Laboratoire d'Architecture des Processeurs
IN-F Ecublens
CH-1015 Lausanne
http://lap.epfl.ch

Semester or Diploma Project
Spring 2017

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

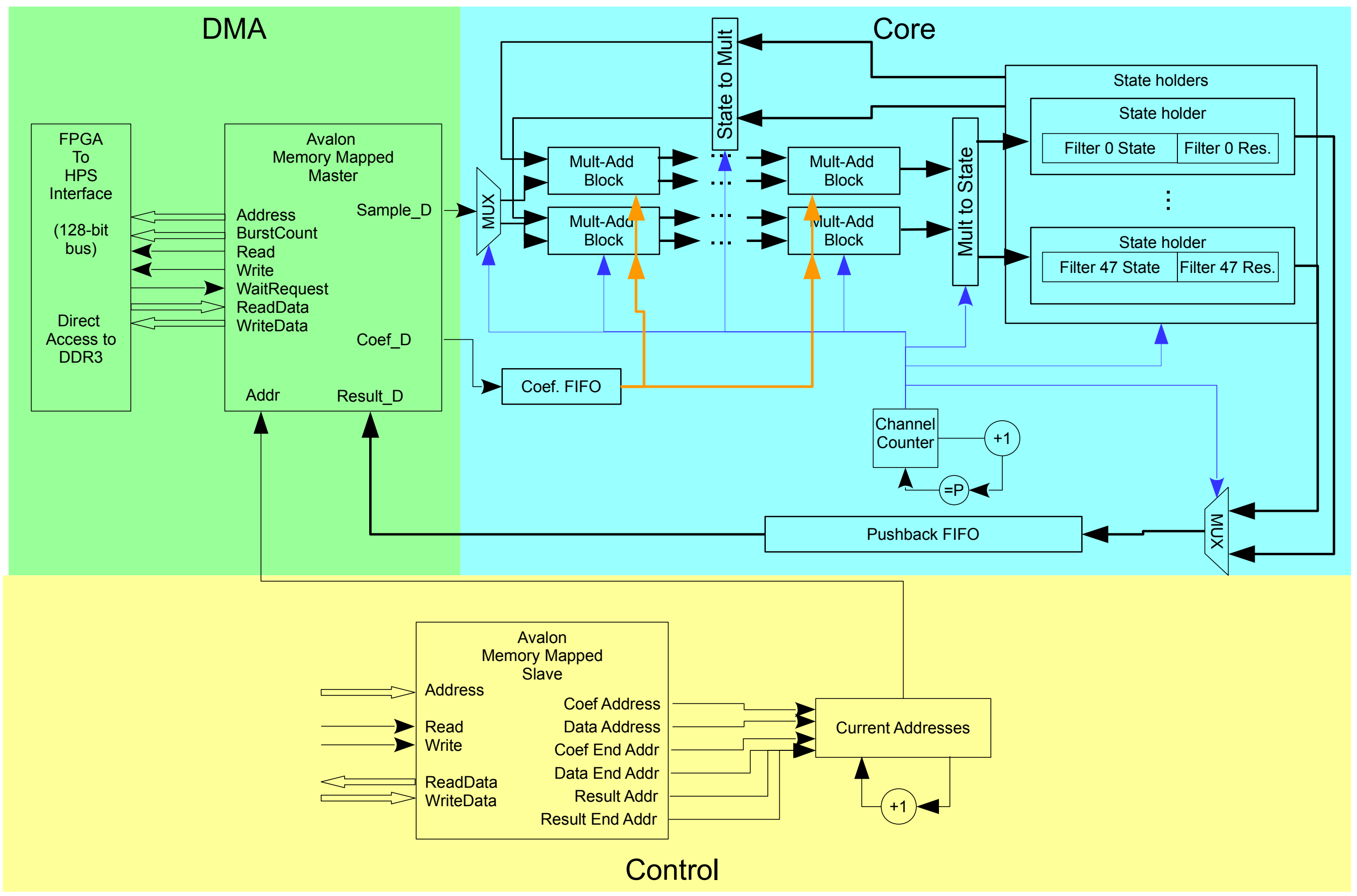
Implementation of FIR filters for fast multi-channel processing

Corentin Ferry **Semester Project**

Description of the project

Digital signal processors are ubiquitous in electronics, with applications ranging from sound processing to software-defined radio. Finite impulse response (FIR) filters are among the components that are used for the processing; the implementation is tailored to the user's needs, whether they specifically need performance or configurability.

Handling numerous channels in a single filter block can be achieved in hardware by running multiple filters in parallels, with a high space expense and controller overhead. This project proposes and discusses an implementation template for a pipeline that simultaneously processes a desired number of channels, while keeping coefficients configurability.



Architecture of the designed system

The implementation we propose is economic in terms of area while meeting the theoretical timing requirements to process long filters.

SEMESTER PROJECT

titled

Implementation of FIR filters for fast multi-channel processing

done at the

PROCESSOR ARCHITECTURE LABORATORY
and the
AUDIOVISUAL COMMUNICATIONS LABORATORY
(Lausanne, Switzerland)

under the supervision of

Dr. Robin SCHEIBLER (EPFL)
Prof. René BEUCHAT (EPFL)

Corentin FERRY

February to May 2017

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Abstract

Digital signal processors are ubiquitous in electronics, with applications ranging from sound processing to software-defined radio. Finite impulse response (FIR) filters are among the components that are used for the processing; the implementation is tailored to the user's needs, whether they specifically need performance or configurability.

Handling numerous channels in a single filter block can be achieved in hardware by running multiple filters in parallels, with a high space expense and controller overhead. This project proposes and discusses an implementation template for a pipeline that simultaneously processes a desired number of channels, while keeping coefficients configurability.

The implementation we propose is economic in terms of area while meeting the theoretical timing requirements to process long filters.

Contents

Acknowledgements

I warmfully thank René Beuchat and Robin Scheibler for their will to accompany me throughout my projects at LAP and LCAV.

I add specific thanks for Sahand Kashani-Akhavan and his brother Sepand, for their help and their involvement in the demo with the Pyramic array performed in February 2017 at ICASSP. All the team at LCAV who participated to ICASSP also get warm thanks!

1 Notations

The following notations will be used throughout this document:

- $\mathbf{N}$ denotes the set of natural numbers,
- $\mathbf{R}$ denotes the set of real numbers,
- $\mathbf{C}$ denotes the set of complex numbers,
- L denotes the filter length,
- B denotes the the length of a block of samples taken from the input (we call this the *block length*),
- P denotes the number of pipelines that the data is going through,
- C denotes the number of channels that the filter has to process,
- $(x_{i,t})_{(i,t)\in\mathbf{N}^2}$ denotes the sequences of samples which is the input to our system, where i is the identifier of the channel, starting from 0, and t is the time, starting from 0.
- $(y_{i,t})_{(i,t)\in\mathbf{N}^2}$ is the same notation as the input, for the output,
- $(\alpha_{i,j})$ denotes the sequences of coefficients of the filter, where i is the identifier of the channel and j is the position of the coefficient.

2 Introduction

Digital filters are ubiquitously found in electronic devices today, with uses ranging from noise reduction on audio signals, to tuned circuits made using software-defined radio. They are the digital counterparts of analog signal filters, that are usually made of interconnected R-L-C circuits; they are expected to have the same behavior (or perform more advanced transfer functions than those of simple RLC filters), by processing sequences of integers instead of analog voltages, as can be seen on Figure 1.

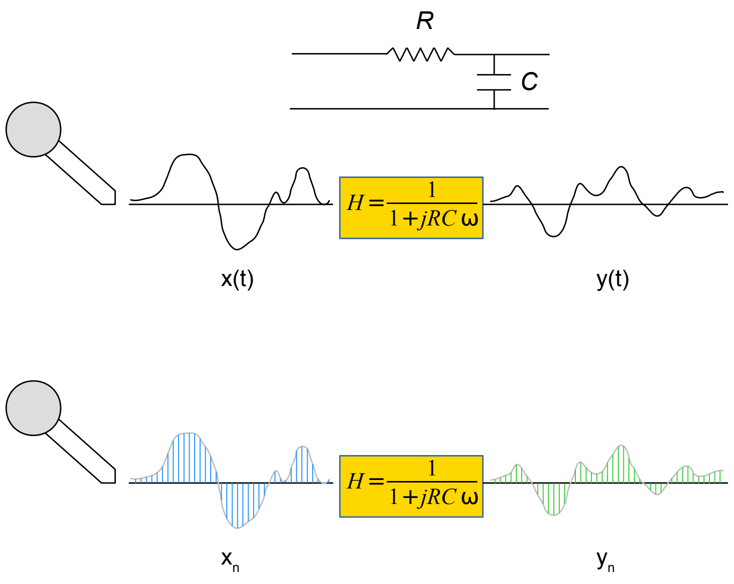


Figure 1: Analog versus digital filters

Finite-impulse response (FIR) filters are one of the kind of digital filters that fit to hardware implementation. Numerous such implementations can be found as of today; each of them uses specific mathematics and assumptions to provide a tradeoff between:

- a high precision on coefficients
- a longer impulse response
- a faster processing

We illustrate on Figure 2 the tradeoff between area and ease of configurability. This notion is quite ambiguous since configuring a filter can involve writing code as well as passing numerical values through a hardware interface.

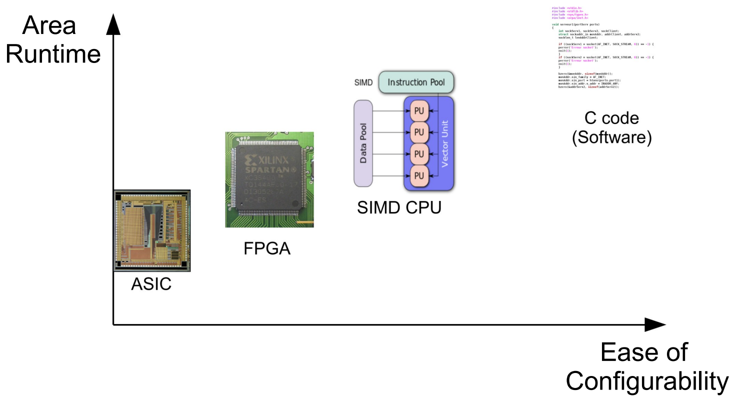


Figure 2: Configurability of a filter vs. performance

The fastest and most optimized filters are implemented as application-specific integrated circuits (ASICs), with a fixed set of coefficients. These implementations allow the hardware synthesizers to generate better-tailored multiplier and adder circuits, reducing the area needed and the power consumption of the device. Wireless sensor tuners, that do care about power efficiency, are likely to use such filters. Usually, these filters only need to be provided with a clock, an input, and give back their output on another port.

On the other hand, software implementations of FIR filters provide the most versatility and configurability: one can dynamically tune all the filter parameters. Software-defined radio users might prefer this kind of filters for this reason.

The main motivation of this project was to build a filter solution for the Pyramic array system [6]. It features a tetrahedral array on which 48 microphones synchronously capture an audio signal, and a system-on-a-chip embedding both a hardware processor and a field-programmable gate array (FPGA), that process the incoming signals. As of the beginning of this project, the system implemented on the FPGA used 7% of the available logic resources on the embedded FPGA, 13% of the available RAM blocks (mainly with FIFOs) and 0% of its signal processing resources (DSP blocks). We thus had cells left there in order to implement a digital filter.

Our project aims at providing a hardware implementation of an FIR filter that:

- is written in VHDL as an IP component (that can be reused),
- fits in the target FPGA along with the existing system,
- supports a fixed number of channels, determined at compile time,
- processes all the channels as simultaneously as possible,
- is configurable, i.e. its impulse response can be dynamically changed.

This report presents some filter implementation techniques that match our criteria and have been implemented before, along with the mathematical background and the hardware used. It then introduces the hardware implementation that we proposed with these criteria, and draws some conclusions about the outcome of this implementation.

Property	**Value**
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	2,162 / 32,070 (7 %)
Total registers	3688
Total pins	176 / 457 (39 %)
Total virtual pins	0
Total block memory bits	14,336 / 4,065,280 (< 1 %)
Total RAM Blocks	52 / 397 (13 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	3 / 6 (50 %)
Total DLLs	1 / 4 (25 %)

Table 1: Resource usage of the original Pyramic design

3 Related work

Some efficient implementation techniques for FIR filters on FPGAs already exist. Some implementations do not even involve multipliers: PLonga [5] proposes a multiplier-less approach to an FIR filter, taking advantage of the 4-bit lookup tables to fetch results in pre-computed tables. These results only have to be summed up to give the final result of the filtering. This approach, however, limits the bit depth of the input samples to 4 bits, which is 4 times lower than the samples that we have. Such a precision loss is not suitable for audio applications.

Many optimized techniques also exist when the coefficients are constant or follow certain hypotheses. J.B. Evans [4] proposes simplifications in the case the coefficients are sum or differences of two powers of 2; in this case, full multipliers are to be replaced by shifters. We cannot make such an hypothesis on the coefficients, since they are provided by the user themselves at runtime. More recently, T. Thingom [7] proposes an implementation where the coefficients are assumed to be constant, with an area-efficient recoding technique called RADIX-2^r.

The optimizations described here cannot be applied in our case. Typically, FIR filters are implemented with static coefficients, which makes it possible to generate smaller and faster multipliers instead of using full DSP blocks. The choice of user configurability is a pitfall against both speed and area, since it leads us to use full multipliers. The optimization target thus has to be at the algorithm level.

4 Contribution

4.1 Target hardware

The target board is a DE1-SoC from Altera. It notably features:

- An ARM Cortex-A9 hardware processor,
- A Cyclone V FPGA [3],
- An audio controller (Wolfson) à retrouver dans la fiche technique de la DE1-SoC

All of the available features are detailed in Figure 3; the resources are detailed on Figure 4.

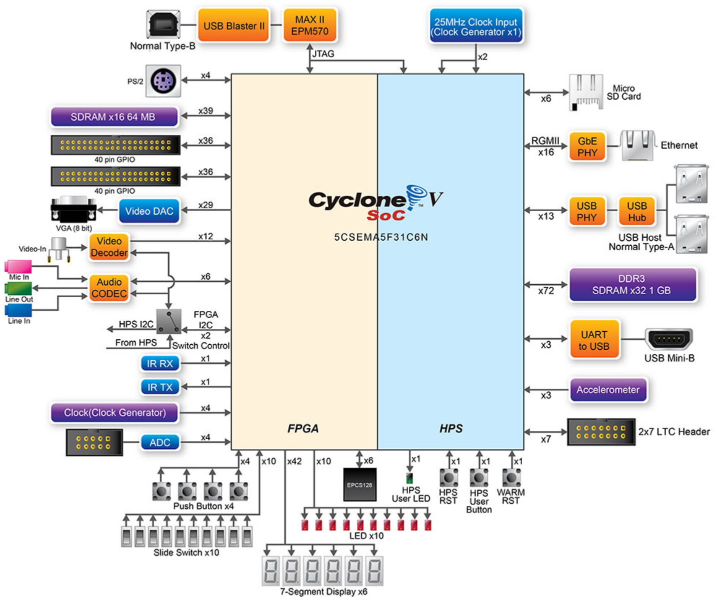


Figure 3: Components available on the DE1-SoC board

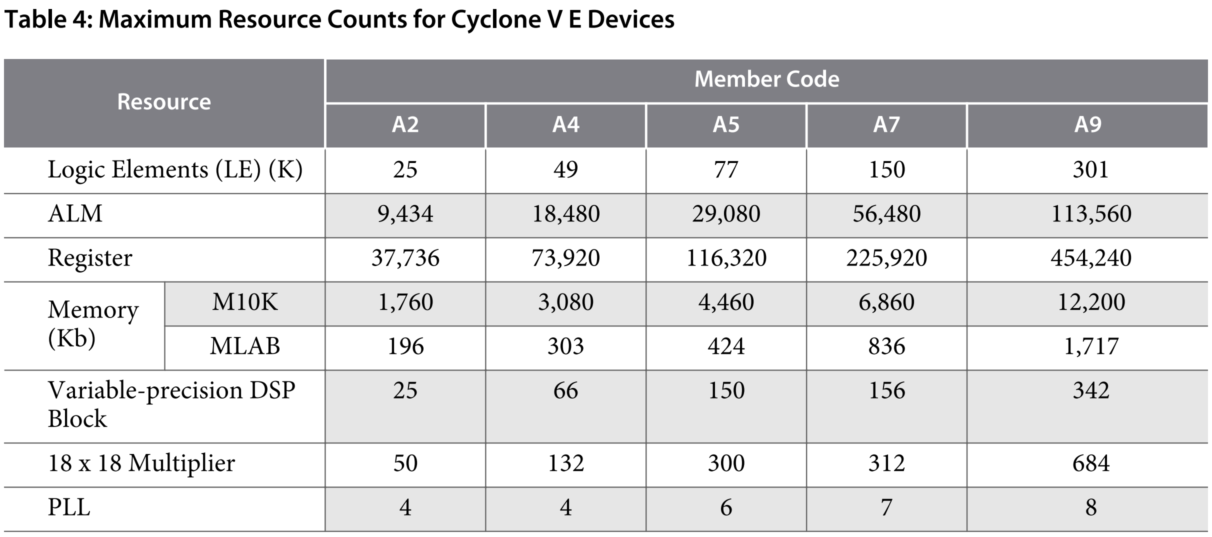


Table 4: Maximum Resource Counts for Cyclone V E Devices

Resource		Member Code				
		A2	A4	A5	A7	A9
Logic Elements (LE) (K)		25	49	77	150	301
ALM		9,434	18,480	29,080	56,480	113,560
Register		37,736	73,920	116,320	225,920	454,240
Memory (Kb)	M10K	1,760	3,080	4,460	6,860	12,200
	MLAB	196	303	424	836	1,717
Variable-precision DSP Block		25	66	150	156	342
18 x 18 Multiplier		50	132	300	312	684
PLL		4	4	6	7	8

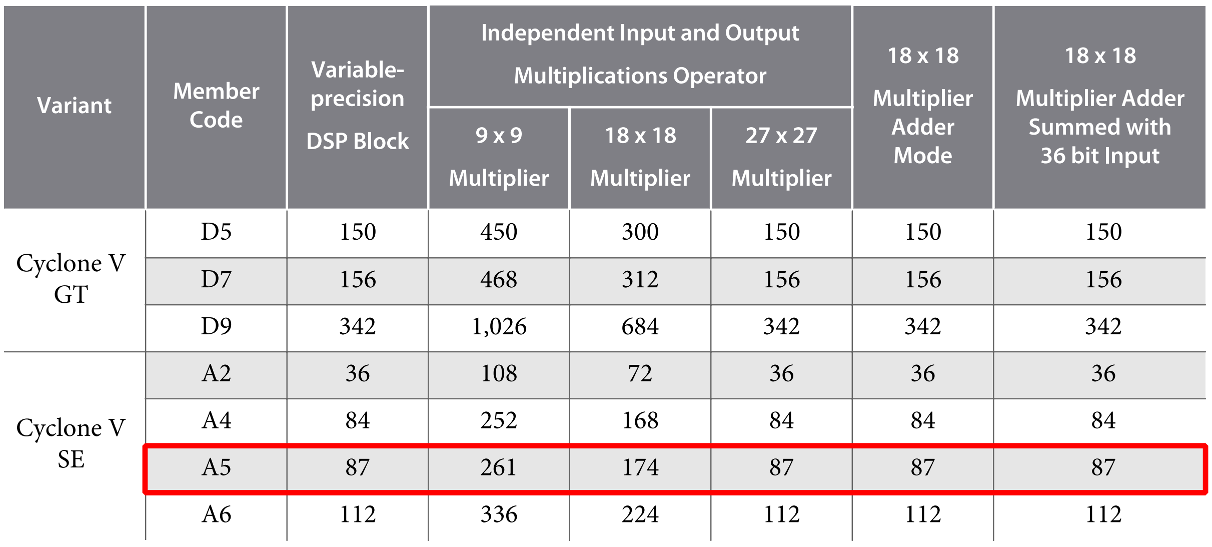
Figure 4: Resources on the Cyclone V FPGA. We are using the A5 model.

The presence of an audio controller enables building a full processing chain, from the input to the output.

We will notably use digital signal processor (DSP) blocks from the onboard FPGA. Figure 5 from Altera's documentation [3] shows the maximum number of multipliers available to us in various configurations. In order to benefit from the maximum number of multipliers (as DSP blocks inside the FPGA), we will be doing integer multiplications. Thus, we will be using fixed-point arithmetic.

Should the design for a particular system other than the Pyramic array not fit in the current target board, one would either have to upgrade their FPGA, or reduce the coefficient and sample widths.

On the Pyramic array, the microphones are located on boards such as the one on Figure 6. Assembled together in a tetrahedral way, these boards are connected to another base board that links them to the GPIO1 port of the DE1-SoC. There is one analog-to-digital converter (ADC) on each board; it features a Serial Peripheral Interface (SPI) which is directly connected to the FPGA's GPIOs. Samples captured by the ADCs go through the SPI to a controller, that writes them to a shared memory available to the



Variant	Member Code	Variable-precision DSP Block	Independent Input and Output Multiplications Operator			18 x 18 Multiplier Adder Mode	18 x 18 Multiplier Adder Summed with 36 bit Input
			9 x 9 Multiplier	18 x 18 Multiplier	27 x 27 Multiplier		
Cyclone V GT	D5	150	450	300	150	150	150
	D7	156	468	312	156	156	156
	D9	342	1,026	684	342	342	342
Cyclone V SE	A2	36	108	72	36	36	36
	A4	84	252	168	84	84	84
	A5	87	261	174	87	87	87
	A6	112	336	224	112	112	112

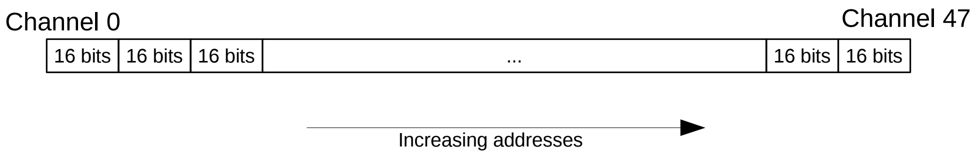
Figure 5: Maximum number of multipliers available for some Cyclone V FPGA models. The one we are using is squared in red.

Figure 6: A 8-microphone board found on the Pyramic array

FPGA and the hardware processor system (HPS).

Our goal is to have a controllable filter that is able to handle the signal captured by the Pyramic. Thus, we will use some of its characteristics:

- There are 48 channels,
- The samples are 16-bit signed (as two's complement) integers,
- The samples are interleaved in memory, like in this diagram:



  Thus, when doing burst memory accesses, we get chunks of 48 samples, captured simultaneously by the microphones, each one belonging to one channel.

- The sampling frequency is 8000 Hz (and not 48000 Hz as the Pyramic captures); this is achieved by using a decimator filter available as an Intellectual Property (IP) component from Altera.

4.2 Mathematical background

An FIR filter is one of the two kinds of digital filters that are driven by a linear differential equation. The general form for these equations is the following:

$$\alpha_0 x_0 + \alpha_1 x_1 + \cdots + \alpha_n x_n = \beta_0 y_0 + \beta_1 y_1 + \cdots + \beta_p y_p \text{ where } n, p \in \mathbf{N}$$

In FIR filters, the output depends neither on the output itself, nor on the past values of the output. Thus, we can simplify the equation into:

$$y_n = \alpha_0 x_0 + \alpha_1 x_1 + \cdots + \alpha_n x_n$$

This equation is simple to compute using hard-wired multiply-accumulate blocks that are found in most FPGAs today. However, performing such a computation will require as much multiply-accumulate blocks as there are terms in the sum, and this sum must be computed for each of the channels, which leads to a need of $L \times C$ multiply-accumulate blocks. Since $C = 48$ is the target number of channels, and the DE1-SoC only has 174 multiply-accumulate blocks in the configuration we chose[1], the limit on the computational resource is that at most 174 multiplications can take place in parallel.

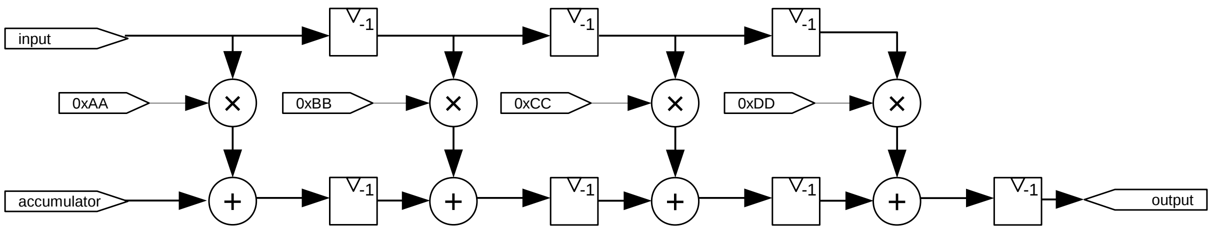


Figure 7: Simplest one-channel generic FIR filter with static coefficients

Some techniques exist to process a very long signal using the same number of multipliers, for instance, by dividing it into chunks. We are using one of them, which is called overlap-add.

[1]We are using 18×18 multipliers

4.2.1 Overlap-add technique

A naive FIR filter takes the samples at the sampling rate of the input. We are using DMA to perform such a fetch operation, which adds a significant overhead related to burst initialization (usually between 5 and 10 cycles). We would like to minimize this overhead. One idea is thus to process chunks of data instead of individual samples: the longer the burst, the smaller the effect of DMA initialization.

The overlap-add method consists in processing the signal in blocks instead of as they arrive. We choose to process blocks of length B; thus, we convolve these blocks with the filter which is of length L. More formally, we compute:

$$\forall i \in [1, B] : y_i = \sum_{k=1}^{L} x_i \alpha_{k-i}$$

We want to compute $y_1, \ldots, y_n$ where $L \ll \text{length}\,(x)$ (which is the case since we're continuously capturing audio).

The blocks that we convolve against the filter are taken such that they do not overlap. If we take a block $X_1 = x_1, \ldots, x_B$, and $X_2 = x_{B+1}, \ldots, x_{2B}$, then each convolution yields $B + L - 1$ nonzero samples. The last $M = L - 1$ samples obtained in convoluting X_1 are overlapping with the first samples of X_2's convolution.

We must thus add these overlapping samples to those obtained when convoluting X_2 so as not to break the results. We do this by keeping track of the spillover :

$$\begin{cases} y_1 = x_1\alpha_L & \text{Non-overlapping sample} \\ y_2 = x_1\alpha_{L-1} + x_2\alpha_L & \text{Non-overlapping sample} \\ \vdots & \\ y_B = \sum_{k=1}^{L} x_k\alpha_{L-B+k} & \text{Non-overlapping sample} \\ y'_{B+1} = \sum_{k=1}^{L} x_k\alpha_{L-(B+1)+k} & \text{Overlapping sample} \\ \vdots & \\ y'_{B+L-2} = x_{B-1}\alpha 1 + x_B\alpha_2 & \text{Overlapping sample} \\ y'_{B+L-1} = x_B\alpha_1 & \text{Overlapping sample} \\ y_{B+1} = \sum_{k=1}^{L} x_{k+B}\alpha_k + y'_{B+1} & \text{Overlapped sample} \\ y_{B+2} = \sum_{k=1}^{L} x_{k+B+1}\alpha_k + y'_{B+2} & \text{Overlapped sample} \\ \vdots & \end{cases}$$

Figure 8 illustrates how this is done in practice.

The algorithm would be written as such:

Algorithm 1 FIR filtering using overlap-add

```
1: procedure FIR($X, H, M$)        ▷ Convolves $X$ with $H$ adding overlapping samples $M$
2:     $R[0..B-1] \leftarrow (0, \ldots, 0)$
3:     $M'[0..L-1] \leftarrow (0, \ldots, 0)$        ▷ New overlapping part
4:     for $i = 1$ to $B - 1$ do
5:         $R[i] \leftarrow \sum_{k=0}^{L-1} X[k]H[L-i+k]$
6:     end for
7:     for $i = B$ to $B + L - 2$ do  ▷ This is equivalent to taking a zero-padded longer input and going
   on with the previous for loop
8:         $R[i] \leftarrow \sum_{k=0}^{L-i} X[k+B]H[L-i+k]$
9:     end for
10:    return $(R, M')$
11: end procedure
```

In hardware, we will zero-pad input, to be always performing the same loop. This will allow us to devise an execution pipeline, which will be the core of our implementation.

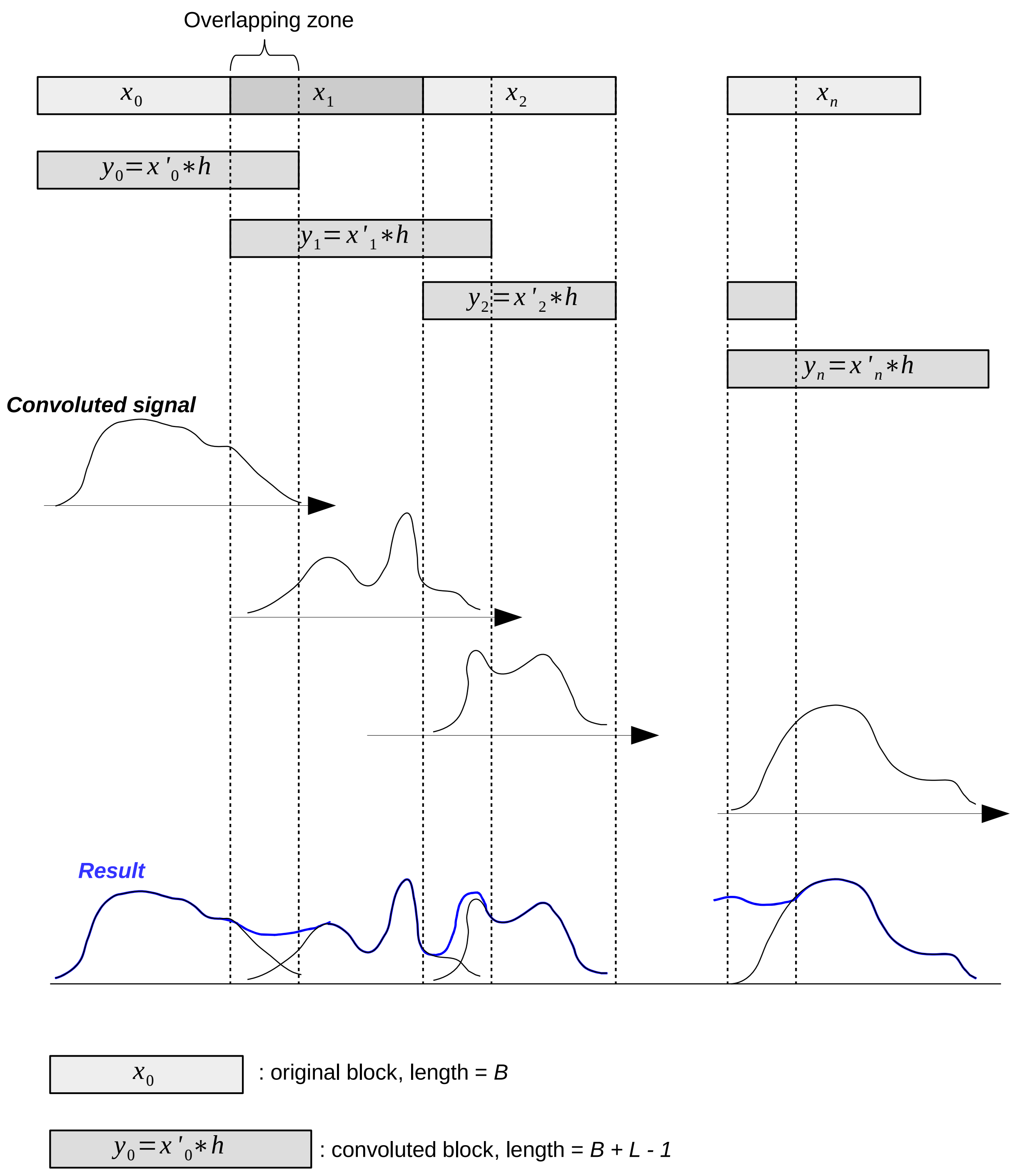


Figure 8: Computation of overlap-add

4.3 Complexity

We can measure the complexity of the chosen algorithm in terms of number of multiplications.

Each sample of a block of goes through L multiplications; we also count the multiplications that involve the zeroes added after the original samples (although the result is zero, the multiplier is still used); there are $\sum_{i=1}^{B} i = \frac{B(B-1)}{2}$ of these per block. Thus, each block involves

$$N = LB + \frac{B(B-1)}{2} = \frac{B(B-1+2L)}{2}$$

multiplications, which is $\mathcal{O}\left(B^2 + BL\right)$.

4.4 Configurability

The filter coefficients have to be updatable at runtime. We have seen two ways to provide an update mechanism:

- We can store all the coefficients in the FPGA's block RAM, then update this memory when the user asks to reload the coefficients,
- or we can reload the coefficients from the main system memory each time they are used, so that those that we use are always up-to-date.

If we want to opt for the first solution, we have to find enough memory blocks to host the coefficients. An ideal target for the filter length being around 3000 samples (which, at 8000 Hz, make up 375 ms), the coefficients being 16-bit wide, we would thus need:

$$3000 * 16 * 48 = 2304000 \text{ bits} = 2.19 \text{ MB}$$

in order to store all the coefficients. This roughly corresponds to half of the total memory bits that are available to us.

We chose to reload coefficients each time they are used. This makes the design area-efficient, but creates a time constraint in that we have to ensure the memory is free to fetch the coefficients when needed.

5 Implementation details

From a global point of view, the implementation is described by the block diagram on Figure 9.

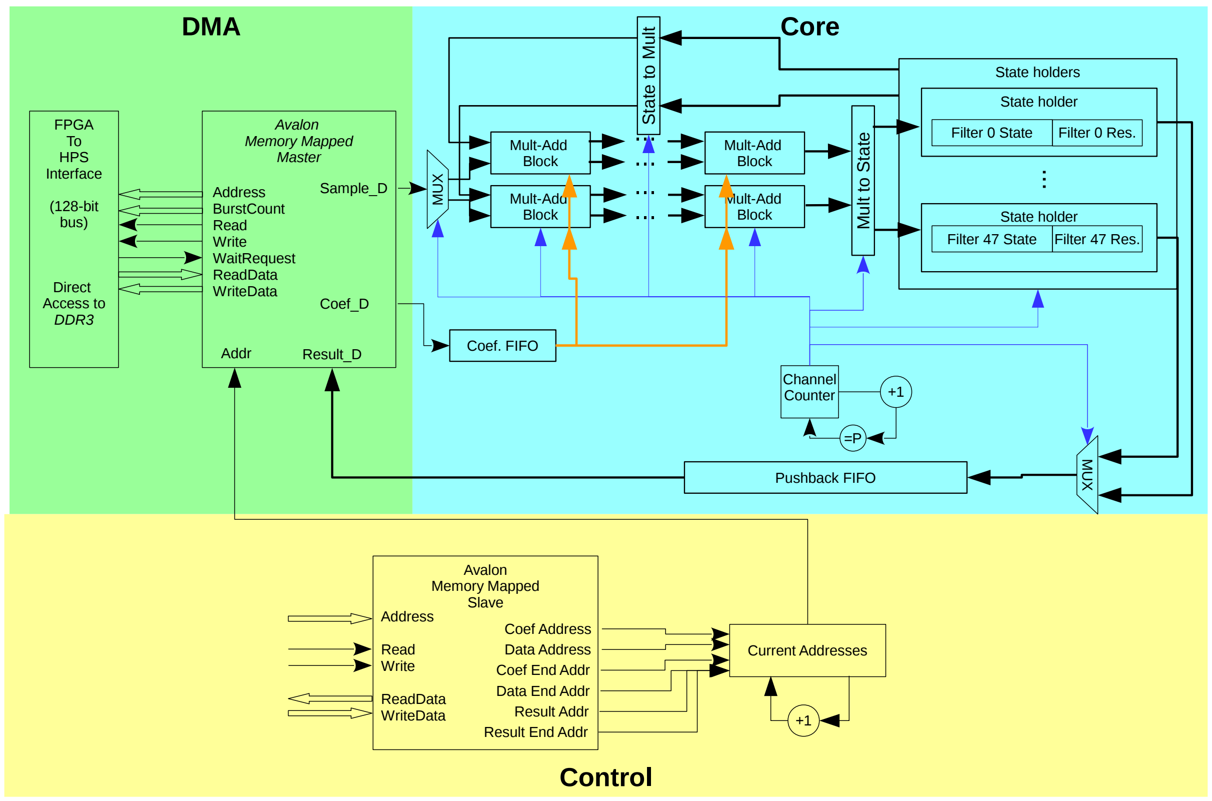


Figure 9: Global view of the system

We can divide the implementation into three main parts:

- The computational core, that performs the actual computations,
- The memory manager, that performs the read / write and data distribution operations,
- The slave, that handles the filter registers and allows the hardware processor to write to the registers.

Each of the sections thereafter is detailing one of these points.

5.1 Communication with the memory: DMA

We implemented the DMA using three distinct synchronous state machines, driven by a synchronous state machine. These state machines are shown on Figure 10.

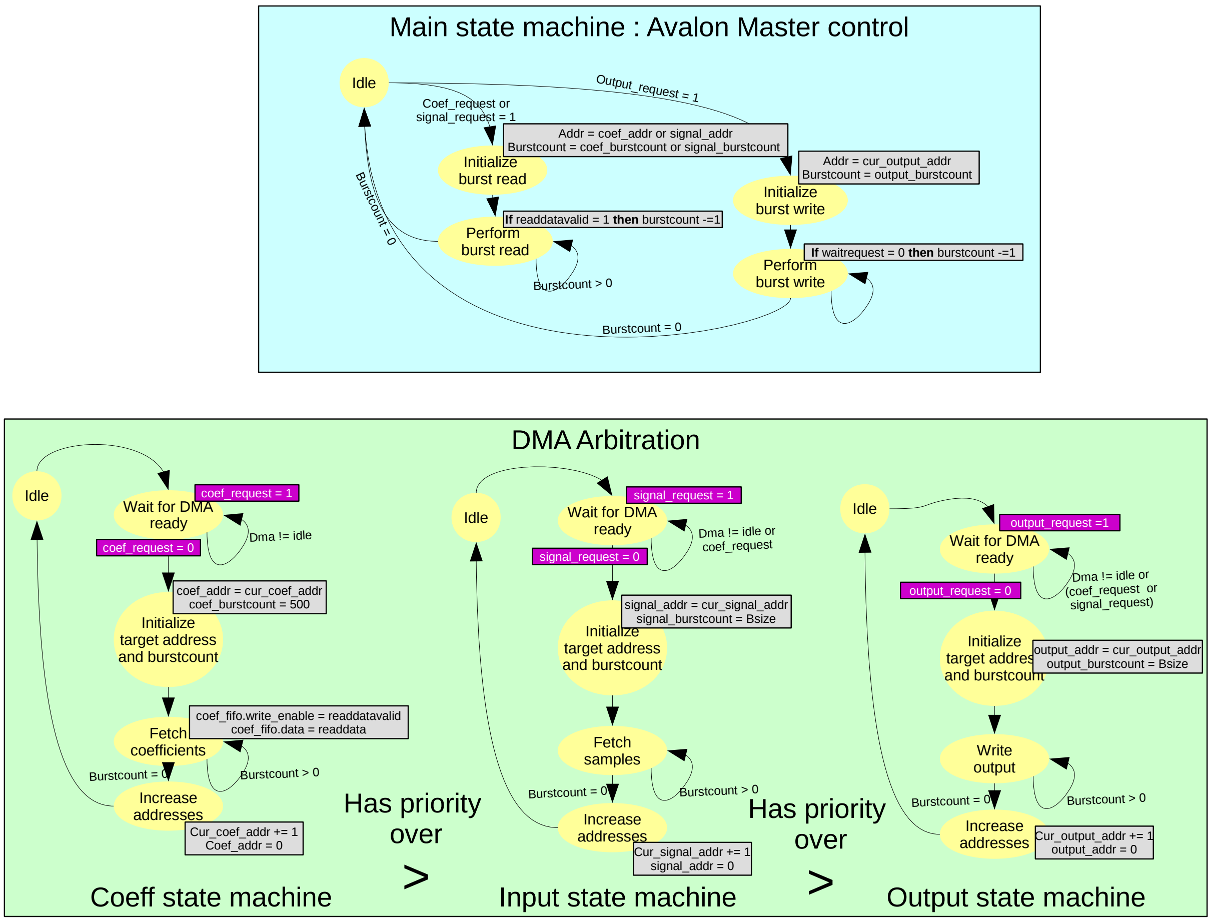


Figure 10: State machines driving the DMA. The main state machine controls the Avalon bus; its parameters are given by three state machines (coefficients, input, output); these three state machines also route the Avalon bus' input and output values to the rest of the system.

The main state machine drives the actual DMA read / write process through burst transfers. Several parts of the design can make use of the memory:

- The input of samples is requested at regular intervals ($\frac{B}{fs}$ cycles),
- Coefficients are asked on-demand, when the global coefficient FIFO is less than half full,
- Output is requested by the state holders, when one output block is ready (so that output gets written at the same rate as input).

Since these parts may request memory access at concurrent times, priorities have been set: the coeffcients have priority over the input, which itself has priority over the output. We run an example of Figure 11 where there are such conflicts.

In order to avoid DMA access conflicts, one has to make extensive use of the on-FPGA block RAM components and extend the FIFOs.

The signals available at the DMA component are shown on Figure 12.

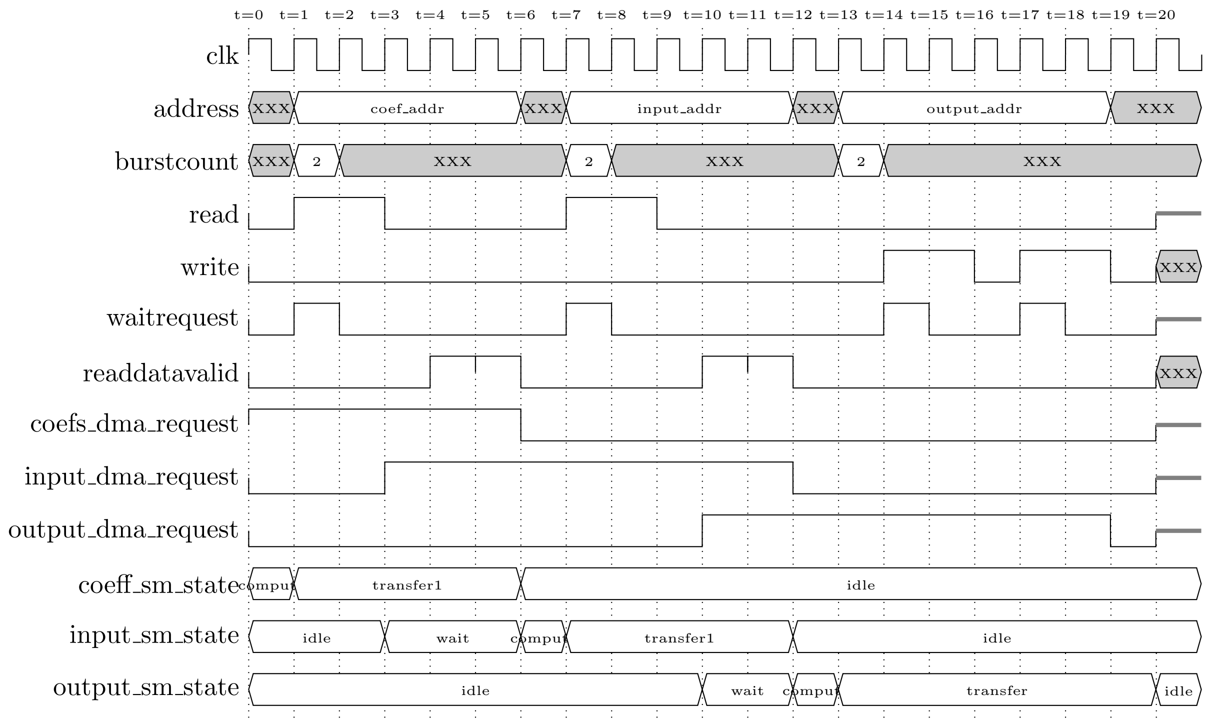


Figure 11: Example of a run of the state machines with coefficient reading, then input reading, then output writing. Note that for this graph to fit on the page, burst transfers are limited to 2 elements.

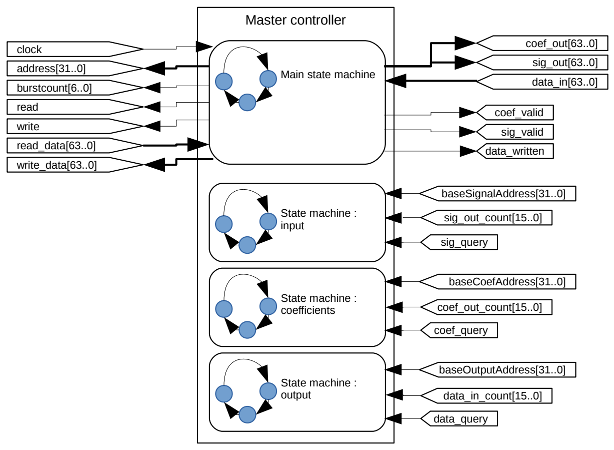


Figure 12: DMA block diagram

5.2 Configuration : Avalon slave

The slave is the part of the hardware that holds the settings defined by the client system and passes them to the rest of the filter. It has the following register map:

Register:offset	Name	Width	Direction	Description
0:0	status	1	in	Enable / disable flag
0:1	status	1	out	Filter running indicator
1	dataInAddr	32	in/out	Input buffer address in memory
2	dataOutAddr	32	in/out	Output buffer address in memory
3	coefsInAddr	32	in/out	Coefficient pool address in memory
4	dataLength	32	in/out	In & out buffer size (both buffers must have the same)
5	coefsLength	32	in/out	Coefficient pool size

Table 2: Register map of the IP component

The slave is a *memory-mapped* slave. This means it has to be connected to its master through a memory-mapped interface: a memory address range available at the master's side will be mapped to the slave registers.

5.3 Core : sample filtering in pipelines

In our architecture, the input samples go through one of P pipelines. P is user-configurable; this number determines how many channels are processed in parallel. All the pipelines have the same length, which is B (the length of the block for the overlap-add).

5.3.1 Periodicity

The whole filter is entirely periodic. Thus, it is possible with a single counter that goes from 0 to $\lceil \frac{C}{P} - 1 \rceil$ to know the current filter state at all times.

5.3.2 Multiplier blocks

We devised a block for implementing the multiply-and-add feature of the FIR fiter. A sample block includes:

- A multiplier and an adder, which signal widths do match Altera's DSP blocks requirements -thus, Quartus will recognize them
- A FIFO that holds some coefficients; this number is adjusted so that the rate of the coefficients arriving via DMA do not prevent multiplications from happening. We compute the right number of coefficients to be contained here in section 5.4.

The multiply-and-add block is illustrated on Figure 13 in an architectural point of view.

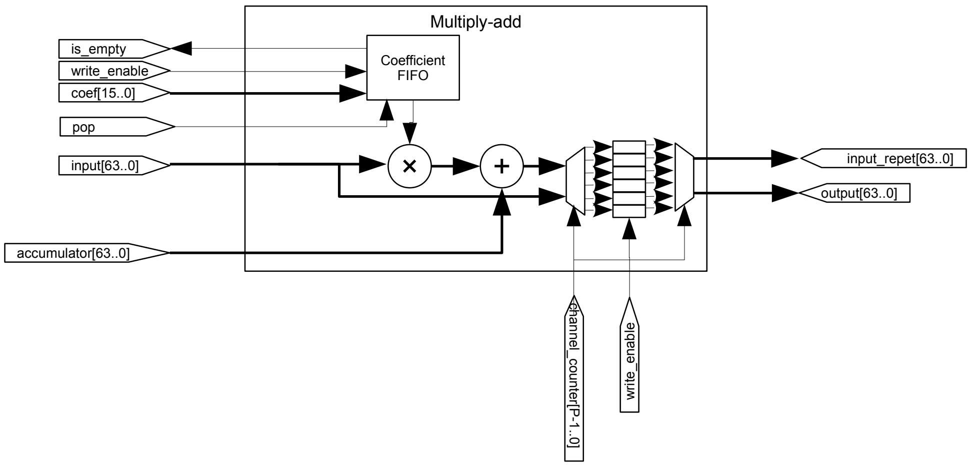


Figure 13: Multiply-and-add block, architecture

Altera provides in [2] an architectural template one has to follow in order to synthesize a DSP multiplier. Figure 14 shows this template.

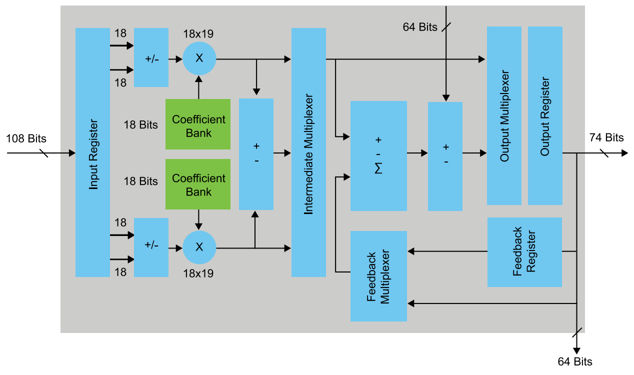


Figure 14: Altera reference DSP implementation

This block is made of a 18×18 multiplier (of which only 16 bits for each multiplication are used), and a 64-bit adder. Altera's DE1-SoC has 174 DSP blocks with such a component ([3]).

The coefficients are taken from the FIFO; to this effect, the "pop" signal is asserted before a sample gets fed into the pipeline; the coefficient is available at the FIFO's output one cycle after the "pop" signal is asserted.

This block is part of a pipeline. Each pipeline will process $\lceil \frac{C}{P} \rceil$ channels, thus each block has to store the intermediate results it has got for each channel, in order to pass all the channels to its successor in respect to the order of the channels.

Figure 15 shows a behavioral description of the multiply-and-add block.

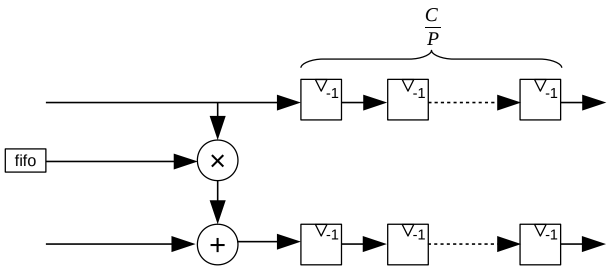


Figure 15: Multiply-and-add block, behavior

The DSP blocks are able to compute the 18×18 multiplication in one cycle at a frequency of 48 MHz, given Altera's documentation shown on Figure 5. We make the hypothesis that the accumulation is done in the same cycle[2]. One cycle later, the result is stored in the right register.

The timing diagram on Figure 16 shows a simulation that uses this hypothesis; we inject a new sample every two cycles in the pipeline.

[2]In order to check this, hypothesis, one has to run the actual filter on the FPGA

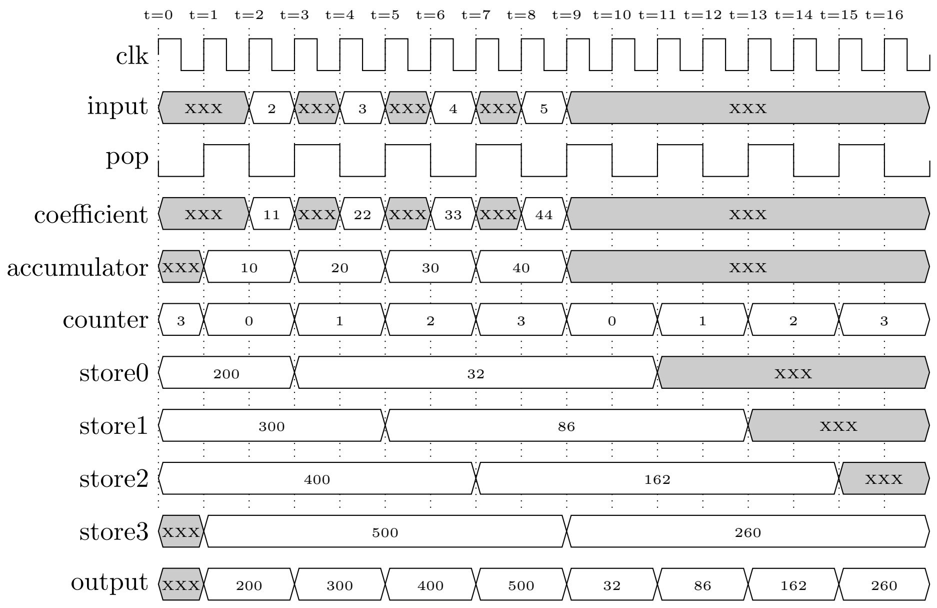

t=0 t=1 t=2 t=3 t=4 t=5 t=6 t=7 t=8 t=9 t=10 t=11 t=12 t=13 t=14 t=15 t=16
clk
input
XXX 2 XXX 3 XXX 4 XXX 5 XXX
pop
coefficient
XXX 11 XXX 22 XXX 33 XXX 44 XXX
accumulator
XXX 10 20 30 40 XXX
counter
3 0 1 2 3 0 1 2 3
store0
200 32 XXX
store1
300 86 XXX
store2
400 162 XXX
store3
XXX 500 260
output
XXX 200 300 400 500 32 86 162 260

Figure 16: Example of a run of a multiply-and-add block with $\lceil \frac{C}{P} \rceil = 4$ channels per pipeline

Each computed sample waits for $\lceil \frac{C}{P} \rceil$ cycles, since during this time, the next stage of the pipeline is processing another channel. For a given channel, the time between the injection of each sample is thus $\lceil \frac{C}{P} \rceil$.

5.3.3 Pipeline architecture

The pipelines are mostly concatenations of the multiply-and-add blocks described in the paragraph above. Figure 17 shows a block diagram of the pipeline system where $P = 2$, $B = 4$. The following interconnections are done:

- Each multiply-and-add block's accumulator output is connected to the accumulator input of its successor;
- Each block's input replica is connected to the its successor's sample input,
- All the "pop" signals for the coefficient FIFOs are connected together to a signal on the DMA controller, that is asserted one cycle before the next sample is inserted into each pipeline,
- Each block's FIFO "write enable" and "data" signals are linked to a global coefficient FIFO. The data bus is shared, but only the target FIFO has its "write enable" signal asserted.
- The channel counter is routed to all the multiply-and-add blocks through a single bus.
- The "write enable" signal for the intermediate registers is the same for all the multiply-and-add blocks; it comes from the DMA as well as the "pop" signals do.

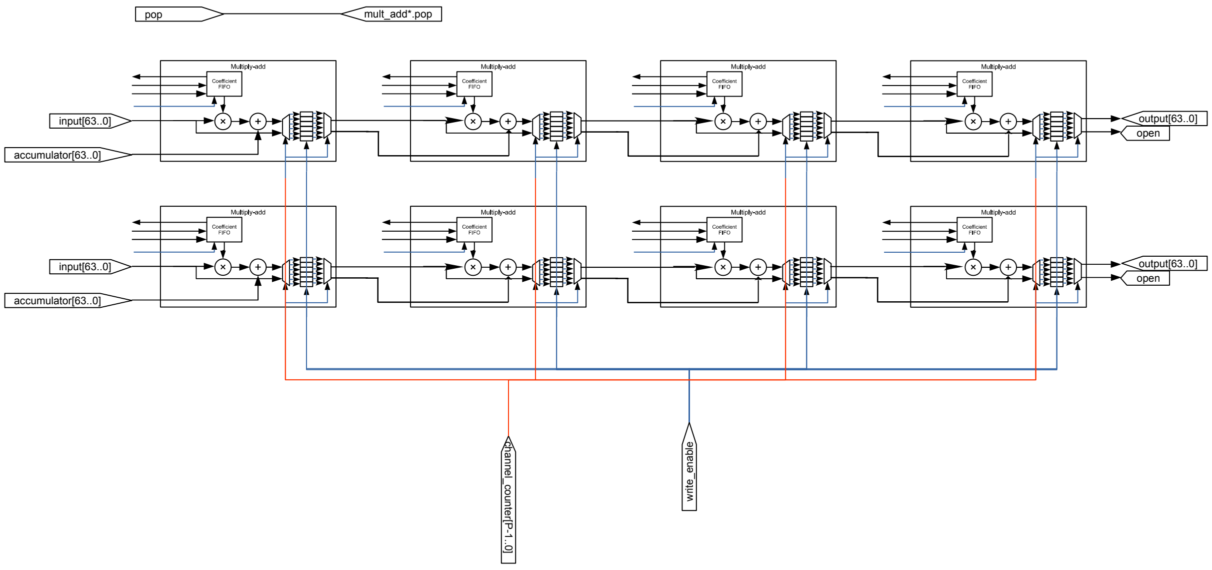


Figure 17: Architecture of pipelines

The P pipelines must have C outputs, one output per channel. To achieve this, a combinational function maps the outputs using the following formula : if k is the current value got from the global counter, p is the number of the pipeline, then the corresponding channel c is :

$$c = p + k \times P$$

A "valid" signal is asserted for the channels that have a sample coming out of a pipeline.

We can compute the time one sample spends into the pipeline : since each pipeline has $\lceil \frac{C}{P} \rceil$ channels to process, and the filter is of length L, processing one sample every two cycles, one sample takes

$$n_{cycles} = 2 \times L \left\lceil \frac{C}{P} \right\rceil$$

cycles from the beginning of the pipeline to the end.

The resulting sample actually becomes available when it is written into memory : there is a block write request every B completed samples; thus, the first sample of the block (that will spend the longest time in the system) will spend approximately

$$N_{cycles} = 2 \times \left(L \left\lceil \frac{C}{P} \right\rceil + B \right)$$

cycles in the system. Note that we don't count here the overhead of DMA initialization.

If $L = 100$, $C = 48$, $P = 4$, $B = 43$ then the samples will spend at most 2486 cycles in the pipeline, plus the DMA transfer time (that can be two cycles if the bus is available, or longer if it's not, since the output has the lowest priority). Since the sampling frequency is 8000 Hz, each sample has at most 6000 cycles to spend in the pipeline, which allows this configuration to fit in the timing requirements.

5.3.4 Implementation of overlap-add

Overlap-add requires memorizing some samples. We created a component that takes as an input a computed sample, to feed it into:

- a FIFO, if it is part of the completed samples (which is the blue result on figure 8),
- a register file if it is part of the overlapping samples.

There are as many such components as there are channels. Their data comes from the interconnect provided at the end of the pipelines.

A block schematic for the state holder component is shown on figure 18.

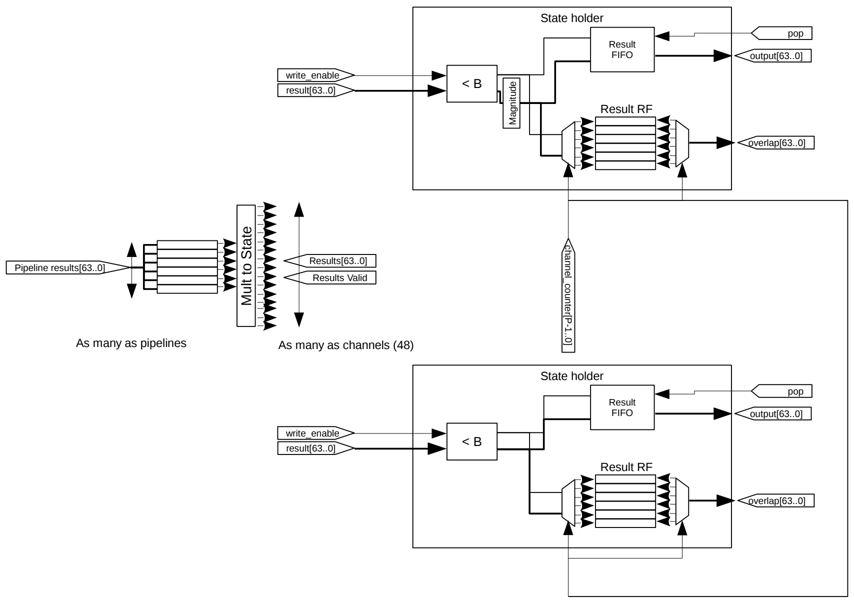


Figure 18: Two state holders, and the interconnect coming from the pipelines

Both the register file and the FIFO are connected to the exterior of the state holder.

Handling of overlapping samples The overlapping samples will be fed back to the pipelines. A dispatcher combinational function maps the registers' contents to the right pipeline; the block schematic for it is shown on Figure 19.

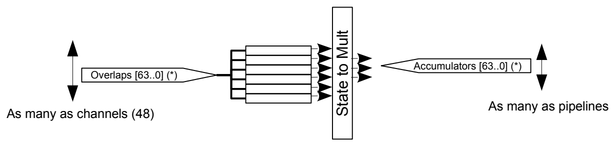


Figure 19: State holder to pipeline wayback block

In order to determine which pipeline the accumulator should be injected into, we use the following formula:

$$p = (c \mod P) \text{ if } 0 \leqslant c - k\left\lceil\frac{C}{P}\right\rceil \leqslant P \text{ else discard}$$

Handling of completed samples The completed samples wait in the FIFO until the output state machine in the DMA pops them. This state machine is triggered when the `usedwords` signal on one FIFO reaches the size of a block B.

5.4 Coefficient reloading

The filter coefficients have to be able to be modified by the user at runtime. Thus, they are taken from the shared memory.

Coefficients in each multiply-add block are held inside a FIFO which is internal to the multiply-add block. In order to avoid fetching coefficients too often, there is a global FIFO that can hold up to 128 16-bit coefficients.

This FIFO is visible on figure 9 as `coef_fifo`. It has an output on 8 shared buses, that distributes the coefficients to all the individual FIFOs of the multiplier blocks. When doing this, the right FIFOs are selected by asserting their "write enable" signal.

The global FIFO is 128-bit wide; thus, it can feed 8 coefficient FIFOs at the same time. The coefficients are distributed at each cycle, in the order of the multiply-add blocks[3].

Loading coefficients from the memory involves using the DMA master. To this effect, the following procedure happens :

- At each cycle, the number of remaining words in the global FIFO (`num_words`) is checked. If `num_words` < 4, then the `coef_query` signal (that goes to the DMA) is asserted.
- The coefficient state machine in the DMA arbitration raises the `coefs_dma_request` signal to the DMA controller.
- The coefficients are fetched like can be seen on Figure 11. The data is pushed to the global FIFO.

We have to make sure that there will be enough coefficients for processing samples at all times (i.e. no coefficient FIFO will get empty while processing). Obviously, there are much more coefficients consumed at each cycle than we are able to provide. The multiply-and-add coefficient banks must thus be large enough to compensate the deficit of rate.

We can compute the minimal size for a multiply-and-add FIFO coefficient bank.

To make the computation simple, let's begin by making the hypothesis that the DMA is always free when it's needed by the global FIFO. Then, 8 coefficients are distributed each time one coefficient is popped from all the FIFOs.

The length of the filter is L, we inject a coefficient every $\frac{PB}{8}$ cycles, and one coefficient is consumed every 2 cycles (since we assume we inject a sample every 2 cycles in the pipeline).

Thus the consumption rate of coefficients is $\frac{1}{2}$ coefficient / cycle, and the arrival rate is $\frac{8}{PB}$ coefficient / cycle. The total rate is thus

$$r = -\frac{1}{2} + \frac{8}{PB} = \frac{16 - PB}{2PB}$$

Since we have to have coefficients for $2L\frac{C}{P}$ cycles, we will need a minimum number of coefficients to be

$$L_{coefbank} = 2L\frac{C}{P} \times \left|\frac{PB - 16}{2PB}\right| = \frac{LC(PB - 16)}{2P^2B}$$

When $L = 100$, $P = 4$, $C = 48$, we obtain $L_{coefbank} = 545$ coefficients per multiply/add block.

Note that this result doesn't seem correct : we have exactly 4800 coefficients, and we need to store $545 \times PB = 545 * 172 = 93740$ coefficients in the banks.

6 Results

The current implementation is not complete, since we haven't tested it yet. This section details the results that we can get at this stage of the implementation.

6.1 Obtained filter

The reader is reminded that we call B the block size, L the filter size (cf. section 1). The number of taps in the filter is equal to B. The filter length is currently limited to the filter size, thus $L = B$.

We can compute the maximum length of the filtering part.

Two hypotheses can be made: either the number of taps is the bottleneck (which is the case in this implementation), or it is the processing time.

Hypothesis: the bottleneck is the number of taps In this case, let's consider P pipelines. We have 174 multipliers available, thus the maximum length is simply

$$L = \frac{174}{P}$$

Here, we impose $B = L$ (since there's no return path for the moment), which validates the hypothesis.

We thus have, in function of P, the number of pipelines that we can have:

[3]It is thus advised that the block length is a multiple of 8

Pipelines	1	2	3	4	5	6	7	8
Maximum length	174	87	58	43	34	29	24	21

Hypothesis: the bottleneck is the processing time In this case, we can compute the time available for doing the computation when the block size is B : it is $t_{avail} = B \times t_{sample} = B\frac{1}{f_s}$.

In terms of clock cycles,

$$c_{avail} = t_{avail} f_{clock} = B \times \frac{f_{clock}}{f_s}$$

The total time it takes to process one block is the time to introduce it into the pipeline, plus the time the last samples take to run through it. More formally, in clock cycles, it is $c_{taken} = c_{introduce} + c_{pipeline}$.

We have

$$c_{pipeline} = 2 \times \left(L \left\lceil \frac{C}{P} \right\rceil + B \right)$$

since each step of the pipeline takes two clock cycles for each sample processed by a tap; the introduction time in cycles is

$$c_{introduce} = \left\lceil \frac{B+m}{P} \right\rceil$$

The maximal filter length is thus found by taking $c_{avail} = c_{taken}$, i.e.

$$L = \left\lfloor \frac{1}{\lceil \frac{C}{P} \rceil} \left(\frac{1}{2} \left(B \times \frac{f_{clock}}{f_s} - \left\lceil \frac{B+m}{P} \right\rceil \right) - B \right) \right\rfloor$$

With $B = 43$, $P = 4$, $m = 1000 = L$, $\frac{f_{clock}}{f_s} = 6000$, $C = 48$, we find $L = 9909$ coefficients at most.

The current implementation doesn't however allow such a length, for it would require for each sample to do several turns of pipeline, and we haven't implemented a return path yet. Still, it is theoretically possible to achieve such a filter length.

6.2 Missing components

A return path for samples inside the pipeline can be used to make it longer. Such a return path would take the form of an interconnect between the end of a pipeline and its beginning; however, if samples spend longer in the pipeline, the periodicity of the filter changes (it is multiplied by the number of turns spent in the pipeline).

Implementation of such a return path would require reviewing all the components that depend on the global counter to adapt their formulae to the new period.

Also, we are using fixed-point arithmetic; thus, after the multiplications, we have to shift the result by some quantity. We did not implement this feature yet.

7 Conclusion and further work

We have implemented, yet not tested, an FIR filter that is able to process simultaneously the 48 channels that the Pyramic array has.

The filter that we obtain is more area-efficient than an implementation that would involve storing all the coefficients onto the FPGA itself. We didn't choose to apply specific optimization techniques that are usually applied to filters with static coefficients, since that would have involved doing strong hypotheses on the coefficients. Nevertheless, the algorithm we implemented is not the best for what can be done using the overlap-add technique.

G. Clark in [1] reduces the computational complexity to $\mathcal{O}(L \log(L))$ by doing a Fast Fourier Transform. The implementation in hardware is quite different because of the circular convolution theorem, because convolution is no longer needed when in the Fourier domain. In such algorithms, the most significant part of complexity comes from tge FFT and inverse FFT that are performed on the signal.

Regarding the hardware, some work is still to do, in particular in allowing longer filter lengths by optimizing the pipeline and allowing the samples to do several turns of pipeline.

The Pyramic array is already capable of capturing and outputting stereo audio; one further step in the project would involve devising a hardware beamformer, that sums up the filtered signals. With such a component, one can build a complete hardware chain from the capture to the output.

This project, along with the previous semester project on an extension for the original FPGA design, will be presented at ENS Rennes as Master's projects.

References

[1] G. Clark, S. Parker, and S. Mitra. A unified approach to time- and frequency-domain realization of fir adaptive digital filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31(5):1073–1083, Oct 1983.

[2] Altera Corporation. *Enabling High-Performance DSP Applications with Arria V or Cyclone V Variable-Precision DSP Blocks (White Paper WP-01159-1.0).* 101 Innovation Drive, San Jose, CA 95134, USA, 2011.

[3] Altera Corporation. *Cyclone V Device Overview (CV-51001).* 101 Innovation Drive, San Jose, CA 95134, USA, 2016.

[4] J.B. Evans. Efficient FIR filter architectures suitable for FPGA implementation. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 41(7):490–493, jul 1994.

[5] Patrick Longa and Ali Miri. Area-efficient FIR filter design on FPGAs using distributed arithmetic. In *2006 IEEE International Symposium on Signal Processing and Information Technology.* Institute of Electrical and Electronics Engineers (IEEE), aug 2006.

[6] Juan Azcarreta Ortiz. Pyramic array - An FPGA based platform for many-channel audio acquisition. Master's thesis, École polytechnique fédérale de Lausanne, Lausanne, Switzerland, 2016.

[7] Imopishak Thingom and Pinky Khundrakpam. FPGA implementation of FIR filter using RADIX-2 r. In *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET).* Institute of Electrical and Electronics Engineers (IEEE), mar 2016.

A Project architecture

A repository containing an up-to-date version of the code is available at `https://github.com/cferr/dsp.git`.

The structure of the project is as follows:

- The `hdl` directory contains the VHDL files that make up the FIR design, plus a toplevel file for the DE1-SoC. Namely, these files are:
  - `dsp_toplevel.vhd` is the top-level file that interconnects the Avalon master, the Avalon slave and the DSP core together.
  - `avalon_master.vhd` is the file containing the Avalon master and the DMA state machines.
  - `slave.vhd` is the file containing the Avalon slave.
  - `state_to_mult.vhd` is the file containing the dispatcher function from the states to the pipelines.
  - `mult_to_state.vhd` is the file containing the dispatcher function from the pipelines to the states.
  - `dspcore.vhd` is the file containing the implementation of the pipelines (shown as the "core" part on Figure 9).
  - `dsp_utils.vhd` is a package that contains instatiation templates for all the components used in the filter's core, plus global user-defined constants. See the following appendix for more details.
  - `dma.vhd` contains the DMA master and its state machines.
  - `channel_counter.vhd` contains the global counter.
  - `DE1_SoC_top_level.vhd` is the top-level entity that instantiates the whole system.
  - `mult_block.vhd` contains the multiply-and-add block.
  - `fifo.vhd` contains the instantiation template for Altera IP FIFOs.
- The `quartus` directory contains an Altera Quartus project file (system.qpf). It has been created with Quartus 16.0; any later version of Quartus should work with this project, provided that it uses the same implementation of the Altera IP FIFO core.
  It also contains a Qsys file (system.qsys), that incorporates an (unfunctional at this stage) hardware processor system (HPS) configuration, and an instance of the FIR filter.
  A makefile can be found inside this directory; in order to use it, one has to be in an "Embedded command shell" from Altera (that can usually be found at `QUARTUS_ROOT/embedded/embedded_command_shell.sh`).

B User-defined constants

The following constants are defined by the user in `dsp_utils.vhd`:

Constant	**Description**
INPUT_SIGNAL_WIDTH	Width of the input samples (default : 16 bits). Should be a power of two.
RESULT_WIDTH	Width of the result (default: 64 bits).
NUM_PIPELINES	Number of parallel pipelines (default: 4). Corresponds to P.
FILTER_BLOCK_LENGTH	Length of the blocks (default: 24). Corresponds to B.
OVERLAP_LENGTH	Length of the overlapping part.
NUM_CHANNELS	Number of channels (default: 48).
CHANNEL_COUNTER_WIDTH	Width of the channel counter.
GCOEF_FIFO_NUM_WORDS	Number of words in the global coefficient FIFO.
GCOEF_FIFO_NUM_WORDS_LOG2	Log2 of the previous number.
COEF_FIFO_NUM_WORDS	Number of words in each multiply-add block FIFO.
COEF_FIFO_NUM_WORDS_LOG2	Log2 of the previous number.
COEF_WIDTH	Width of the coefficients (default: 16).
CHANNELS_PER_PIPELINE	Number of channels per pipeline (equals $\lceil \frac{C}{P} \rceil$).