
DIOPSIS 740: A dual processors architecture.

Authors: Michele Panzeri, Giordano Tamburrelli
Revisori: Marco Domenico Santambrogio
Data: 10/06/2005
Version: 1.4
State: Draft.
Document spreading

This document is totally free.
## Revisions

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>State</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>05/05/2005</td>
<td>1.0</td>
<td>Draft</td>
<td>Written by Michele Panzeri</td>
</tr>
<tr>
<td>20/05/2005</td>
<td>1.1</td>
<td>Draft</td>
<td>Revision by Giordano Tamburrelli</td>
</tr>
<tr>
<td>08/06/2005</td>
<td>1.2</td>
<td>Draft</td>
<td>Revision by Michele Panzeri</td>
</tr>
<tr>
<td>09/06/2005</td>
<td>1.3</td>
<td>Draft</td>
<td>Revision by Giordano Tamburrelli</td>
</tr>
<tr>
<td>10/06/2005</td>
<td>1.4</td>
<td>Draft</td>
<td>Revision by Michele Panzeri</td>
</tr>
</tbody>
</table>
Figure index

Figure 1 - Inside the processor................................................................. 8
Figure 2 - Peripherals and buses .............................................................. 11
Figure 3-Operators' block...................................................................... 16
Figure 4-Principle of localization ............................................................ 19
Figure 5 - A general schema of the application ....................................... 19
Figure 6-Formula of cross-correlation.................................................... 20
Figure 7-Formula of discrete cross-correlation........................................ 20
Figure 8-C-code for cross-correlation..................................................... 21
Figure 9-Profile of c-code...................................................................... 22
Figure 10-Inner loop.............................................................................. 23
Figure 11-Assembler code of the inner loop.......................................... 23
Figure 12-VLIW scheduling of c-code.................................................... 24
Figure 13-Assembler/C code of the inner loop....................................... 26
Figure 14-Assembler code of the inner loop.......................................... 26
Figure 15-Profile of Assembler-code..................................................... 27
Figure 16-VLIW scheduling of Assembler-code...................................... 27
Figure 17-Profile of Assembler-code with sw-pipelining....................... 28
Figure 18-VLIW scheduling of Assembler-code with sw-pipelining........ 29
Figure 19-VLIW scheduling of Assembler-code with unrolling.............. 29
Figure 20-Profile of Assembler-code with unrolling.............................. 30
Figure 21-How is compiled a "for".......................................................... 31
Figure 22-Flow chart of the for............................................................... 31
Figure 23-How is compiled a "do-while"............................................... 31
Figure 24-VLIW scheduling of Assembler-code with while.................... 32
Figure 25-Profile of Assembler-code with while................................... 32
Figure 26-Speed-up of the inner loop.................................................... 33
Figure 27-Speed-up of the inner loop.................................................... 33
Figure 28-Speed-up of the function....................................................... 34
Figure 29-Percentage of time spent in the inner loop............................ 34
Figure 30 - A schema of the synchronization........................................ 36
Figure 31 - A diagram of the application............................................... 39
Figure 32 - Layers of software on Diopsis 740...................................... 40
Figure 33 - A view of Jtst ................................................................... 42
Figure 34 - A screenshot of Made .......................................................... 44
Introduction

The main purposes of this project are basically two:

- To study DIOPSIS 740 architecture and in particular to stress the limits of this architecture through a case study.
- To beta-test the development tools for DIOPSIS 740 pointing out advantages and disadvantages of the methodology of development proposed by Atmel.

This project has been conducted by a team of two people this led to achieve both of the goals above described.

At the beginning we spent a lot of time reading a huge amount of documentation about this architecture to achieve a deep knowledge of all useful details.

After that we focused on the limits of the architecture and we pointed it out studying a specific application to use as a particular case study that could easily show this limits.

During the real implementation of this case study we could deeply test the tools provided with Diopsis 740 and the methodology proposed by Atmel to develop applications.

Last but not least our real purpose is to find a general high-quality method to solve this kind of problems in a dual processors system.

But what is Atmel and what Diopsis really is?

Atmel Corporation is an industry leader in the design and manufacture of advanced semiconductors, with focus on microcontrollers, nonvolatile memory, logic, radio frequency (RF) components and sensors. These functions are marketed as standard products, application-specific standard products (ASSPs) or customer-specific products (ASICs).

DIOPSIS 740 instead is a dual core system for audio, communication and beam-forming applications integrating: a microcontroller(32-bit RISC ARM7TDMI) and a DSP(40-bit floating point VLIW DSP, the ATMEL mAgic DSP).

DIOPSIS 740 combines the best in RISC with ARM, and the best in DSP with mAgic, giving the user an integrated platform for a wide spectrum of applications such as: ultrasound, High end audio and Speech, Audio enhancement, beam-forming base stations, beam-
forming for High Comfort Audio Conference, Radar, Sonar, Cruise Control, High speed, modem, digital radio etc etc.

This report describe precisely all our work: first of all we’ll introduce the architecture of Diopsis 740 and the we will focus on mAgic and his features.

After that we'll expose the case study that we have developed and we'll show how to get the best from this DSP processor stressing the capabilities of partitioning of work between the two processors and parallelism, we have chosen indeed the computation of cross correlation routine thought to be useful in the analysis of the waves of an **earthquake** or a **tsunami**.

Finally we’ll show and analyze the tools used to develop this application and other features like the operating systems available on Diopsis 740.
**Diopsis 740: the architecture.**

*A dual processor system.*

Diopsis 740 is a Dual CPU processor integrating a mAgic DSP and an ARM7TDMI, plus a total of 245 Kbytes of SRAM.

The ARM7TDMI is a general purpose processor that gives the right flexibility to the system in order to apply this architecture to every field of interest. Instead the mAgic is a very high performance, VLIW oriented, DPS processor that led the architecture to manage efficiently every application with hard constraints of mathematical computation or real time requisites.

![Figure 1 - The Diopsis 740](image)

![Figure 1-Inside the processor](image)
This architecture is indeed optimally suited for FP applications with a significant need for complex domain computations like FFT and frequency domain phase-shift algorithms, requiring high dynamic range and maximum numerical precision.

The general purpose processor is a member of Advanced RISC Machines (ARM) family of 32-bit microprocessors, which offer high performance combined with a very low power consumption.

The ARM family is based on a Reduced Instruction Set Computer (RISC) principles, and the instruction set and the related decode mechanism are much simpler than those of micro programmed Complex Instruction Set Computers.

This simplicity, which results in a high instruction throughput and impressive real-time interrupt response, combined with a rich set of peripheral and 32 Kbytes internal memory provide a highly flexible and integrated solution for a wide field of possible applications.

Pipelining is employed so that all parts of the processing and memory system can operate continuously.

The ARM7TDMI processor is built with a bank of 37 32-bit registers and six status registers and seven operation modes are supported:

1. User (usr): The normal ARM program execution state
2. FIQ (fiq): Fast Interrupt request; it’s connected to the mAgic Halt signal
3. IRQ (irq): Used for general-purpose interrupt handling
4. Supervisor (svc): Protected mode for the operating system
5. Abort mode (abt): Entered after data or instruction prefetch abort
6. System (sys): A privileged user mode for the operating system
7. Undefined (und): Entered when an undefined instruction is executed

Mode change can be made under software control or can be brought about by external interrupts or execution processing.

Most application programs execute in User mode. The non-user modes are entered in order to service interrupts or exceptions, or to access protected resources. Each operating mode has dedicated banked registers for fast exception handling.

The ARM processor moreover operate in little-endian mode.

The system has two buses: the main bus is the ASB (ARM System Bus), and the other one is the APB (ARM Peripherals Bus) which is designed for accesses to on chip peripherals.
An interface between the ASB and the APB is provided by the AMBA Bridge. The D740 is also equipped with a large set of useful peripherals controlled by the ARM through an On-chip Peripheral Data Controller (PDC) that transfer data to the memories in the DMA without the intervention of the processor which can continue it’s normal computation.

Moreover the PDC removes the processor interrupt handling overhead reducing significantly the number of clock cycles dedicated to data transfer.

Each peripherals has a 16K-byte address space allocated in the upper 3M bytes of the 4GByte address space. To maintain an efficient management of the peripherals register set ( which are composed of control, mode, data, status and interrupt registers ) frequently the registers are mapped into three memory locations.

The available peripherals are:

- **EBI (External Bus Interface)**: the EBI generates the signals that controls the access to the External Memory or peripherals devices.
- **ADDA (Analog to Digital and Digital to Analog interface)**: the ADDA provides 4 channel serial interface toward stereo audio 24-bit ADC and DAC.
- **SPI (Serial Peripherals Interface)**: two four-wire serial interfaces provide a simple industry-standard communication way managed by the Peripheral Data Controller.
- **AIC (Advanced Interrput Controller)**: the AIC in an 8-level priority, individually maskable, vectored interrupt controller.
- **PIO (Parallel I/O Controller)**: The PIO features 32 programmable I/O lines, 28 PIO lines are available on D740 pads, while the remaining 4 are only internal.
- **TC (Timer Counter)**: the TC contains three identical 16-bit timer/counter channels
- **WD (WatchDog Timer)**: the WD can be used to guard against system lock-up is the software becomes trapped in a deadlock. If an overflow occurs, the watchdog timer generates processor interrupts via the Advanced Interrput Controller (AIC) and an external low pulse through the PIO.
- **CLKGEN (Clock Generator)**: the clock generator provides divided clocks for several peripherals: the Timer Counter, the Watchdog, The USARTs and the SPIs.
Diopsis: A dual processor architecture

Figure 2 - Peripherals and buses

(See the Atmel documentation for a complete description about the ARM processor and its peripherals).

mAgic instead is a high performance VLIW DSP delivering 1 Giga FP operations per seconds at a clock rate of 100 MHz. It also has 512 data registers, 16 address registers, 10 independent operating units and 2 independent address generation units.

The main components of the processor are the core processor, the on-chip memories and the interfaces to and from the ARM subsystem. The core processor above mentioned is composed by the operators block, the register file, the address generation unit and the program decoding and sequencing unit.

To achieve the best performance pipeline is adopted with a depth dependent from the specific operator used.

For additional information about mAgic see the section: “The DSP processor: mAgic”.

The Arm interface (mAAr).

The D740 master is the ARM processor, the mAgic instead behaves as a standard AMBA ASB slave device, allowing access to different resources depending in the operating mode.

In the system mode, mAgic halts its execution and the ARM takes control of it. When mAgic is in System mode the ARM can access many mAgic internal devices. The ability of
the ARM to access internal mAgic resources in System Mode can be used for initialization and debugging purposes. By accessing the Command Register, the ARM can change the operating status of the DSP (Run/System Mode), initiate DMA transactions, force single or multiple step execution, or simply read the DSP operating status. In Run Mode, mAgic works under direct control of its own VLIW program and the ARM has access only to the 1K x 40-bit dual ported shared memory (PARM) and to the mAgic Command Register. In order to allow a tight coupling between the operations of mAgic and the ARM at run time, they can exchange synchronization signals, based on interrupts. Actually the synchronization of the processors is based on semaphores, for a complete description see the section: "Synchronization & Parallelism".
The DSP processor: mAgic

mAgic is a floating and integer point DSP based on a Modular Very Long Instruction Word (VLIW) architecture, built around a large multiport register file. It operates on IEEE 754 40-bit extended precision floating-point and 32-bit integer numeric format and produces 1Gflops - 1.5 Gops at 100 MHz in 0.18 µm Atmel CMOS technology. In a single instruction four floating point multiplications, four floating point additions and two floating point subtractions, six memory accesses and six address updates can be executed simultaneously. The DSP supports floating point paired vector and complex arithmetic domain computations.

*The DSP is designed for easy cooperation with the on-chip ARM7TDMI™ ARM® Thumb® Processor Core.*

mAgic DSP lives in two main different states, the Run Mode and the System Mode. In the System Mode, the DSP halts and the ARM controller can read and write the mAgic Local Data Memories acting as input/output buffers. Moreover in the System Mode it is possible to access internal resources for debugging purpose.

The Run Mode is the state in which the mAgic processor works under direct control of its own VLIW program. In this mode only part of the core memories (one side of PARM double port memories) are accessible by the ARM.

*The DSP is specialized to obtain good performances in complex and vectorial operations.*

As a matter of facts, the assembly instruction set is supplied with complex and vectorial instructions, as well as with single instructions (with floating point and integer numbers).

The main features of the mAgic DSP are summarized in the following:

- External Data/Program Memory access: up to two 16-Mw banks where each word is 40 bits long
- Complex, real and vector floating point arithmetic handling
- Single and vector integer arithmetic, single and vector logic & shift capabilities
- Autonomous flow control capability with dedicated instructions for loop management
- Autonomous multiple address generation capability
The mAgic DSP contains:

- A VLIW core
- Core memories
- The mAgic ARM interface (mAar)
- The external memory interface (P1XM)

The VLIW Core consists of:

- Data Register File (RF): 16 port, 512 register organized in two 4in-4out 256 reg × 40 bits. The two banks distribution ensures an optimal management in complex arithmetic
- Address Register File (ARF): 16 registers
- The arithmetic operators block (4 Multipliers, 4 adder/subtractors, 2 subtractors)
- The Multiple Address Generation Unit (MAGU)
- A Flow Control System equipped with:
  - a Condition Generation Unit
  - a set of logical operators acting on a Condition Code Stack
  - a Program Counter Generation Unit with a PC stack and a Status Unit
- The global controller which arbitrates the communication with the external memory and the ARM

The Core Memories consist of:

- On Core VLIW Program Memory: 8 Kwords × 128 bits
- On core Data Memory:
  - six 2 Kwords × 40 bits banks (P0,P1,P2 left & right) for internal computation
  - two 2 Kwords × 40 bits banks (Buffer P3 left & right) for external memory access
  - two 512 words × 40 bits banks (P4=PARM left & right) for the communication with ARM
mAgic DSP is an architecture based on a Very Long Instruction Word (VLIW) structure. The use of this structure and of a large multiport Register File allows to obtain a large bandwidth and an high computational power.

The VLIW instructions allow the control of all the DSP devices cycle by cycle. The length of these instructions is of 128 bits and they are collected in a 8 Kwords Program Memory. A compression mechanism allows to expand the capacity of this memory of a factor between 2.5 and 3, depending on the application.

The instructions are organized in fields, with an almost direct correspondence between fields and devices groups driven by them.

<table>
<thead>
<tr>
<th>DMO</th>
<th>FLOC</th>
<th>RFA</th>
<th>MAGU</th>
<th>DMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUX</td>
<td>FLOC</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 4-Vliw of mAgic

- FLOC: it is a 11-bit field containing the codes for the Flow Control and VLIW Decoder Unit
- RFA: it is a 64-bit field divided in eight sub-fields of 8 bits each one. They drive the input - output ports of the Register File, from port 0 to port 7
- MAGU: it is a 22-bit field divided in two sub-fields of 11 bits each one. They drive the Multiple Address Generation Unit, 11 bits for the Left Address Generation Unit (LAGU) and 11 bits for the Right Address Generation Unit (RAGU)
- DMO: it is a 23-bit field driving the operator block and the multiplexer in output from it.

It is divided in three sub-fields: 7 bits for the adder opcode, 7 bits for the multiplier and shifter opcode and 7 bits for the bank selection of ports 0, 1, 2, 3, 4, 5 and 6. The MuxSel sub-field drives the multiplexer in output from the operator block for the direct connection with the internal memories.

The register File is connected to the operators block trough his ports as detailed in the following figure. The output ports of each Register File bank are the inputs for the operators and the input ports are the outputs of the same operators.
In each Register File the ports are numbered as in the following description:

- output port RF0I (RF0Q): it is the first input for the multipliers (or for the shift/logic unit)
- output port RF1I (RF1Q): it is the second input for the multipliers (or for the shift/logic unit)
- output port RF2I (RF2Q): it is the first input for the adders
- output port RF3I (RF3Q): it is the second input for the adders
- input port RF4I (RF4Q): it is the output of the multiplier (or of the shift/logic unit)
- input port RF5I (RF5Q): it is the “−” output of the adder (used to store subtraction results in ADDSUB instructions)
- input port RF6I (RF6Q): it is the “+” output of the adder
- input port RF7I RF7Q: they take inputs from the memory and other peripherals

According to this description, for instance, performing an addition or a subtraction with every kind of operands (complex, real or integer) the two source registers are read from Register File port 2 and port 3, while the result is written in port 6. For a product (complex, real or integer) or a shift/div/conversion instruction the sources are read from port 0 and port 1 and the result is written in port 4.
mAgic is a VLIW engine but, from an user point of view, it works like a RISC machine, implementing triadic computing operations on data coming from the register file, and data move operations between the local memories and the register file. The operators are pipelined for maximum performance. The pipe-line depth depends on the operator used. The operations scheduling and parallelism are automatically defined and managed at compile time by the assembler-optimizer, allowing efficient code execution. To give the best support to the RISC-like programming model, mAgic is equipped with a complex 256-entry register file. It can be used as a complex register file (real and imaginary part), or as dual register file for vectorial operations. When performing single instructions the register file can be used as an ordinary 512 register file. Both the left and right side of the register file are 8-ported, making a total of 16 I/O port available for the data move to and from the operator block and the memory. The total data bandwidth between the register file and the operator block is 70 bytes per clock cycle, avoiding bottlenecks in the data flow between the two units. The operators' block, the register file, the address generation unit and the program-sequencing unit compose the core processor.

The hardware that performs arithmetical operations is contained in the Operators Block. It works on 32-bit integers and IEEE 754 extended precision 40-bit floating-point data. The Operator Block is composed of four integer/floating point multipliers: an adder, a subtractor and two add-subtract integer/floating point units. It has two shift/logic units, a Min/Max operator and two seed generators for efficient division and inverse square root computation also. The operator block is arranged to support complex arithmetic (single cycle complex multiply or multiply and add), fast FFT (single cycle butterfly computation) and vectorial computations. The mAgic peak performance is achieved during single cycle FFT butterfly execution, when it delivers 10 floating-point operations per clock cycle.

mAgic is equipped with two independent address generation units. It is able to generate up to two couple of addresses, one to access the left and right memory for reading and one to access the left and right memory for writing. It is also used in the loop control to test if the end of a loop is reached. The Multiple Address Generation Unit (MAGU) supports indexed addressing, linear addressing with stride, circular addressing and bit reversed addressing. The address generation unit is composed by 16 registers.

The Program Address Generation Unit is devoted to manage the correct Program Counter generation according to the program flow. It generates addresses for linear code execution as well as for non-sequential program flow. The Condition Generation Unit
Diopsis: A dual processor architecture combines the flags generated by the operators to produce complex conditions flags used to control the program execution. Predicated instruction execution is supported for different groups of instructions: arithmetical instructions, memory write, immediate load, or all of them. The Program Address Generation Unit allows also to perform conditioned and unconditioned branch instructions, loops, call to subroutines and return from subroutines.
Case study: Computation of the delay between two signals using cross-correlation function

Now we want to show quickly a problem that can be resolved and implemented on mAgic. The problem is estimate the delay between two signals, in this way is possible to know where the signal is generated. For example the signal is a sound and is captured by two microphones. Knowing the delay between the two signals and the sound speed is easily possible to compute the localization of the sound source.

![Figure 4-Principle of localization](image)

With simple consideration it’s possible to extend the basic algorithm to the case of locating a signal source in a three dimensional environment: in this way is possible for example to compute the epicentre of an **earthquake** or of a **tsunami**.

Many existing implementations of sound source localization often require special purpose multichannel A/D hardware which generate significant amounts of signal data and require intensive computation. Our architecture is what we need!

![Figure 5-A general schema of the application](image)
There are a lot of details to consider on the calculus of the delay (such as prefiltering, postfiltering) but we want to focus our attention on the core of the solution of this problem that is the calculus of cross-correlation.

When dealing with multiple random processes, it is important to be able to describe the relationship, if any, between the processes: in order to do this we use the cross-correlation function.

Looking at the generalized formula for the crosscorrelation, we will represent our two processes by $U=U(t)$ and $V=V(t-\tau)$. We will define the crosscorrelation function as

$$R_{uv}(t,t-\tau) = E[UV]$$

$$= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} uvf(u,v) \, dv \, du$$

**Figure 6-Formula of cross-correlation**

Just as the case with the autocorrelation function, if $U(t)$ and $V(t)$ are at least jointly wide sense stationary, then the cross-correlation does not depend on absolute time; it is just a function of the time difference. This means we can simplify our writing of the above function as

$$R_{uv}(\tau) = E[UV]$$

If we deal with two real signal sequences, $x[n]$ and $y[n]$, then we arrive at a more commonly seen formula for the discrete cross-correlation function. See the formula below:

$$R_{xy}(n,n-m) = R_{xy}(m)$$

$$\quad = \sum_{n=-\infty}^{\infty} (x[n] \, y[n-m])$$

**Figure 7-Formula of discrete cross-correlation**

Skipping all the details: once we computed the cross-correlation the delay is just the value of $\tau$ (or $m$ for discrete cases) where the cross-correlation assumes the maximum value. So the problem is resolved by a simple function like this.
int delay(__vector__ float * y1,__vector__ float * y2){
    [...]/initialization steps for the function
    for(h=1;h<=N/2;h++){ [...]/initialization steps for the loop
        for(k=0;k<(N-h);k++){ //Positive delay
            zc[h]=zc[h]+z1[k]*z2[k+h];
        }
        for(k=0;k<(N-h);k++){ //Negative delay
            zc[h]=zc[h]+z1[k+h]*z2[k];
        }
        [...]/Look for the maximum value
    }
    //return the delay with the maximum cross-correlation
    return MaxDelay;
}

Figure 8-C-code for cross-correlation

Have we finished? We have developed this simple algorithm but the performances are not good! We’ll show in the next chapter why and how we can improve the algorithm using mAgic!
An example: Profile performance and optimization of the algorithm

C-code

What we did before is just to develop a function without considering the potential enrichment due to the hardware! Is it possible to improve the algorithm? Yes, it’s and we’ll show that we could have an incredible speed-up of 50 (50! Not 50%) with simple considerations.

First of all we want to understand where optimization are necessary.

The development environment allows the user to profile a resource of mAgic. What does profiling means? Simple it is the counting of the number of cycles in which the resource assumed a specific value.

We decide to profile PMA (Program Memory Address): This profile allows a user to evaluate at a glance where the program spends the major number of cycles, and so it is possible for example to decide which routines must be optimized.

In this chapter we always simulate our function with 4 streams of 64 samples.

We start with the profile of the simplest algorithm explained above:

![Figure 9-Profile of c-code](image)
The histogram in red counts the number of times (the Y-axis value) the PMA assumes the corresponding Xaxis value for each value in the monitored range. This means that the current program has spent a greater number of cycles where the boxes are higher.

In our case we spend almost all the time in the sector of memory between 20F and 268 so we go backward and we discover that our bottleneck is the part of code corresponding to

```c
for(k=0;k<(N-h);k++){ //Positive delay
    zc[h]=zc[h]+z1[k]*z2[k+h];
}
for(k=0;k<(N-h);k++){ //Negative delay
    zc[h]=zc[h]+z1[k+h]*z2[k];
}
```

And this is the correlated assembler code for one cycle:

```assembly
........ RF2A 2 508 MEM2RF 8 2 MEM2RF 6 2
LABEL .L15 RF2A_P 2 0x1FD MEM2RF 6 3 FADD 6 6 7
RF2A 3 508 IMM2A 2 A=5 L=0 M=1 IADD 6 8 6 RF2L 2 6
RF2A_P 3 0x1FD MEM2RF 6 2 DIMM2A 2 S1= p0.z2 RF2A 3 508
IMM2A 3 A=5 L=0 M=1 DIMM2A 4 S1= p0.z2 A1=0 L1=0 M1=1 P1=0 RF2A_P 3 0x1FD
RF2A 2 508 A1=0 L1=0 M1=1 P1=0 S2=0 A2=0 L2=0 DIMM2A 3 A=5 L=0 M=1
RF2A_P 2 0x1FD 0 S2=0 A2=0 L2=0 RF2A 2 508
IMM2A 2 A=6 L=0 M=1 M2=0 P2=0 SH 2 6 494 RF2A_P 2 0x1FD
MEM2RF 8 2 SH 2 6 494 RF2A_A 2 2 IMM2A 2 A=5 L=0 M=1
DIMM2RF 6 16 1 0 RF2A_A 4 2 MEM2RF 8 4 MEM2RF 6 2
IADD 7 6 -8 RF2A 2 508 MEM2RF 6 2 IADD 6 6 502
MEM2RF 6 3 RF2A_P 2 0x1FD FMUL 7 8 6 RF2L 3 6
IGE 6 7 IMM2A 2 A=5 L=0 M=1 DIMM2A 2 S1= p0.sum BR .L15
BRIF_NOT .L18 RF2A 3 508 L1=0 A1=0 M1=1 P1=0 LABEL .L16
BR .L16 RF2A_P 3 0x1FD 0 S2=0 L2=0 A2=0.....
LABEL .L18 IMM2A 3 A=6 L=0 M=1 M2=0 P2=0
```

Figure 10-Inner loop

Figure 11-Assembler code of the inner loop
This code is a disaster! Indeed:

- It doesn’t use special operation provided by mAgic;
- It loses all the time to access in memory to load the values, in particular I take the pointer to the vectors in memory.

The result is that our vliw scheduler is not able to find much parallelism.

---

**Assembler-code**

After this simple but important consideration we decide to optimize that part of the code writing assembler code first of all we decide to take significant values in the registers, in this way we gain a lot of time! Next we decide to use special instructions provided by the ISA of mAgic, in particular we use VFMUL(to multiply), VMS2RF_U(to load the values from memory), CADD(to add).

These instructions are all vectorial in other words they compute two values. For example:

\[
\text{CADD(destA,sourceA,sourceB)} \quad \text{means} \quad \text{destA}[L]=\text{sourceA}[L]+\text{sourceB}[L];\text{destA}[R]=\text{sourceA}[R]+\text{sourceB}[R];
\]

Moreover the special instruction VMS2RF_U computes the next value to load. For example:

\[
\text{VMS2RF_U(dest,point)} \quad \text{means} \quad \text{dest}[L]=\text{point}[L];\text{dest}[R]=\text{point}[R];\text{point}++; \quad \text{Last but not least we decide to compute more than one sum in each clock using values already load from memory.}
\]
Our new code is:

```c
#define VFMUL(destA,sourceA,sourceB) asm volatile("VFMUL %0 %1 %2":="j" (destA):"r" (sourceA),"r" (sourceB))
#define VMS2RF_U(destA,sourceA) asm volatile("VMS2RF_U %0 %1":="j" (destA):"r" (sourceA))
#define CADD(destA,sourceA,sourceB) asm volatile("CADD %0 %1 %2":="j" (destA):"r" (sourceA),"r" (sourceB))

//registers of type ARF that point at arrays
register __vector__ float * punty1 asm("520");
register __vector__ float * punty2 asm("521");

//registers that contain the first values of the arrays
register __vector__ float valy1 asm("20");
register __vector__ float valy2 asm("22");

int delay(__vector__ float * y1,__vector__ float * y2){
    [...]
    for(k=0;k<(M-h);k++){
        //read the values
        VMS2RF_U(valy1,punty1);
        VMS2RF_U(valy2,punty2);
        VMS2RF_U(valy1i,punty1i);
        VMS2RF_U(valy2i,punty2i);
        VMS2RF_U(valy1h,punty1h);
        VMS2RF_U(valy2h,punty2h);

        //multiply them
        VFMUL(muly1y2i,valy1,valy2i);
        VFMUL(muly1y2h,valy1,valy2h);
    }
    [...]
}
```
VFMUL(muly2y1i,valy2,valy1i);
VFMUL(muly2y1h,valy2,valy1h);

//adding to temporary sumxxxx variable
CADD(sumy1y2i,sumy1y2i,muly1y2i);
CADD(sumy1y2h,sumy1y2h,muly1y2h);
CADD(sumy2y1i,sumy2y1i,muly2y1i);
CADD(sumy2y1h,sumy2y1h,muly2y1h);

}
Let's look now the new profile:

![Figure 15-Profile of Assembler-code](image)

Now is possible to compare the above figure (that shows the profile of our function with assembler code) with the Figure 9-Profile of c-code (the profile of the original function). The gain is already visible: indeed the new area of the histogram is smaller and in the end of this chapter we'll show this fact with an accurate numerical analysis.

The next figure shows how vliw are now scheduled.

![Figure 16-VLIW scheduling of Assembler-code](image)
Now is possible to compare the above figure with the Figure 12-VLIW scheduling of c-code(vliw scheduling of the original function). The new vliw scheduling is denser than the previous one, this implies that we have more parallelism.

**Assembler-code with software pipelining**

This schedule is denser than the previous one but we have again a lot of free space. We decide to pipeline our code because we have a lot of data dependencies inside each iteration and this give us less parallelism!

Let's look now the profile of the code with software pipelining

![Figure 17-Profile of Assembler-code with sw-pipelining](image)

The new loop is faster than the previous one. Indeed is possible to compare the area of this histogram with the area of the histogram in Figure 15-Profile of Assembler-code: The height is the same but now we have a narrower region, this implies that the new code is faster!

Why? Simply because we have more parallelism with this code. Indeed, as we can see from how vliw are scheduled now, some instructions now are overlapped
A dual processor architecture

Figure 18-VLIW scheduling of Assembler-code with sw-pipelining

Again: the new vliw scheduling is denser than the previous one because with software-pipelining we have reduced data-dependencies in the same iteration.

Assembler-code unrolled

What we can do now? The simple idea is to unroll the loop (in our case 4 times) to gain more parallelism and to reduce the overhead of the loop.

This is simple but give us a lot of advantages infact look now at the new scheduling:

Figure 19-VLIW scheduling of Assembler-code with unrolling

This vliw scheduling is denser than the previous one because with unrolling we have a bigger Basic Block (that is a portion of code with single entry and single exit point) and the compiler can find more parallelism in the code. Moreover we reduce the overhead in a single iteration due to the control of the loop.
And this is the profile of the unrolled code:

![Graph showing the profile of unrolled code](image)

Figure 20-Profile of Assembler-code with unrolling

The area of the new histogram is more little but is larger than the histogram in the Figure 17-Profile of Assembler-code with sw-pipelining so this time is not immediate to view the speed-up. For this reason a concise numerical analysis is provided at the end of this chapter.

**Assembler-code with while-loop**

What we can do now? We could see from the graph that we have a strange peak at the beginning of the loop, what is it?

Let's look the assembler code

```
LABEL .L33
DIMM2RF 6 32 1 0
IADD 6 6 -52
IGE 51 6
BRIF_NOT .L36
BR .L34
```
Diopsis: A dual processor architecture

\[\text{LABEL .L36}\]

[...]

\[\text{DIMM2RF 0 0 7 4}\]

\[\text{IADD 51 51 7}\]

\[\text{BR .L33}\]

\[\text{LABEL .L34}\]

[......]

Figure 21-How is compiled a "for"

This assembler code is very brainless!! The compiler translate the "for" in a strange way! Look at his flow chart!

![Flow chart of the for](image)

So we decide to replace the "for" with a "do-while". Look how simpler is now the assembler code:

\[\text{LABEL .L33}\]

\[\text{DIMM2RF 0 0 7 4}\]

\[\text{IADD 51 51 7}\]

[......]

\[\text{DIMM2RF 6 32 1 0}\]

\[\text{IADD 6 6 -52}\]

\[\text{IGE 51 6}\]

\[\text{BRIF_NOT .L33}\]

Figure 22-Flow chart of the for

Figure 23-How is compiled a "do-while"
Look now the new profile:

Figure 24-VLIW scheduling of Assembler-code with while
Comparing the above figure with the Figure 20-Profile of Assembler-code with unrolling we can imply that the density (from an intuitive point of view) of the vliw is not changed and the parallelism is the same. So where is the improvement of the code?

look now at the new profile:

Figure 25-Profile of Assembler-code with while
It’s possible to compare the above figure with Figure 20-Profile of Assembler-code with unrolling: now we have eliminated the “noisy” peak at the beginning of each iteration of the loop so we have decrease the overhead due to the control of the loop.
**Diopsis: A dual processor architecture**

**Analysis**

With a spread-sheet we decide to make a precise numerical analysis of the profiling.

First of all we could compute the theoretical speed-up of the loop. This is simple, it’s enough to count the number of cycles for one loop looking how vliw are scheduled. We have to multiply it for a correcting factor(as example we know that a unrolled loop for each iteration computes 4 times a normal loop). And we collect everything in a table:

Legend:
- U+S+A+V: Code unrolled+software pipelining+assembler+while
- U+S+A+F: Code unrolled+software pipelining+assembler+for
- S+F+A: software pipelining+assembler+for
- A: assembler
- C: c code

<table>
<thead>
<tr>
<th></th>
<th>U+S+A+V</th>
<th>U+S+A+F</th>
<th>S+F+A</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles number</td>
<td>31</td>
<td>51</td>
<td>29</td>
<td>34</td>
<td>93</td>
</tr>
<tr>
<td>Correction factor</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Cycles number*Correction factor</td>
<td>31</td>
<td>51</td>
<td>116</td>
<td>136</td>
<td>1488</td>
</tr>
<tr>
<td>U+S+A+V</td>
<td>31</td>
<td>1,645161</td>
<td>3,741935</td>
<td>4,387097</td>
<td><strong>48</strong></td>
</tr>
<tr>
<td>U+S+A+F</td>
<td>51</td>
<td>2,27451</td>
<td>2,666667</td>
<td><strong>29,17647</strong></td>
<td></td>
</tr>
<tr>
<td>S+F+A</td>
<td>116</td>
<td>1,172414</td>
<td>18,76931</td>
<td><strong>12,82759</strong></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>136</td>
<td></td>
<td>14,80448</td>
<td><strong>10,94118</strong></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1488</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 26-Speed-up of the inner loop**

We could see that we have a theoretical speed-up of 48

We show now the real speed-up computed using the profile informations

<table>
<thead>
<tr>
<th></th>
<th>U+S+A+V</th>
<th>U+S+A+F</th>
<th>S+F+A</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum of profiled accesses</td>
<td>19680</td>
<td>31104</td>
<td>58608</td>
<td>74304</td>
<td>1100032</td>
</tr>
<tr>
<td>U+S+A+V</td>
<td>19680</td>
<td>1,580488</td>
<td>2,978049</td>
<td>3,77561</td>
<td><strong>55,89593</strong></td>
</tr>
<tr>
<td>U+S+A+F</td>
<td>31104</td>
<td>1,884259</td>
<td>2,388889</td>
<td><strong>35,36626</strong></td>
<td></td>
</tr>
<tr>
<td>S+F+A</td>
<td>58608</td>
<td>1,267813</td>
<td>18,76931</td>
<td><strong>14,80448</strong></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>74304</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1100032</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 27-Speed-up of the inner loop**
The results are similar!
Look that the most significant improvement are due to the unroll and to writing assembler code.
Until now we just consider only the inner loop but what about the entire function? This is the speed-up of the function computed using the profile informations

<table>
<thead>
<tr>
<th></th>
<th>U+S+A+V</th>
<th>U+S+A+F</th>
<th>S+F+A</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum of profiled accesses</td>
<td>47615</td>
<td>57743</td>
<td>87599</td>
<td>100830</td>
<td>1121572</td>
</tr>
<tr>
<td>U+S+A+V</td>
<td>47615</td>
<td>1,212706</td>
<td>1,839735</td>
<td>2,11761</td>
<td>23,55501</td>
</tr>
<tr>
<td>U+S+A+F</td>
<td>57743</td>
<td>1,51705</td>
<td>1,746186</td>
<td>19,42351</td>
<td></td>
</tr>
<tr>
<td>S+F+A</td>
<td>87599</td>
<td></td>
<td>1,151041</td>
<td>12,80348</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>100830</td>
<td></td>
<td></td>
<td>11,1234</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1121572</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 28-Speed-up of the function

This is just to explain that a optimization is more important when it affects the greatest part of the computation.
Look how is important the loop in the different cases!!

<table>
<thead>
<tr>
<th></th>
<th>total cycles of the loop</th>
<th>total cycles of the function</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+S+A+V</td>
<td>19680</td>
<td>47615</td>
<td>41,33</td>
</tr>
<tr>
<td>U+S+A+F</td>
<td>31104</td>
<td>57743</td>
<td>53,86</td>
</tr>
<tr>
<td>S+F+A</td>
<td>58608</td>
<td>87599</td>
<td>66,90</td>
</tr>
<tr>
<td>A</td>
<td>74304</td>
<td>100830</td>
<td>73,69</td>
</tr>
<tr>
<td>C</td>
<td>1100032</td>
<td>1121572</td>
<td>98,07</td>
</tr>
</tbody>
</table>

Figure 29-Percentage of time spent in the inner loop

Is it necessary every time optimize our code? No, the simple idea of Atmel is to give at developers a simple well-known high-level language as C. All the common intensive computation for a DSP application(such as fft, conv) are provided by a library with all the code optimized for mAgic
Synchronization & Parallelism

The two major features in Diopsis 740 are parallelism and a dedicated processor for heavy computation, we have chosen our project in order to exploit and experiment both this features.
The cross-correlation indeed is a heavy computational problem and for sure it’s an appropriate work to test the capabilities of mAgic.
On the other hand generate the signals, format the input and output and finally coordinate the all work is a job for the Arm processor.
The synchronization become consequently a key factor for the success of the program.
In other words Arm’s job is the generation of two collection of samples that represent two not correlated signals arrived for example from ADDA.
To make the things more tricky and the signals more real we add to each of them a different noise to stress the capability of our implementation of cross correlation routine.
After generating these branch of signals the ARM have to pass them to the mAgic for the real computation of the delay.
The passage of these data is accomplished exploiting the shared memory between ARM and mAgic, so the Arm fill a buffer with the two collection of samples and then the mAgic can compute the result which is returned to the Arm using once again the shared memory.
However to exploit the parallelism of the architecture is necessary to use a second buffer, in fact during the computation of the mAgic the Arm is basically idle if we use only a buffer.
Instead with two buffer when mAgic is working the Arm can begin to fill the second buffer with other two branch of signals for the next computation.
The overall mechanism is coordinated by two semaphores, located obviously again in the shared memory, that show the Arm processor if the mAgic is working on a specific buffer or if this buffer is ready to be filled by the Arm because the DSP has yet used the data that are inside.
We decided to use semaphores instead interrupts in order to develop a stand alone solution without any operating system. This choice guarantee a simpler and lighter application: characteristics very useful in a testing application like the ours is.
A useful optimization of this solution is rendering the size of the buffers dynamic depending on an evaluation of the time spent from the Arm polling the semaphore and waiting for mAgic. Indeed if the Arm waste too much time polling the semaphore it is possible to reduce the size of the buffers making the mAgic work easier.

On the other hand if the Arm never polls the semaphore it’s possible to increment the size of buffers in order to give more job to the mAgic in a single cycle of filling by the Arm.

In this particular application however this solution is not suitable because with all the optimizations done on the DSP code of the correlation the bottleneck is became paradoxically the work of the Arm that fills the buffers.

So increasing or decreasing the buffers modifies the size of the work of mAgic but also the amount of work of Arm and so it’s no useful to reduce the buffers in order to make faster the Arm’s job because also the dsp’s job will be faster coming back to the initial situation.
Furthermore, like we said, the bottleneck is now the work of Arm and is self-defeating reducing further a limit the size of buffers because if the buffers are too small the overall performance of the system decreases.
So in this particular case is sufficient to use static buffers instead of dynamic, but this is a particularity of the specific application.
Finally it’s useful to note that in this application we have exploited all the advantages of a dual processor system like the architecture of Diopsis 740 is.
Indeed we used parallelism making a parallel computation between Arm and mAgic and in particular we used the powerful mathematical skills of the DSP for the more complex part of our program: the cross correlation computation.
The code of the main cycle of Arm is then the following (some code not really needed to understand the behaviour of the program was cut to enhance the readability):

```c
//Initially is necessary to fill the two buffers before starting mAgic
load_buffer();
load_buffer();

//let’s Start mAgic
RUNMAGIC;

//An unbounded loop is the main cycle of the program
while(1)
{
    //Waiting to fill a free buffer, this is a blocking procedure!
polling_semaphore();

    //Let’s print the result derived from the computation on the data that
    //were on the buffer.
    printf("\nResult catched is : %d\n", (int) catch_result());

    //The buffer can be now filled, which buffer have to be filled is checked by
    //the procedure itself
    load_buffer();
}
```
Instead the code of the main cycle of mAgic is trivially the following (some code not really needed to understand the behaviour of the program was cut to enhance the readability):

//An unbounded loop is the main cycle of the program
while(1)
{
    //These two procedures take the data from the right buffer and convert it in the
    //format needed for the computation of cross-correlation
    prepare_data();
    prepare2_data();

    //Depending on which buffer give us the data we decide where put the results of
    //our computation. After that we must remember to free the semaphore!
    if(!switch_buffer)
    {
        p_R1 = (int) delay(x1, x2, x1i, x2i);
        p_S1 = 0;
        switch_buffer = 1;
    }
    else
    {
        p_R2 = (int) delay(x1, x2, x1i, x2i);
        p_S2 = 0;
        switch_buffer = 0;
    }
}
Figure 31-A diagram of the application
The Operating Systems: eCos vs. mArmOS.

A typical Diopsis application can exploit the services of one of the two following operating systems:

eCos
mArmOS

This section is so dedicated to a brief description of these OS.
eCos, which is an acronym for Embedded Configurable Operating System, is very useful in the development of complex applications like applications that run in a multithreading fashion.

In origin eCos, released in 1998, was created to build a simple but effective solution in terms of cost and quality for embedded system marketplace.
The model adopted to create this OS was the Open source model, so eCos is actually available with no costs, royalty free.
Moreover developers have full access to entire source code in order to customize the OS and to achieve a better satisfaction of all the possible needs.
Furthermore the presence of a complete open source development and debug environment is included in eCos system.
Features of eCos:
- Fully configurable.
- Implemented in C++.
- A choice of scheduler implementations.
- Interrupt and exception handling.
- Thread and synchronization support.
- Timers, counters and alarms.

The second OS is mArmOS which is an acronym for mAgic Arm Operating System. Basically mArmOS is a collection of inizalization routines and macros.

In origin mArmOS was used stand alone to develop simple ARM programs and to test the harware, it’s very fast and small, but it doesn’t support multithreading, libc and memory allocation.

Consequently mArmOS is very useful for simple but critical applications that have hard real time constraints. A brief summary of all characteristics of these OS is here following:

<table>
<thead>
<tr>
<th>Features</th>
<th>Avail. on eCos</th>
<th>Avail. on mArmOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highly configurable.</td>
<td>Yes</td>
<td>Nothing to configure!!!</td>
</tr>
<tr>
<td>Open source.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Supports ISO C.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Supports math library.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>µITRON POSIX compatible.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handling.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for the peripherals.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Multithread.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Footprint.</td>
<td>Optimized</td>
<td>Minimal</td>
</tr>
<tr>
<td>Very hard real time.</td>
<td>Not always</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Development tools.

**Jtst (JigTeST for D740): a development board.**

JTST is a stand-alone, general-purpose module that provides the appropriate resources in order to evaluate DIOPSIS 740 DSP performances in a wide range of applications.

![Figure 33-A view of Jtst](image)

JTST provides the following resources to DIOPSIS 740:

- **Memories:** mAgic SSRAM, ARM FLASH and ARM SRAM
- **Stereo Audio CODECs (4 in + 4 out)**
- **Serial I/O:**
  - 1 USB 2.0 Full (12 Mbps)
  - 2 RS232/LVTTL asynchronous/synchronous serial I/O lines
  - 2 SPI serial I/O lines
- **Reset Logic (Power ON, Push Button, WDG)**
Diopsis: A dual processor architecture

- PLL - Clock Logic
- Configuration DIP SWITCH & Status 7-segment Display
- Voltage Regulators 5V/3.3V & 5V/1.8V
- Connectors (USART, SPI, USB, PIO, AUDIO, JTAG-ICE, EXT CLK, PSU)

In our project we didn’t use the Jtst but we only emulated the behaviour of the dual processor system on Made (described in the following section) so for a more precise description and analysis of Jtst see the Atmel official documentation.

**Made: Multicore Application Development Environment.**

To develop our project we used the powerful environment Made which is an integrated development environment (IDE) that can be used to develop either simple Diopsis™ 740 applications or complex applications. It includes the C compilers for both ARM and mAgic DSP.

Basically Made is based on GNU compiling tools (GCC), an high-level mAgic DSP macro-assembler/optimizer, an editor tool (Scite) with syntax highlighting and a unified debugging environment interfacing with a cycle accurate simulator or with a Diopsis board, through the GNU debugger tool (GDB).

Unfortunately Made is still a beta version and it still have some deficiency that we came up and overtook during the development of our work.

The editor Scite works very well and is syntax highlighting feature is very useful to discover every syntax error.

The problems arose using the output console of Made, indeed it isn’t very intuitive to use. To give an example the scrolling bar function on the console is actually unusable.

Another problem which arose using the console is the length that the output must have. Once the output length has reach this limit the console is stuck and to continue to work on the project is necessary to restart all Made !.

Probably a simple solution exists to solve this little problem, but actually it’s so unintuitive that we didn’t find it.

We encountered some weird problems and error messages also in the management of the projects and of the files related which is sometimes too much complicated but this is probably due to the complexity of the emulation that Made have to do without the Jtst.
A curious particular of Made is the warning massage that often appear at the first compilation of a correct program that immediately disappear at every successive compilation of the source code.

The last comment that we have to do to Made is the impossibility to observe the evolution of the memory in the Array View panel during the execution of a program (in a Cycle Accurate simulation) except for the page 0. Another time it a method exists to monitor this part of the memory, it’s absolutely unintuitive and so unusable for the programmer. Over this flaws Made is a promising environment indeed it’ currently only a beta version and therefore some errors are unavoidable and moreover there are some features that we have really appreciated in Made.

One of them is for sure the possibility to use Scite in a separate window from that of Made in order to write code in a full screen fashion and in order to have a bigger console output.
Conclusion

With this project not only we had implemented a simple application but also we developed a general method to solve this kind of problem with a cooperative system like diopsis. Indeed our real purpose is to try to understand how to design an application on this class of architectures.

We found a lot of new clashing problematic as:
- Less communication VS more cooperation;
- Good balanced loading VS Simplicity;
- Performance code VS effortlessness code;

Maybe we don’t have the base-knowledge to tackle this type of problems but we hope to take advantage from our ingenuity.

For example we need to design a simple way to allow a efficient communication between the two processors: not only with the usage of two buffers we achieved this goal but also we discovered an alternative design that is simpler than the one proposed by Atmel.

With cross-correlation as a general dsp-problem we showed the capabilities of mAgic optimizing the code we showed the features of a general VLIW processor. We illustrated that it’s achievable to get a speed-up of 50.

This project show us a first real powerful multiprocessor architecture so we are very glad to have worked on this topic. Moreover our initial goals are been achieved and we hope to have learned something useful and maybe other people can in the future take advantages from our experience.
References

- Datasheet Short(7001s.pdf)
- Datasheet Complete(doc7001.pdf)
- mAgic Reference Manual(doc7002.pdf)
- JTST User Manual(doc7003.pdf)
- MADE User Manual(doc7004.pdf)
- MCC User Manual(doc7005.pdf)
- mAgic Binary Tools User Manual(doc7006.pdf)
- DSP Library User Manual(doc7007.pdf)
- OS User Manual(doc7008.pdf)
- mAgic Assembler and Preprocessor Directives User Manual (doc7009.pdf)
- eCos training(DiopsisEcos.ppt)
- JTST(jtst.ppt)
- Overview on diopsis(RN_00_diopsis740_Overview.ppt)
- Overview on ARM(RN_01_diopsis740_ARM_System.ppt)
- Overview on eCos(RN_02_diopsis740_eCos.ppt)
- Overview on mArmOS(RN_03_diopsis740_mArmOS.ppt)
- Tutorial on diopsis (RN_04_diopsis740_SwitchToTarget.ppt)
- Overview on mAgic(RN_01_diopsis740_mAgicProgramming.ppt)
- Active Stereo Sound Localization(GreidPaper.pdf)
- “A structural model for binaural sound synthesis”, Brown, Duda, IEEE Transactions on speech and audio processing.
- “Techniques for low cost spatial audio”, Burgess, Graphics Visualization and Usability Center - Multimedia Group Georgia Institute of Technology.
- “Discriminative binaural sound localization”, Reuven & Singer, School of Computer Science & Engineering, The Hebrew University, Jerusalem 91904, Israel.
- “Robust Sound Source Localization Using a Microphone Array”, Valin, Michaud, Rouat, & Letourneau, Department of Computer Engineering Università de Sherbrooke

46