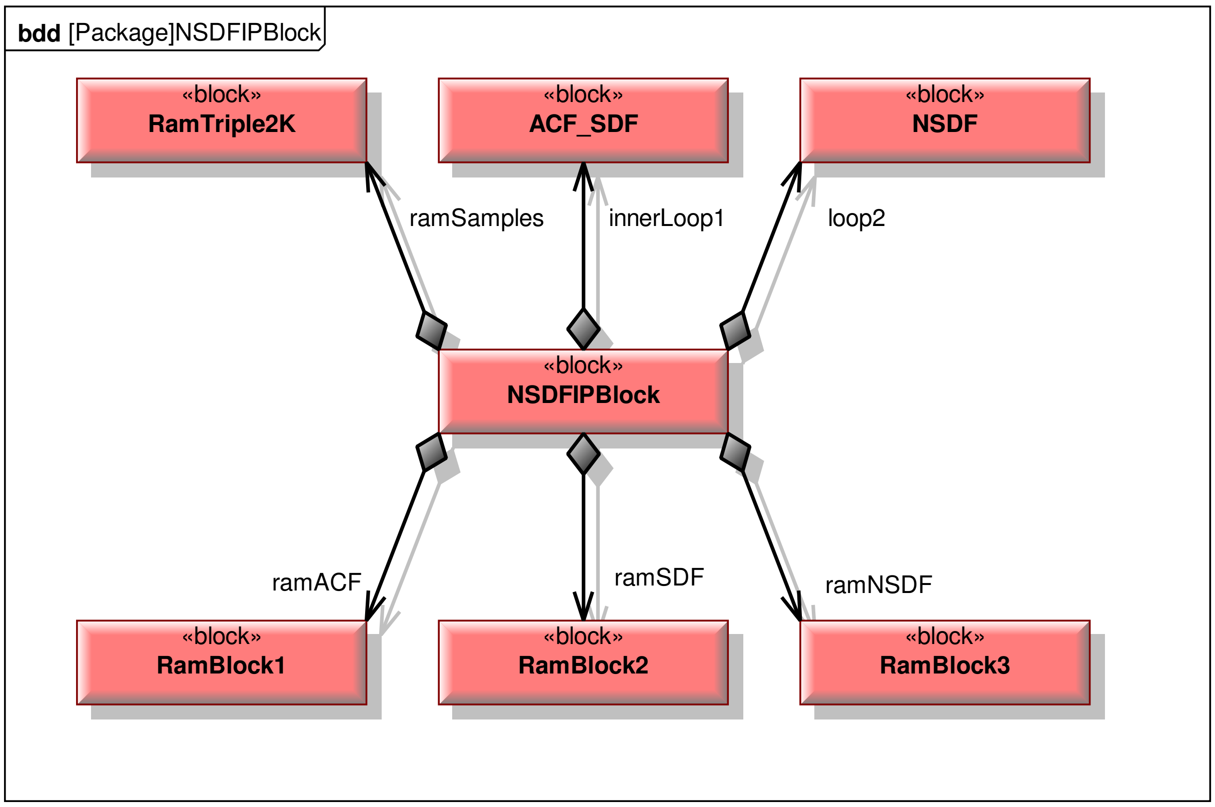


Figure 33 SysML block definition diagram for the composition of the NSDFIPBlock

All ram blocks uses the internal M4K ram blocks part of the Cyclone II FPGA. The RamBlock1-3 is configured to store 1024 words of 32 bits. The RamTriple2K is configured to store 2048 works of 32 bits using a dual buffer. This block is composed to contain one buffer for writing new samples and one buffer for reading and processing the NDSF computation. In this way it will be possible to compute the NDSF in parallel while buffering new samples.

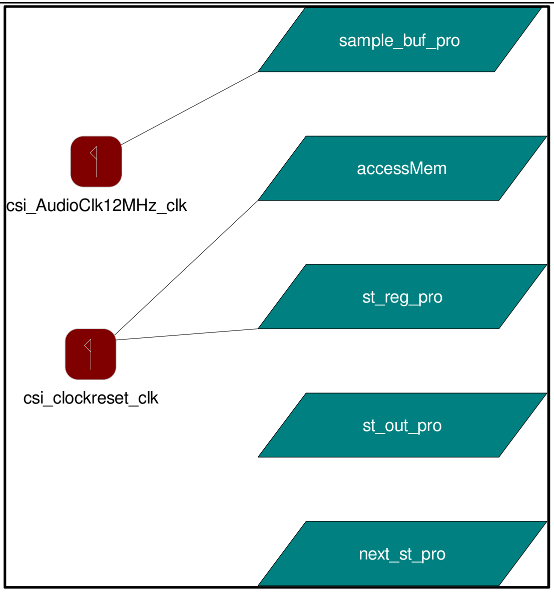


Figure 34 Concurrent diagram, Behavioral processes in the NSDFIPBlock

The NSDFIPBlock is composed by 5 parallel VHDL behavioral processes as illustrated in the concurrent diagram in Figure 34. Sample_buf_pro, accessMem and st_reg_pro are clocked processes.

Sample_buf_pro handles buffering of samples. AccessMem handles the memory mapped registers that can be accessed from the Nios II processor that includes status, control and reading the ramNSDF buffer. St_reg_pro, st_out_pro, st_next_pro are the processes that control the data path implementing the finite state machine FSMD described in Figure 35.

The FSMD is operates in 3 main states. In idle state it waits for a new block of 1024 samples. The signal buf_ready indicates a new sample buffer is ready in the ramSamples block. The pre and post states are used in pre and post processing of the pipeline for computation of the inner loop1 and loop2. Further details of the FSMD implementation are to be found in the source code of the VHDL file: NSFD_IPBlock.vhd.

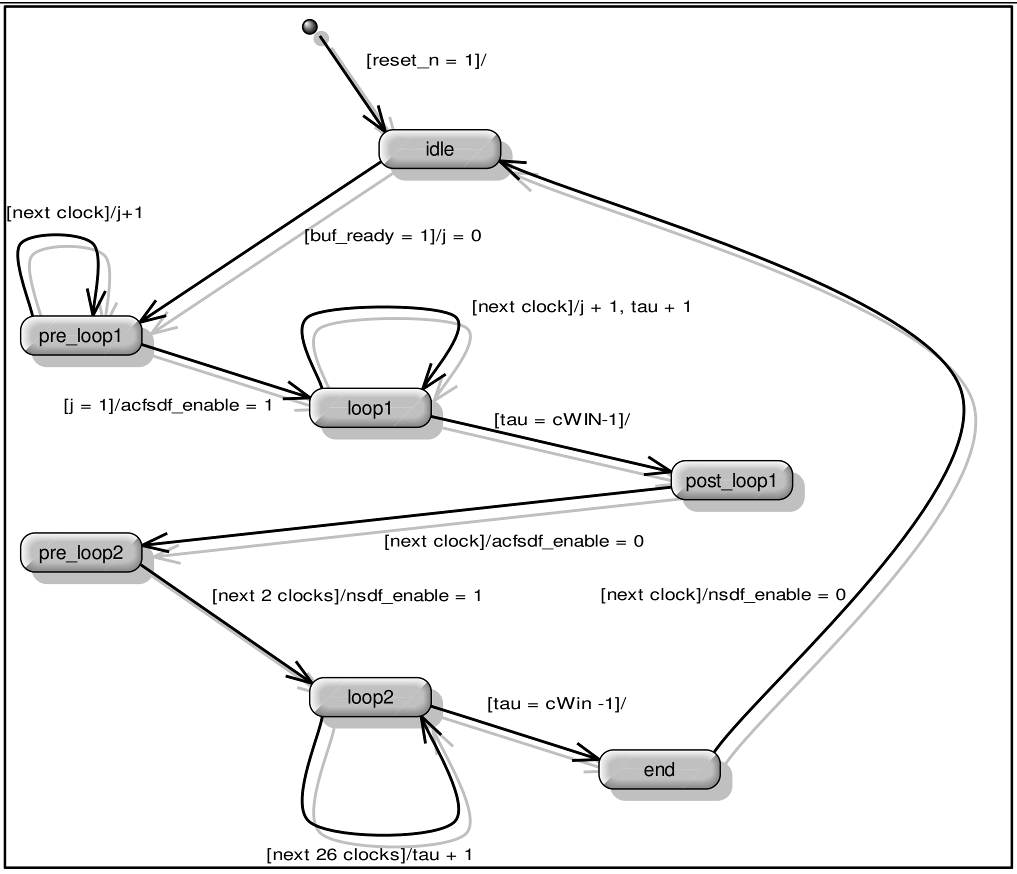


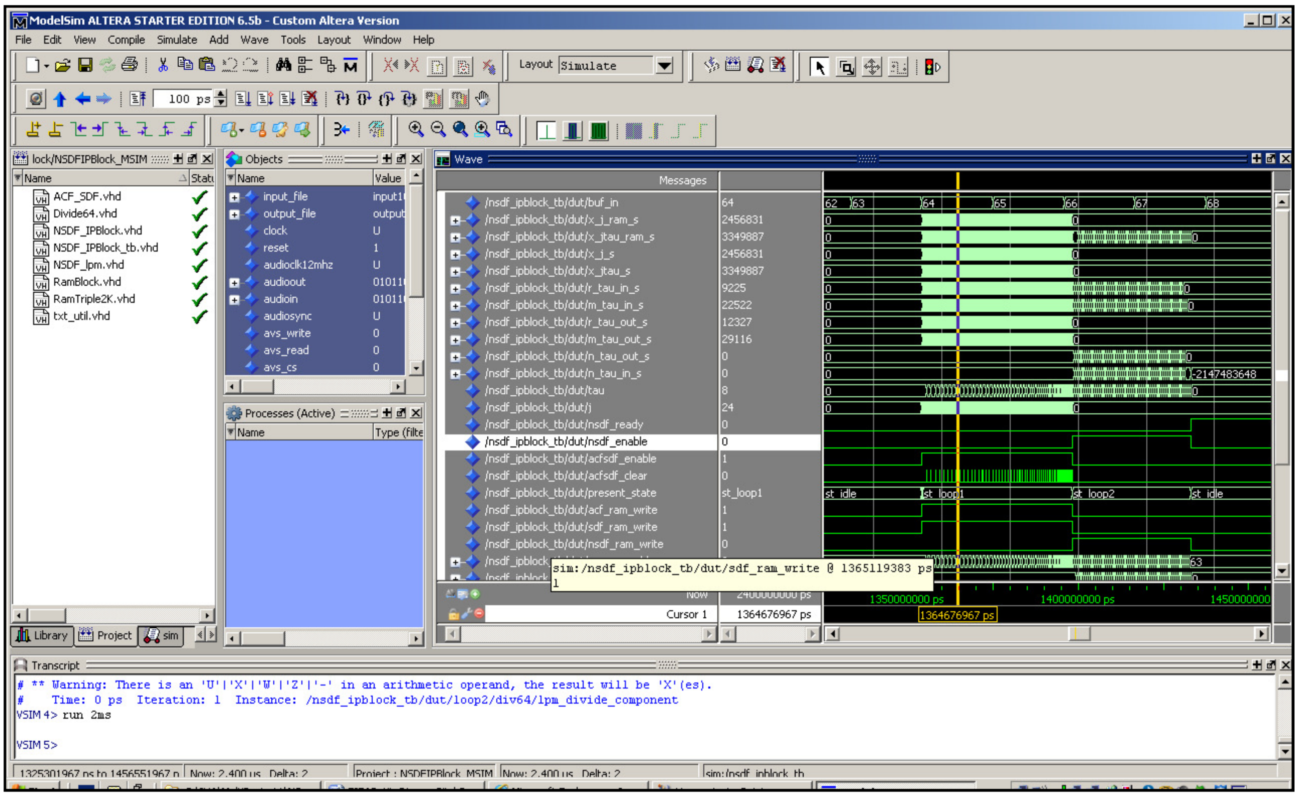
Figure 35 Finite State Machine (FSM) for controlling the data path of the NSDFIPBlock

The VHDL entity description of the NSDFIPBlock is illustrated below.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity NSDF_IPBlock is

  port (
    -- Audio Interface
    csi_AudioClk12MHz_clk       : in  std_logic;                    -- 12MHz Clk
    csi_AudioClk12MHz_reset_n   : in  std_logic;                    -- 12MHz Clk
    coe_AudioIn_export          : in  std_logic_vector(23 downto 0); -- To Codec
    coe_AudioOut_export         : out std_logic_vector(23 downto 0); -- From Codec
    coe_AudioSync_export        : in  std_logic;                    -- 48KHz Sync
    -- Avalon Interface
    csi_clockreset_clk          : in    std_logic;                  -- Avalon Clk 50 Mhz
    csi_clockreset_reset_n      : in    std_logic;                  -- Avalon Reset
    avs_s1_write                : in    std_logic;                  -- Avalon wr
    avs_s1_read                 : in    std_logic;                  -- Avalon rd
    avs_s1_chipselect           : in    std_logic;                  -- Avalon Chip Select
    avs_s1_address              : in    std_logic_vector(11 downto 0); -- Avalon address
    avs_s1_writedata            : in    std_logic_vector(31 downto 0); -- Avalon wr data
    avs_s1_readdata             : out   std_logic_vector(31 downto 0)  -- Avalon rd data
    );

end NSDF_IPBlock;
```

3.4.2 NSDFIPBlock verification

The NSDFIPBlock is verified in the simulated environment using ModelSim. The files created in the architectural model are used to create stimuli and monitor the result of the simulation using the wave view editor in ModelSim. The stimuli files contains samples in binary format and a test bench is written in VHDL to test the design under test (DUT) as described in chapter 7.1 in [9]. In the figures below is illustrated some of the wave views used to test and verify the computation of the inner loop1 and loop2 controlled by the FSM as described the previous chapter.


ModelSim ALTERA STARTER EDITION 6.5b - Custom Altera Version
File Edit View Compile Simulate Add Wave Tools Layout Window Help
Layout Simulate
Wave
Objects
Processes (Active)
Library
Project
sim
Transcript

```
/nsdf_ipblock_tb/dut/buf_in            64
/nsdf_ipblock_tb/dut/x_j_ram_s         2456831
/nsdf_ipblock_tb/dut/x_jtau_ram_s      3349887
/nsdf_ipblock_tb/dut/x_j_s             2456831
/nsdf_ipblock_tb/dut/x_jtau_s          3349887
/nsdf_ipblock_tb/dut/r_tau_in_s        9225
/nsdf_ipblock_tb/dut/m_tau_in_s        22522
/nsdf_ipblock_tb/dut/r_tau_out_s       12327
/nsdf_ipblock_tb/dut/m_tau_out_s       29116
/nsdf_ipblock_tb/dut/n_tau_out_s       0
/nsdf_ipblock_tb/dut/n_tau_in_s        0
/nsdf_ipblock_tb/dut/tau               8
/nsdf_ipblock_tb/dut/j                 24
/nsdf_ipblock_tb/dut/nsdf_ready        0
/nsdf_ipblock_tb/dut/nsdf_enable       0
/nsdf_ipblock_tb/dut/acfsdf_enable     1
/nsdf_ipblock_tb/dut/acfsdf_clear      0
/nsdf_ipblock_tb/dut/present_state     st_loop1
/nsdf_ipblock_tb/dut/acf_ram_write     1
/nsdf_ipblock_tb/dut/sdf_ram_write     1
/nsdf_ipblock_tb/dut/nsdf_ram_write    0
sim:/nsdf_ipblock_tb/dut/sdf_ram_write @ 1365119383 ps
1
Cursor 1   1364676967 ps
```

```
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ps  Iteration: 1  Instance: /nsdf_ipblock_tb/dut/loop2/div64/lpm_divide_component
VSIM 4> run 2ms

VSIM 5>
```

Figure 36 ModelSim verification of the NSDF computation

The total computation time is measured in ModelSim to 11.1 ms. Few more cycles was needed in preparing access to the memory blocks RamBlock and RamTrible2K.

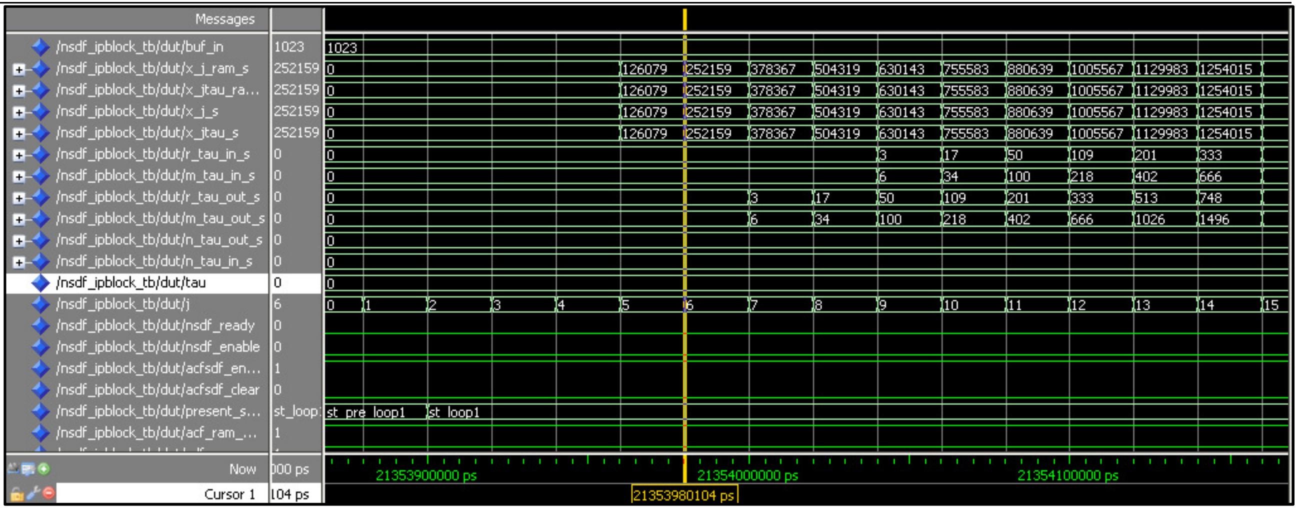

Messages
/nsdf_ipblock_tb/dut/buf_in
/nsdf_ipblock_tb/dut/x_j_ram_s
/nsdf_ipblock_tb/dut/x_jtau_ra...
/nsdf_ipblock_tb/dut/x_j_s
/nsdf_ipblock_tb/dut/x_jtau_s
/nsdf_ipblock_tb/dut/r_tau_in_s
/nsdf_ipblock_tb/dut/m_tau_in_s
/nsdf_ipblock_tb/dut/r_tau_out_s
/nsdf_ipblock_tb/dut/m_tau_out_s
/nsdf_ipblock_tb/dut/n_tau_out_s
/nsdf_ipblock_tb/dut/n_tau_in_s
/nsdf_ipblock_tb/dut/tau
/nsdf_ipblock_tb/dut/j
/nsdf_ipblock_tb/dut/nsdf_ready
/nsdf_ipblock_tb/dut/nsdf_enable
/nsdf_ipblock_tb/dut/acfsdf_en...
/nsdf_ipblock_tb/dut/acfsdf_clear
/nsdf_ipblock_tb/dut/present_s...
/nsdf_ipblock_tb/dut/acf_ram_...
st_pre_loop1
st_loop1
Now
Cursor 1
21353900000 ps
21354000000 ps
21354100000 ps
21353980104 ps

Figure 37 Inner loop1 verification of m_tau and r_tau computation

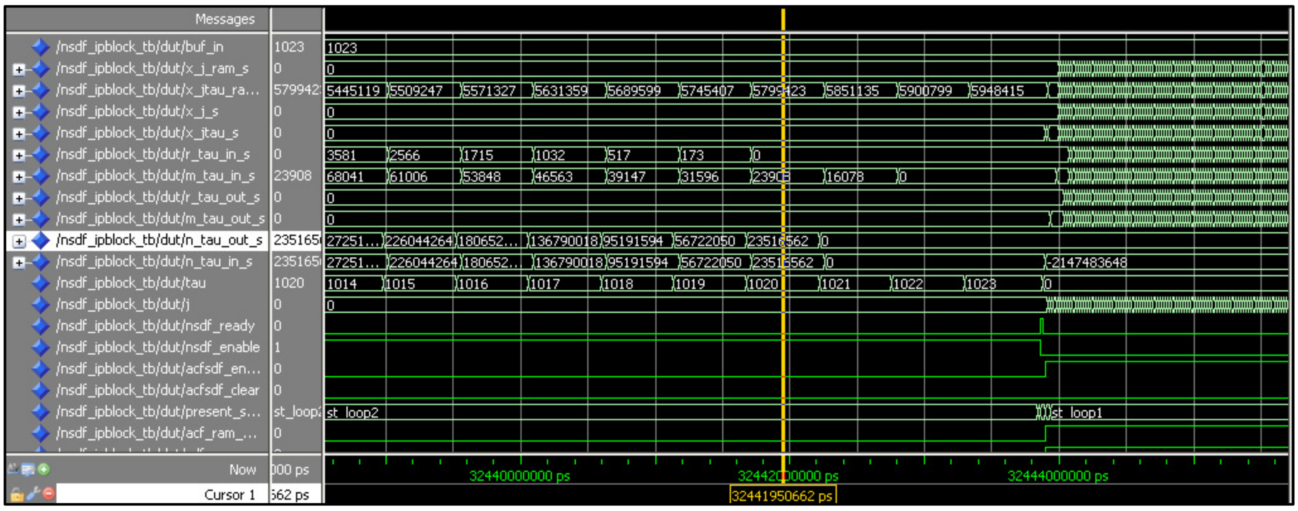
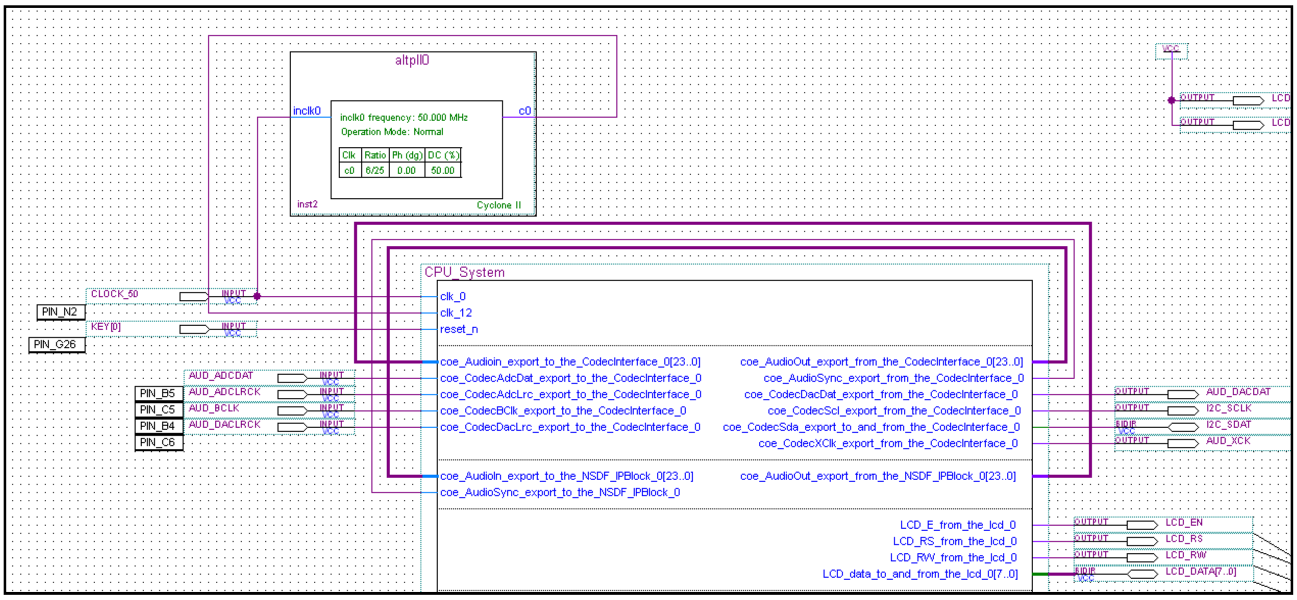

Messages
/nsdf_ipblock_tb/dut/buf_in
/nsdf_ipblock_tb/dut/x_j_ram_s
/nsdf_ipblock_tb/dut/x_jtau_ra...
/nsdf_ipblock_tb/dut/x_j_s
/nsdf_ipblock_tb/dut/x_jtau_s
/nsdf_ipblock_tb/dut/r_tau_in_s
/nsdf_ipblock_tb/dut/m_tau_in_s
/nsdf_ipblock_tb/dut/r_tau_out_s
/nsdf_ipblock_tb/dut/m_tau_out_s
/nsdf_ipblock_tb/dut/n_tau_out_s
/nsdf_ipblock_tb/dut/n_tau_in_s
/nsdf_ipblock_tb/dut/tau
/nsdf_ipblock_tb/dut/j
/nsdf_ipblock_tb/dut/nsdf_ready
/nsdf_ipblock_tb/dut/nsdf_enable
/nsdf_ipblock_tb/dut/acfsdf_en...
/nsdf_ipblock_tb/dut/acfsdf_clear
/nsdf_ipblock_tb/dut/present_s...
/nsdf_ipblock_tb/dut/acf_ram_...
st_loop2
st_loop1
Now
Cursor 1
32440000000 ps
32442000000 ps
32444000000 ps
32441950662 ps

Figure 38 Loop2 verification of n_tau computation

The simulated results are compared with the fixed point results stored in text files generated from the architectural SystemC model.

3.5 Software design and implementation

The NSDFIPBlock is created as a component and inserted into the SOPC design using the SOPC builder.

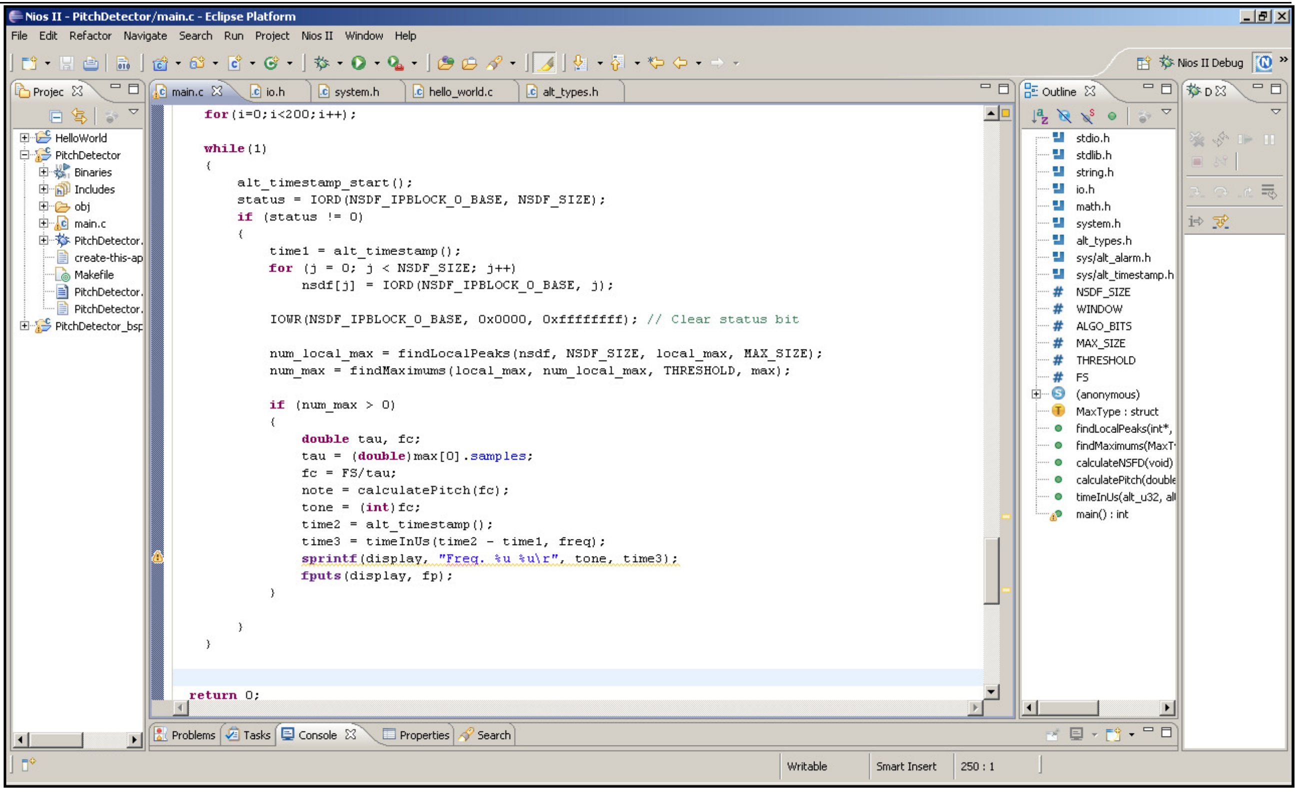

altpll0
inclk0
c0
inclk0 frequency: 50.000 MHz
Operation Mode: Normal
inst2
Cyclone II
CPU_System
clk_0
clk_12
reset_n
CLOCK_50
PIN_N2
KEY[0]
PIN_G26
AUD_ADCDAT
AUD_ADCLRCK
AUD_BCLK
AUD_DACLRCK
PIN_B5
PIN_C5
PIN_B4
PIN_C6
coe_AudioIn_export_to_the_CodecInterface_0[23..0]
coe_CodecAdcDat_export_to_the_CodecInterface_0
coe_CodecAdcLrc_export_to_the_CodecInterface_0
coe_CodecBClk_export_to_the_CodecInterface_0
coe_CodecDacLrc_export_to_the_CodecInterface_0
coe_AudioOut_export_from_the_CodecInterface_0[23..0]
coe_AudioSync_export_from_the_CodecInterface_0
coe_CodecDacDat_export_from_the_CodecInterface_0
coe_CodecScl_export_from_the_CodecInterface_0
coe_CodecSda_export_to_and_from_the_CodecInterface_0
coe_CodecXClk_export_from_the_CodecInterface_0
coe_AudioIn_export_to_the_NSDF_IPBlock_0[23..0]
coe_AudioSync_export_to_the_NSDF_IPBlock_0
coe_AudioOut_export_from_the_NSDF_IPBlock_0[23..0]
AUD_DACDAT
I2C_SCLK
I2C_SDAT
AUD_XCK
LCD_E_from_the_lcd_0
LCD_RS_from_the_lcd_0
LCD_RW_from_the_lcd_0
LCD_data_to_and_from_the_lcd_0[7..0]
LCD_EN
LCD_RS
LCD_RW
LCD_DATA[7..0]
VCC
LCD

Clk	Ratio	Ph (dg)	DC (%)
c0	6/25	0.00	50.00

Figure 39 Final SoPC design including the NSDF_IPBlock

The software part is implemented in a simple main program that polls the NSDFIPBlock by reading the status register. Bit 1 in this register tells that a new NSDF buffer has been computed. The computed NSDF block is read from the memory mapped interface and the status bit is cleared. The functions findLocalPeaks and findMaximums are the “C” implementation of the Matlab functional model.

After finding the peak the pitch is calculated. The number samples from start of the NSDF buffer to the peak and the sampling rate is used to calculate the frequency. The timestamp function provide by Altera is used to measure the computation time.



```
    for(i=0;i<200;i++);

    while(1)
    {
        alt_timestamp_start();
        status = IORD(NSDF_IPBLOCK_0_BASE, NSDF_SIZE);
        if (status != 0)
        {
            time1 = alt_timestamp();
            for (j = 0; j < NSDF_SIZE; j++)
                nsdf[j] = IORD(NSDF_IPBLOCK_0_BASE, j);

            IOWR(NSDF_IPBLOCK_0_BASE, 0x0000, 0xffffffff); // Clear status bit

            num_local_max = findLocalPeaks(nsdf, NSDF_SIZE, local_max, MAX_SIZE);
            num_max = findMaximums(local_max, num_local_max, THRESHOLD, max);

            if (num_max > 0)
            {
                double tau, fc;
                tau = (double)max[0].samples;
                fc = FS/tau;
                note = calculatePitch(fc);
                tone = (int)fc;
                time2 = alt_timestamp();
                time3 = timeInUs(time2 - time1, freq);
                sprintf(display, "Freq. %u %u\r", tone, time3);
                fputs(display, fp);
            }

        }
    }

  return 0;
```

Figure 40 Main loop in C program that reads the NSDF and calculates the pitch

The calculated frequency and measured computation times in us is displayed on the LCD display of the DE2 board.

```c
int findLocalPeaks(int *buf, int sbuf, MaxType *max, int smax)
{
        int rising = 0;
        int idx = 0;
        int n_1 = 0;
        int n;

        for (n = 0; n < sbuf; n++)
        {
                if (buf[n] > 0)
                {
                        if (buf[n] > n_1) rising = 1;
                        if (buf[n] < n_1)
                        {
                                if ((rising == 1) && (idx < smax))
                                {
                                        max[idx].samples = n - 1;
                                        max[idx].value = n_1;
                                        idx++;
                                }
                                rising = 0;
                        }
                }
                n_1 = buf[n];
        }

        return idx;
}

int findMaximums(MaxType *lmax, int slmax, int threshold, MaxType *max)
{
        int rising = 0;
        int idx = 0;
        int n_1 = lmax[1].value;
        int n;

        for (n = 2; n < slmax; n++)
        {
                if (lmax[n].value > n_1) rising = 1;
                if (lmax[n].value < n_1)
                {
                        if ((rising == 1) && (n_1 > threshold))
                        {
                                max[idx].samples =
                                           lmax[n - 1].samples;
                                max[idx].value = n_1;
                                idx++;
                        }
                        rising = 0;
                }
                n_1 = lmax[n].value;
        }

        return idx;
}
```

The architectural model of the NSDF calculation has been tested as a pure software version running on the Nios II processor. This version has been optimized by rewriting it to use the build in Altera definitions for signed integers variables (alt_32, alt_64) see listing below:

```
// Compute normalized squared difference function (NSDF), optimized for Nios II
void calculateNSFDOpt(void)
{
  unsigned tau, j;
  alt_32 m_x[WINDOW];    // Window of input samples
  alt_32 r_tau[WINDOW]; // Autocorrelation ACF type II (ACF)
  alt_32 m_tau[WINDOW]; // Squared difference function modified SDF (SDF)
  alt_32 n_tau[WINDOW]; // Normalized squared difference function (NSDF)
  alt_32 sample;

  // Generate test data
  for (tau = 0; tau < WINDOW; tau++)
           m_x[tau] = tau;

  for (tau = 0; tau < WINDOW; tau++)
  {
    n_tau[tau] = m_x[tau];
    for (j = 0; j < WINDOW-1-tau; j++)
    {
      r_tau[tau] = r_tau[tau] + ((alt_64)(m_x[j]*m_x[j+tau]) >> ALGO_BITS);
      m_tau[tau] = ((alt_64)(m_x[j]*m_x[j]) >> ALGO_BITS) +
                                ((alt_64)(m_x[j+tau]*m_x[j+tau]) >> ALGO_BITS);
    }
  }

  for (tau = 0; tau < WINDOW; tau++)
  {
    if ((int)m_tau[tau] == 0)
      n_tau[tau] = 0;
    else
      n_tau[tau] = (((alt_64)r_tau[tau] << (ALGO_BITS-1))/m_tau[tau]) << 1;
    sample = n_tau[tau];
  }
}
```

3.6 Results

The performance of the final pitch detector is listed below. The compiler optimization level 3 is turned on in achieving the best performance for the software version. See performance results below:

- **SW Calculation NSDF = 3.027 sec (Nios II SW function)**
- **SW Calculation NSDF = 620 ms (Nios II SW function, opt level 3)**
- **HW Calculation NSDF = 11.1 ms (NSDFIPBlock equal to 286 [5]MOPS)**
- **SW Calculation main = 5.6 ms (Peak detection and frequency)**
- **SW Calculation main = 2.1 ms (Peak detection and frequency, opt level 3)**

The picture below shows a photo of the LCD display on the DE2 prototype. The first line of the display shows the time measured (us) calculating the NSDF in software. The second line displays the pitch frequency in Hz and the time (us) to find the peak and computing the frequency.

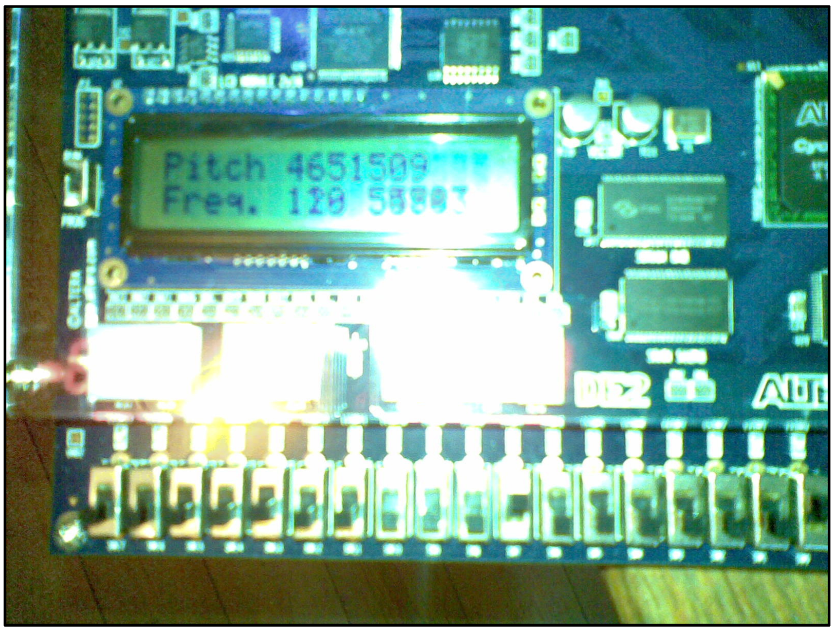


Figure 41 Photo of LCD display of the DE2 pitch detector prototype

[5] MOPS = Million Operations Per Second, calculated based on number of iterations of inner loop1 and number of operations (* and + equal 6) . MIPS cannot be used here since we are not speaking about "CPU" instructions.

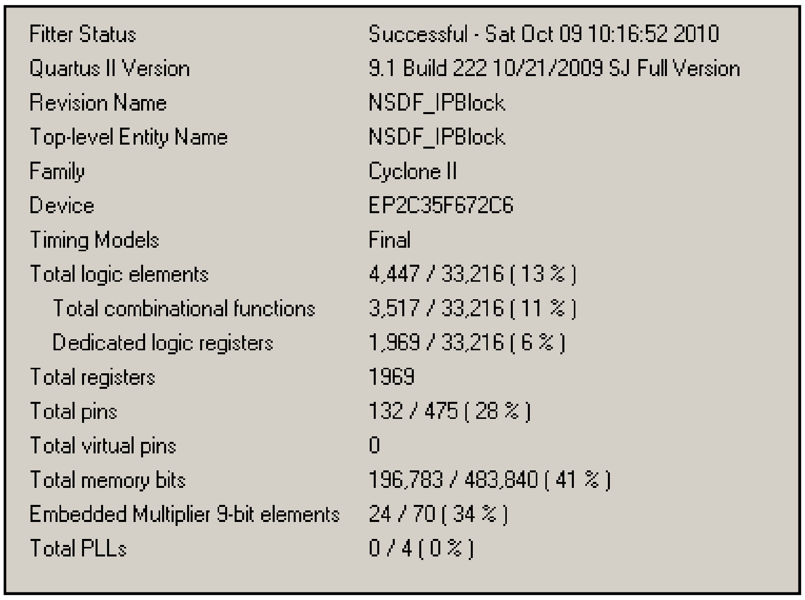
The table below shows that the frequency measured is stable starting from around 45 Hz (Theoretical frequency was 46.875 Hz see functional Matlab model)

The maximum frequency able to be measured is 655 Hz.

Freq	**Measured**
45	47
50	50
55	55
60	60
75	75
100	100
200	200
250	250
300	300
350	350
400	400/396
450	448
500	500/494
550	551
600	600/604
650	648
655	***657***
660	220
700	350
725	241
750	Unstable
800	Unstable

The table below is a summary of the FPGA area usages for the NSDFIP_Block and the final SoPC design that includes the Nios II processor and all IP core components. The percentage is compared with the total amount of resources of the Cyclone II FPGA on the DE2 board.

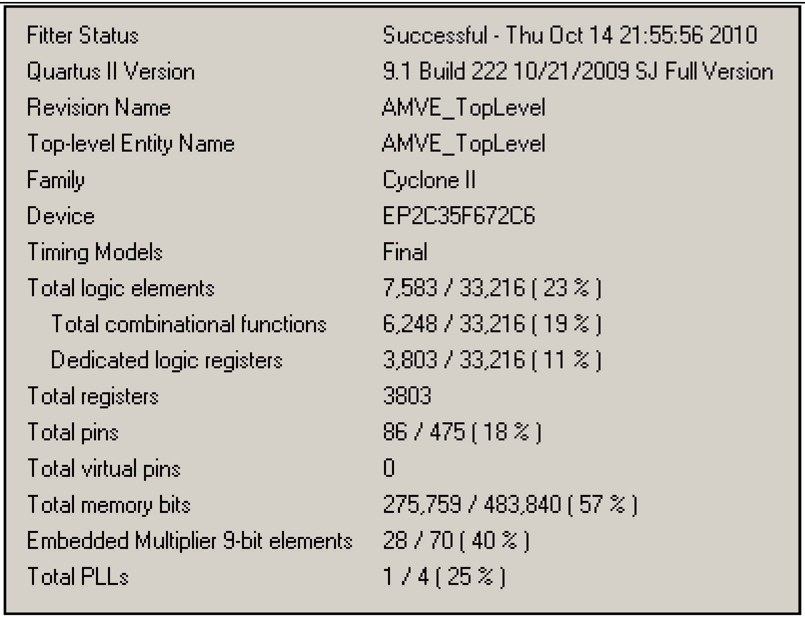
	LE[6]	**Memory (M4K) (bits)**	**Multipliers (9 bits)**
NSDFIPBlock	4447 (13 %)	196783 (41%)	24 (34%)
Final SoPC design	7583 (23%)	275759 (57%)	28 (40 %)
Difference	3135 (10%)	78976 (16%)	4 (6%)



```
Fitter Status                        Successful - Sat Oct 09 10:16:52 2010
Quartus II Version                   9.1 Build 222 10/21/2009 SJ Full Version
Revision Name                        NSDF_IPBlock
Top-level Entity Name                NSDF_IPBlock
Family                               Cyclone II
Device                               EP2C35F672C6
Timing Models                        Final
Total logic elements                 4,447 / 33,216 ( 13 % )
   Total combinational functions     3,517 / 33,216 ( 11 % )
   Dedicated logic registers         1,969 / 33,216 ( 6 % )
Total registers                      1969
Total pins                           132 / 475 ( 28 % )
Total virtual pins                   0
Total memory bits                    196,783 / 483,840 ( 41 % )
Embedded Multiplier 9-bit elements   24 / 70 ( 34 % )
Total PLLs                           0 / 4 ( 0 % )
```

Figure 42 Hardware NSDF_IPBlock, FPGA resource usages

[6] Logical Elements



Fitter Status	Successful - Thu Oct 14 21:55:56 2010
Quartus II Version	9.1 Build 222 10/21/2009 SJ Full Version
Revision Name	AMVE_TopLevel
Top-level Entity Name	AMVE_TopLevel
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	7,583 / 33,216 (23 %)
Total combinational functions	6,248 / 33,216 (19 %)
Dedicated logic registers	3,803 / 33,216 (11 %)
Total registers	3803
Total pins	86 / 475 (18 %)
Total virtual pins	0
Total memory bits	275,759 / 483,840 (57 %)
Embedded Multiplier 9-bit elements	28 / 70 (40 %)
Total PLLs	1 / 4 (25 %)

Figure 43 Final pitch detector SoPC platform, FPGA resource usages

3.7 Discussion on achieved results

The pitch detection is good in the area 47 Hz to 655 Hz hereafter the peaks that is found seems not to be correct. The reason could be related to the way the peaks are found and calculated. This part of the algorithm needs more investigation. Already in the functional Matlab model it seems not to be stable for higher frequencies.

It is quite amazing that we can achieve a performance optimization of approximately **56 times** moving the NSDF calculation to an optimized parallel hardware design. Since effort has been made in optimizing the architectural software implementation the software execution time could not be improved significant. The total pitch computation time is:

Pitch computation time = HW Calculation NSDF + SW Calculation main = 13.1 ms

It is interesting to compare the FPGA area usage of the NSDFIPBlock. The area usage is actually bigger than the Nios-II processor. The Nios-II/s processor requires between 1200-1400 LE and two M4K blocks. That is only 5% of the whole FPGA resources. The NSDFIPBlock uses quiet a large amount of area (13% LE, 41% RAM, 34% Multipliers) even bigger that the SoPC design without the NSDF block. It would not be possible to compute the NSDF in software using a big number of parallel Nios-II processors even though there is space for 20 of them in the Cyclone II FPAG. If we would be able to parallelize the computation on 20 CPU's the processing time will still be in the area of 620/20 = 31 ms.

The improved performance has defiantly a penalty of increased FPGA area usage.

3.8 Suggestion to improvements

The method described could be improved by reuse of the architectural model in SystemC to test the NSDFIPBlock directly with [7]ModelSim reusing the SystemC modules in co-simulation with the VHDL code. This is possible with a special multi language license for ModelSim. It would be more efficient to reuse the architectural model and replacing modules written in VHDL for verification instead of rewriting a test bench in VHDL.

The NSDFIPBlock only computes the NSDF buffer each second time the window size of samples are received. The design is prepare being able to buffer samples the same time as computing, but more work needs to be done to complete this part of the implementation.

An alternative optimization would be to start with the software NSDF computation and then accelerate the inner part of the two “for loops” by creating a customized instruction for the Nios II processor. This approach would be a faster way to achieve a better performance than writing everything in VHDL.

The algorithm for finding the maximum peaks needs more investigation and improvement. Currently it is not stable for frequencies over 660 Hz. Finding a more exact maximum could also be improved by using a parabolic interpolation between the samples before and after the maximum peak.

Instead of polling the status bit the use of interrupt to the Nios II processor would be a better way to transfer the processed NSDF buffer to the Nios II processor for computation of the pitch. The next step in making an embedded SW application would be to add a RTOS kernel and creating a separate task that would be responsible for processing the NSDF buffer and updating the pitch on the LCD screen. More functionality could be added to handle user operation and communication with other connected devices. Such an extension will follow a traditional process for development of real-time embedded systems based on object oriented analysis and design with UML.

Finally it would be interesting to see if it would be possible to reduce the development time for the VHDL part of the NSDFIPBlock by using high level synthesis (HLS) coming from a C-based design to gate level implementation. Would it be possible to achieve the same performance in an automatic process as in the manually process? ImpulseC[8] is a promising HLS approach for FPGA development that could be a choice for such a work.

[7] SystemC verification with ModelSim from Mentor Graphics
http://www.mentor.com/products/fv/techpubs/systemc-verification-with-modelsim-24133

[8]ImpulseC homepage: http://www.impulseaccelerated.com/

4 Project effort

In the following list is summarized the effort of work that has been performed making this project. It contains an approximate of the used time on each step in the process and where the different models and source code that can be found on the CD that contains the project work.

1. Matlab functional model of the pitch detector validated with parameters like: window size, tones, peak detection, NSDF calculation.
   MultiProject1\MatlabModel
   (10 hours)

2. SystemC functional model in floating and fixed point. Signal processing blocks. Kahn processing network using SystemC fifo buffers.
   MultiProject1\SystemC\SmartPitchDetector (Functional)
   (10 hours)

3. SystemC architecture model – HW refinement adding clock, sample bus and adapters. Calculation in integer's preparation for VHDL implementation.
   Golden reference model for the final VHDL test bench.
   MultiProject1\SystemC\SmartPitchDetector (Architecture)
   (10 hours)

4. Audio loop back SoPC design using codec to create an initial NIOS II FPGA platform.
   MultiProject1\DE2AudioLoop
   (5 hours)

5. VHDL implementation computing inner loops of the NSDF algorithm computing ACF, SDF. Estimation of cycle usages. Pipelining of ACF and SDF computation. NSDF using the Quartus MegaWizard Plug-In Manager in generating the division part of NSDF. Use timing analysis output from the Quartus tool in back annotation of timing estimates to the SystemC architecture model to give an estimate of the final HW implementation.
   MultiProject1\NSDFInnerLoop
   MultiProject1\NSDFLoop2
   (10 hours)

6. VHDL implementation of the control unit of the Datapath implementation in 6. Verification of the algorithm in ModelSim and comparing result with the architecture model in SystemC. Using test stimuli files generated from the SystemC model in the VHDL test bench.
   MultiProject1\NSDFStates
   (20 hours)

7. SoPC design adding the verified NSDF_IPBlock to a NIOS II design. Found that this design requires far too many LE elements 239239 – 33216. Synthesis time =5:47 hours. Problem is related to 4 * 1024 * 32 bits buffers mapping to LE and not using the M4K RAM blocks.
   MultiProject1\DE2PitchDetector (ModelSim project)
   (5 hours)

8. NSDF redesign using internal RAM blocks storing input sample buffer, ACF, SDF and NSDF results. Refinement and verification of new design in ModelSim. Changed state machine to handle pipelining accessing the ramblocks. Computing ACF and SDF in one clock cycle.
   MultiProject1\NSDFIPBlock
   (15 hours)

9. Final pitch detector SoPC project design, implementation and verification of Hardware NSDFIPBlock and C-main program on target.
   MultiProject1\DE2PitchDetector – the final DE2 prototype
   (10 hours)

	Duration (Project hours)
Functional model (1+2)	20 (21%)
Architectural model and platform prototype (3+4)	15 (16%)
Hardware implementation and verification (5+6+8)	45 (47%)
Software implementation, system integration and test (7+9)	15 (16%)
Project total	95 hours

5 Conclusion

It has been a very learning exercise doing this project. The theory has mainly been adapted from four different courses: Embedded Real-Time Systems (TIIRTS), DSP in applied digital signal processing (TISPRC), Hardware/Software Co-design (TIHSC1) and advanced VHDL programming (VDL2). The project has combined the learning from these courses and provided a methodology for developing of signal processing algorithms targeting a FPGA SoPC platform where parts of the algorithm is implemented in hardware and software.

The methodology explained is exemplified with implementation of the smart pitch detecting algorithm. The method presented is based on manually transformation coming from the functional and architectural model to the final implementation in hardware and software. The project has also provided with information about the time that it takes to perform the different phases of the process. Hardware design, refinement, test and implementation is the far most time consuming part (45%) therefore the time used in doing a proper model before starting the implementation is very important. Without the golden reference model it would not be easy to verify the hardware implementation or even be sure it was the right part of the functionality to implement in hardware.

The project has cover all goals as defined in the original project description and even used more time in refining the hardware design and implementation than original specified. The project description stated that is was less important to transform the SystemC model to implementation and demonstrate the functionality in a final DE2 prototype. This process has been very important especially in being able to evaluate the value of the architectural SystemC model.

This project has shown it is possible to use a combination of common tools like Matlab, UML/SysML and SystemC to define a manual HW/SW Co-design development process. The methodology is for signal processing algorithms covering functional, architectural design and implementation targeting a FPGA SoC platform that includes hardware components and software functionality.

6 References

[1] Philip McLeod, Geoff Wyvill, “A Smarter Way To Find Pitch, University of Otago,” Department of Computer Science.

[2] Wayen Wolf, ”A Decade of Hardware/Software Codesign,” Cover feature IEEE, 2003

[3] Fabina Mischkalla, Da He, Wolfgang Mueller, “Closing the Gap between UML-based Modeling, Simulation and Synthesis of Combined HW/SW Systems,” University of Paderborn, C-LAB, DATE 2010.

[4] E. Riccobene, A. Rosti, and P. Scandurra, “Improving SoC Design Flow by means of MDA and UML Profiles,” in Pro. 3rd Workshop in Software Model Engineering, 2004.

[5] Mauro Prevostini, Elena Zamsa, “SysML Profile for SoC Design and SystemC Transformation,” University of Lugano, 2007

[6] Thorsten Grötker, Stan Liao, Grant Martin, Stuart Swan, “System Design with SystemC,” Kluwer Academic Publishers, 2004

[7] Davic C. Black and Jack Donovan, “SystemC: From The Ground Up,” Springer, 2004

[8] J. Bhasker, “A SystemC Primer,” Second Edition, Star Galaxy Publishing, 2004

[9] Daniel D. Gajski, Samar Abdi, Andreas Gerslauer, Gunar Schirner, “Embedded System Design,” Springer, 2009.

[10] J. Staunstrup, W. Wolf, “Hardware/Software Co-Design Principles and Practice,” Kluwer Academic Publishers, 1997.

[11] Stefan Doll, VHDL Verification course http://www.stefanvhdl.com/vhdl/html/index.html

[12] Kenneth L. Short, VHDL for Engineers, Pearson

[13] Open SystemC Initiatiove OSCI, www.systemc.org

[14] Doulos, Expert VHDL Verification course

[15] Xilinx, http://www.xilinx.com/

[16] Altera, http://www.altera.com/

[17] Axilica, http://www.axilica.com

[18] Artisan Studio, http://www.atego.com/

[19] QEMU, http://wiki.qemu.org/Main_Page

[20] Simulink HDL Coder, http://www.mathworks.com/products/slhdlcoder/

[21] Adnam Shaout, Ali H. El-Mousa, and Khalid Mattar, ”Specification and Modeling of HW/SW Co-Design for Heterogeneous Embedded Systems,”
World Congress of Engineering 2009 Vol I

[22] Sanford Friedenthal, Alan Moore, Rick Steiner, “A Practical Guide to SysML, The System Modeling Language,” Morgan Kaufmann and Object Management Group, 2008, p. 95-147