Master Thesis

Circular microphone array based beamforming and source localization on reconfigurable hardware.

Performed at:
Signal Processing and Speech Communications Laboratory,
Graz University of Technology.

Author:
Boris Clénet.

Supervisor:
Dr. Harald Romsdorfer.

Graz, September 2010.

Abstract

Source localization and beamforming using microphone arrays are common strategies for speech enhancement and source separation in various applications related to voice capturing. This thesis focuses on their implementation on hardware with real time constraints. Several algorithms are used and the idea is to increase performances as much as possible. Therefore, mathematical and physical optimizations are considered, as well as improvements due to the board's specificities. Nevertheless, the work focuses more on handling of basic solutions by the hardware features. A hybrid plateform is at disposal; it puts a reprogrammable computation unit and a more classical CPU together. The improvements are significant and the real-time constraints are achieved notwithstanding the fact that hardware capacities and ressources are limited regarding this application that deals with multiple microphone outputs.

Acknowledgements

I first want to thank Harald Romsdorfer and Gernot Kubin that made this master thesis possible by welcoming me at the Signal Processing and Speech Communication Laboratory, respectively in their capacity as supervisor and head of the laboratory.

Then, Manfred Mücke and Thang Viet Huynh for helping me with the board and Tania Habib who helped me to learn more about beamforming. Wolfgang Jäger and I had very close work together on this project and we gave each other advice and hints: I am really grateful to him for all this.

I also want to thank all the workers at the SPSC laboratory, the administration members, administrators, and Ph.D students whom have ever been kind to me. Finally, all the master students with whom I have been working during these six months are to acknowledge, especially Florian Krebs, Martin Schickbichler, Susanne Rexeis, Niklaus Hammler, and Anna Fuchs.

Contents

List of Figures

List of abbreviations.

ALU	Arithmetic Logic Unit.
AR	ALU's associated register.
AU	Arithmetic Units.
BM	Blocking Matrix.
CC	Cross Correlation.
CPU	Central Processing Unit.
D-CACHE	Data-Cache.
DATARAM	Dual Port RAM.
DMA	Direct Memory Access.
DOA	Direction Of Arrival.
DSB	Delay and Sum Beamforming.
EI	Extension Instruction.
EMAC	Ethernet Media Access Controller.
ER	Extension Register.
EU	Extension Unit cycles.
FPGA	Field-Programmable Gate Array.
FPU	Floating Point Unit.
FR	FPU's associated register.
GIB	Generic Interface Bus.
GSC	Generalized Sidelobe Cancelling.
IDE	Integrated Development Environment.
IRAM	Inherent ISEF's RAM.
IR	Issue Rate.
ISEF	Instruction Set Extension Fabric.

MC	Multiple input Canceler.
MU	Multiplication Units.
NLMS	Normalized Least Mean Square.
PA	Processor Array.
PE	Processor Entities.
SBIOS	Stretch BIOS.
SCC	Stretch-C Compiler.
SCP	Software Configurable Processor.
SIR	Source to Interferences Ratio.
SNR	Signal to Noise Ratio.
TOA	Time Of Arrival.
WR	Wide Register.

Introduction.

Beamforming is of essential importance for a large number of current works in application fields such as telephony, telecommunications, teleconferencing, speech enhancement, and many others. In order to perform properly, all these applications also require precise source localization. It is furthermore well known that microphone arrays can carry out both source localization and beamforming. Many algorithms using sets of microphones have already been developed to perform these operations. The principle of basic algorithms are described in this thesis.

Besides theory of beamforming and source localization, this thesis deals especially with the implementation of these algorithms in real conditions on a hardware plateform. Therefore it explains the porting process between mathematical definitions of the algorithms and their final implementation on a board.

In order to make the computations gain performance, the chosen board (from Stretch® Inc.) is dedicated to multimedia applications: it was designed to process audio and video data. Moreover, it contains a specific calculation unit which is comparable to Field-Programmable Gate Arrays (FPGA) in the way it works. It is also close-connected to the Central Processing Unit (CPU) of the board so that they can interact efficiently. Hence, the flexibility of a CPU and the high computation potential of programable hardware are brought together.

Goals.

The initial aim of this work is to show the reader how to port computation algorithms from their mathematical definition to their implementation on the Stretch® board; Matlab® is used as a link. The main avantages and constraints of the board in terms of optimization, as well as the problems to avoid are described.

The algorithms are related to source localization and beamforming with *circular microphone arrays* as this is a current field of research of the SPSC laboratory. The idea is to optimize the routines as much as possible. Real time considerations have therefore to be taken into account.

Scope of this work.

In oder to make the reader become familiar with the Stretch® board, an introduction-chapter is first going to be made about it (cf. chapter 1). An introduction to the project and its important notions is made as well (cf. chapter 2).

In a second time, chapter 3 describes the handling of audio content for the application. Then, for each algorithm, a chapter describes the porting process from the mathematical and physical definitions to the implementations on the board (cf. chapters 4 and 5). Finally, chapter 6 gathers all the performance results and the general thoughts about the implementations efficiency.

Chapter 1

Introduction to the board.

1.1 Overview of the board.

This section describes the Stretch® S6 PCIe DVR Add-in VRC6416 card that was used for the project. Figure B.1 shows a picture of a similar VRC6016. Figure B.2 additionally lists chip identification numbers. Figure B.3 gives the block diagram of the board with the previous numbered references. The core of the board is the Processor Array (PA). It consists in three S6105 and one S6100 Processor Entities (PE). The S6100 processor has extended features compared to the S6105 which makes it the master of the processor array.

On the left side of processor entities in figure B.3, A/D converters (Techwell® TW2864) are connected to each processor. The A/D converters are for video and audio purposes but only the audio paths is needed in the application.

The main memory of each processor entity appears on their right side in figure B.3. Each PE has 128MBytes of main memory. Figure B.4 focuses on the processor. The main blocks that were used for the application are the DDR2 controller and the S6SCP Engine. The Quad Dataport—which is for video applications only—, the Low-Speed Peripherals, the Enhanced Generic Interface Bus (eGIB), and the Ethernet Media Access Controller (EMAC) were not used.

1.2 The Xtensa processor.

The Xtensa LX Dual-Issue VLIW processor consists in three calculation devices as figure 1.1 shows. The Floating Point Unit (FPU) with its corresponding register (FR) and the Arithmetic Logic Unit (ALU) with its register (AR) are the two "classical" ones.

The Instruction Set Extension Fabric (ISEF) with the inherent ISEF RAM (IRAM) and the Wide Registers (WRS) is the reprogrammable part of the chip that was referred in the introduction.

Normal C functions are executed either on the ALU or on the FPU whereas time critical or costly parts of the code can be outsourced to the ISEF, where specific calculations are performed in parallel. This

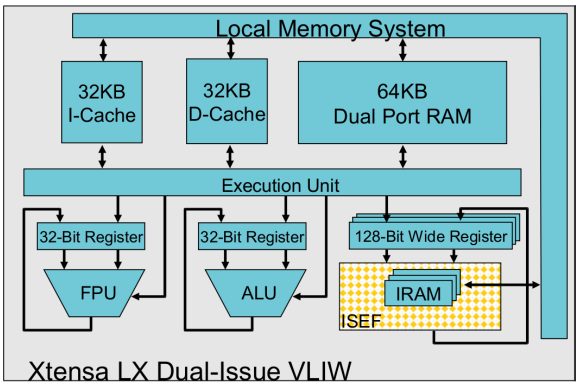


Figure 1.1: Organization of the Xtensa.

There are several possibilities to provide data from the main memory to the calculation devices.

- When using the ALU, data can be transfered over the Data-Cache (D-CACHE) or over the Dual Port RAM (DATARAM).
- When using the ISEF, data can be transfered over D-CACHE, DATARAM, or IRAM.

Direct Memory Access (DMA) enables the IRAM to be filled directly from the main memory. Otherwise it has to be done over the WRs. It is also possible to use the Extension Register (ER) from inside the ISEF. Intermediate results can be stored there, but no access from outside the ISEF is possible.

1.3 The pipeline structure.

The Xtensa processor uses a five stages hardware *pipeline*. It is possible to launch at most one instruction in the pipeline per CPU cycle. Every stage takes one cycle and the pipeline mechanism enables to output one result per cycle. The stages are listed in the following. Figure 1.2 shows the schedule of the pipeline.

An ISEF configuration results in n extra cycles known as Extension Unit cycles (EUs). These cycles are depicted on the right side of figure 1.2. The ISEF configuration gives rise to a pipeline extension. It is possible to add up to 31 EUs to the normal pipeline stages.

I	Instruction fetch.
R	Register file read and instruction decode.
E	Execute.
M	Data cache read.
W	Register file write.
EUn	Extension Unit cycles.

If dependencies between consecutive instructions happend, *stalls* appear in the pipeline. Avoiding these time-consuming stalls requires careful programming. Figure B.5 illustrates the effect of dependencies. At address 0x4001c0db, the instruction `wraputi` needs the result of the ISEF's Extension Instruction (EI) at 0x4001c0cd. The EI is not finished when its result would be needed by the `wraputi` instruction and therefore, the whole pipeline is stalled for 13 cycles.

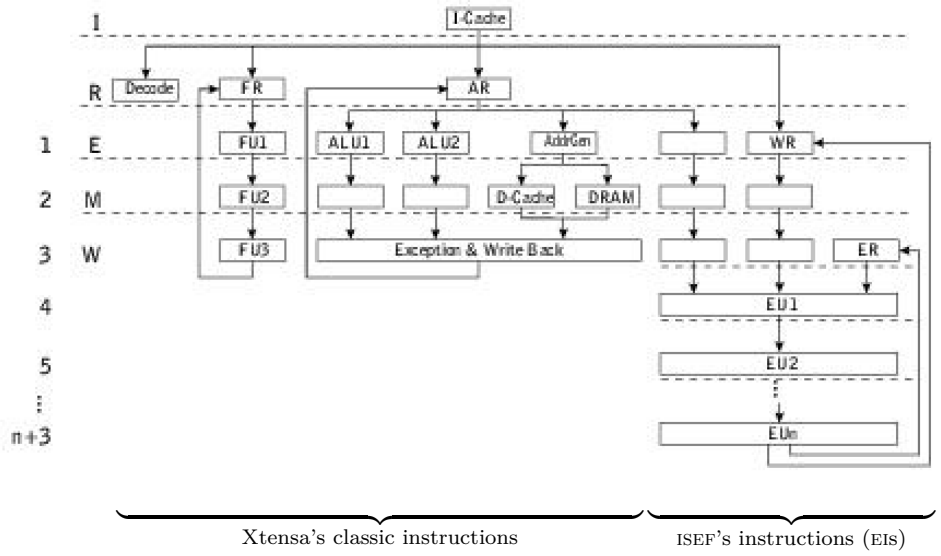


Figure 1.2: The extended pipeline structure.

1.3.1 Issue Rate.

In the previous sections, CPU cycles are often mentioned. It is important to notice that the actual CPU and the ISEF do not run necessarily at the same frequency. The cycles of the ALU and the ones of the ISEF are linked by the "Issue Rate" (IR). It is a number that gives the ratio between the Xtensa's running frequency and the ISEF's one.

For example, an IR of 1 means that ISEF issues an instruction every time the ALU issues one. An IR of 3 means that ISEF issues an instruction at the

beginning of every group of three instructions from the ALU. The default value for the IR is 1:1. It the compiler can not reach the target frequency, it indicates the achieved frequency in a report file (cf. section 1.5). The IR has then to be increased so that the target frequency for the ISEF is achievable.

1.4 Use of software.

The Stretch® Software Configurable Processor (SCP) is programmable in ANSI-C. Although there are special hardware-related parts of the code (i.e., especially those that command the ISEF) which have to be programmed in Stretch-C. Stretch-C varies from ANSI-C; there are additional, more width-flexible data types and certain hardware-related functions, definitions, and intrinsics.

1.4.1 Defining and using EIs.

Apart from the standard ANSI-C files (∗.c), Stretch-C files (∗.xc) are used to define EI. The following code example demonstrates the construction of an EI.

```
1  #include <stretch.h>
2  static se_sint<64> sumver[4];
3  SE_FUNC void CROSSONISEF
4      (SE_INST CC_MAC, SE_INST CC_INIT_MAC, SE_INST CC_FIN_MAC,
5      WRA A, WRB B, WRA *Y_1, WRB *Y_2)
6  {
7      se_sint<16> b;
8      ....
9      *Y_1 = (sumver[1], sumver[0]);
10 }
```

In line 1, the preprocessor includes the standard stretch library for Stretch-C files. It is necessary to use the WR data type and all the specific instructions. Line 2 defines a static array. If a variable is defined as static, it is stored in the ERs. The data type `se_sint<64>` defines a signed integer with a width of 64bits.

Lines 3, 4, and 5 define an EI. The keyword `SE_FUNC` identifies the function as EI and `SE_INST` gives it its name (i.e., how the function can be called from ordinary C code). Common parts of several EIs can share the same hardware resources of the ISEF. In our case, the instructions `CC_MAC`, `CC_INIT_MAC` and `CC_FIN_MAC` differ slightly from each other but the main part is nevertheless the same. Therefore, all three instructions are outlined in one function.

In total, each EI call can transfer data from 3 WRs to the ISEF and output data to 2 WRs. Line 5 defines four variables, located in the WRs (2 input values, 2 output values). It is possible to tell the compiler which WR (A or B) should be used. Line 7 defines a local signed integer with a width of 16bits. Data types `se_sint<n>` and `se_uint<n>` refer respectively signed and

unsigned integers with a bit-width of n. In line 9, the two return values (of width 64bits) are stored in one WR (of width 128bits).

1.4.2 Handling the wide registers.

Figure 1.1 shows that the ISEF is accessible over the WRs and the over the later-discussed IRAM via DMA. To load the WRs with data to be processed on the ISEF, it is necessary to use a couple of special functions within the *.c file.

```
1 ....
2 WRGETINIT(0, p_x2);
3 WRGET0I(&wr_x1, 1);
4 ....
5 WRPUTINIT(0, p_acc);
6 WRAPUTI(wr_y_1, 4);
7 WRPUTFLUSH0();
8 WRPUTFLUSH1();
9 ....
```

Line 2 and 3 enable data transfer to the WRs. Line 2 initializes the transfer from a memory place in the DATARAM to the WRs. The zero declares, that the source pointer should be incremented after each access. Line 3 copies the data (1 byte) to the WR. Line 5 to 8 retrieve data from the WRs. Line 5 initializes the transfer from the WR to the destination located in the DATARAM. Again, the destination pointer should be incremented after each access. Line 6 copies 4 bytes of data from the WR to the destination. Line 7 and 8 are additionally necessary to complete the data transfer.

1.4.3 Handling the IRAM.

Figure 1.1 shows the ISEF inherent location of the IRAM. Two data paths to the IRAM are possible. The first one, over the WRs is not efficient. The second one is more direct and to prefer: it uses DMA to transfer data from the main memory to the IRAM.

```
1 *.c file:
2 se_iram_handle *hA;
3 hA = se_iram_get_handle(crossisef, A, SE_IRAM_ROW_MAJOR, 0);
```

The IRAM handle `hA` is an "access gate" to array A (line 3) in the IRAM (`crossisef` is the name of the ISEF configuration). If the array as several dimensions, one should specify along which dimension the increment is done first (with arguments `SE_IRAM_ROW_MAJOR` or `SE_IRAM_COL_MAJOR`). In total, there are 32 banks of IRAM, each bank should be accessed only once per ISEF cycle, otherwise stalls of many cycles may occur. Therefore the distribution of variables has to be wide spread over the banks. The following code example shows the possibilities defining variables within the IRAM structure.

```
1 *.xc file:
2 static se_sint<16> A[1024][8];
3 SE_MEM(A);
```

```
4 static se_sint<32> B[1024][4];
5 SE_MEM(B);
6 static se_sint<64> C[1024][2];
7 SE_MEM_LOCAL(C);
8 static se_sint<128> D[1024][1];
9 SE_MEM_LOCAL(D);
```

The maximum depth of an array is 1024 and only sizes that are powers of two are possible. A group of 8 banks can be used in four different ways: the width of the data type can vary between 16, 32, 64 and 128bits. Increasing the width of the data type decreases the second dimension of the array. After the desired array has been defined, it has to be mapped into the IRAM by using the intrinsic SE_MEM or SE_MEM_LOCAL. It is not possible to access the array over DMA with the latter.

1.4.4 The BIOS.

The Stretch® BIOS (SBIOS) provides the foundations for application running on the S6000 family of processors. It important parts regarding the project's applications are the DMA, the memory management, clock routines, some utility functions, and the data types. The used SBIOS functions are going to be demonstrated with an example.

```
1  #include <sx-misc.h>
2  #include <sx-mm.h>
3  #include <sx-mmdma.h>
4  #include <sx-timer.h>
5  //create memory pools
6  static sx_int8 ddr_pool_space[567] SX_DDR;
7  static sx_int8 dram_pool_space [567] SX_DATARAM;
8  sx_mm_pool *ddr_pool;
9  sx_mm_pool *dram_pool;
10 ddr_pool = sx_mm_create(ddr_pool_space, sizeof(ddr_pool_space));
11 dram_pool = sx_mm_create(dram_pool_space, sizeof(dram_pool_space));
12 //allocate memory
13 p_samples_1 = sx_mm_zalloc(ddr_pool, framesize_100ms_inbytes);
14 p_frame_1 = sx_mm_zalloc(dram_pool, framesize_100ms_inbytes);
15 //initialize dma channel
16 sx_mmdma_chan *p_channel_1;
17 sx_mmdma_chan_config *p_ch1_conf;
18 p_ch1_conf = (sx_mmdma_chan_config *)sx_mm_zalloc(ddr_pool,
19      sizeof(sx_mmdma_chan_config));
20     (*p_ch1_conf).chan_num = 5;
21     (*p_ch1_conf).priority = 2;
22     (*p_ch1_conf).src_stride = 0;
23     (*p_ch1_conf).src_skip = 0;
24     (*p_ch1_conf).dst_stride = 0;
25     (*p_ch1_conf).dst_skip = 0;
26 init_ch1_error = sx_mmdma_chan_init(p_ch1_conf, &p_channel_1);
27 //count cycles, copy data
28 count1 = sx_get_ccount();
29 memcpy_ch1_error = sx_mmdma_memcpy(p_channel_1,
30      p_frame_1, p_samples_1, framesize_100ms_inbytes, 1);
31 while(sx_mmdma_get_num_pending(p_channel_1) != 0);
32 count2 = sx_get_ccount();
33 cycles1 = count2 - count1;
34 //close dma channel, free memory
```

```
35  close_ch1_error = sx_mmdma_chan_close(p_channel_1);
36  sx_mm_free(ddr_pool, p_samples_1);
37  sx_mm_free(dram_pool, p_frame_1);
```

The four necessary SBIOS .h files are listed below.

sx-misc.h	includes intrinsics, stretch data types are also declared here.
sx-mm.h	manages memory allocation.
sx-mmdma.h	handles DMA.
sx-timer.h	counts cycles in order to evaluate performance.

The available stretch data types are: `sx_int8`, `sx_uint8`, `sx_int16`, `sx_uint16`, `sx_int32`, `sx_uint32`, `sx_int64`, and `sx_uint64`. The numbers specify the bit width of the data type and "u" means that the data type is unsigned. In lines 6 and 7, the memory for two memory *pools* is reserved. Arrays of `sx_int8` are located in the main memory (`SX_DDR`) and in the DATARAM (`SX_DATARAM`). The intrinsics `SX_DDR` and `SX_DATARAM` are provided by the file sx-misc.h.

After generating pointers to pools in lines 8 and 9, the memory pools have to be created in lines 10 and 11. Functions that create memory pools are defined in sx-mm.h. In lines 13 and 14 two variables of a certain length, each located in one pool, are initialized (`zalloc()` allocates memory and initializes it with zeros).

The DMA channel has now to be initialized. At first, the necessary pointers (lines 16–18) and the configuration's specifications (lines 20–25) are created. The DMA channel-number can be chosen between 0 and 11 for the programmer's use (line 20). Line 21 sets the priority of each channel between 0 and 3 (0 is the highest one). The stride-skip mechanism in lines 22–25 provides the possibility to skip bytes in a recurrent pattern at the source or at the destination (see also chapter 3).

In line 26, the channel is initialized and line 29 realizes the data transfer. While the DMA transfer is pending (line 31), a waiting period is imposed. Afterwards, line 35 closes the DMA channel. The processor cycles are counted between line 28 and line 32 and calculated in line 33. Finally, the reserved data pools have to be freed (line 37).

1.5 Development environment.

Stretch®, Inc. provides a graphical Integrated Development Environment (IDE) for code development. The IDE is a conglomeration of development tools; this chapter gives a short overview of them. Figure 1.3 shows the building process for an ISEF configuration. There are two files: the ISEF configuration file (fir8.xc) and the corresponding (fir8.c) file from where the ISEF configuration is called.

The $*$.xc file needs the inclusion of stretch.h in order to use Stretch® intrinsics. From the ISEF's configuration, the Stretch-C Compiler (`scc`) gen-

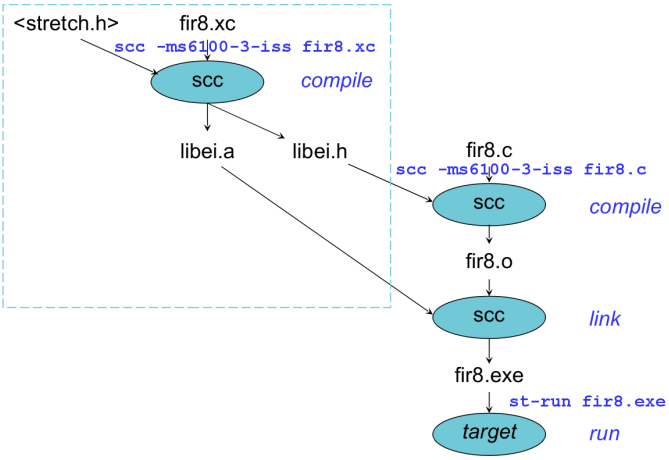


Figure 1.3: Building Process of an ISEF configuration.

erated several files. The compilation process creates an ∗.xo object file and a ∗.xr report file.

```
1 scc -c -ms6100-3-iss fir.xc
2 scc -ms6100-3-iss -stretch-link-config fir8.xo
```

In a second step, the linker creates two additional files. libei.h—different names are possible by specifying the option `-o calling scc`—has to be included in the ∗.c file, where the ISEF fEIs are called from. The file libei.a is an archive file and it is used for the linking process of the ISEF configuration and the ∗.c file. The linking process also needs the object code from the ∗.c file. It is generated by the scc compiler.

```
1 scc -c -ms6100-3-iss fir8.c
```

The arising executable file can be started with the command `st-run`. Further compilation options are possible. The most important ones are listed in the following.

First, several compilation modes are available they are called *native, simulation* and *remote*.

-ms6-native	compiles the EIs into native code for the host machine. Useful for executing and debugging EI quickly before compiling them for the ISEF.
-ms6100-3-iss	targets the S6SCP Instruction Set simulator for C/C++ and Stretch-C code.
-ms6100	targets the S6100 and S6105 SCP processor for C/C++ and Stretch-C code.

The following commands ask for changes in default modes specificities.

-stretch-effort[0-10]	sets the effort level for compiling EI into bitstreams. This affects compilation of Stretch-C files only. In general, a higher effort level takes more time, but produces better ISEF resource usage.
-stretch-freq	sets target frequency (in MHz). Default value is 300.
-stretch-issue-rate	sets IR. Default value is 1.
-stretch-nobits	compiles EI with no bit-file generation (cf. next section): only informations required by the ISS to run the EIs are produced. Note that this switch can only be used in combination with -ms6-iss.

The following commands control optimizations for advanced compilation handling.

-OPT:alias disjoint	=	Assume that memory references through different named pointers do not alias with each other, nor with any direct memory references.
-OPT:Olimit size	=	Do not optimize functions that exceed the specified size.
-OPT:unroll times	=	Do not unroll any loop more than the specified number of times. The default is 8, and unroll=1 disables loop unrolling.

1.5.1 Report files.

The size of the ISEF is limited. There are 4096 units of Arithmetic Units (AU) and 8192 units of Multiplication Units (MU). The IRAM has 32 banks, each with 2048 bytes of memory. The resource usage report of a configuration is located in the ∗.xr file. The final version of the report can be found at the end of this file.

```
1  /*****************************************************\
2  *          Final resource usage report                *
3  *-----------------------------------------------------*
4  * Configuration cross16:
5  * Total AUs                  =  3512 out of 4096
6  * Total MUs                  =  5120 out of 8192
7  * Total SHIFTs               =     0
8  * Total IRAMs                =     0 out of   32
9  * Total PRIENC bits          =     0 out of  256
10 * Target Issue Rate          =     1
11 * Target chip frequency      = 300.0 Mhz
12 * Target ISEF frequency      = 300.0 Mhz
13 * Achieved ISEF frequency    = 176.1 Mhz
```

```
14 * Maximum output write cycle =    11
15 * Warning: ISEF cannot run at the required frequency.
16 * Compile time               =  3263 seconds
17 \*************************************************/
```

Besides the resources, the IR, the target chip and ISEF frequencies, and the execution cycles of the configuration are listed. If the configured ISEF frequency could not meet the required frequency, a warning occurs and the achieved frequency is given instead. The informations are available only if a "bit-file" generation was requested. This term designs the computation by the compiler of the ISEF setup as it would really be on the board. Without this bit-file, the values of the report file are rough approximations and there is absolutely no certification that the designed configuration will actually fit the ISEF.

1.5.2 Pipeline View.

Information about possible stalls produced by an ISEF configuration could be found with the pipeline view (cf. figure B.5). The lines of code in question are executed step by step by the debugger after the pipeline view of these lines is generated. If stalls occur, redesigning parts of the code or of the whole underlying model are a solution. Reducing stalls improves the execution speed of an ISEF configuration.

1.5.3 Profiling.

Execution speed is the main information about configurations's performances. The process that measures the execution speed is called profiling (cf. figure B.6). Performing profiling lists particular functions of the code with the number of taken cycles for each of them. Functions with a great amount of cycles are preferred candidates to be implemented on the ISEF.

Chapter 2

Introduction to the application.

This chapter is intended to give the reader an overview of the project. It first describes the application goals and the microphone arrays. Then informations on source localization and beamforming are given. Finally, important notions are explained in order to make the project more understandable.

2.1 Application purpose.

The system lies in a conference room where several speakers talk either alone or together. Ideally, it localizes the dominant speaker (e.g., the one who has the floor) and returns an audio output in which all the signals coming from other sources are vanished.

In order to do so, the system disposes of a microphone array. The first step of the process is called *source localization* and is intended to derive the position of the speaker. Then, this information is given to the *beamforming* operation that derive the desired audio data from it. Figure 2.1 summarizes the process.

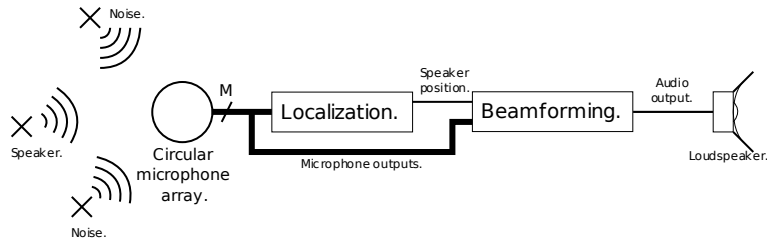


Figure 2.1: Description of the application's process.

More precisely, the aims of the system is to isolate and amplify the dominant speaker in the case of multispeakers. For a single speaker, it

should enhance the speech by removing ambient noises. Note that one of the main advantage of this solution is that no personal microphone is needed: all the classical means of amplification no longer bother the orators.

2.2 Microphone arrays theory.

A microphone array is defined as an arrangement of multiple spatially separated microphones. Several configurations exist; they are divided into three groups: linear, planar and volumetric arrays. Each of these groups has its own limitations regarding the spatial range that it covers (i.e., the places that array processing can reach). For example, a linear array covers a range of 180°azimuth while a planar array covers a range of 360°azimuth.

The array that was used is a planar one. As figure 2.2 shows, 24 equally-spaced microphones compose its circular shape of radius $r = 27.5cm$. The main advantages of this array are that it covers a range of 360°azimuth and that it is less affected by spatial aliasing than linear arrays (cf. section 2.5).

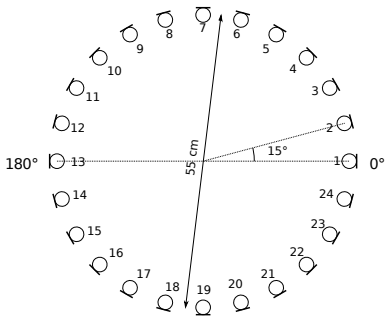


Figure 2.2: Scheme of the microphone array.

2.3 Source localization.

Microphone arrays can be used to estimate the position of a sound source relatively to the array's position, especially when a *dominant source* and several *interfering sources* overlap. The cross correlation operation is the heart of this source localization algorithm.

In order to understand the basic concepts of source localization and why the cross correlation operation is important for it, let us assume a simple situation. Two microphones receive the signal coming from a single source in far field of the array (cf. 2.5); no other source is present. Figure 2.3 sketches this case.

Depending on the speaker's position, the Time Of Arrival (TOA) of his speech to each microphone is likely to be different because his relative distance to each microphone is not the same.

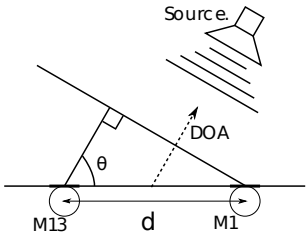


Figure 2.3: Simple case of source localization.

From the TOA difference between microphones m_1 and m_{13} ($\tau_{m_1,m_{13}}$), one can derive the angular Direction Of Arrival (DOA) (θ) of the speech according to (2.1). If m_1 is fixed as the reference: the wavefront arrives at m_{13} with a relative time delay. Therefore $\tau_{m_1,m_{13}}$ has a negative value and $\tau_{1,13} = -\frac{d}{c}\, cos(\theta)$, hence (2.1). The notations of figure 2.3 are used, c is the sound of speed in the environment of the array ($c \approx 343 m.s^{-1}$).

$$\theta = \arccos\left(-\tau_{m_1,m_{13}} \frac{c}{d}\right) \tag{2.1}$$

It is well known that the cross correlation indicates the likeness of two signals. It reaches its maximum when the signals match the most. Left-hand side graph of figure 2.3 shows the cross correlation between the outputs of microphones m_1 and m_{13} (in this order) when only one speaker is present. Right-hand side graph zooms in the range ± 80.

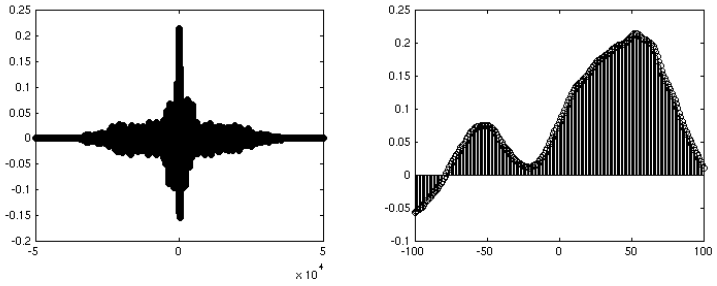


Figure 2.4: Cross correlation between m_1 and m_{13}. Analysis time $\approx$ 1s, sampling frequency = 48kHz.

The position of the maximum $n_{\max}$ and $\tau_{m_1,m_{13}}$ are linked by (2.2).

$$\tau_{m_1,m_{13}} = \frac{n_{\max}}{f_s} \tag{2.2}$$

f_s is the sampling frequency. Combining 2.1 and 2.2, the DOA can easily be derivable from $n_{\max}$. In figure 2.3, $n_{\max} \approx 50$, which means a DOA of $\theta \approx 130°$.

Because $\tau_{m_1,m_{13}}$ can not be greater than $\frac{d}{c}$, the values among which looking for a maximum is relevant are in $[-\frac{d\ f_s}{c}; \frac{d\ f_s}{c}]$. Depending on the sign of $n_{\max}$, one can say if the speaker is closer to m_1 or m_{13} (i.e, if $\theta < 90°$or $\theta > 90°$). Indeed, if $\theta < 90°$, as figure 2.3 shows, the signal has a time advance at M1. Therefore, the cross correlation between the outputs of M1 and M13 (in this order) has a maximum that is retarded ($n_{\max} > 0$).

With a linear microphone array, θ can be derived if ti belongs to $[0; \pi]$. It is nevertheless impossible to know if θ or $-\theta$ is actually detected because these source's position would return the same cross correlation. In the case of a circular array, it is indispensable to widen the range to $[0; 2\pi]$. Therefore, one computes the cross correlation between another pair of microphones who is perpendicularly-oriented to the first one.

2.4 Beamforming.

Microphone arrays can be also used to balance signals in a sound field by *steering* at their incoming direction. This processing technique is referred as *beamforming*. It is well known that a single microphone has a certain *directivity pattern*. This directivity enhances the signal emitted by a source and attenuates signals arriving from other directions in the same time, by aiming the *beam* at the desired source.

Microphone arrays have a significant advantage compared to a single microphone because the output signals of the microphones can be combined to form a beam in any desired direction without moving the array. The theory and the mechanisms of beamforming are explained more into details in section 5.1 as it is one of the algorithms that were implemented on the board.

2.5 Important notions.

2.5.1 Spatial aliasing.

Similarly to the *Nyquist criterion* in temporal sampling—which has to be fulfilled in order to avoid temporal *aliasing*—there are restrictions for spatial sampling. Let us consider the studied case of section 2.3 with a planar-wavefront signal of maximal frequency $f_{\max}$. In order to avoid spatial aliasing, the phase difference between the two microphones has to stay in the range $\pm\pi$. Otherwise, it is no longer possible to know if the calculated $\tau_{m_1,m_{13}}$ was induced by the actual phase shift or from a modulo π version of it. The criterion $2\pi f_{\max} \times \frac{d}{c}\ cos(\theta) \leq \pi$ summarizes this constraint.

From this criterion, (2.3) gives the maximal distance between microphones $d_{\max}$ that is will not lead to aliasing for a known $f_{\max}$. This equation assumes the case $\theta = 0°$, which is the worst one.

$$d_{\max} \leq \frac{c}{2\ f_{\max}} \tag{2.3}$$

From (2.3), a maximal frequency of 1000Hz would lead to $d_{\max} \leq 0.17cm$. One can easily see that even for the human speech range of fundamental frequencies (about 100-1500Hz), it is hard to design an array with M microphones so that the distance between each pair is smaller than this $d_{\max}$. Therefore, the microphone array that is used for this application is very likely to be constrainted to spatial aliasing.

Let us assume first a linear array impinged by a signal which has a maximal frequency that is higher than the *aliasing frequency*. Each pair of microphone is affected by aliasing because distance between them is always greater or equals d. In the case of a circular array, the DOAs are different for each microphone. The TOA differences between each pair are not the same, which means that for a frequency higher than the critical frequency, some pairs are affected by spatial aliasing while other pairs are not.

Hence, it is more efficient to face spatial aliasing with a circular array than with a linear one.

2.5.2 Near and far field assumptions.

If the distance between a source and the microphone array is significantly greater than the dimensions of the array, the source is said to be in *far field* of the array. In this case, the curvature of the arriving wavefront is negligible with respect to the aperture size. The wavefront appears to be planar when arriving at the microphones.

Inversely, the source is said in *near field* of the microphone array when it is so close to it that the curvature has to be considered. Therefore, the signal arrives at each microphone with a different DOA as shown on figure 2.5.

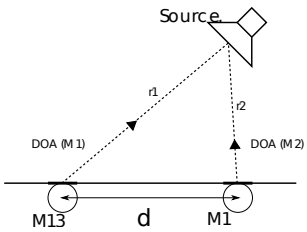


Figure 2.5: Simple case of near field configuration.

The far field assumption is the easiest to handle because there is only one DOA to derive for all the microphones (cf. figure 2.3). Near field assumption leads to harder calculations but it enables also to estimate the distance of the speaker. This thesis only reports implementations that use the far field assumption because of simplicity and lower computational cost.

Chapter 3

Managing audio inputs.

3.1 Simulating audio inputs.

Audio inputs were simulated by audio RIFF-WAVE files during all the implementations testing. Their samples were coded with a 8bits, 16bits, or 24bits resolution. Their possible sampling frequencies are 8kHz, 16kHz, 32kHz, or 48kHz.

From byte 0x00, a RIFF-WAVE file begins with a header. This is intended to describe the file contents on a size of 44bytes; it is composed of three chunks. The elements of each one of them are listed below in the order they appear in the file.

- RIFF chunk (declaration of the RIFF-WAVE format).

Name	Bytes	Description
FileTypeChunkID	4	"RIFF" (0x52,0x49,0x46,0x46).
FileSize	4	size of the file in bytes (minus 8B).
FileFormatID	4	"WAVE" (0x57,0x41,0x56,0x45).

- Audio format chunk.

FormatChunkID	4	"fmt" (0x66,0x6D, 0x74,0x20).
BlocSize	4	chunk size in bytes minus 8 (0x10).
AudioFormat	2	storage format (1 for PCM, ...).
Channels	2	nb. of channels (0 for Mono, 1 for Stereo).
Frequency	4	sampling frequency (Hz).
BytePerSec	4	nb. of bytes per second.
BytePerSample	2	nb. of bytes per sample (among all channels).
BitsPerSample	2	nb. of bits per channel's sample (8, 16, or 24).

- Data chunk.

DataBlocID	4	"data" (0x64,0x61,0x74,0x61).
DataSize	4	nb. of bytes of data.

Then, the data is stored in *little endian* sample after sample (i.e., the k^{th} sample of channel n is written before the $k+1^{th}$ of channel n-1).

In order to handle RIFF-WAVE files and to retrieve the data that are in the header, a structure was created.

```
1  struct wavefile
2  {
3       // RIFF - CHUNK
4       char riff_name[4];
5       long riff_length;
6       char riff_type[4];
7       // FMT - CHUNK
8       char fmt_name[4];
9       long fmt_length;
10      short formattyp;
11      short canalnb;
12      long samplerate;
13      long b_per_sec;
14      short b_per_sample;
15      short Bits_per_sample;
16      // DATA - CHUNK
17      char data_name[4];
18      long data_length;
19 }
```

These informations can further be useful to extract a desired number of sample. The following extract from the source code gives an example of how this structure can be used. It stores a frame of 100ms in the data pool ddr_pool at the address given by p_samples_1.

```
1  struct wavefile wf1;
2  FILE *p_file1;
3  void *p_samples_1; //pointer to the frame
4
5  p_file1 = fopen("file.wav", "rb");
6  fread((void *) &wf1, sizeof(wf1), 1, pwfile1); //reads the header
7
8  //number of bytes in the frame (corresponds to 100ms)
9  frame_length_in_bytes = (int)(0.1 * wf1.samplerate * wf1.
       bytes_per_sample);
10
11 //allocates memory
12 p_samples_1 = sx_mm_zalloc(ddr_pool, frame_length_in_bytes);
13 fread(p_samples_1, 1, vefile1.data_length, pwfile1); //store the frame
```

It is a bit different when the samples are coded with 8 bits in the RIFF-WAVE file. In this case, retrieved samples are unsigned integers with values in the range 0–255. The process retrieves these exact values even though they might represent negative or positive values from a physical point of view. In order to find out the exact meaning of the signal, one can subtract 128 to all the values of the frame. Because 128 corresponds to 0 in unsigned 8bits coding, one obtains signed integers between -128 and 127.

3.2 The frame mechanism.

In order to deal with the real purpose of the application, the cross correlation has to be calculated as soon as a certain amount of data (referred as *frame length*) is available at the microphones outputs.

It is useless to compute a cross correlation when the frame length is too small because then the maximum (cf. section 2.3) might not be detected. But the frame shall not be too long either because the computation has to run in real time (e.g., it can not last longer than the frame duration).

The specifications also impose that each new frame starts a certain time after the calculations for the previous one began. This time is called *frame shift*. The frame length and frame shift are given as time values: therefore the constraints that the frame mechanism sets also depends on the sample frequency.

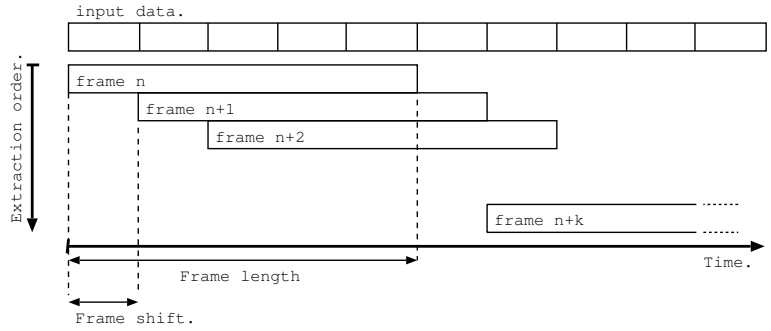


Figure 3.1: The frame mechanism.

Figure 3.1 shows the principle of the frame mechanism. The same is used for the two input data streams. Then, a cross correlation or any other computation of size N equaling the frame length can be done. In the case of the cross correlation, the output frames give directly the searched information so there is no need to do any further operation on them.

3.3 DMA transfers and resolution.

When using DMA, the resolution of audio data in the file can be a problem because IRAM can not handle 8bits and 24bits datatypes (cf. section 1.4): 16bits and 32bits tables are respectively used instead. Indeed, DMA transfers always consider amounts of bytes (i.e., no information about datatypes can be filled in). Thus, it is impossible to cast datatypes in a normal way.

In order to cast the 8bits files's data into IRAM's 16bits tables when using the DMA, the skip and stride mechanism is used (cf. section 1.4). In IRAM, a byte is skipped between each data-byte coming from the main memory. This is achieved with the following setup of the DMA channel.

```
1      (*p_ch1_conf).src_stride = 1;
2      (*p_ch1_conf).src_skip = 0;
3      (*p_ch1_conf).dst_stride = 1;
4      (*p_ch1_conf).dst_skip = 1;
```

`src_stride` and `dst_stride` have to be equal: they represent the number of data-bytes that DMA actually transfer. Line 2 means that all the bytes in the main memory will be transfered. Line 4 means that one byte will be skipped in the destination memory place. Hence each block of 2bytes in the IRAM can be considered as a 16bits-casted value.

The configuration would be slightly different regarding the second problem (24bits). Here, data has to be casted into 32bits values. Therefore, a byte is skipped between each group of three bytes coming from the source.

```
1      (*p_ch1_conf).src_stride = 3;
2      (*p_ch1_conf).src_skip = 0;
3      (*p_ch1_conf).dst_stride = 3;
4      (*p_ch1_conf).dst_skip = 1;
```

`dst_skip` still makes skip one byte in the destination memory place but this time, 3bytes are stridden.

Chapter 4

The cross correlation computation.

4.1 Theory of cross correlation.

4.1.1 Motivations.

One usually uses the cross correlation operation to detect similarities between two signals and give the time at which they match the most. As seen previously in section 2.3, this operation is really helpful in source localization systems. Knowing the relative position of two microphones and computing the cross correlation between their outputs, it is straightforward to derive the TOA difference between them. Then one can calculate exactly the detected speaker's direction of emission; this is the result of the localization system. The beamforming process will also need this information further on.

For these reasons, a survey of the cross correlation algorithm as well as its optimization was the starting point for the project.

4.1.2 Mathematical description of the cross correlation.

For two discrete signals x and y defined $\forall n \in \mathbb{Z}$, the cross correlation [4] is defined as:

$$(x \star y)[n] \triangleq \sum_{m=-\infty}^{+\infty} x^*[m]\, y[n+m] \tag{4.1}$$

* is the complex conjugate operator.

In the case of real, finite, and discrete signals of size N ($0 \leq n \leq N-1$), the right part of (4.1) becomes a sum between $m = 0$ and $m = N-1$. In this case, the cross correlation does not exist for $n < -(N-1)$ and $n > N-1$; its size is $2 \times N - 1$. From now, n will always refer to the index of the cross correlation elements.

In order to have a better view of what happens during this computation, figure 4.1 depicts it from a multiplication point of view in the case N = 32.

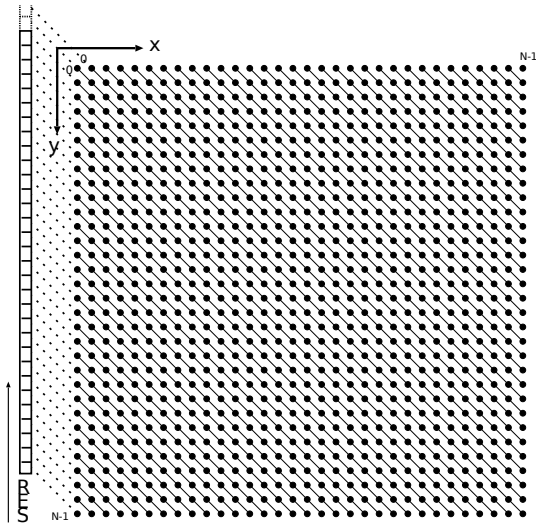


Figure 4.1: The cross correlation algorithm.

Each dot of the grid is a product between one element of x and one of y. The sum of the dots over a diagonal represents one element of $(x \star y)$ as it is given by (4.1). For example, the longest diagonal in figure 4.1 refers to (4.2) that is the result of the cross correlation for $n = 0$.

$$(x \star y)[0] \; = \sum_{m=0}^{m=N-1} x[m]y[m] \tag{4.2}$$

In our case, the sum (4.1) was subdivided in order to deal with the specific hardware architecture that is going to be explained in section 4.3. In the following, the terms "square decomposition", "diagonal decomposition", and "linewise decomposition" refer to the algorithms that were founded out to do this splitting. They are going to be described in the two next sections.

Square decomposition.

The subdivision in small blocks (squares) is the principle of the algorithm. Figure 4.2 ($N = 32$) describes this algorithm with $b = 4$ as block size. b samples per frame are needed in order to compute all the multiplications that are included in a square.

Furthermore, one can see that a square also represents the cross correlation between b samples from x and b samples from y (let us call $c_{i,j}$ this function).

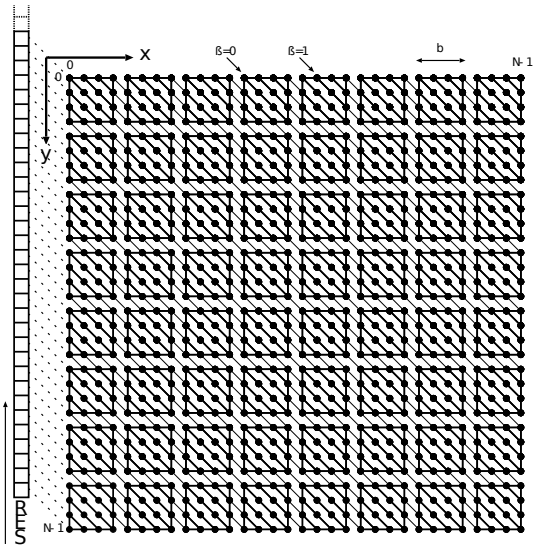


Figure 4.2: Principle of square decomposition.

$$c_{i,j}[n] = \sum_{m=0}^{m=3} x_i[m] \times y_i[n+m] = (x_i \star y_j)[n] \tag{4.3}$$

$$x_i[m] = \begin{cases} x[m] & \text{if } i \leq m < i+b \\ 0 & \text{otherwise} \end{cases}$$

$$y_j[m] = \begin{cases} y[m] & \text{if } j \leq m < j+b \\ 0 & \text{otherwise} \end{cases}$$

Let us introduce $\alpha = \left\lfloor \frac{|n|}{b} \right\rfloor$ and $\beta = |n| \bmod b$, respectively the quotient and the remainder of the division of $|n|$ by b. α can be interpreted as a block index and β as an offset in this block.

A diagonal (i.e., an element of $x \star y$) goes through a certain amount of squares depending on n. When $\beta = 0$, n is a multiple of b and the diagonal is the assembly of the main diagonals of the squares it crosses. Only one square per square-line is crossed. The expression of the cross correlation for the square decomposition and $\beta = 0$ can be written as:

$$\begin{cases} (x \star y)[n] = \sum_{i=\alpha}^{\frac{N}{b}-1} c_{i,i-\alpha}[\beta] & \forall n < 0 \\ (x \star y)[n] = \sum_{j=\alpha}^{\frac{N}{b}-1} c_{j-\alpha,j}[\beta] & \forall n \geq 0 \end{cases} \tag{4.4}$$

Otherwise, for $\beta \neq 0$, the diagonal is a combination of small diagonals with two possible lengths that are α and $b - \alpha$. Two square per square-line

are crossed. It adds terms to the sums in (4.4) in order to take into account all the extra squares that are crossed. The expression of the cross correlation for the square decomposition and $\beta \neq 0$ is:

$$\begin{cases} (x \star y)[n] &= \sum_{i=\alpha}^{\frac{N}{b}-1} c_{i,i-\alpha}[\beta] + c_{i+1,i-\alpha}[-(b-\beta)] \quad \forall n < 0 \\ (x \star y)[n] &= \sum_{j=\alpha}^{\frac{N}{b}-1} c_{j-\alpha,j}[-\beta] + c_{j-\alpha,j+1}[b-\beta] \quad \forall n \geq 0 \end{cases} \tag{4.5}$$

In order to show the diagonals and the squares they cross for different values of β, two diagonals are pointed on figure 4.2. They represent the cases $\beta = 0$ and $\beta = 1$ ($\beta \neq 0$).

In (4.4) and (4.5), one can notice that i for $n < 0$ and j for $n \geq 0$ have the same role and inversely. This comes from the property (4.6) of the cross correlation.

$$(x \star y)[n] = (y \star x)^*[-n] \tag{4.6}$$

Figure 4.3 shows the principle of the merging algorithm in the case $N = 32$ and $b = 4$. Each "i *cc* j" rectangle represents the result of the cross correlation between the i^{th} b-samples group of the first frame and the j^{th} b-samples group of the second frame. In other words, it depicts the non-zero results of $c_{i,j}$.

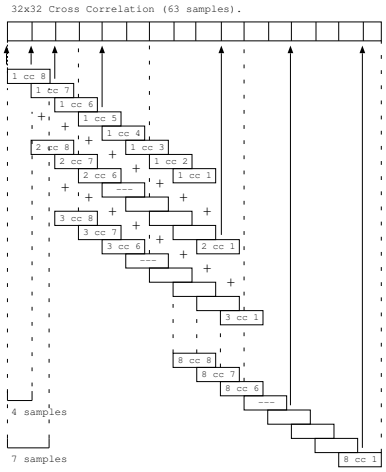


Figure 4.3: Principle of the merging algorithm for square decomposition.

Diagonal decomposition.

The diagonal decomposition is similar to the previous one. In this case the squares are angled so that the merging of a block needs less access to the final result. Figure 4.4 depicts this algorithm for a frame of $N = 32$ and $b = 4$.

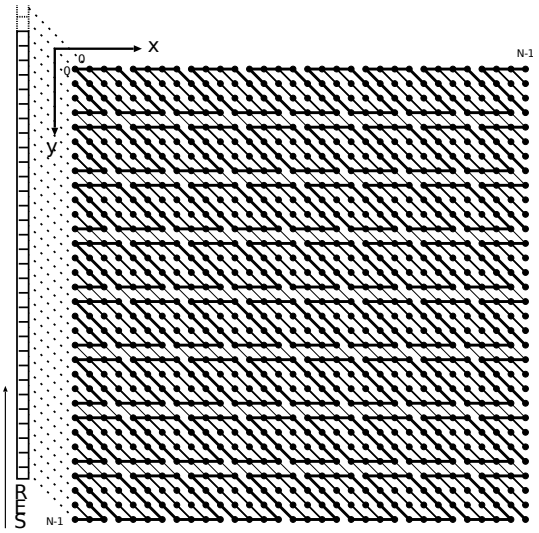


Figure 4.4: Principle of diagonal decomposition.

Merging the blocks into the final result is simpler here because each diagonal of a block belongs to only one diagonal of the final grid. Suppose a block-size of $b \times b$ where N is a multiple of b; (4.7) gives the diagonal decomposition of the cross correlation.

$$(x \star y)[n] \;=\; \sum_{m=0}^{\frac{N}{b}-1} \sum_{k=0}^{b-1} x[m \times b + k]\; y[m \times b + k + n]. \tag{4.7}$$

One can notice that the whole frame can not be filled completely with blocks that have a rhomboid shape. The side blocks have a triangular shape and they compute less multiplications than the others. It is interesting to replace these triangular blocks in order to always compute the same number of products per block. To do so, multiplications that are intended to the opposite side of the frame can be computed in the same block as suggested in figure 4.5. This side effect is going to be discussed in section 4.3.

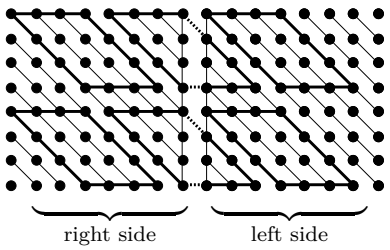


Figure 4.5: Diagonal decomposition, border case.

Linewise decomposition.

The linewise decomposition differs from the two previous approaches. Although the block's shape is linear here, it still computes the same number of multiplications. A segment of length b represents the multiplications between one element of x and b elements of y. Figure 4.6 depicts this algorithm for a frame of $N = 32$ and $b = 16$.

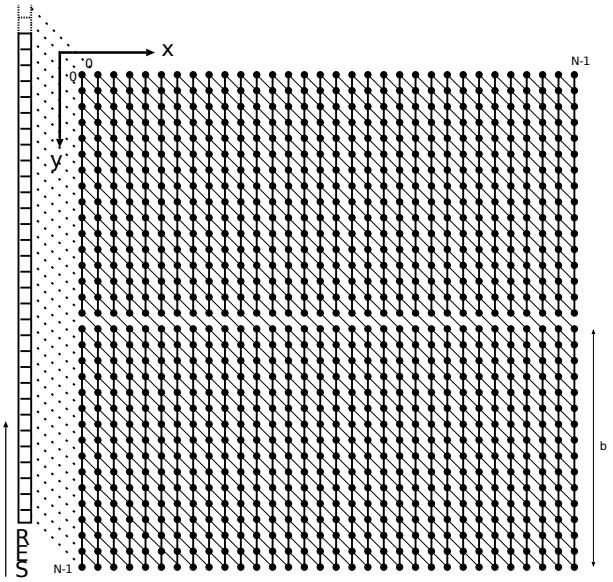


Figure 4.6: Principle of linewise decomposition.

The merging of the blocks is rather simpler than for the other decompositions. Indeed, one can notice that the contribution of a block to the final result takes place on b diagonals (i.e., b final elements of the cross correlation frame) but only two of them differ from the previous block. Therefore one needs to add the blocks to the final cross correlation frame with a shift of one sample between them as figure 4.7 shows for the case $N = 32$ and $b = 16$.

On this figure, "block 1" refers to the bottom-left block of figure 4.6, "block 2" to the top-left one, "block 3" to the second in the bottom-left, and so on... "block 32" would refer the top-right one.

One can also notice that a $b \times b$ cross correlation is easily derivable from b linear blocks. Similarly to the square decomposition case, let us call $l_{i,j}$ the function that gives the expression of such blocks.

$$l_{i,j}[m] = \begin{cases} x[i]y[j+m] & \text{if } 0 \leq m < b \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

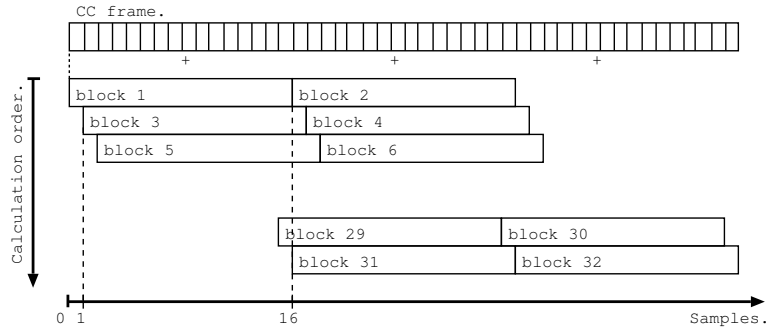


Figure 4.7: Merging algorithm for the linewise decomposition.

(4.9) gives the expression of a the cross of a $b \times b$ block as a function of (4.8). The $c_{i,j}$ function is used as it was defined in (4.3).

$$c_{i,j}[n] = \sum_{k=0}^{k=b-1} l_{i+k,j}[k-n] \tag{4.9}$$

Finally, replacing the $c_{i,j}$ from (4.9) in (4.4) and (4.5) gives the final expression of the whole cross correlation frame of size $2N-1$ as it was done for the square decomposition case.

In this case, all the multiplications are done in the same way they are in (4.1). Only their order of computation changes; section 4.3 details it. Note also that for linewise decomposition, b is generally greater than the b of other decomposition algorithms.

4.2 Cross correlation implementation in Matlab®.

Although this thesis mainly describes the ways how to port computation algorithms from Matlab® to the studied Stretch® board, this is not what is developed in this section. In Matlab®, indeed, the cross correlation code is straightforward.

In this application the Matlab® implementation is used for verifications purpose and therefore no optimization attempt was given on it. The idea is to compare the outputs from the Matlab® process and from the Stretch® board process.

To do so, the Matlab® algorithm has to run the same way as the one on the board does and it need to be accurate enough to behave like a reference (i.e., the frame length/frame shift mechanism has to be implemented as well).

4.2.1 Comparison of two audio files in Matlab®.

The loading of audio. WAVE files is simplified in Matlab® compared to the usual C method.

The function

```
1   [Y,FS,NBITS]=WAVREAD(FILE,N,FMT);
```

enables to get the data samples in a "native" or "double" format (FMT). It also returns the sampling frequency (FS) and the number of bits per sample (NBITS). The native formats are integers of the same resolution that the one of the file's samples. In the double case, the data is a double which represents the value normalized between -1 and 1.

Once the frames to compare are in two vectors, one can for example derive the mean difference between the samples in order to have an idea of the solution accuracy.

4.2.2 The cross correlation operation in Matlab®.

Having the input frames in vectors A and B, the cross correlation vector C can be computed with the following function.

```
1   C = XCORR(A,B);
```

The function uses formula (4.10) to compute the cross correlation. Therefore, the order of the results is inverted compared with the one given by (4.1).

$$(x \star y)[n] = \sum_{m=-\infty}^{+\infty} x^*[m]\ y[n-m] \tag{4.10}$$

The function in line 1 below can be used to give the right order to the frame so that it is comparable with the board's output (note that $x \star y \neq y \star x$ in the general case). The lines 2 and 3 would return the same result.

```
1   FLIPLR(X);
2   XCORR(A,B)';
3   FLIPLR(XCORR(B,A)');
```

In order to be sure of the comparisons that were made, the cross correlation function was further rewritten in the exact same way as it is done by the simplest implementation on the Stretch® board. The Matlab® inner function was not used anymore since then.

4.3 Cross correlation implementation on the board.

This section describes objectively the different solutions that were set up to make the Cross Correlation (CC) run on the board. Appendix B provides schemes that explain data flow and memory management of the solutions. Appendix A reports their source code notice. Section 6.1 gives comparisons and puts figures on the results of these implementations.

4.3.1 Simple implementation without the ISEF.

This implementation gives a reference for the following versions of the algorithm. It uses the Xtensa as a normal C code would do it: with no use of the extra capabilities given by the board. It is the slowest version but it is the easiest to verify. Therefore the speed improvement quantifications and the results can always be compared to it.

The data to work on is placed in the main memory. No use of DMA or DATARAM is requested explicitly. The actual signal processing part takes place in the ALU. It computes the CC as (4.1) and figure 4.1 describe it.

Several pointers handle the memory places where the data circulate.

- p_samples (1 & 2) - point on the data of audio files in main memory.
- p_corr - points on result the vector in main memory.

4.3.2 Using the ISEF and WRs.

The best way to improve the performances of the basic solution described in section 4.3.1 is to move some computations to the ISEF. One way of feeding the ISEF with data is to do it over the WRs.

As figure 1.1 shows, there is a possible close connection between WRs and DATARAM. Furthermore, the DATARAM size of 64KB enables to store the whole frames. Hence the first step is to make the data transfer from the main memory of the board to the DATARAM. This operation is achievable either with or without DMA. Both approaches were used and compared (cf. section 6.1) but the main differences between the implementations appear in the way they compute the data once it is in the DATARAM.

Figure 4.8 shows the flow of data for all the solutions that use the ISEF with the WRs. Depending on the implementation version, the dashed arrows may represent DMA transfers or classical transfers.

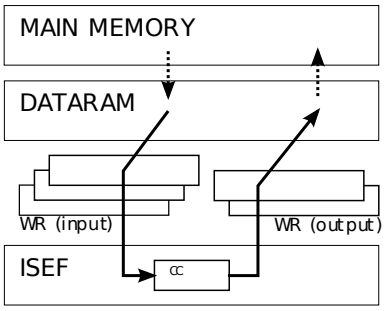


Figure 4.8: Data flow in the case of WRs use.

This version needs some more pointers than the previous one in order to handle the added features.

- p_frame (1 & 2) - point on audio data frames in DATARAM.
- p_channel (1, 2, & 3) - point on DMA channels (when needed). Usually, one channel was dedicated to each input frame and another one to the output stream.

Once the data is in the DATARAM, there are several ways to proceed. As announced in section 4.1, some decompositions of the CC were found so that ISEF can compute it. For each EI the correct set of samples has to be provided to the ISEF via the WRs. The choice of this set makes the main difference between the algorithms. Then, the way to merge the EI's outputs into the $N \times N$ differs also.

Using ISEF with the square decomposition.

Figure B.7 shows memory management for this implementation.

In the square decomposition case, the ISEF runs a $b \times b$ CC in one EI cross_isef(). It means that each EI receives $b \times b$ inputs via the WRs and outputs $2b - 1$ samples also via the WRs. The operation is repeated $(N/b)^2$ times so that each block is computed. The merging of a block is done just after its calculation, after it is issued from the WRs.

The handling of the current positions in the input frames (i.e., the indexes i and j of a "i cc j" rectangle in figure 4.3) is done outside the ISEF by pointers. The same principle is used to point the right place in the result frame (i.e., where the current block has to be added).

Using ISEF with the diagonal decomposition.

Figure B.8 shows the data flow in memory of this implementation.

For the diagonal decomposition, the computation of a block needs $2b-1$ samples from the first frame x and b from the second frame y. For the blocks that are not border cases, cc_mac() is called. It needs the following set of samples from x:

- $b-1$ "old" samples that the previous block also needed[1]. They can be found in olda[ ], in the ERs.
- b "new" samples that are in WRA. $b-1$ of them are immediately stored in olda[ ] for the next EI.

[1] As the blocks overlap along the x dimension, several input samples can be used from one EI to another.

The b samples from y are in WRB and they stay the same for all the blocks of a block-line. Each time that one of these samples is loaded into the ISEF, a horizontal set of multiplications is done and they are added to the right diagonal. sumhor[ ] is a table of size b in the ERs that stores each diagonal of a block. When the computation of a block is finished, the value of the current position indicator run is incremented of b.

Once run equals the input frame length, it means that a border is reached. In this case, $b-1$ new samples of x come from the beginning of the frame and 1 is the last one of the frame. The elements of sumhor[ ] are calculated only with the $b-1$ new samples and they represent the diagonals of the left-side triangles of figure 4.5. sumver[ ] stores the computations involving the old samples and the last sample. They represent the right-side triangles; they can be retrieved by the EI getfromer().

run is previously reset by cc_fin_mac() before changing of block and cc_init_mac() is called to reset the table sumhor[ ].

Using ISEF with the linewise decomposition.

Figure B.9 gives an overview of memory management for this implementation.

Concerning data flow in memory, the approach here is different. The first step is to transmit a whole input block from the first frame into the ERs of the ISEF over the WRs. Then, samples from the second frame are transfered one by one into the ISEF. Each EI receives one sample; after b iterations, a $b \times b$ CC is performed. Note that in this case, b is greater than for the previous decompositions (cf. section 6.1).

In the same time, after each EI, one sample of the $b \times b$ CC is ready. Therefore its storage is handled just after the EI. That is $b-1$ stored output samples after $b-1$ iterations. b samples remain uncomputed in order to reach the $2b-1$ normal output frame size of a $b \times b$ CC. These samples are stored back into the result vector over the WRs after the b^{th} iteration. Storing back is done alternately over WRA and WRB, due to performance reasons[2].

4.3.3 Using the IRAM.

In these solutions, the WRs mechanisms are removed completely. They make use of the IRAM, a memory space which was created to be close to the ISEF's EIs and also accessible from outside the ISEF. In fact, each occurrence of an EI can process data from and to the IRAM, thus it is a good alternative to the previous processing via the WRs. Furthermore, this memory space reaches 64KB, what enables the storage of a large amount of samples (e.g., entire frames).

[2]Figure B.9 refers to this alternation with dashed arrows

Figure 4.9 shows the flow of data for all the solutions that use the IRAM. The data is directly transfered from the main memory to the IRAM via DMA. The dashed arrows represent these transfers. It is important to notice that IRAM can not handle 8bits datatypes (cf. chapter 1): 16bits tables are used instead. Chapter 3 explained the skip-stride mechanism that one needs to cast 8bits audio-file's data into 16bits values when using DMA.

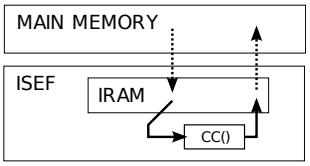


Figure 4.9: Data flow in the case of IRAM use.

The implementations are very similar regarding data flow. Therefore, only one explanation is going to be done but it is valid for both square and diagonal decompositions. As shown on figure B.10, the extension instruction cross_calc() gets data from the frames in the IRAM and issues a result in the ERs (buffer). Then, several calls of the extension instruction add_cross() do the merging operation: they add the buffer to the final result.

In order to perform the calculations and the merging between the right elements, the exact places (ranks of the tables) have to be known and update after every iteration of these extension instructions. For all these matters, several tables are needed.

- frames1, frames2 and res. As the place dedicated to frames and to the final result has to be accessible from inside and outside the ISEF, it is stored in the IRAM.
- xcorr1. The intermediate result does not need to be accessible from outside the ISEF, hence it is stored in the ERs.
- rank. This table gives the current places in the previous tables. It is also stored in the ERs as all the ranks updates are made in ISEF's extensions instructions. Its elements are:
  - rank[0] - offset from the beginning of frames1
  - rank[1] - offset from the beginning of frames2
  - rank[2] - offset from the beginning of res
  - rank[3] - offset from the beginning of xcorr1

Chapter 5

The beamforming implementation.

5.1 Theory of beamforming.

A short introduction to beamforming was given in chapter 2. This section intends to present the theory of several algorithms that were used for beamforming. It also relates the space configuration during the test files recording.

5.1.1 Precisions and measurements specifications.

Notations.

Since now and for the next sections, the following notations are going to be used. Bold font refers to vectors and capital versions of the signal's names point out their Fourier transform. The spatial origin is the center of the array.

M	number of microphones of the array ($M = 24$).
r	radius of the array ($0.275m$).
c	speed of sound ($343m.s^{-1}$).
$\mathbf{x}$	cartesian coordinates of the aimed source.
m_i	name of microphone i.
$\mathbf{m_i}$	cartesian coordinates of microphone i.
$\{\rho_i, \theta_i\}$	polar coordinates of microphone i. ($\rho_i = r, \ \forall i$).
θ	source's direction of emission.
θ_s	direction of steering of the array.
τ_{m_i}	time delay from the source to the microphone i.
τ_{m_i,m_j}	relative time delay between microphones i and j.
$x_i(t)$	signal at output of microphone i.

Sources layout for test files recording.

Figure B.11 shows the sources's organization relatively to the array. The sources surround the array and they are all at a distance of $200cm$. The inner stars represent the microphones and the outer stars the sources.

5.1.2 Delay and sum beamforming.

Delay and Sum Beamforming (DSB) [1], [2], [12], [13] and [19] is a widely used algorithm because of its simplicity. Its principle is similar to source localization in the way it uses the likeness of the signals coming from the different outputs of the microphones. The aim is to shift the signals at each microphone output so that the dominant source's signal in each output is aligned (i.e., it has the same phase shift from a reference time). In order to do so, the source position has to be formerly known: this is why source localization happends earlier in the global process of figure 2.1.

Knowing the source position, the first task is to derive the TOA of its emitted signal at each microphone. In the near field of the array, (5.1) gives the value of the TOA (τ_{m_i}) at each microphone i from $\mathbf{m_i}$ and $\mathbf{x}$.

$$\tau_{m_i} = \frac{|\mathbf{x} - \mathbf{m_i}|}{c} \qquad 0 \leq i \leq M \tag{5.1}$$
$$\mathbf{x}^T := [x\ y\ z] \quad \mathbf{m_i}^T := [x_i\ y_i\ z_i]$$

Under the far field assumption, the exact position of the source is not known exactly in the space. Indeed, source localization returns only the direction of emission of the source signal θ. In this case, the closest microphone to the source (m_c) and its azimuth (θ_c) are derived from θ. From this microphone, all the relative TOA delays (τ_{m_c,m_i}) to the other microphones are computable as figure 5.1 shows. (5.2) gives the expression of these delays in our case of circular array with equally-spaced microphones.

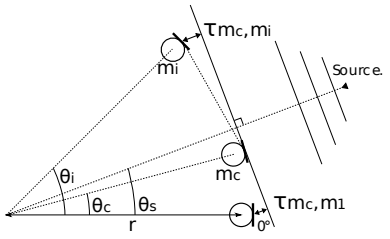


Figure 5.1: Highlighting of τ_{m_c,m_i} under the far field assumption.

$$\tau_{m_c,m_i} = 2\ \frac{r}{c}\ sin\left(\frac{\theta_i - \theta_c}{2}\right) sin\left(\frac{\theta_i - \theta_c}{2} + \theta_c - \theta_s\right) \tag{5.2}$$

Figure 5.2 plots the time delays τ_{m_c,m_i} (in s.) as a function the microphone's angles θ_i (in °). The angle of steering $\theta_s = 25°$ was used. Hence the closest microphone angle is $\theta_c = 30°$ (m_3); therefore $\tau_{m_c,m_3} = \tau_{m_3,m_3} = 0$.

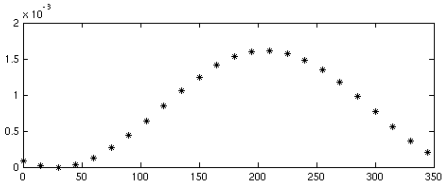


Figure 5.2: Values of τ_{m_c,m_i} under the far field assumption, with and $\theta_s = 25°$.

Once all the τ_{m_c,m_i} are known, the DSB algorithm takes place. Complex weights are applied on each microphone output according to figure 5.3. The weight at each microphone equals $w_i = e^{j\omega\tau_{m_i}}$ in the frequency domain. Indeed, if one considers that each microphone output contains a shifted version of the desired source signal $s(t)$ as (5.3) shows, applying the weights leads to (5.4a) or, in the frequency domain, (5.4b).

$$x_i(t) = \underbrace{s(t-\tau_{m_i})}_{\text{desired signal}} + \underbrace{\eta(t)}_{\text{noise and interferences}} \tag{5.3}$$

$$\begin{aligned} x_i(t)\delta(t+\tau_{m_i}) &= s(t-\tau_{m_i})\delta(t+\tau_{m_i}) + \eta(t) \\ &= s(t) + \eta_i(t) \qquad (5.4a) \\ X_i(f) \times w_i &= \left(S(f)e^{-j\omega\tau_{m_i}}\right) e^{j\omega\tau_{m_i}} + \eta(f) \\ &= S(f) + \eta_i(f) \qquad (5.4b) \end{aligned}$$

It is visible that applying the weights enables the desired signal to be "aligned" in all the microphone outputs: this is what is referred as "steering the array".

In a second time, the sum over all the shifted outputs is done so that the desired signal is added constructively while the interferences are likely to be added destructively.

Figure B.12 shows a configuration with two sources: one emits the desired signal (bold sinewave) and the other adds interferences (dashed sinewave). Figure B.13 shows how each source is processed by the DSB algorithm. Although the dashed and bold signals are mixed together by the array, they are displayed separately for a better understanding of the principle.

Figure 5.3 summarizes the DSB algorithm as a block diagram. Note that it is possible to normalize the output by the number of microphones M so that the output has the same order of magnitude than the inputs.

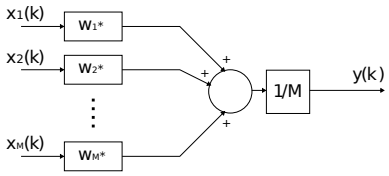


Figure 5.3: The delay and sum beamformer.

Directivity pattern of the DSB algorithm.

For a given steering direction θ_s and when using the DSB algorithm, the general expression of beamforming in time domain is given by (5.5a). In frequency domain, it becomes (5.5b).

$$y_{\text{DSB}}(t) = \sum_{i=1}^{M} x(t)\delta(t - \tau_{m_i})\delta(t + \tau_s) \tag{5.5a}$$

$$Y_{\text{DSB}}(f) = \sum_{i=1}^{M} X(f)e^{-j\omega\tau_{m_i}}e^{j\omega\tau_s} \tag{5.5b}$$

τ_{m_i} is the relative time delay that is due to the distance between each microphone and the source depending on its actual position θ: it is given by (5.1) for any possible value of θ. τ_s is the delay that the weight w_i causes at m_i: it is fixed for each microphone depending on θ_s. It is also given by (5.1) but this time, with $\theta = \theta_s$.

Hence, the impulse response of the array is modeled as (5.6). Note that this function has θ as a parameter.

$$H_{\text{DSB}}(j\omega, \theta) = \sum_{i=1}^{M} e^{j\omega(\tau_s - \tau_{m_i})} \tag{5.6}$$

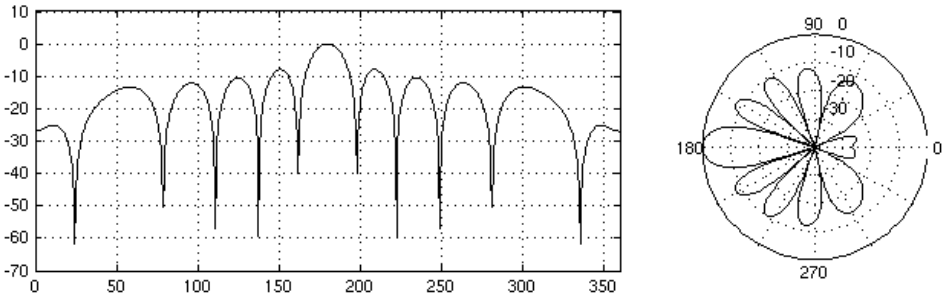


Figure 5.4: Directivity pattern of the circular array for an input frequency $f = 1500Hz$ and $\theta_s = 180°$.

Figure 5.4 plots $H_{\text{DSB}}(j\omega, \theta)|_{dB}$ for a constant input frequency and $\theta \in [0, 2\pi]$. It is called the *directivity pattern* of the array at this frequency for a given steering angle. On this figure, a polar plot of the same function is also available in order to picture clearly how DSB acts over space.

As discussed earlier in chapter 2, spatial aliasing is a problem when space filtering. In order to notice the influence of the maximal frequency of a narrowband signal on spatial aliasing for this array, figure 5.5 shows the comportment of circular array's directivity pattern over frequency.

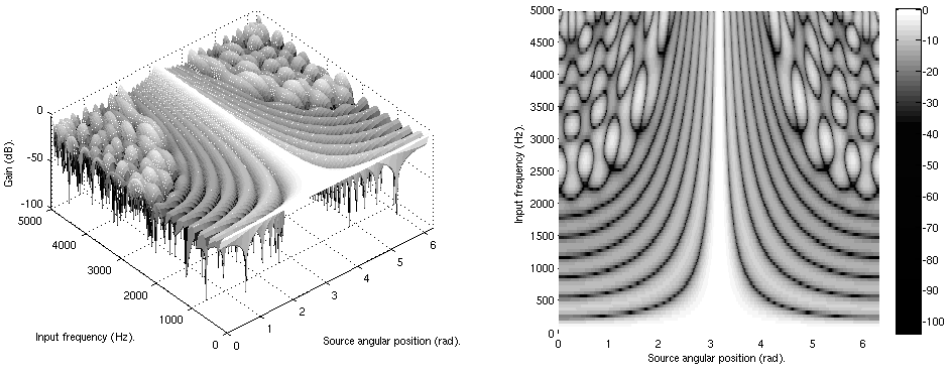


Figure 5.5: Effect of input frequency on directivity pattern, $\theta_s = 180°$.

For low frequencies, the directivity pattern is very bad because there is almost no spatial selectivity. As frequency increases, the main lobe at θ_s becomes narrower. But when the frequency is too high, spatial aliasing causes the apparition of extra side lobes that are impeding.

5.1.3 Generalized sidelobe cancelling.

In section 5.1.2, DSB was detected as problematic regarding sidelobes caused by spatial aliasing. Indeed, these lobes amplify unwanted zones for high frequencies. The Generalized Sidelobe Cancelling (GSC) algorithm [7], [8], [17], [21] can encounter this problem. Figure 5.6 shows the global block diagram of a GSC beamformer. The different blocks of this diagram are going to be explained in this section.

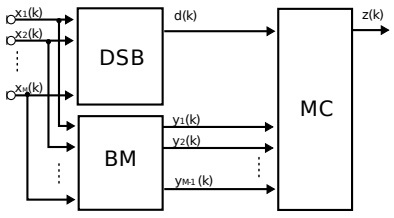


Figure 5.6: The GSC beamformer.

The algorithm is based on the DSB algorithm as it is intended to ameliorate it. Therefore, the first block (DSB) refers to a classic delay and sum beamforming as it was introduced in section 5.1.2. In the same time, the Blocking Matrix (BM) also processes the microphones outputs. The goal is to isolate the interferences (i.e., every signal that is not coming from the steering direction). Finally, the Multiple input Canceler (MC) removes the resultant noise to the (already purified) DSB-beamformer's output.

From a mathematical point of view, [1] and [17] describe the GSC beamformer as on figure 5.7. (5.7) is the equivalent expression in matrix notations.

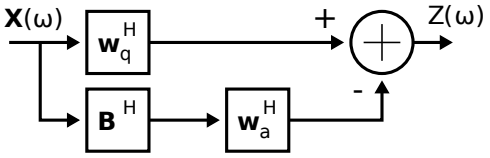


Figure 5.7: Mathematical description of the GSC beamformer.

$$Y = (\mathbf{w}_q - \mathbf{B}\mathbf{w}_a)^H \mathbf{X} \tag{5.7}$$

In (5.7), $\mathbf{w}_q$ refers to the quiescent weights w_i of section 5.1.2 and $\mathbf{w}_a$ are constrainted active weights that varies throughout the algorithm execution. $\mathbf{B}$ is the blocking matrix and $.^H$ refers to the Hermitian transpose. The instantaneous output $y(t)$ acts like a command for the weights $\mathbf{w}_a$. Therefore GSC is an adaptive algorithm.

[1] simplifies the expressions of $\mathbf{B}$ and $\mathbf{w}_q$ when the input $\mathbf{X}$ on figure 5.7 is pre-steered (i.e., the weights of section 5.1.2 have already been applied to it).

$$\mathbf{B}^T := \begin{bmatrix} 1 & -1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & -1 \end{bmatrix} \quad \mathbf{w}_q := \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

$\mathbf{B}$ is a $M \times (M-1)$ that is defined so that the elements of $\mathbf{B} * \mathbf{X}$ reads (5.8).

$$y_j(t) = x_i(t + \tau m_i) - x_{i+1}(t + \tau_{m_{i+1}}) \tag{5.8}$$

Then, each weight of $\mathbf{w}_a$ balances one of the $M-1$ results. As the input is pre-steered, the weights $\mathbf{w}_q$ become ones.

[18] and [21] replace each weight of $\mathbf{w}_a$ by an adaptive L-taps filter whose coefficients are given by a Normalized Least Mean Square (NLMS) algorithm. Let us call $\mathbf{W}_m(k)$ the coefficient vector for the m^{th} output's filter ($m = 0, 1, \cdots, M-1$) and $\mathbf{y}_m(k)$ a vector of L elements from this output.

$$\mathbf{W}_m(k) := [w_{m,0}(k), w_{m,1}(k), \cdots, w_{m,L-1}(k)]^T$$
$$\mathbf{y}_m(k) := [y_m(k), y_m(k-1), \cdots, y_m(k-L-1)]^T$$

(5.10) is the function of the NLMS algorithm that enables coefficients renewing at each step. α is the step size for coefficient adaptation ($0 < \alpha < 2$) and $\|.\|$ is the Euclidean norm. K is a threshold for the total squared-norm of the weight vectors $\mathbf{W}_m(k)$. The resulting new MC is depicted on figure 5.8.

$$\mathbf{W}'_m = \mathbf{W}_m(k) + \alpha \frac{z(k)}{\sum_{j=0}^{M-1} \|\mathbf{y}_j(k)\|^2} \mathbf{y}_m(k)$$
$$\Omega = \sum_{m=0}^{M-1} \|\mathbf{W}'_m\|^2$$
$$\mathbf{W}_m(k+1) = \begin{cases} \sqrt{\frac{K}{\Omega}} \mathbf{W}'_m & \text{for} \quad \Omega > K \\ \mathbf{W}'_m & \text{otherwise} \end{cases} \tag{5.10}$$

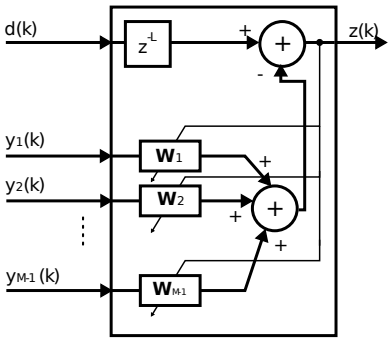


Figure 5.8: Multiple input canceler for the NLMS algorithm.

5.2 Beamforming implementation in Matlab®.

It is important to notice that Matlab®'s weak point is the computation of loops. Meanwhile, it is much more performant to use vector an matrix operations. Therefore, the implementations use these features instead of loops each time it is possible.

5.2.1 The DSB algorithm.

The first input information of the beamforming process is the θ_s (s). From this angle, the closest microphone to the source has first to be derived.

`alphas` is a vector that stores the θ_i in ascending order and `alpha` is $\frac{2\,\pi}{M}$. Line 1 gives the position in `alphas` of the angle that is immediately lower than the speaker's one. Then, by calculating the angle difference from the two closest microphones's angles to θ_s, the lines 2–5 derive the actual closest microphone.

```
1 nb_prev = floor(s/alpha)+1;
2 nb_clos = nb_prev;
3 if(abs(alphas(nb_prev)-s) > mod(abs(alphas(nb_prev+1)-s),alpha))
4       nb_clos = mod(nb_clos + 1,nb_mics);
5 end
6 thosteer = 2*r/c*sin(alphas/2) .* sin(alphas(nb_clos) - s + alphas/2);
7 thosteer = [thosteer(24-nb_clos+1:24), thosteer(1:24-nb_clos)];
8 delays = round(thosteer*fs);
```

Then, the calculations of the delays is done according to (5.2). The term $(\theta_i - \theta_c)$ is replaced by the vector `alphas` as it is inevitably one of its elements for all i. Therefore, line 6 calculates a $1 \times M$ vector that represents the τ_{m_c,m_i} values. Nevertheless, this vector stores the values in the wrong oder. Indeed τ_{m_c,m_c} is always the first element as `alphas` represents the relative angles $(\theta_i - \theta_c)$. Line 7 shifts the result vector the position of each element corresponds to its associated-microphone number. The delays are later rounded so that they can be used as a number of samples (line 8).

In order to obtain input data, the handling of audio files can be done as in 4.2. The input data frames of size n are stored in a $n \times M$ matrix (`x`). The delays are applied by inserting zeros at the beginning of each line of the matrix.

```
1 for k=1:nb_mics
2       xShift(:,k) = [zeros(delays(k),1); x(1:end-delays(k), k)];
3 end
```

The sum of `xShift` along its lines gives the result of the DSB beamformer. Note that applying the delays is made completely in the time domain.

5.2.2 The GSC algorithm.

After that the DSB algorithm is done, the GSC program computes the BM with a matrix multiplication on `x`. `bm` is defined as in section 5.1 and therefore inter is a $(M-1) \times n$ matrix that represents the BM outputs (i.e., the interferences).

```
1 bm = [diag(ones(1,nb_mics-1),0) zeros(nb_mics-1,1)] + [zeros(nb_mics
      -1,1) diag(-ones(1,nb_mics-1),0)];
2 inter = bm*x';
```

From now, the algorithm becomes adaptive and therefore it has to run step by step; there no choice but to set up a loop. At each step k, the computation of the filters as well as the sum over the filtered-lines can be done as described below. `w` is a $L \times (M-1)$ vector which columns are the $\mathbf{W}_m(k)$.

```
1 z(k) = y(k-Q) - sum(sum(inter(:,k-L+1:k)'.*w));
```

`z(k)` is the GSC beamformer's output. This value is further needed in the weights renewing algorithm. At each step, a new matrix is calculated according to (5.10).

5.2.3 Audio comparisons.

In order to compare several implementations from an audio enhancement point of view, their outputs can be normalized to the same references. An energy normalization of the signal can be done. The energy of a real, discrete, and finite signal is given by (5.11).

$$E_{x_i} = \sum_{n=0}^{N-1} |x_i(n)|^2 \tag{5.11}$$

One can easily see the normalization of the signal's samples by $\sqrt{1/E_{x_i}}$ provides a signal with an unitary energy. Applying this normalization to each implementation result enables, for example, to compare their noise levels directly.

5.3 Beamforming implementation on the board.

This section describes the different algorithms of beamforming that were tried on the board. Appendix B provides schemes that explain data flow and memory management of the solutions. Appendix A reports their source code notice. Section 6.2 gives the achieved performance results of these implementations. These implementations deal with 16bits integers audio inputs and a sampling rate of 48kHz.

5.3.1 DSB implementation.

Analyzes on the addition operation with ISEF.

The main advantage of the DSB algorithm regarding implementation on hardware is its simplicity. Indeed it only comes down to an addition when the $\frac{1}{M}$ normalization is not performed. On the other hand, this simplicity raises the questions of the usefulness of optimization for this part of the application. Knowing that every beamforming algorithm needs a source localizer upstream from it—and that this part is generally costly in therms of ressources consumption—the latter has to have priority to the access of the board's fast computation areas.

Nevertheless, taking into account the apparently low computational cost of the DSB beamforming, it is interesting to know if it can fit the ISEF's units that a cross correlation implementation left for free. Furthermore, as no connexion between these two implementations are needed, the lack of routing ressources due to excessive use of ISEF (cf. section 6.1) is not

a problem. The idea is therefore to strike a balance between ressources usage and algorithm performance for the beamforming implementation on the ISEF. In fact, if it requires too much computation units one can consider that ressources are somehow wasted in regards to the others (more costly) operations.

The comparison between ALU's and ISEF's addition operation was made for a reduced sum calculation. Two implementations on the ISEF were tested and each one of them has a reference implementation that is completely executed on ALU. These classical additions between two frames of n-samples use loops.

For the first implementation that uses the ISEF, groups of samples are loaded in two WRs and the direct sum between the WRs is made and stored in one output WR. It puts the samples side by side in the 128bits-wide WRs. Therefore 8 samples can fit the whole capacity of a WR. Then, a loop over these groups retrieves the n-samples result.

This implementation is however dangerous considering the possible overflows of the additions. Because there are 24 microphones in the array, 24 times 16bits-samples are added to obtain the DSB output. An overflow of at most $\lceil \log_2(24) \rceil = 5$bits can happend. Therefore, the output result should be coded with at least 5bits more than the inputs.

The second implementation avoids overflows completely by casting the result samples on 32bits and by outputting them via 2 WRs. It means also that the data has to be manipulated outside the WRs and that more operations are needed to retrieve the output samples from the two WRs.

Section 6.2 gives the results of these analyzes in terms of execution speed and ISEF usage.

Concerning the normalization, which is a division operation (or a floating point multiplication), ISEF can not achieve any optimization. Indeed, it can neither handle division nor floating point operations. The normalization has therefore to be done by the FPU with no possible optimization.

The global DSB algorithm.

Concerning the complete computation of the DSB algorithm over the 24 channels, figure B.14 shows the data flow and the structures that are used for this implementation.

As for the cross correlation, the data is initially stored in the main memory and each frame[1] that has to be computed by the ISEF is first transfered to the DATARAM. The result frame always stays in the DATARAM whereas the input frames are loaded one after another at the same address. The function compute_dsb() extracts the current groups of samples that p_dsb and p_frame point (respectively to the result and the input frames). The

[1]NB: the frame mechanism (cf. chapter 3) is not used here. The frames do not overlap; they represent contiguous blocks of data in the input file.

samples are loaded into the WRs and the EI dsb_isef operates. The result is stored back so that it is reused during the addition of the next channel.

Note that the calculation of the delays is done by the CPU in the ALU and the FPU. Then, as in Matlab®, the values are casted to integers so that they can be added to the value of the pointer p_frame. Therefore, a shifting of the beginning of the frame is applied depending on the current channel number and its associated delay.

5.3.2 GSC implementation.

The work on GSC on the stretch was limited in time and no working implementation is currently available. Nevertheless, this section gives the principle of the GSC algorithm that was set up and possible optimizations for it. Figure B.15 summarizes the data flow for this solution. On this figure, the links that are referred by numbers are supposed to be transfers of small amount of data over the WRs. The reader has to imagine that these groups of samples come from the right place as the mechanism of pointers would have been to complex to depict.

Arithmetic operations.

The first principle of the GSC optimization is to run the upper and lower branches of figure 5.7—respectively the DSB the BM algorithms—in parallel.

If one wants to use the ISEF for it, the arithmetic operations of both algorithms can be computed by the same EI in the ISEF. Indeed, as the same data is needed for both algorithm, the number of loadings to the WRs would be reduced.

For a further use of the BM outputs by the filters—that are also supposed to run on the ISEF in a second time—the frame results are stored temporarily in the ERs.

The following memory places have to be allocated in the ERs.

- prev_frame, a table that keeps the previously added frame in order to compute one output of the BM.
- res_bm, a two dimensional table to store the BM outputs.

The adaptive filters.

Several filters have to be applied on the BM outputs. The filters has the priority to be implemented in the ISEF because they use more ressources than the other calculations. The coefficients renewing is not an expensive calculation but it is beneficial to compute it in the ISEF too so that its result can be easily reachable by the filtering operations. For these reasons, the current weights for each filter are stored in the ERs. The major problem of

the renewing operation is that for each step, the previous result of the GSC beamformer has to be available.

The filters needs several tables in order to work properly.

- res_bm, the two dimensional table that stores the BM outputs.
- weights, a two dimensional table for the filter weights (it represents the $\mathbf{W}_m(k)$).

One of the main problems of the adaptive filter algorithm is that (5.10) returns floating points values. These values cannot be directly used in ISEF because it is now well known that it can not handle such formats. Furthermore, it is part of the algorithm that the values of the weights stay so low that casting them to integers would destroy their physical meaning. The values have first to be scaled up to preserve the desired number of significant digits and then a cast can be operated. It affects the filter's output by the same scaling factor. Therefore, the inverse scaling has to be performed by the FPU as soon as the data leaves the ISEF.

Chapter 6

Performance analyses.

6.1 Performances of cross correlation.

This section gives a complete report of the performance achievements for the different algorithms of cross correlation that were described in section 4.3. It also discusses the drawbacks and the advantages of each solution.

In order to refer to the cross correlation solutions easily, the following references are given:

i. straightforward implementation for which the ISEF is not used.

ii. square decomposition ($b = 4$) with ISEF and WRs.

iii. diagonal decomposition ($b = 4$) with ISEF and WRs.

iv. linear decomposition ($b = 40$) with ISEF and WRs.

v. square decomposition ($b = 4, 6, \text{and } 8$) with ISEF and IRAM.

vi. diagonal decomposition ($b = 4$) with ISEF and IRAM.

For each solution that uses the ISEF, the memory transfers between the main memory and the DATARAM can be done with or without DMA. These cases are going to be reported separately.

6.1.1 ISEF's ressources usage.

Tables 6.1 and 6.2 respectively report the ISEF's usage of implementations that use the WRs (ii, iii, and iv) and the IRAM (v[1]) for the 8bit resolution case.

[1]In implementation v, the extension instruction *add_cross()* merges 3 output samples in the $b = 8$ and $b = 6$ cases. For $b = 4$, either 2 or 7 samples are merged; both versions are referred in the tables respectively as (1) and (2).

f always refers to the simulated achieved ISEF's frequency for an issue rate of 1; the issue rate (IR) is derived from it. Max. cycles refer to the maximum output write cycles of the EI.

In the second table, 10 banks of the IRAM over 32 were used for each implementation. Note that there is no difference in ISEF's usage whether DMA is used or not.

EI	b	MUs/8192	AUs/4096	f(MHz)	ERs/4096	IR	Max. cycles
ii	4	2048(25%)	180(55%)	300	0(0%)	1	14
iii	4	4096(50%)	2448(55%)	232	574(14%)	2	20
iv	40	5120(63%)	2216(86%)	154	1552(37%)	2	8

Table 6.1: ISEF's usage of the extension instructions for cross correlation with WRs.

EI	b	MUs/8192	AUs/4096	f(MHz)	ERs/4096	IR	Max. cycles
v	8	8192(100%)	3533(86%)	-	339(8.3%)	-	-
v	6	4608(56%)	2926(71%)	185.2	273(6.7%)	2	10
v	4(1)	2048(25%)	2951(72%)	234.5	52(1.3%)	2	16
v	4(2)	2048(25%)	3400(83%)	105	172(4.2%)	3	14
vi	4	2048(25%)	3464(84%)	128	53(1.3%)	3	17

Table 6.2: ISEF's usage of the extension instructions for cross correlation with IRAM.

6.1.2 Cycles count and execution time.

Table 6.3 and 6.4 reports the Xtensa's cycles count and the corresponding execution time for the best versions of implementations i, ii, iii, iv, and v. These numbers provide the Xtensa's capabilities to run both a frame of 100ms and a whole file when applying the frame mechanism see in 3. The input resolution is 8bits and the sampling frequency is 8kHz.

EI	b	DMA	$\text{cycles}_{\text{frame}}$	t_{frame} (ms)	$\text{cycles}_{\text{file}}$	t_{file} (ms)
i	-	✗	2,138,951	7.1	124,950,739	417
ii	4	✗	447,503	1.5	40,751,422	136
ii	4	✓	423,605	1.4	44,353,896	148
iv	40	✗	296,061	1.0	30,955,049	103
iv	40	✓	294,757	1.0	35,085,588	117

Table 6.3: Cycles count for algorithms that use WRs and a 8bits input resolution.

EI	b	DMA	$\text{cycles}_{\text{frame}}$	t_{frame} (ms)	$\text{cycles}_{\text{file}}$	t_{file} (ms)
i	-	✗	2,138,951	7.1	124,950,739	417
v	6	✓	593,050	1.9	-	-
v	4(1)	✓	600,038	2.0	54,284,649	181
vi	4	✓	562,839	1.9	52,386,719	175

Table 6.4: Cycles count for the algorithms that use IRAM and a 8bits input resolution.

N.B.: as seen in section 4.3, the meaning of the value of b for linear decomposition slightly differs from the others decompositions. It is therefore normal to have a greater value here; it does not mean necessarily that more multiplications are performed per EI.

Note also that table 6.4 does not give the results for the $b = 6$ and $b = 8$ cases. Indeed problems occur during the compilation of the $b = 8$ case with bit-file generation (cf. section 6.1.5). When $b = 6$ the simulator's watchdog can not handle the computation of the whole file and no tests were made with the remote mode. Table 6.5 reports the 16bits resolution case, only the linear version (iv) was tested because it was the most efficient for the 8bits case. It is compared with the reference-solution i.

EI	b	DMA	$\text{cycles}_{\text{frame}}$	t_{frame} (ms)	$\text{cycles}_{\text{file}}$	t_{file} (ms)
i	-	✗	6,372,732	21.2	344,308,116	1148
iv	20	✗	1,129,227	3.8	88,019,443	293
iv	20	✓	1,126,868	3.7	95,584,729	319

Table 6.5: Cycles count for algorithm iv with a 16bits input resolution.

6.1.3 Real time considerations.

In order to match the system's expectations, the following references were given: a frame size of 100ms and frame shift of 20ms would enable to detect correctly the maximum of the cross correlation of the signals (cf. section 2.3). The real-time condition is then to achieve a cross correlation between two frames of 100ms in less than 20ms. Otherwise, either frame size or frame shift has to be adjusted in compliance with performance results and global application matters (i.e., which size and shift are needed as a minimal way to detect the phenomena correctly).

Table 6.6 shows the approximative limitations of configurations i and v (the most critical ones) for a 8bits input resolution. It gives the maximal number of input samples that can be processed by each of them in less than 20ms. And as a result, if it is possible to use them for each of the

given sampling frequencies. In each case, the maximum duration of an hypothetical frame is given.

EI	b	Max. frame size	8kHz	16kHz	32kHz	48kHz
i	-	1400	175ms ✓	88ms ✗	44ms ✗	29ms ✗
v	8	3000	375ms ✓	188ms ✓	94ms ✗	63ms ✗
v	6	2000	250ms ✓	125ms ✓	63ms ✗	42ms ✗
v	4(1)	1800	225ms ✓	113ms ✓	56ms ✗	38ms ✗
v	4(2)	1080	135ms ✓	68ms ✗	34ms ✗	23ms ✗

Table 6.6: Limitations of the algorithms in terms of input frame size, considering the real-time problem.

For a sampling frequency of 8kHz, all configurations are fast enough for real-time in this case. But when the the amount of samples gets bigger, some of the configurations can not handle real-time any more. Let us not forget either that this table is valid in the 8bits resolution case and that the audio inputs of the board can also handle 16bits resolution.

6.1.4 ISEF's capacity analysis.

The ISEF has to be exploited to its maximum as it is the fastest place for computation on the board. From the number of available MU in the ISEF, the number of possible multiplications between samples of a given resolution can be derived. Then, from the number of needed multiplications for a CC, the maximum number of input samples is easily derivable.

Let us assume the example of a CC between two frames of size N whose elements have a 8bits resolution. The square decomposition algorithm as it is described in figure 4.2 is used. The number of MU in the ISEF equals 8192; the largest computable CC is an 8×8 CC. It means that two MUs compute the result and the carry of the multiplication between two bits.

This reference can be used to predict the number of MU that an implementation will need. Furthermore, Stretch® provides reference that gives the number of AU and MU for each possible operation in the ISEF. Nevertheless, these predictions are not always right; indeed the number of needed ressources as well as the way ISEF uses them is variable and most of the time unpredictable. The coding style has also an influence on ISEF's usage.

6.1.5 The routing ressources problem.

As seen earlier in this section, some implementations are not runable on the board. Even if they fit the ISEF in terms of MUs and AUs, the compilation is not possible on the board because the bit-file can not be created.

It happens often when the implementation uses the ISEF too much—more than 85% of the computation units in the tested implementations.

This is due to an excessive use of its routing ressources. Indeed, even if all the computation units of the ISEF are not used, there might not be enough paths available to transfer data between them. Those routes are the so-called *routing ressources*.

The main problem is that it is not possible to predict exactly the number of routing ressources that an implementation will use before trying to compile with bit-file generation. Contrary to the computations units—for which Stretch® documents some ways of quantification (cf. previous section)—the lack of routing ressources is always hard to predict and the developper should always try to use them as few as possible.

To do so, the best solution would be to limit the flow of data in the ISEF (therefore, one should have an vision of how the EI is interpreted in it). The most straightforward solution is to reduce the implementation depth into the ISEF (i.e., in our case, to reduce the value of b). Although the number of needed multiplications always stays the same for the calculation of a frame, the smaller is b, the more complicated is the handling and the smaller becomes performance.

6.2 Performances of beamforming.

6.2.1 The addition operation optimization.

This section gives the result of the reduced-addition operation analysis that was described in section 5.3. The ISEF version uses only a small amount of ressources: only 128 arithmetic bits are needed. Note that 128 corresponds to the bitwidth of a WR: each arithmetic bit computes an addition between two bits and outputs a carry.

Furthermore, this ISEF configuration returns an output in only 5 EU and is runable with an issue rate of 2. It means that the computation of the addition operation between two groups of eight 16bits-samples takes 9 CPU-cycles (I, R, E, W, 5×EU) with the WR whereas it takes about 60 cycles with the ALU. The major problem is that for each group of 8 samples, the WRs have to be loaded again and the result has to be stored in memory. These are the most time-consuming operations.

MUs/8192	AUs/4096	f(MHz)	IR	Max. cycles
0(0%)	128(3.1%)	234	2	5

The implementation that outputs a result on 32bits needs obviously more ressources as more computations are made by the ISEF to process the same number of input samples. The usage it makes of the ISEF is given here.

Figure 6.1 gives the evolution of needed cycles when the number of total additions n increases. On the figure:

MUs/8192	AUs/4096	f(MHz)	IR	Max. cycles
0(0%)	160(3.9%)	298	2	3

- squares represent the case with no ISEF with a result on 16bits.
- rhomboids represent the case with ISEF with a result on 16bits.
- stars represent the case with no ISEF with a result on 32bits.
- circles represent the case with ISEF with a result on 32bits.

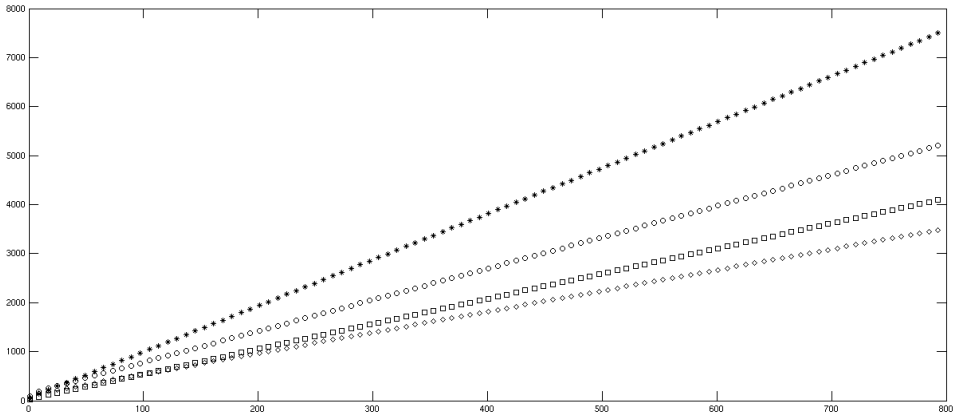


Figure 6.1: Number of cycles needed by the addition implementations.

One can easily see that the ISEF implementations curves have a lower slope their respective reference without ISEF. They become lower for values of n over about 100 (which is likely to be always the case in the application). It is also visible that the curves for ISEF implementations are not linear and that therefore, the square line might go over the circle one (i.e., all the ISEF implementation might end up to be more efficient that the others).

6.2.2 Results of the DSB implementation.

For the global computation, table 6.7 gives the calculation times and the number of cycles for the implementations of the addition operation on a whole file of 1s. It represents the performances of the delay and sum algorithms on 24 channels, with ($\cdot_{\text{norm}}$) or without the $\frac{1}{N}$ normalization.

The previous assumption was right: all the ISEF configurations become more efficient when n is great enough. Concerning real time, these implementations compute the DSB algorithm on 24 files of 1s in less than 50ms. This obviously fast enough to compute this algorithm in realtime.

Output resolution	ISEF	cycles	t (ms)	$\text{cycles}_{\text{norm}}$	t_{norm} (ms)
16bits	✗	11,032,850	36.8	13,749,752	45.8
16bits	✓	7,797,643	26.0	10,575,283	35.3
32bits	✗	11,697,912	39.0	14,552,414	48.5
32bits	✓	10,314,584	34.4	13,166,149	43.9

Table 6.7: Cycles count for addition algorithms.

6.2.3 Audio considerations.

As told before, the aim of beamforming on a single speaker is to enhance its speech. In order to quantify the improvements of a beamformer in this case, the SNR improvement between one of the inputs $x_i(t)$ and the beamformer's output is a precious information.

In the two speakers case, the beamformer performs a separation of the speeches. Here, several signals are mixed and it is hard to quantify the fall of one speech in particular. Along SNR, [10] gives other means of performance measurement in blind source separation. In particular, the Source to Interferences Ratio (SIR) gives a good idea of the algorithms performances.

The principle of the performance measures described in [10] is to decompose the beamformed output signal $s(t)$ of a source $s_i(t)$ as the sum $s(t) = s_i(t) + e_{\text{interf}}(t) + e_{\text{noise}}(t) + e_{\text{artif}}(t)$ where $e_{\text{interf}}(t)$ represents the sources which accounts for the interferences of the unwanted sources, $e_{\text{noise}}(t)$ is the disturbing noise (but not the sources), and $e_{\text{artif}}(t)$ is an "artifact" term that may correspond to artifacts of the separation algorithm. Then, the SIR is given as (6.1).

$$SIR = 10 \log_{10} \frac{\|s_i\|^2}{\|e_{\text{interf}}\|^2} \tag{6.1}$$

It means that the exact original signal s_i or an accepted version of it (i.e., a good capture of it with no interferences) has to be at disposal.

Qualitatively, the estimate of the undesired speech removal is done by listening at the output result. Although an improvement can be heard, the results are not promising. Indeed, the tested algorithm are not performant in echoic environments. In this case, some reverberations of the undesired signals arrive to the array in the steered direction: they are therefore considered as part of the desired signal by the beamformer and can not be removed efficiently with basic algorithms.

6.3 Conclusions on performances.

6.3.1 Conclusions on the cross correlation performances.

All the implementations that use the ISEF are faster that the computation with the ALU only (i). It confirms that using the ISEF is a way of increasing performances. Furthermore, implementations that do not use the IRAM (ii, iii, and iv) are in any case faster than the other ones (v, vi). Use of IRAM is a bit complicated and should therefore be used when the application really needs it only. When one needs to use IRAM, it is however recommended to use DMA transfers to load the data directly into it: the performances are in this case much better.

Implementation iv is clearly the fastest. Compared with implementation i and from a frame point of view, iv decreases the number of cycles about 7.3 times in the 8bits case and about 5.7 times in the 16bits case. The difference between implementations is often the number of loaded samples per block; the level of block overlapping (i.e., the number of samples that are needed from one block to the other) of the decomposition is an important factor. Furtermore, the way that the algorithm merges the blocks into shall not be too complicated either: the diagonal and linear decompositions are the best regarding that.

Looking at the calculation of one frame, the approaches that use DMA are faster than the other but it is the opposite for a whole file computation. DMA initialization causes partly this difference and therefore, using DMA for a streaming algorithm would be preferable. Finally, ISEF's ressources usage is to consider relatively to algorithm performances. Indeed, the application has to run not only the cross correlation algorithm but also other computations that may need the ISEF as well. The main problem resides in the prediction of the amount of ressources that an implementation needs.

6.3.2 Conclusions on the beamforming performances.

The analysis of the addition operation on reduced cases showed the advantages of the ISEF solutions compared to the others. Nevertheless, for real cases of computations, they are less than 1.5 times faster that the others considering the number of CPU cycles. It is hard to improve addition efficiently mainly because the ALU computes this operation already fast. Indeed, it is efficient enough to run in real time.

Therefore, the DSB algorithm shall not have priority for implementation on the ISEF. Nevertheless, regarding the low cost of this operation in terms of ressource consumption, its implementation can only be beneficial if a part of the ISEF is still available.

Concerning the implementation of the GSC, there is no possible consideration of speed performances as told in section 5.3. Some advantages regarding its implementation on the board are however noticeable. First,

let us notice that data transfers to the ISEF are often costly as its input width is somewhat narrow. Some processes of the GSC algorithm are highly related between each other; they share common inputs and results. This characteristic reduces the ISEF weak points as input data have to be loaded only once for different operations.

Moreover, let us not forget that the beamforming and the source localization algorithms have to run together on the same board in real time. When combining the results of sections 6.1 and 6.2, one can see that it is doable. Access to the ISEF stays really limited though and optimization with this ressource has to be reserved to the most consumming algorithm.

Conclusion.

There are various way to optimize the implementation of an algorithm, each one at its level. Finding mathematical simplifications in the calculations is the first possibility. Then, by analyzing the physical meaning of the computation as well as its precision needs, one can adjust the amount and the quality of both input and output data. Therefore, resolution, sampling frequency, frame length, and frame shift are configurable depending on the desired speed achievements. Finally, modifications on hardware processing of the data are in our case the best possible optimizations. Controlling memory places and calculation units can indeed increase the speed of routines.

However, this way of optimization needs a perfect knowledge of the board functioning. Hence the first task is to acquaint oneself with the board specifications. The Stretch® board has some non negligible advantages regarding fast computation. Its hybrid layout makes it flexible; in fact it is rare to make CPUs and reprogramable hardware interact this way. The high potential of computation of the ISEF makes the algorithms significantly faster. But this thesis also showed the drawbacks of the board. In particular, the ISEF always caught the attention because of its small capacity and its narrow inputs. Because of that, the data is transported little by little to it.

Generally, one of the main problems was to adapt the several algorithms to the board specificities and especially to the ISEF's shape. Considering its relative low capacity of computation, the calculations had to be split into groups; it led to the so-called "decompositions". For each one of them, a merging algorithm had to be found. The amount of data in the decomposition's groups (i.e., the data that the ISEF processes in one EI) was often the limiting factor of the implementations in terms of performances. Indeed, the more this amount is low, the less the processing is efficient; it is notably due to the memory transfers times.

Future works.

Concerning the hardware part, there are still unexplored zone of the board that could be used to increase performances. First, the time of reconfiguration of the ISEF is currently really competitive in the market. This small

latency was not used in this thesis but it enables fast changes of ISEF configuration that could be useful to solve its problem of capacity. Then, one has to use the presence of several processors on the same board to its best advantage. Indeed, multithreading is often beneficial for applications with parallel computations. Therefore, one should study the PA network functioning into details. Besides optimization of the current algorithms, these features could enable the use of more complex processes. That would notably be advantageous to counter the effects of echoic environments on beamforming.

Unfortunately, a deep analysis of the execution of the algorithms on the board and in real conditions has not been carried out. Only a small amount of tests were made. In oder to run the algorithms on the board, the audio inputs mechanisms have to explored as well.

Appendix A

Source code.

Notice on source files.

The source files are available upon request at this address:
`mailto:boris.clenet@insa-rennes.fr`.

Cross correlation.

Regarding source files organization, every implementation of the cross correlation has the same structure. The main function main_*.c is the same for every implementation, it only differs whether DMA is needed or not. main_no_dma.c and main_dma.c refers to these cases. The main functions also handle the interface with audio files. They will be reported only one time each and for the 8bit resolution case. When using IRAM, main_dma_iram.c is used.

Then, each implementation needs a cross_*.c and a cross_*.xc file. For example cross_ii.c refers to algorithm ii. Implementation i is the only one that does not need a *.xc file because it makes no use of the ISEF. Furthermore, it does not use the DMA version of the main file. The cross_*.c handle the computation of a frame and the cross_*.xc of a block. Only the case $b = 4(1)$ is reported for implementation v.

Finally, cross.h is a header file that calls all the needed libraries and declares the structure for RIFF-WAVE files handling.

Beamforming.

Files main_beam.c refers to the main file that is common to all the implementations of the DSB. It mainly handles the interface with audio files.

Then, dsb_functs_*.c contains the functions that calculates the delays and that computes the sum loop. The *.xc files are the ISEF configurations. "32" refers to the case that deals with output samples on 32bits and "wr" when ISEF is used.

Appendix B

Figures.

Figures of chapter 1.

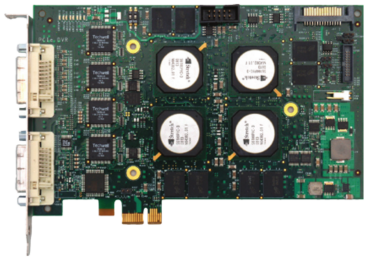


Figure B.1: Picture of the VRC6016 card.

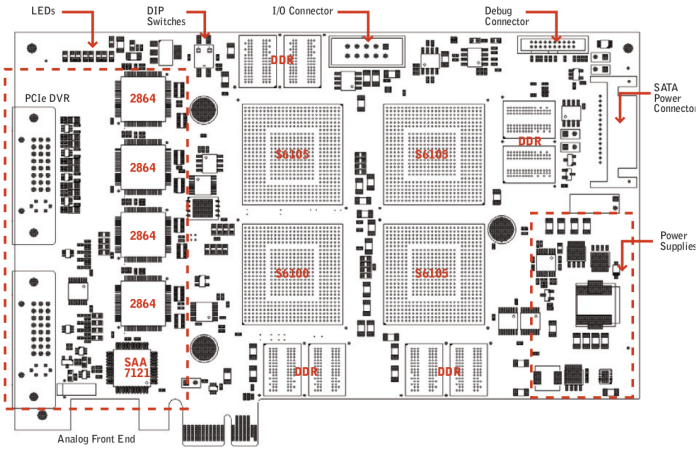


Figure B.2: VRC6016 board layout.

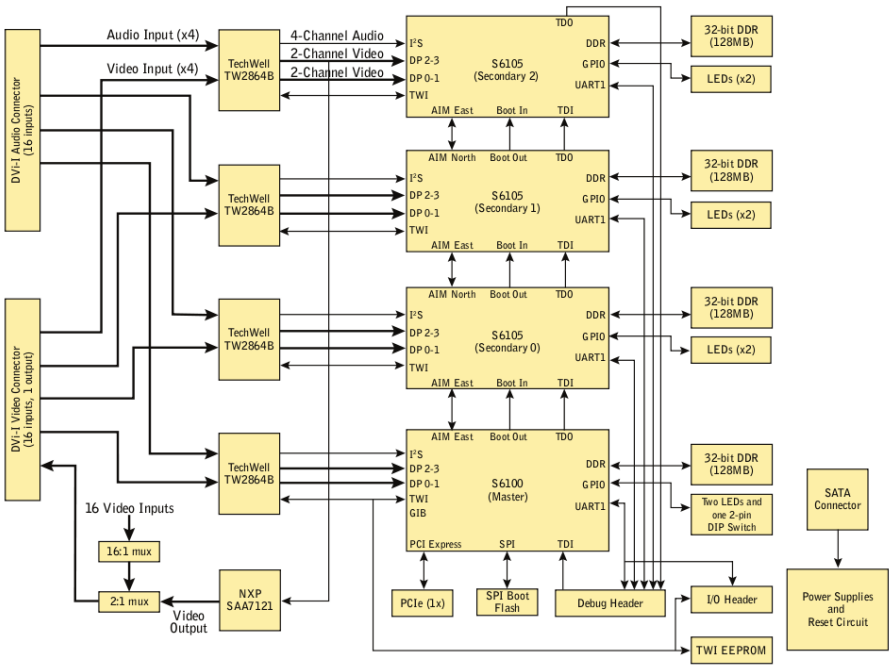


Figure B.3: VRC6016 board block diagram.

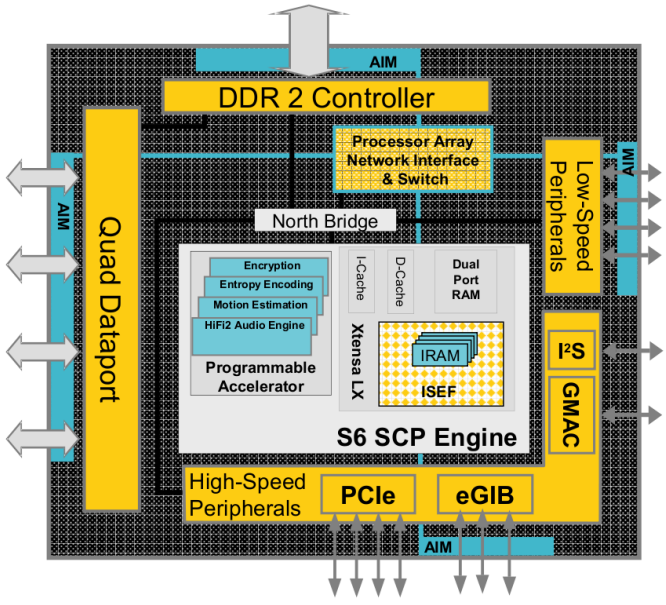


Figure B.4: Architecture of the Xtensa processor.

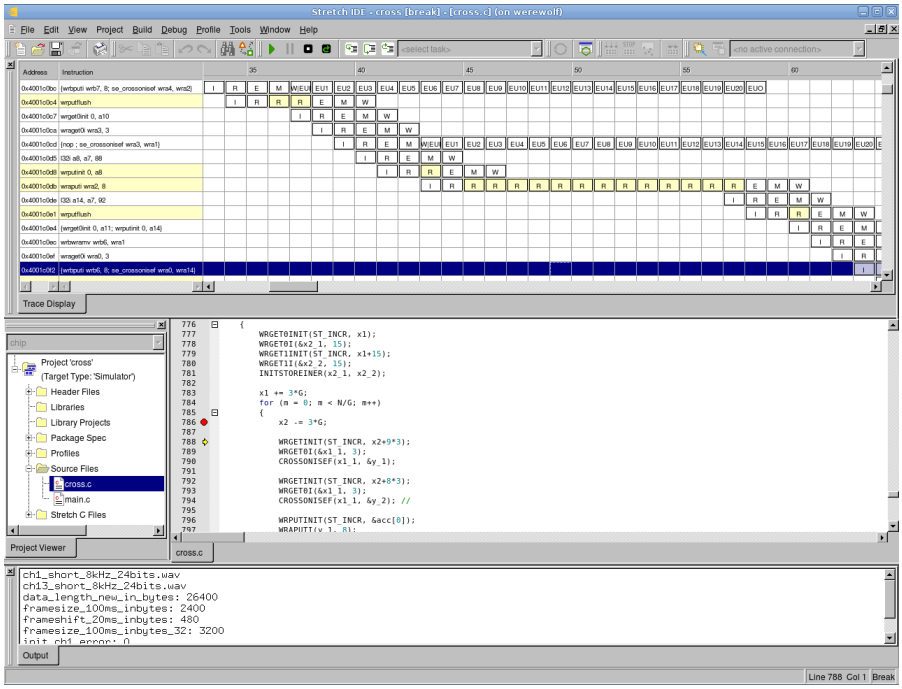
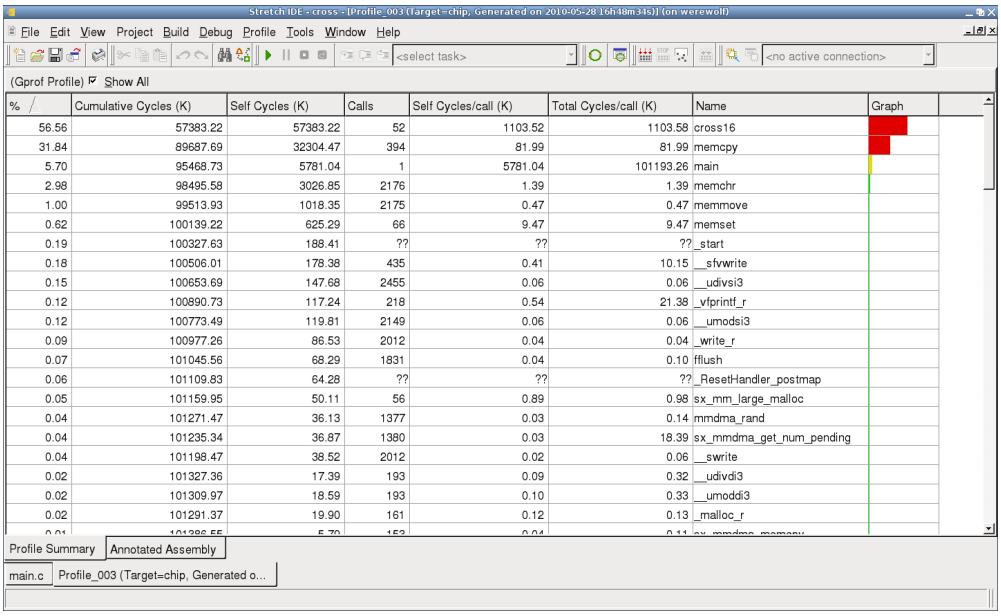


Figure B.5: Pipeline view of the Stretch IDE.



%	Cumulative Cycles (K)	Self Cycles (K)	Calls	Self Cycles/call (K)	Total Cycles/call (K)	Name	Graph
56.56	57383.22	57383.22	52	1103.52	1103.58	cross16	
31.84	89687.69	32304.47	394	81.99	81.99	memcpy	
5.70	95468.73	5781.04	1	5781.04	101193.26	main	
2.98	98495.58	3026.85	2176	1.39	1.39	memchr	
1.00	99513.93	1018.35	2175	0.47	0.47	memmove	
0.62	100139.22	625.29	66	9.47	9.47	memset	
0.19	100327.63	188.41	??	??	??	_start	
0.18	100506.01	178.38	435	0.41	10.15	__sfvwrite	
0.15	100653.69	147.68	2455	0.06	0.06	__udivsi3	
0.12	100890.73	117.24	218	0.54	21.38	_vfprintf_r	
0.12	100773.49	119.81	2149	0.06	0.06	__umodsi3	
0.09	100977.26	86.53	2012	0.04	0.04	_write_r	
0.07	101045.56	68.29	1831	0.04	0.10	fflush	
0.06	101109.83	64.28	??	??	??	_ResetHandler_postmap	
0.05	101159.95	50.11	56	0.89	0.98	sx_mm_large_malloc	
0.04	101271.47	36.13	1377	0.03	0.14	mmdma_rand	
0.04	101235.34	36.87	1380	0.03	18.39	sx_mmdma_get_num_pending	
0.04	101198.47	38.52	2012	0.02	0.06	__swrite	
0.02	101327.36	17.39	193	0.09	0.32	__udivdi3	
0.02	101309.97	18.59	193	0.10	0.33	__umoddi3	
0.02	101291.37	19.90	161	0.12	0.13	_malloc_r	

Figure B.6: The profiling functionality.

Figures of section 4.3

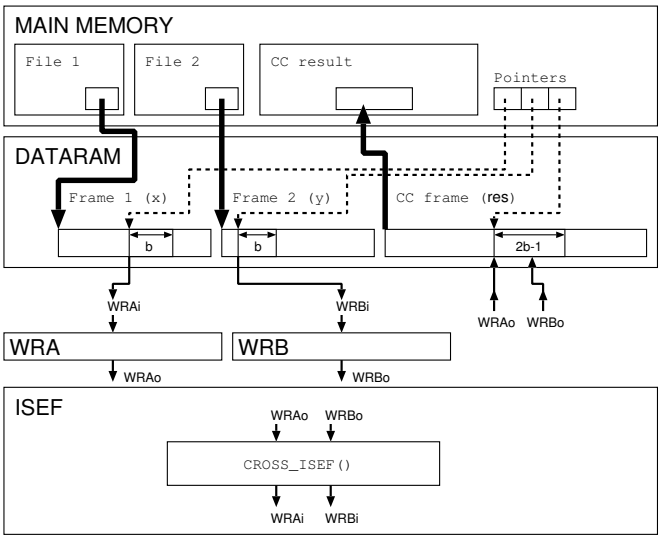


Figure B.7: Data flow in the case of ISEF use with square decomposition.

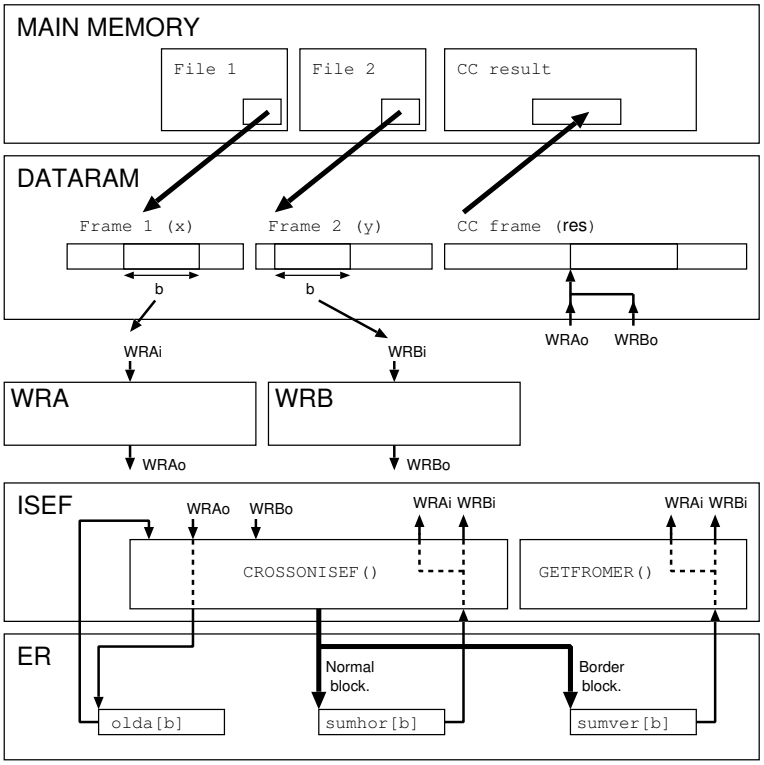


Figure B.8: Data flow in the case of ISEF use with diagonal decomposition.

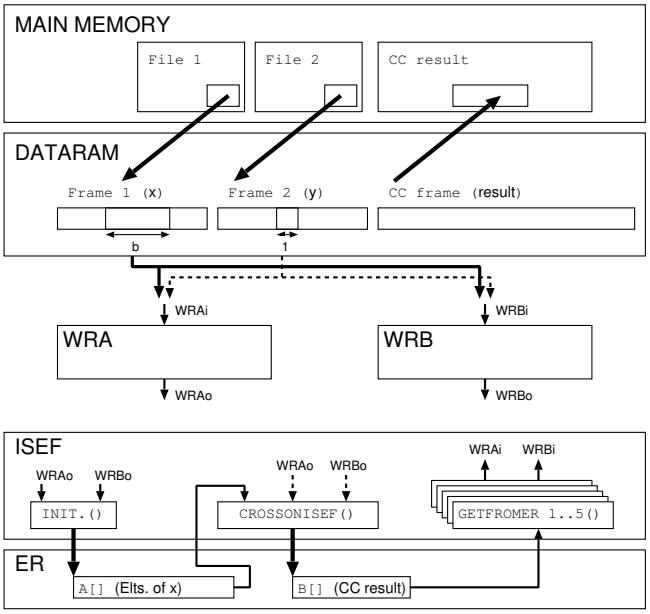


Figure B.9: Data flow in the case of ISEF use with linewise decomposition.

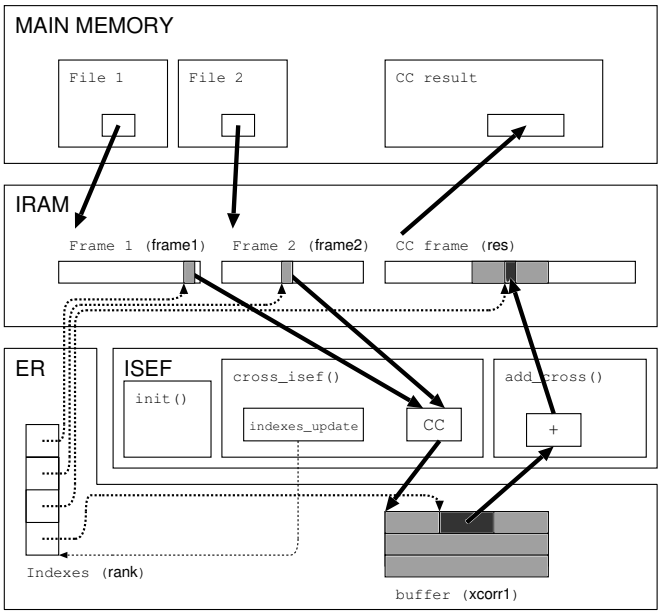


Figure B.10: Data flow in the case of IRAM use.

Figures of section 5.1

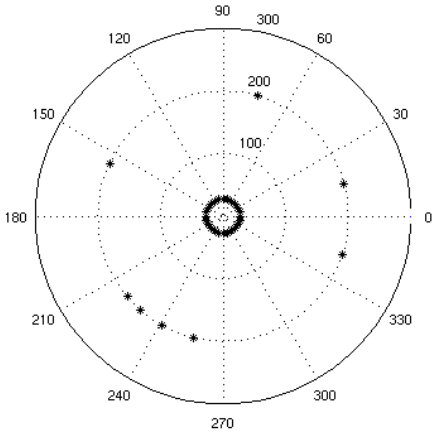


Figure B.11: Room organization for the recordings.

The angular positions of the sources are, by increasing order of number (from 1 to 8), 15, 342, 229, 241, 220, 155, 256, and 74°.

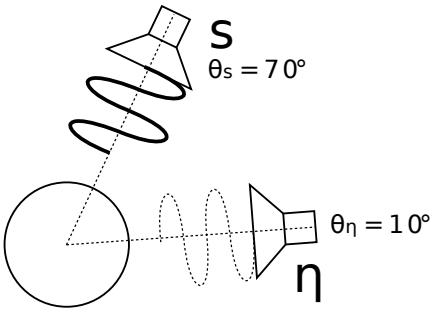


Figure B.12: Configuration with two sources.

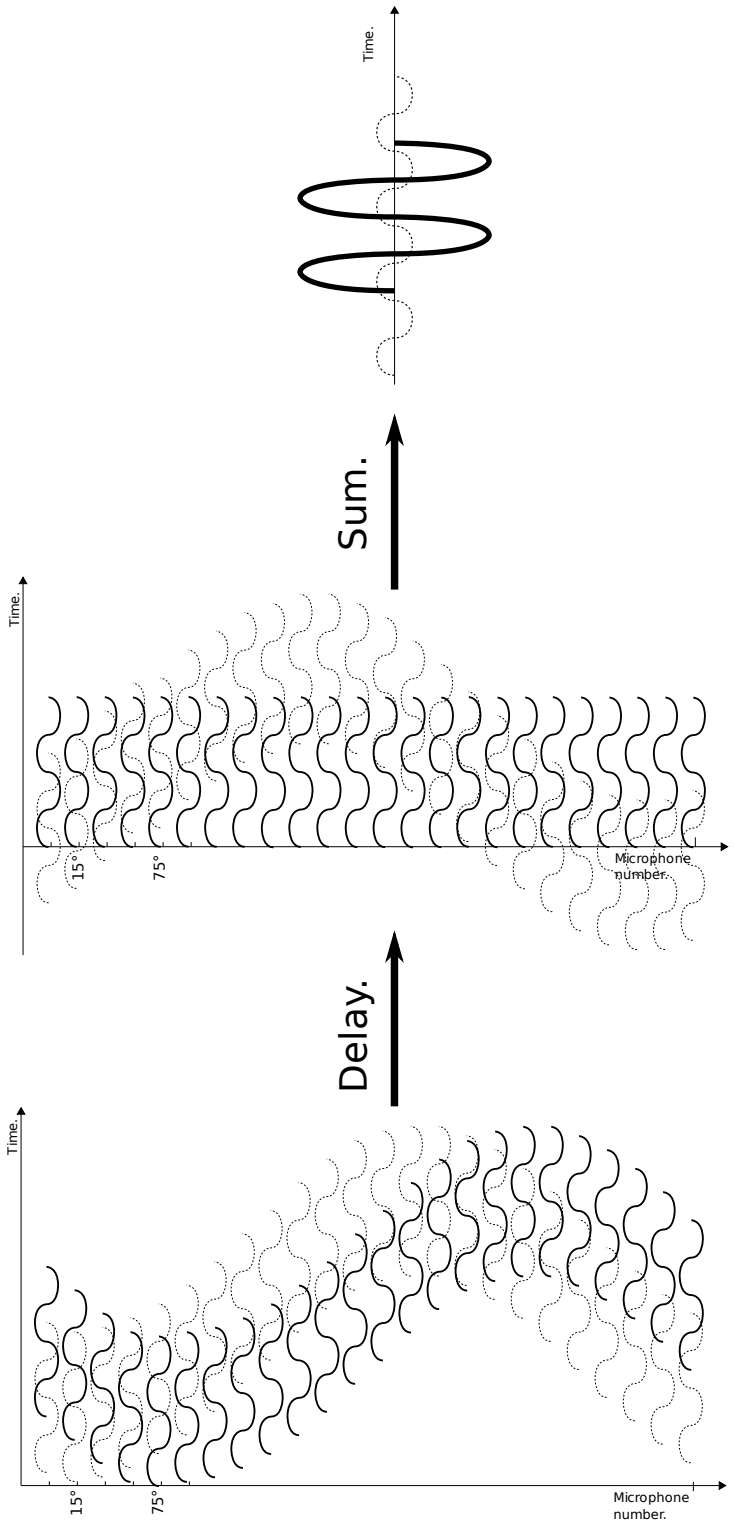


Figure B.13: The DSB "alignment" principle.

Figures of section 5.3

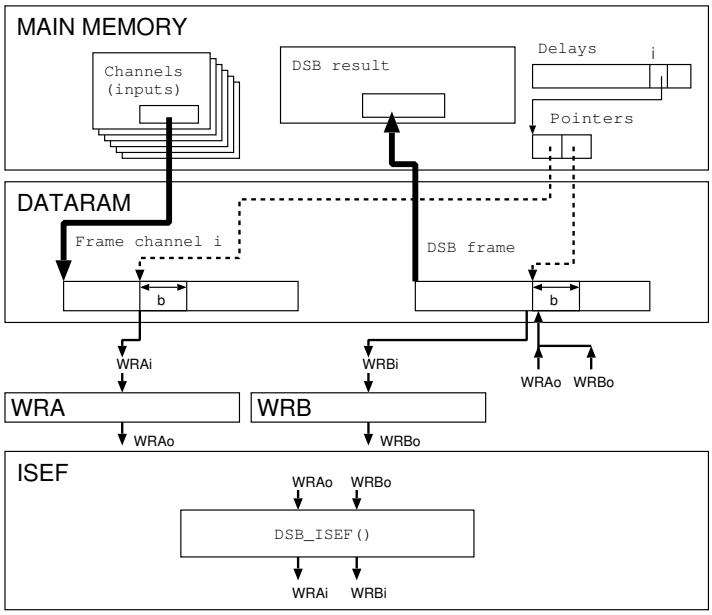


Figure B.14: Data flow of the DSB implementation.

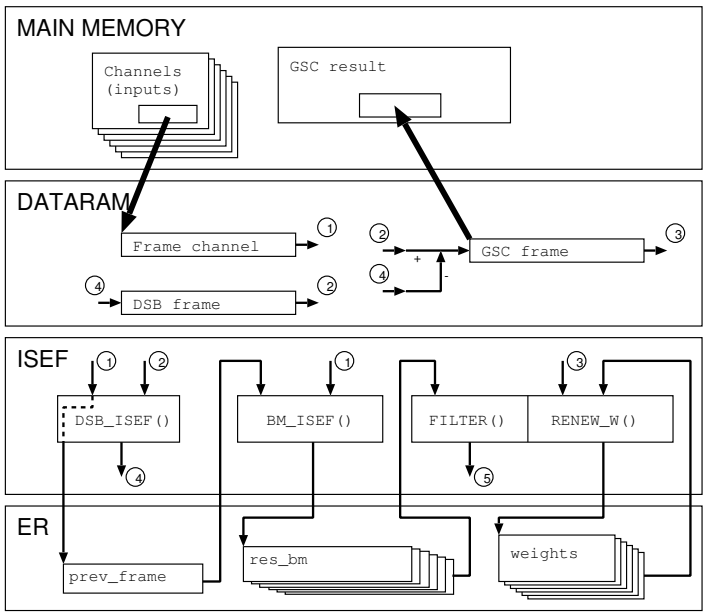


Figure B.15: Data flow of the GSC implementation.

.

Bibliography

[1] Matthias Wölfel and John McDonough, *Distant Speech Recognition*. Wiley, 2009.

[2] Jacob Benesty, Jingdong Chen, and Yiteng Huang, *Microphone Array Signal Processing*. Springer, 2008.

[3] Jian Li and Petre Stoica, *Robust Adaptive Beamforming*. Wiley, 2006.

[4] Kidiyo Kpalma and Véronique Haese-Coat, *Traitement Numérique du Signal*. Ellipses, 2003.

[5] Lukas Ottowitz, *Acoustic Source Localization with a Circular Microphone Array*. Master Thesis, SPSC, TU Graz, March 2008.

[6] *Stretch®, Inc. Documentation and Help*.

[7] Bin Huang, Chong Zhu, Wei Fan, Yu-Fu Tao, and Quing-ning Zeng, *Microphone Array Speech enhancement Based on Filter Bank Generalized Sidelobe Canceler*. Education Technology and Computer Science, March 2009.

[8] Fei Huang, Wei-xing Sheng, and Xiao-feng Ma, *Robust partially adaptive array processing based on generalized sidelobe canceler*. Microwave Conference, December 2008.

[9] Cha Whang, Demba E. Ba, and Zhenyou Zhang, *Maximum Likelihood Sound Source Localization and Beamforming for Directional Microphone Arrays in Distributed Meetings*. Journal of LaTeXclass files, January 2007.

[10] Emmanuel Vincent, Rémi Gribonval, Cédric Févotte, *Performance Measurement in Blind Audio Source Separation*. IEEE Trans. on audio, speech, and language processing, July 2006. `http://bass-db.gforge.inria.fr/bss_eval/`.

[11] Hai Huyen Dam, Seven Nordholm, Siow Yong Low and Kok Lay Teo, *Steerable far-field Circular Array*. Communications Theory Workshop, February 2006.

[12] Nikolaos Mitianoudis and Michael E. Davies, *Using Beamforming in the Audio Source Separation Problem.* Signal Processing and Its Applications, July 2003.

[13] Nikolaos Mitianoudis and Michael E. Davies, *Audio Source Separation of Convolutive Mixtures.* IEEE Trans. on Speech and Audio Processing, September 2003.

[14] Terence Betlehem and Robert C. Williamson, *Acoustic Beamforming Exploiting Directionality of Human Speech Sources.* Acoustics, Speech, and Signal Processing, April 2003.

[15] Jianfeng Chen, Louis Shue, and Senjin Liu, *Fixed Blocking Matrix for Robust Microphone Array Beamforming.* Signal Processing and its Applications, July 2003.

[16] Hiroshi Saruwatari, Shoji Kajita, Kazuya Takeda, and Fumitada Itakura, *Speech Enhancement Using Nonlinear Microphone Array With Complementary Beamforming.* Acoustics, Speech, and Signal Processing, March 1999.

[17] Osamu Hoshuyama, Akihiko Sugiyama, and Akihiro Hirano, *A Robust Adaptive Beamformer for Microphone Arrays with a Blocking Matrix Using Constrainted Adaptive Filters.* IEEE Trans. on signal processing, October 1999.

[18] Osamu Hoshuyama and Akihiko Sugiyama, *A Robust Generalized Sidelobe Canceler with a Blocking Matrix Using Leaky Adaptive Filters.* IEEE Trans. on signal processing, October 1997.

[19] Barry D. Van Veen and Kevin M. Buckley, *Beamforming: A Versatile Approach to Spatial Filtering.* IEEE ASSP Magazine, April 1988.

[20] Henry Cox, Robert M. Zeskind, and Mark M. Owen, *Robust Adaptive Beamforming.* IEEE Trans. on acoustics, speech, and signal processing, 1987.

[21] Lloyd J. Griffiths and Charles W. Jim, *An Approach to Linearly Constrained Adaptive Beamforming.* IEEE Trans. on antennas and propagation, January 1981.