High Performance Processors and Systems:
Homework 2005/2006.

DIOPSIS 740: A dual processors architecture.

Autor: Michele Santoro

Revisor:
Marco Domenico Santambrogio

Data: 10/06/2006

Document spreading

This document is totally free.

Figure index and Table index

Figure 1 - Diopsis 740 .... **Error! Bookmark not defined.**
Figure 2 - mAgic operational block .... **Error! Bookmark not defined.**
Figure 3 - VLIW instruction .... **Error! Bookmark not defined.**
Figure 4 - ARM and mAgic interconnections .... **Error! Bookmark not defined.**
Figure 5 - MADE interface .... **Error! Bookmark not defined.**
Figure 6 - mAgic disassembler .... **Error! Bookmark not defined.**
Figure 7 - PMA profile histogram .... **Error! Bookmark not defined.**
Figure 8 - Input signal (red) and filtered signal (blue) .... **Error! Bookmark not defined.**
Figure 9 - Two sample signals .... **Error! Bookmark not defined.**
Figure 10 - The result of convolution .... **Error! Bookmark not defined.**
Figure 11 - PMA profile of C-code .... **Error! Bookmark not defined.**
Figure 12 - VLIW scheduler of C-code .... **Error! Bookmark not defined.**
Figure 13 - PMA profile fully unrolled .... **Error! Bookmark not defined.**
Figure 14 - VLIW scheduler fully unrolled .... **Error! Bookmark not defined.**
Figure 15 - PMA profile without register dependencies .... **Error! Bookmark not defined.**
Figure 16 - VLIW scheduler without register dependencies .... **Error! Bookmark not defined.**
Figure 17 - PMA profile unrolled (8) + software pipelining .... **Error! Bookmark not defined.**
Figure 18 - VLIW scheduler unrolled (8) + software pipelining .... **Error! Bookmark not defined.**
Figure 19 - PMA profile of whole mAgic code .... **Error! Bookmark not defined.**
Figure 20 - VLIW scheduler of whole mAgic code .... **Error! Bookmark not defined.**

Table 1 - Asm code fully unrolled *19*
Table 2 - Example of register dependencies elimination *21*
Table 3 - Asm code without register dependencies *21*
Table 4 - Optimization steps: number of cycles *27*
Table 5 - Optimization steps: memory occupation *28*
Table 6 - Optimization steps: wrt theoretical number of cycles *28*

Index

Document spreading *2*
Figure index and Table index *3*
Index *4*
Introduction *5*
Atmel Corporation and Diopsis 740 .... *6*
A dual processor system. **6**
mAgic **6**
ARM **8**
MADE **10**
Case study: Computation of the FIR (Finite Impulse Response) *14*
Profile performance and optimization of the algorithm *17*
C-code **17**
Assembler-code with loop fully unrolled. **19**
Assembler-code with register dependencies removed **20**
Assembler-code unrolled (8) and software pipelining **23**
Assembler-code of whole mAgic code **24**
Analysis **27**
Conclusion *29*
References *30*

Introduction

The goal of this project is to better understand the architecture of the DIOPSIS 740 dual core processor made by the Atmel Corporation, and to focus the attention in optimizing the code to fully exploit its power.

Basically, this project consists in a performance analysis, which shows to improvements due to the use of the optimizing techniques.

The first step was to read a huge amount of documentation about the DIOPSIS 740 architecture: how does it work, how the two processors are coordinated and synchronized, how it's possible to share date among them, and how to do the right measurements.

The optimization follows the standard methodology also proposed by Atmel, exploits the particular architecture of the mAgic processor (DSP) allowing the parallelization of the code execution.

This document is structured in the following way:

- A brief overview on Atmel Corporation
- Introduction of the DIOPSIS 740 architecture, and in particular of mAgic
- Presentation of the Development Environment MADE and its tools
- Case of study: FIR (Finite Impulse Response) and optimizations

The need of a fast and efficient computation it's the point that leads to use Atmel dual-core processor in this project, in fact, thanks to the cooperation of the two processors, it's possible to achieve an high number of operations (1GFLOP - 1.5 Gops.).

Atmel Corporation and Diopsis 740

Atmel Corporation is an International Firm leader in designing and development of semiconductors; it's main field is the development of microprocessors, non-volatile memory, logic components, and radio frequencies sensors.

Diopsis 740 is an acronym for Dual Inter Operating Processor for Silicon Systems. It's composed of two processors: 1 ARM processor and 1 DSP processor which work in parallel to offer the best performance in audio applications, or communication and beam-forming applications.

A dual processor system.

In details, Diopsis 740 it's the union of a microcontroller (32-bit RICS ARM7TDMI) and a DSP (40-bit floating point VLIW, ATMEL mAgic Digital Signal Processing).

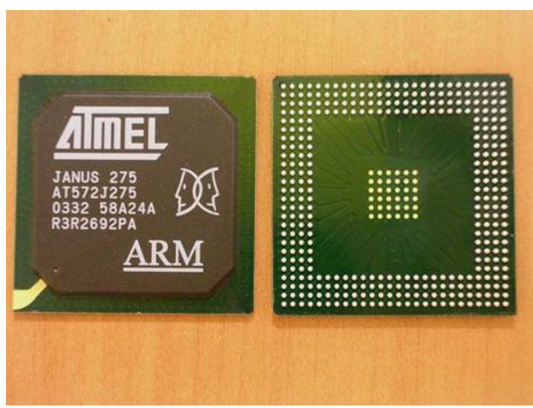


Figure 1 - Diopsis 740

Diopsis 740 couple, hence, the technology RISC of its ARM processor and the computational power of the DSP mAgic, so to give the final user an integrated platform useful for ultrasound applications, language and audio synthesis, radar, sonar applications, and so on...

mAgic

The most important component of the architecture is the mAgic processor; it can perform floating point and integer computation, but it's specialized in computations and operations

on complex numbers. It's based on VLIW architecture (Very Long Instruction Word), and 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. Due to it's peculiar architecture, it's possible in a single instruction to execute in the same time up to 10 operations: 4 floating point multiplications, 4 floating point adders, 2 shift/logic units and seed generators.

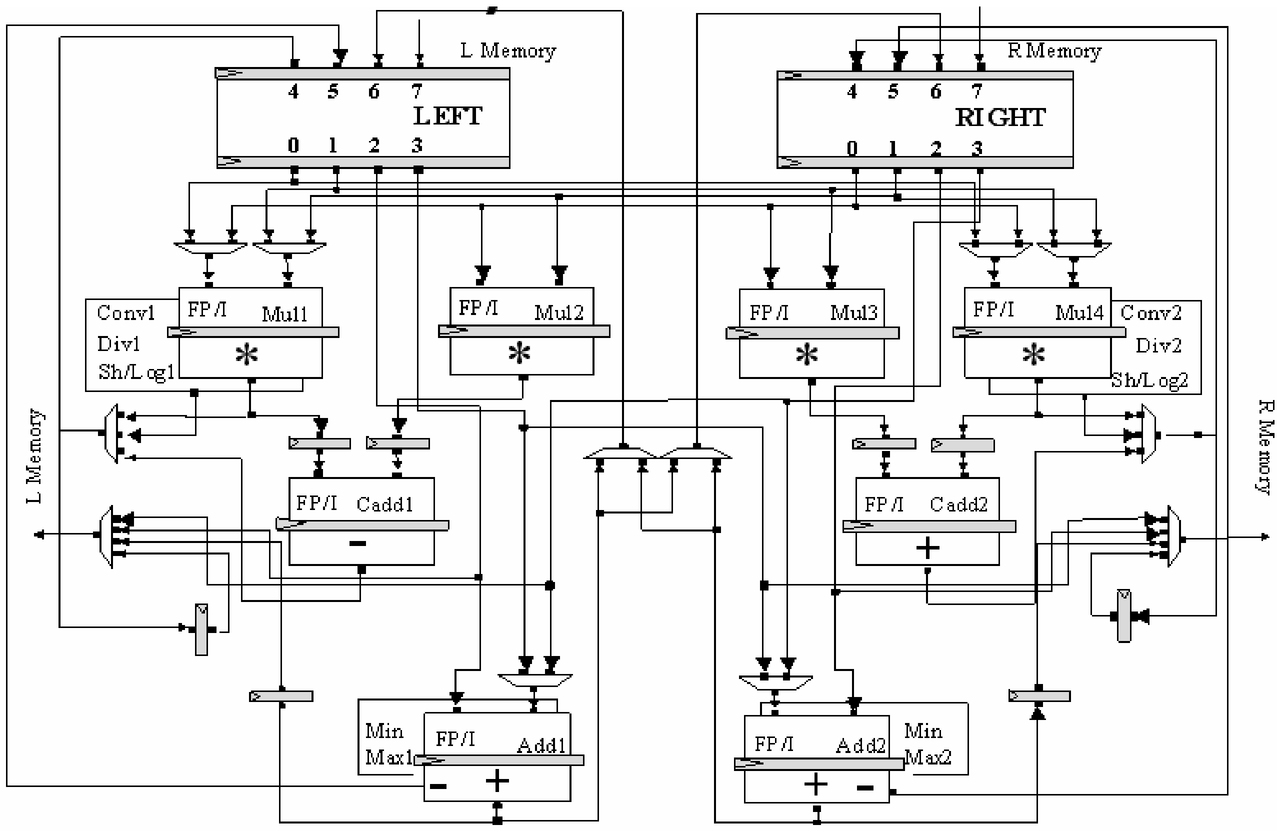


Figure 2 - mAgic operational block

VLIW instruction set allows controlling all the logical components cycle-by-cycle. The length of VLIW instructions is 128 bit, split in 8 parts. Moreover there is a compression mechanism that allows reducing the memory occupation of the code.

DMO	FLOC	RFA								MAGU		DMO		
MUX	FLOC	7	6	5	4	3	2	1	0	LAGU	RAGU	ADD	MUL	RFIO
2	11	8	8	8	8	8	8	8	8	11	11	7	7	7

Figure 3 - VLIW instruction

- 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.

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.

ARM

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.

The ARM can interface and manage all the peripherals needed by applications:

- ADDA: (A/D and D/A converter) interface supporting up to 4 Analog to Digital and Digital to Analog, stereo 24-bit converters;
- CLKGEN: 8 channel clock generator, allows to generate programmable clock signal

- PIO: Parallel I/O Controller, features 32 programmable I/O lines, 28 PIO lines are available on D740 pads, while the remaining 4 are only internal
- EBI: External Bus Interface, generates the signals that controls the access to the External Memory or peripherals devices.
- WD (WatchDog Timer): the WD can be used to guard against system lock-up is the software becomes trapped in a deadlock

The communication between ARM and mAgic is through mAAr (mAgic Arm Interface). Synchronism is achieved using interrupts generated by the Arm or the mAgic. Moreover, Arm and mAgic can share data using the PARM, the shared memory, structured in 2 blocks (left and right) of 512 words of 40-bit.

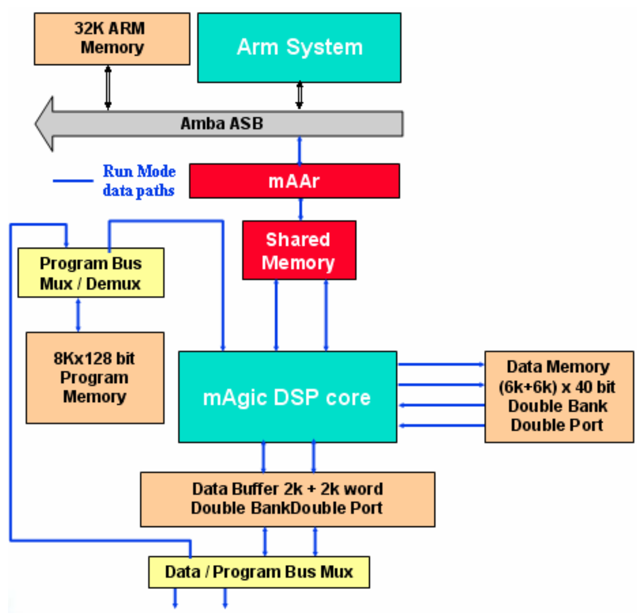


Figure 4 - ARM and mAgic interconnections

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.

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

The ARM processor moreover operates in little-endian mode.

MADE

MADE (Multicore Application Development Environment) is an Integrated Development Environment for developing Diopsis 740 applications: it's a graphical environment for creating and modifying source code in C or in assembly and, in the same time, it's a tool set for simulating and debugging code, or even to value the performance, in particular of the mAgic processor. In fact MADE can simulate with high accuracy the number of cycles needed for the application execution.

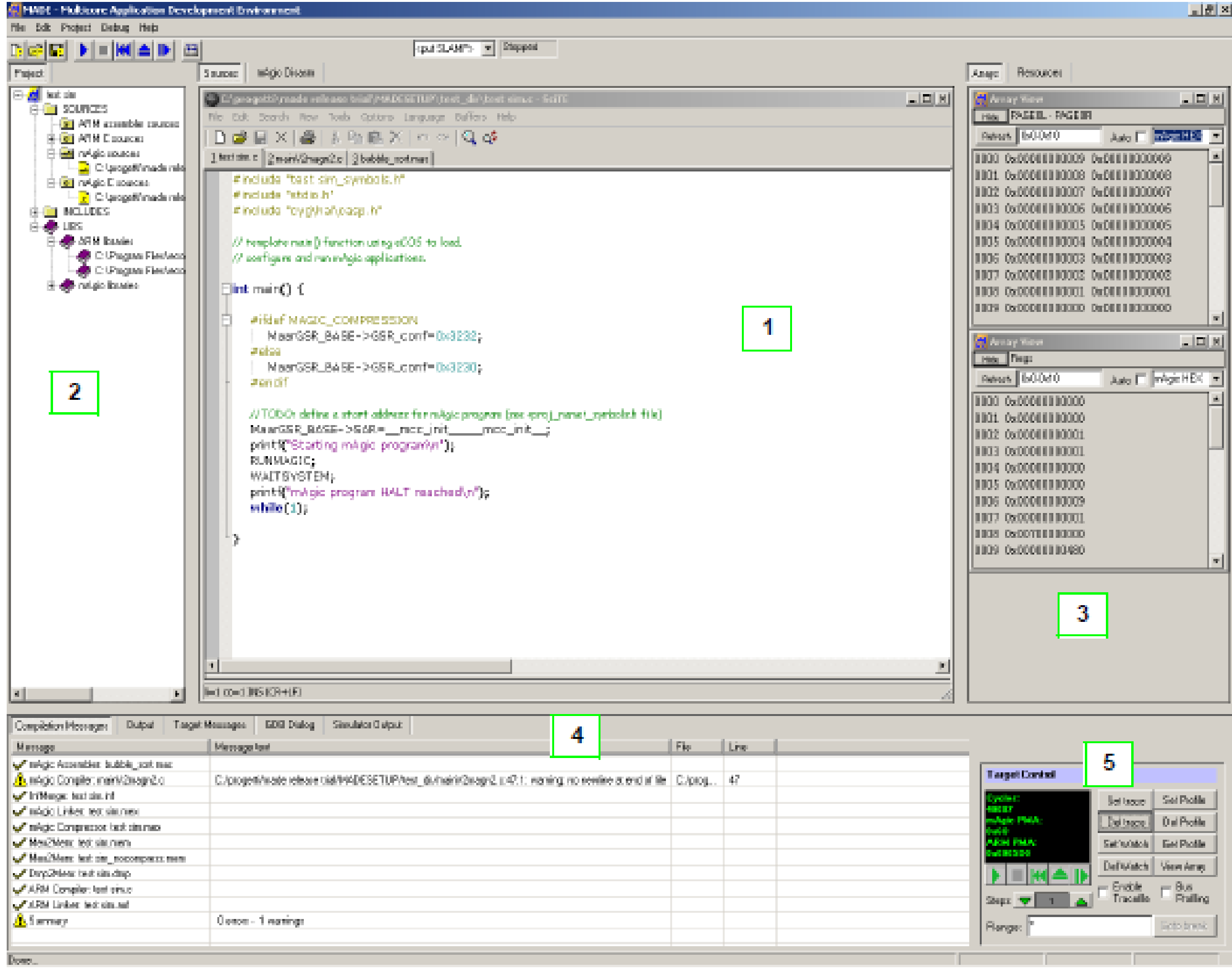


Figure 5 - MADE interface

MADE graphic interface is split in five main panels:

1) Editor (Scite) compatible with many programming language; mAgic Disassembler
2) Project Tree
3) Register Arrays which contein allocated resources

4) Output and Debug panel
5) Panel that allows to control the execution of the program

MADE has all the tools to develope, in a professional fashion, applications well-suited by Diopsis 740 architecture. Very important tools in MADE are the *mAgic disassembler* and the *Resource Profile Diagram.*

- The mAgic disassembler is an advanced tool that allows to check up and select the assembly code built by mAgic.

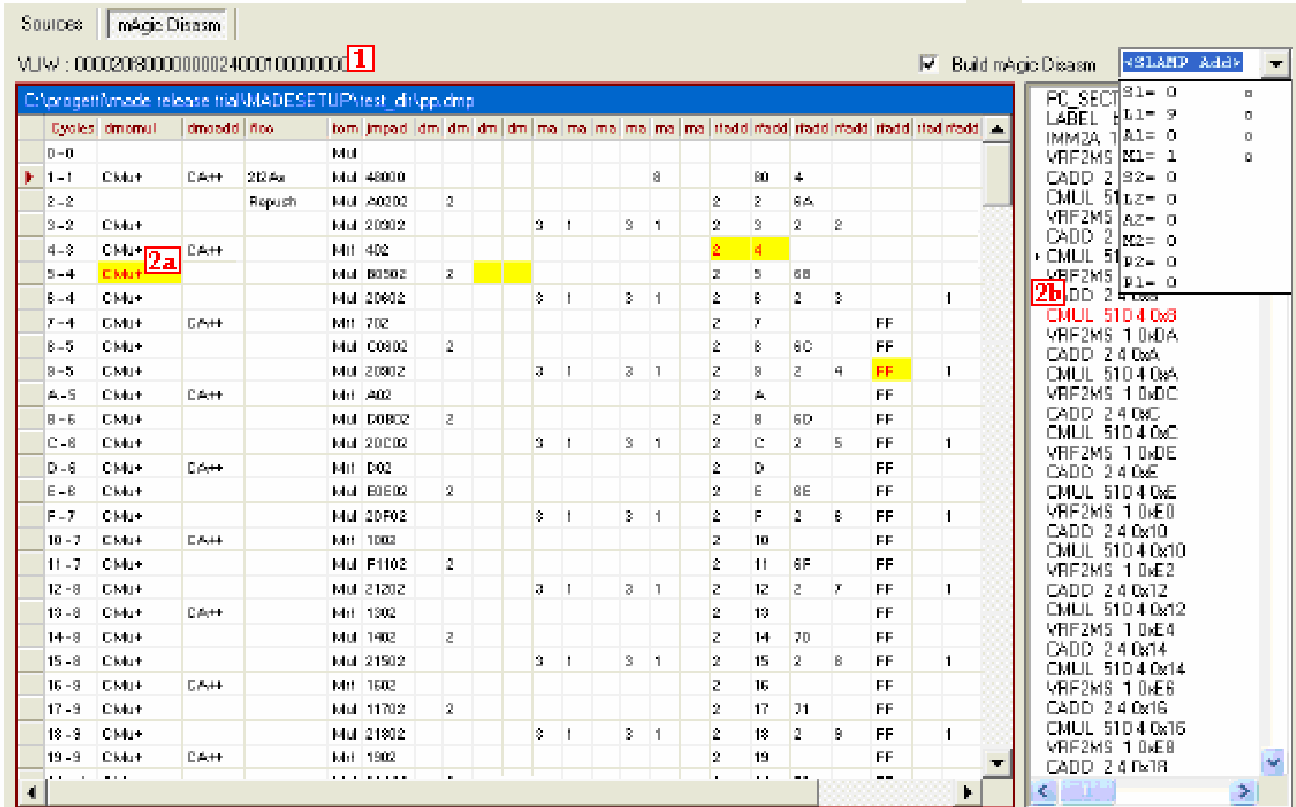


Figure 6 - mAgic disassembler

The mAgic disassembler is split in two panels:

1) On the left a table that shows the list of VLIW command in hexadecimal executed by mAgic, ordered by memory address, in compress and non-compress mode (Cycles of clock)
2) On the right the assembler code of mAgic as results from VLIW commands translation.

The upper label (1) show in binary the code of the VLIW command and in the left table (2a) is highlighted there is how is represented in memory the command *CMUL 510 4 0x8* in the panel on the right (2b).

All this information can be used to understand the mAgic program efficiency and its use of the memory in order to valuate, for example, if the automatic assembler code optimizer did a good job, and, in case it didn't, decide to change the source code.

- The other important tool allows to precisely plot the normal profiling mode of the mAgic program memory address read from the local control device in mAgic, i.e. the amount of the cycles for each part of the program.

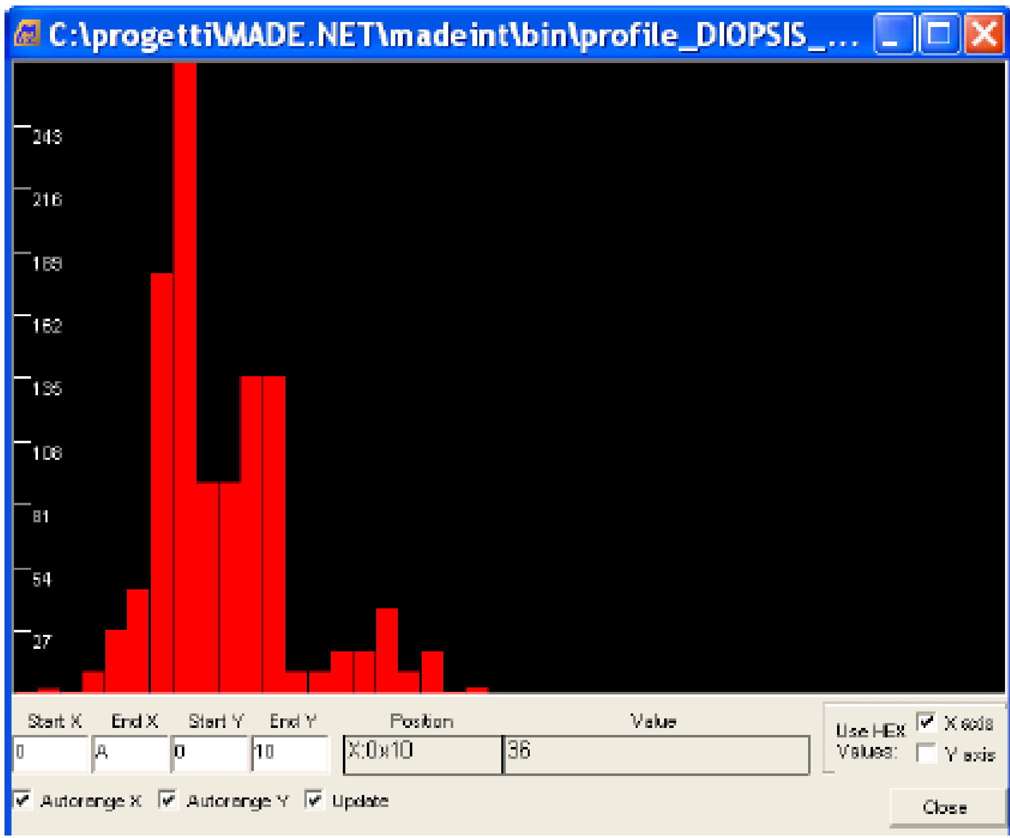


Figure 7 - PMA profile histogram

The histogram in red counts the number of times (the Y-axis value) the PMA assumes the corresponding X-axis value for each value in the monitored range. This means that the application has spent a greater number of cycles where the spikes are higher.

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 order to give a great speedup to the overall system. The exact X value corresponding to a rectangle can be displayed by simply moving the mouse cursor on it, and the Y value associated can be displayed by double clicking.

Hence, this tool is useful to improve a specific function. During optimization phase this tool is very useful, in fact it allows to monitor, step-by-step, the efficiency of the function or of its inner parts.

Case study: Computation of the FIR (Finite Impulse Response)

FIR filters are one of two primary types of digital filters used in Digital Signal Processing (DSP) applications (the other type being IIR).

A finite impulse response (FIR) filter is a type of a digital filter. It is 'finite' because its response to an impulse ultimately settles to zero. This is in contrast to infinite impulse response filters which have internal feedback and may continue to respond indefinitely.

Whenever we want that in a system error signals do not affect the main data flow, we have to attenuate them using filters.

FIR filters are computed opportunely weighting and adding echoes (delayed signal replicas of the input). FIR filters, also known as digital filters, are the ideal solution for complex issue in vocal analysis and synthesis, in images, digital audio elaboration.

For example in the following Figure, the input signal (red) is filtered (blue) using a FIR, in order to reduce the noise. Finally the signal can be processed because cleaned of its irrelevant components.

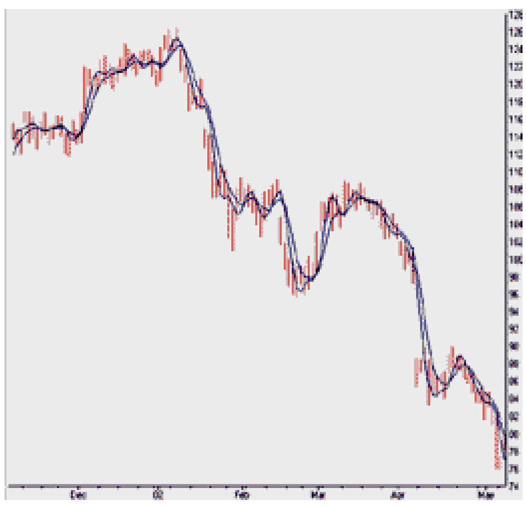


Figure 8 - Input signal (red) and filtered signal (blue)

Due to the fact that FIR filters are so important, a simple algorithm was made to test and stress the Diopsis 740 architecture.

In this project the role of the ARM is pretty trivial, in fact it makes the mAgic processor begin its computation, and eventually it waits for its termination.

In order to avoid more complexity, the samples are created directly in mAgic, while in a future implementation it would be possible to create the signals in the ARM processor and pass them to the mAgic via shared memory, or, even better, it would be possible to sample external signals from the ADC located on the JTST (test board produced by Atmel).

The chosen FIR filter allows doing the convolution between two signals.

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(\alpha) g(x-\alpha) d\alpha$$

Equation 1 - Convolution Function

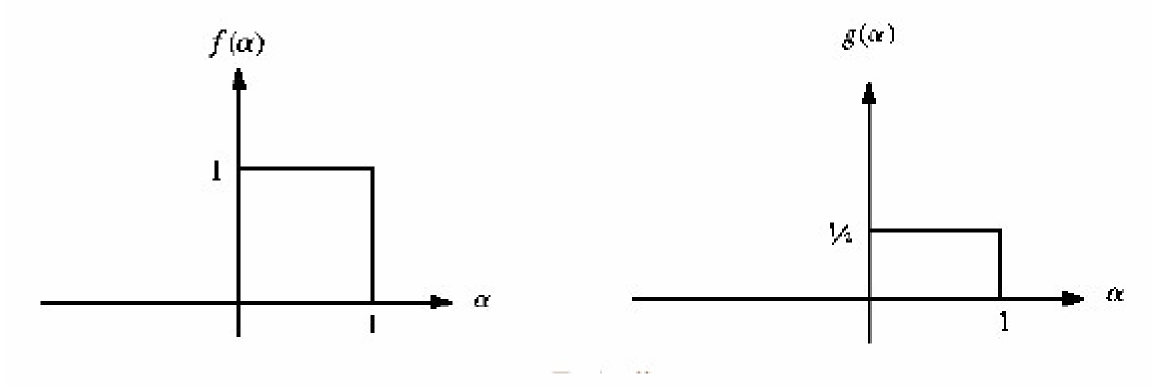


Figure 9 - Two sample signals

The operations done by the algorithm are the turnover, translation and multiplication of a signal wrt the other. The result of the convolution of the upper signals is:

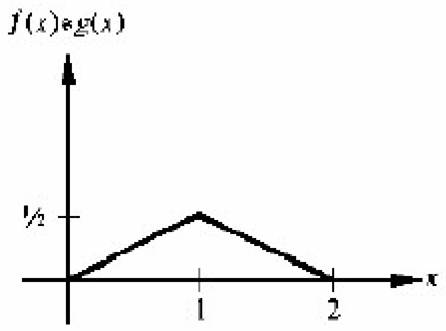


Figure 10 - The result of convolution

A simple algorithm which computes this function on two digital signals (previously sampled), is the following:

```
void conv(__complex__ float *x, __complex__ float *h, __complex__ float *y, int xLEN, int hLEN)
{
	//initialization
	for(n=hLEN-1; n<xLEN; n++)
	{
		temp = 0.0 + 0.0i;
		for(k=0; k<hLEN; k++)
		{
			temp = temp + x[n-k] * h[k];
		}
		y[n-hLEN+1] = temp;
	}
}
```

Obviously, the performance of this program is not good enough, because the C code optimizer cannot recognize and merge more instructions in a optimal VLIW instruction. Basically, in this first step it's done a mere translation with waste of resource and cycles from the C instruction to the correspondent VLIW instruction.

Hence, a user level optimization is needed!

Profile performance and optimization of the algorithm

C-code

The previous implementation of the algorithm is very time and resource wasting, in fact it cannot exploit the hardware architecture of the mAgic processor. In this way, instead of using in the same time all the functional units the DSP offers this algorithm use for each cycle a different functional unit without taking advantage of the intrinsic parallelism.

With simple considerations, and using the standard method of optimization, also suggested by Atmel, it's possible to gain appreciable speed-up and thus to improve performance.
First of all we want to understand where optimization are necessary.
As said in the introduction, MADE has a tool that allows to profile a resource of Diopsis 740. Obviously, the most important resource to profile in this case is the PMA, the Program Memory Address. Analyzing the resulting plot, a user can evaluate at glace where the program spends the major number of cycles, and so it is possible for example to decide which part of the code must be optimized.
The histogram of the PMA for the C-code is the following:

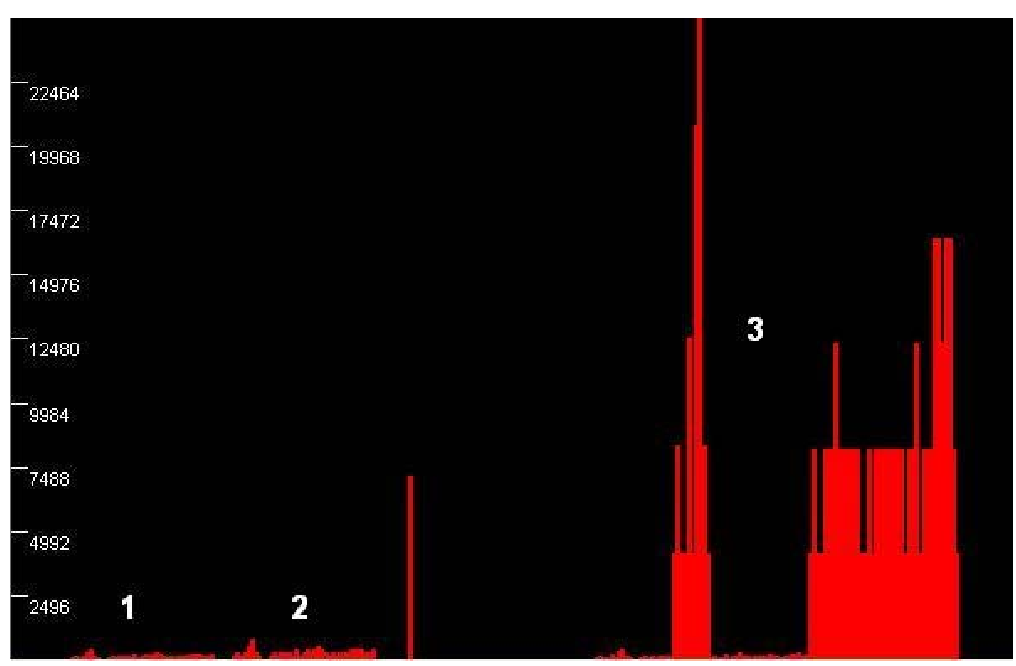


Figure 11 - PMA profile of C-code

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 this Figure it's possible to recognize three parts: (from the left to the right) the first and the second parts are those where the two signals are initialized in a loop; the third part is the algorithm that computes the convolution. Notice that the last part is divided into a first loop and an inner loop.

The most of time is spent in the inner loop, where the hard computation takes place. In order to improve the performance, it's fundamental to look up to the Disasm code generated by MADE, where it's possible to identify if the relevant parts of the algorithm are translated in such a way the mAgic architecture is exploited.

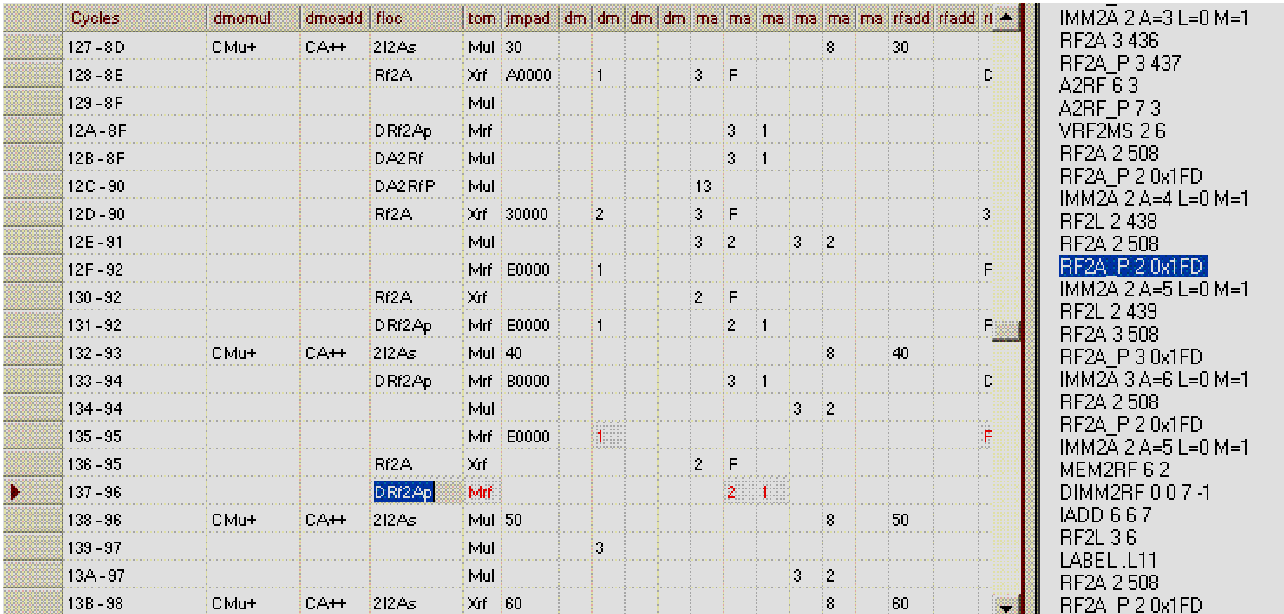


Figure 12 - VLIW scheduler of C-code

This code on the right is not so good. In fact:

- It doesn't use special operation provided by mAgic; hence, to do an addition, the mAgic waste an entire clock cycle.
- 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 VLIW scheduler is not able to find much parallelism: the figure shows a VLIW scheduler quite sparse.

Assembler-code with loop fully unrolled.

After those considerations, it's easy to understand the importance of writing the whole function in assembler; in that way is enabled the use of special operations provided by mAgic.

In this case the loop code in assembly is fully unrolled:

```
...
\dup i=0 to 63
        VMS2RF_U X ARFinX
        CMUL MulReg <H+2*i> X
        CADD AccReg AccReg MulReg
\enddup
...
```

Table 1 - Asm code fully unrolled

The dup asm instruction allows to repeat text in which the part between <...> is substituted by a pre-computed expression.

Let's look now the new profile:

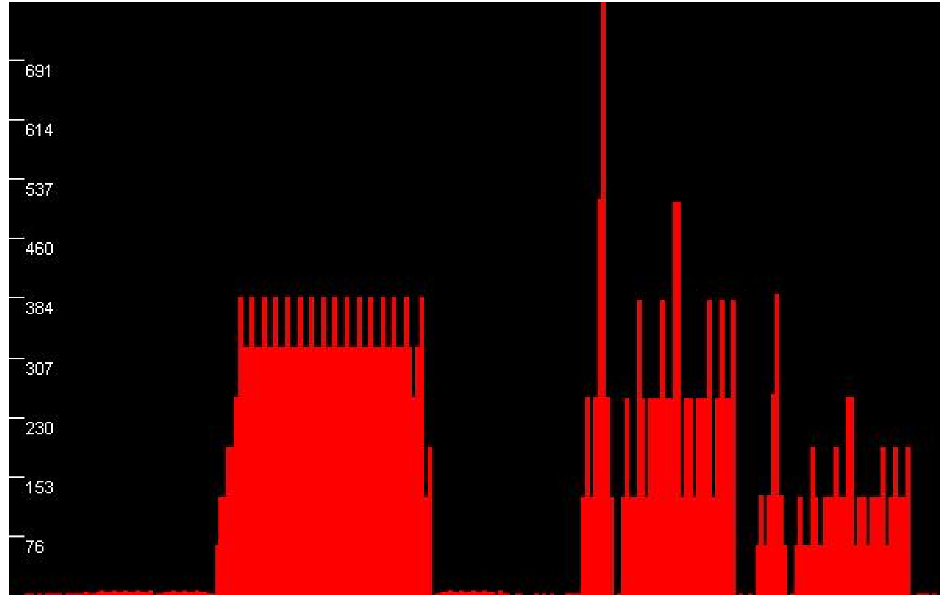


Figure 13 - PMA profile fully unrolled

The above figure shows the profile of PMA when the function that does the convolution is totally translated in assembler. Now the mAgic has two files: the first in C-code, allows

initialization of signals; the second entirely in assembler provides the computation of FIR filter. It's possible to see that now the assembler code is allocated first in PMA wrt the C-code. Notice that the improvement was very effective: the convolution function is computed in a so fast way that now it's directly comparable with the two signals initialization. In the first part the histogram is so compact because of the full unroll of loop.

The next figure shows how VLIW are now scheduled.

Cycles	dmomul	dmoadd	floc	tom	jmpad	dm	dm	dm	dm	ma	ma	ma	ma	ma	ma	rfadd	rfadd	rfadd
AB - 2F				Mul						2	1	2	2	1	2			
AC - 30				Mul						2	1	2	2	1	2			
AD - 30				Mul						2	1	2	2	1	2			
AE - 30				Mul						2	1	2	2	1	2			
AF - 30				Mul						2	1	2	2	1	2			
B0 - 30				Mul						2	1	2	2	1	2			
B1 - 30				Mul						2	1	2	2	1	2			
B2 - 31				Mul						2	1	2	2	1	2			
B3 - 31				Mul						2	1	2	2	1	2			
B4 - 31				Mul						2	1	2	2	1	2			
B5 - 31				Mul						2	1	2	2	1	2			
B6 - 31				Mul	B0000		2			2	1	2	2	1	2			DB
B7 - 32				Mul						2	1	2	2	1	2			
B8 - 32			Rf2A	Mrf						F	4							
B9 - 33				Mul														
BA - 33				Mul														
BB - 34				Mul						2		2	2		2			
BC - 35	CMu+	CA++	2I2Rf	Mul														
BD - 35				Mul						2		2	2		2			
BE - 35		IA++		Mul		1	2							F	3			
BF - 36			IncCo	Mul	307					2		2	2		2	7	3	

VMS2RF_U 38 1
VMS2RF_U 36 1
VMS2RF_U 34 1
VMS2RF_U 32 1
VMS2RF_U 30 1
VMS2RF_U 28 1
VMS2RF_U 26 1
VMS2RF_U 24 1
VMS2RF_U 22 1
VMS2RF_U 20 1
VMS2RF_U 18 1
VMS2RF_U 16 1
VMS2RF_U 14 1
IADD 439 438 -439
IADD 439 439 502
SH 439 439 150
IMM2A 15 S=0 A=0 M=1
RF2A_L 15 439
LABEL LOOP1
DIMM2RF 152 0.0 153 0.0
VMS2RF_U 6 0
CMUL 150 14 6
CADD 152 152 150
VMS2RF_U 6 0
CMUL 150 16 6
CADD 152 152 150
VMS2RF_U 6 0
CMUL 150 18 6

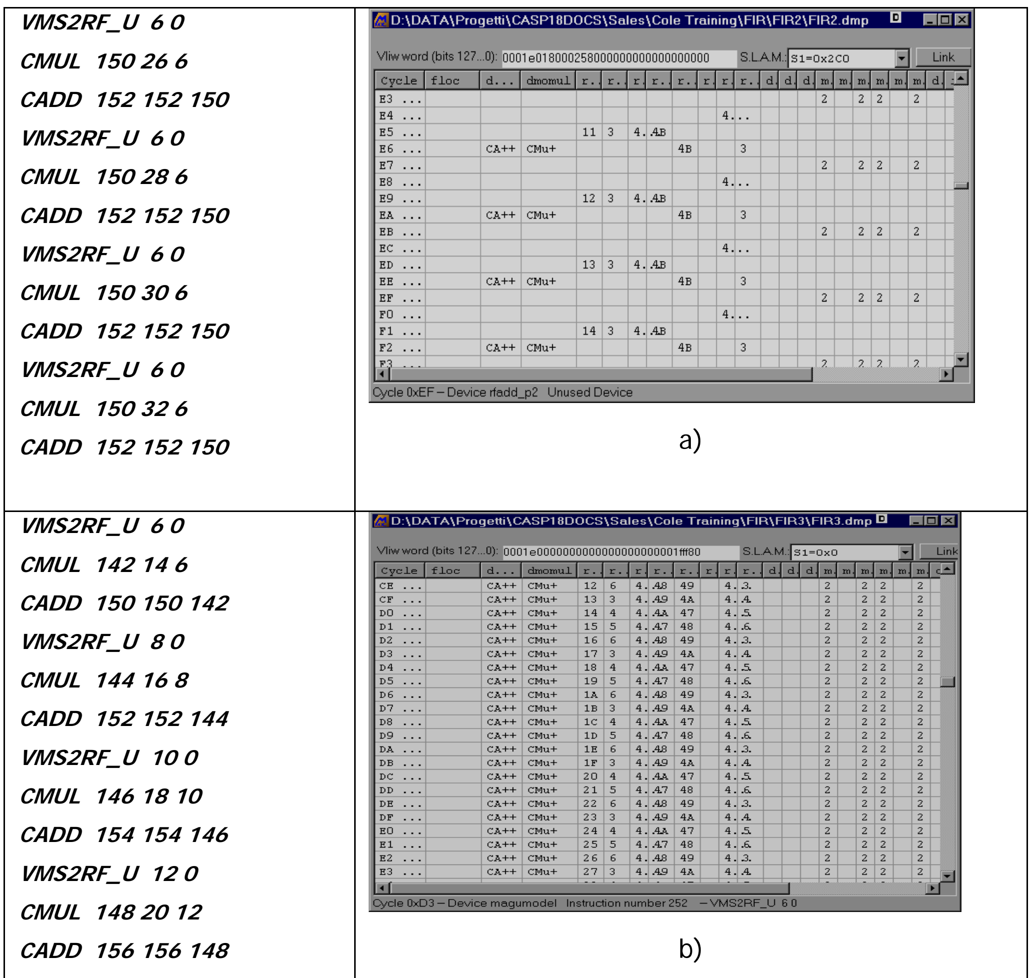
Figure 14 - VLIW scheduler fully unrolled

Now is possible to compare the above figure with the **Error! Reference source not found.**showing VLIW scheduling of the original function. The new VLIW scheduling is denser than the previous one, this implies that we have much more parallelism.

Assembler-code with register dependencies removed

First of all, now it's possible to allocate each vector data in a proper register, so to make easier for mAgic to retrieve the data for the computation. This kind of register dependencies elimination allows exploiting more parallelism between consecutive operations.

For example:



VMS2RF_U 6 0 *CMUL 150 26 6* *CADD 152 152 150* *VMS2RF_U 6 0* *CMUL 150 28 6* *CADD 152 152 150* *VMS2RF_U 6 0* *CMUL 150 30 6* *CADD 152 152 150* *VMS2RF_U 6 0* *CMUL 150 32 6* *CADD 152 152 150*	a)
VMS2RF_U 6 0 *CMUL 142 14 6* *CADD 150 150 142* *VMS2RF_U 8 0* *CMUL 144 16 8* *CADD 152 152 144* *VMS2RF_U 10 0* *CMUL 146 18 10* *CADD 154 154 146* *VMS2RF_U 12 0* *CMUL 148 20 12* *CADD 156 156 148*	b)

Table 2 - Example of register dependencies elimination

Looking at figure, it's possible to compare the result of scheduling in an intuitive way. In a) the scheduled VLIW instructions are sparse, in fact the correspondent assembler code on the left uses for each couple of CMUL and CADD operations the same registers. In b) the register dependencies are eliminated allocating the results of the CADD and CMUL operations specific register cells. The scheduler can exploit the parallelism in the higher way.

In this case the assembler instructions for the loop are the following:

```
...
\dup i=0 to 15
        \dup k=0 to 3
                VMS2RF_U <X+2*k> ARFinX
                CMUL <MulReg+2*k> <H+2*(4*i+k)> <X+2*k>
                CADD <AccReg+2*k> <AccReg+2*k> <MulReg+2*k>
        \enddup
\enddup
...
```

Table 3 - Asm code without register dependencies

Notice that each operation (VMS2RF, CMUL, CADD) have data loaded and stored in different register cells, so to eliminate all register dependencies.

Here you are the new profile:

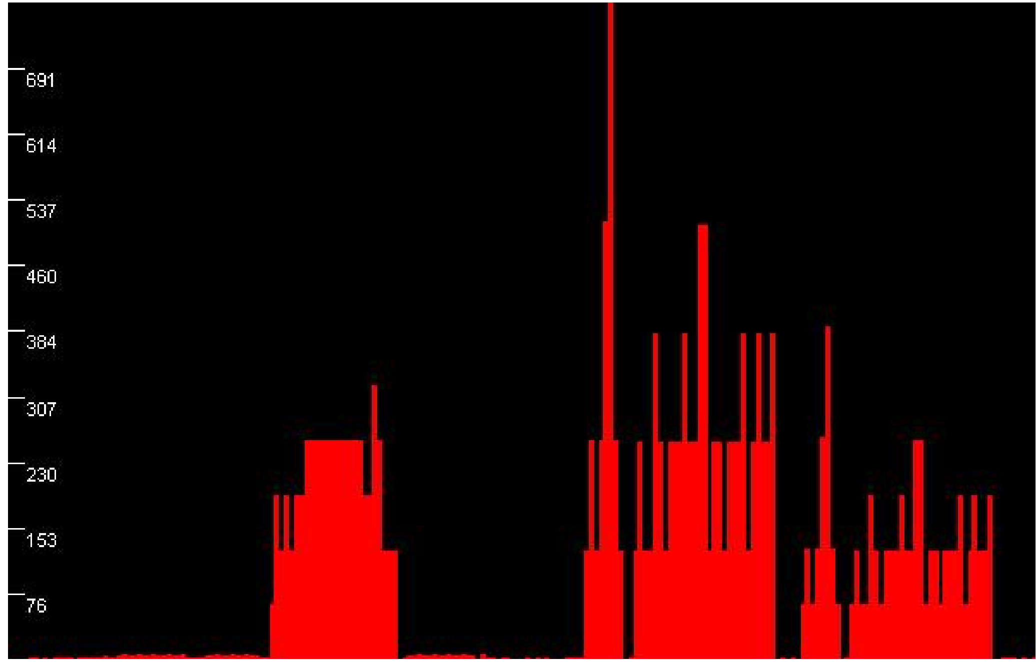
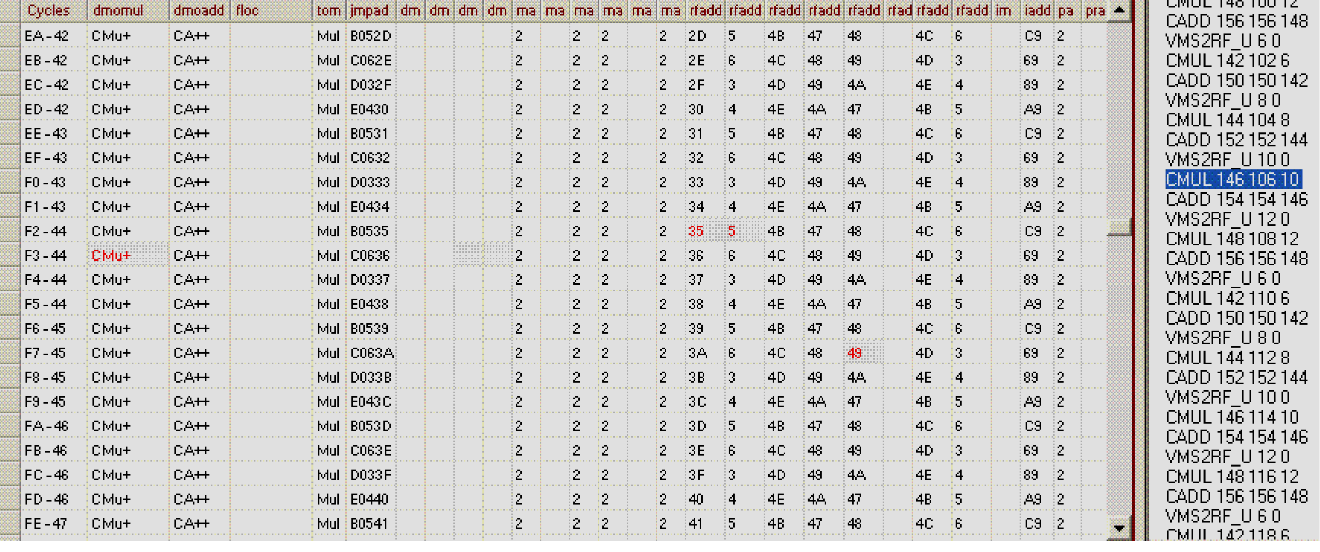


Figure 15 - PMA profile without register dependencies

The optimization is impressive! The convolution function in assembler has a total amount of accesses to the PMA absolutely less than the signal initialization loops in C-code!
Now take a look at new VLIW scheduling:



Cycles	dmomul	dmoadd	floc	tom	jmpad	dm	dm	dm	dm	ma	ma	ma	ma	ma	ma	rfadd	rfadd	rfadd	rfadd	rfadd	rfad	rfadd	rfadd	im	iadd	pa	pra
EA - 42	CMu+	CA++		Mul	B052D					2		2	2		2	2D	5	4B	47	48		4C	6		C9	2	
EB - 42	CMu+	CA++		Mul	C062E					2		2	2		2	2E	6	4C	48	49		4D	3		69	2	
EC - 42	CMu+	CA++		Mul	D032F					2		2	2		2	2F	3	4D	49	4A		4E	4		89	2	
ED - 42	CMu+	CA++		Mul	E0430					2		2	2		2	30	4	4E	4A	47		4B	5		A9	2	
EE - 43	CMu+	CA++		Mul	B0531					2		2	2		2	31	5	4B	47	48		4C	6		C9	2	
EF - 43	CMu+	CA++		Mul	C0632					2		2	2		2	32	6	4C	48	49		4D	3		69	2	
F0 - 43	CMu+	CA++		Mul	D0333					2		2	2		2	33	3	4D	49	4A		4E	4		89	2	
F1 - 43	CMu+	CA++		Mul	E0434					2		2	2		2	34	4	4E	4A	47		4B	5		A9	2	
F2 - 44	CMu+	CA++		Mul	B0535					2		2	2		2	35	5	4B	47	48		4C	6		C9	2	
F3 - 44	CMu+	CA++		Mul	C0636					2		2	2		2	36	6	4C	48	49		4D	3		69	2	
F4 - 44	CMu+	CA++		Mul	D0337					2		2	2		2	37	3	4D	49	4A		4E	4		89	2	
F5 - 44	CMu+	CA++		Mul	E0438					2		2	2		2	38	4	4E	4A	47		4B	5		A9	2	
F6 - 45	CMu+	CA++		Mul	B0539					2		2	2		2	39	5	4B	47	48		4C	6		C9	2	
F7 - 45	CMu+	CA++		Mul	C063A					2		2	2		2	3A	6	4C	48	49		4D	3		69	2	
F8 - 45	CMu+	CA++		Mul	D033B					2		2	2		2	3B	3	4D	49	4A		4E	4		89	2	
F9 - 45	CMu+	CA++		Mul	E043C					2		2	2		2	3C	4	4E	4A	47		4B	5		A9	2	
FA - 46	CMu+	CA++		Mul	B053D					2		2	2		2	3D	5	4B	47	48		4C	6		C9	2	
FB - 46	CMu+	CA++		Mul	C063E					2		2	2		2	3E	6	4C	48	49		4D	3		69	2	
FC - 46	CMu+	CA++		Mul	D033F					2		2	2		2	3F	3	4D	49	4A		4E	4		89	2	
FD - 46	CMu+	CA++		Mul	E0440					2		2	2		2	40	4	4E	4A	47		4B	5		A9	2	
FE - 47	CMu+	CA++		Mul	B0541					2		2	2		2	41	5	4B	47	48		4C	6		C9	2	

CADD 156 156 148
VMS2RF_U 6 0
CMUL 142 102 6
CADD 150 150 142
VMS2RF_U 8 0
CMUL 144 104 8
CADD 152 152 144
VMS2RF_U 10 0
CMUL 146 106 10
CADD 154 154 146
VMS2RF_U 12 0
CMUL 148 108 12
CADD 156 156 148
VMS2RF_U 6 0
CMUL 142 110 6
CADD 150 150 142
VMS2RF_U 8 0
CMUL 144 112 8
CADD 152 152 144
VMS2RF_U 10 0
CMUL 146 114 10
CADD 154 154 146
VMS2RF_U 12 0
CMUL 148 116 12
CADD 156 156 148
VMS2RF_U 6 0

Figure 16 - VLIW scheduler without register dependencies

A great level of parallelism is indeed achieved! Nevertheless more optimizations are still possible.

Assembler-code unrolled (8) and software pipelining

This optimization has the direct effect to reduce the memory occupation of the code in assembly, in fact the loop unroll have to be of the correct size: in very large loops unrolling the program memory occupation grows enormously. Due to the fact that the code now is not totally unrolled, it's important to be sure the part of code inside a loop can be parallelized. The software pipelining is an optimization technique that allows executing instructions in parallel way doing proper initialization.

The effects of this optimization are shown in the following PMA profile:

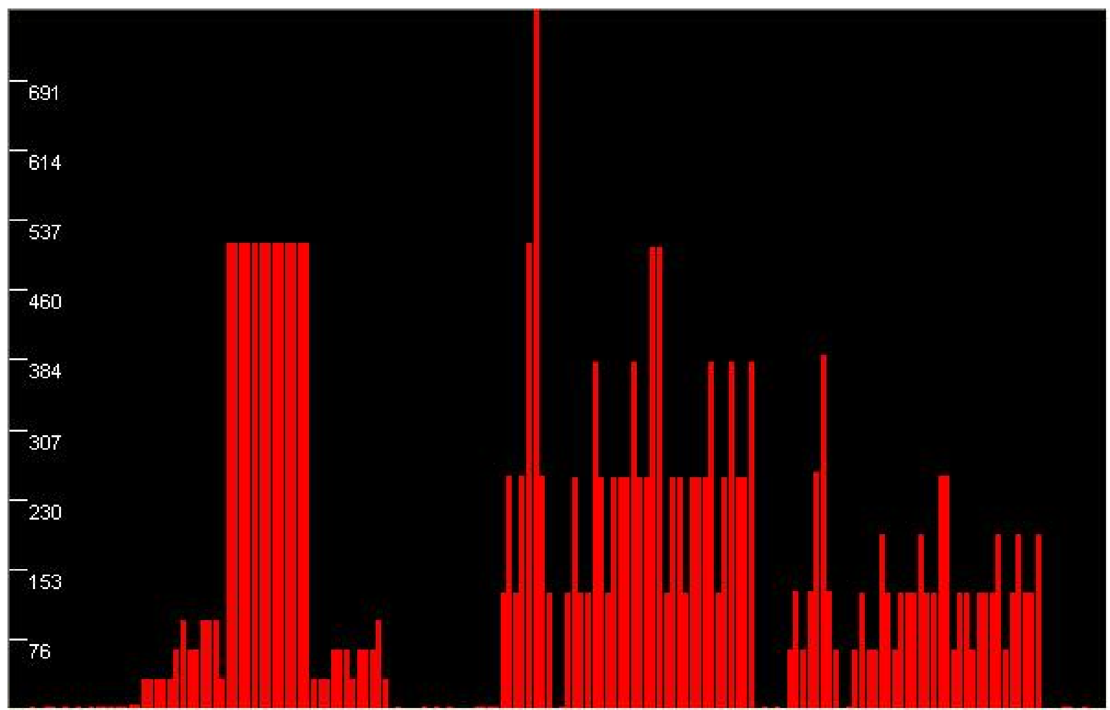
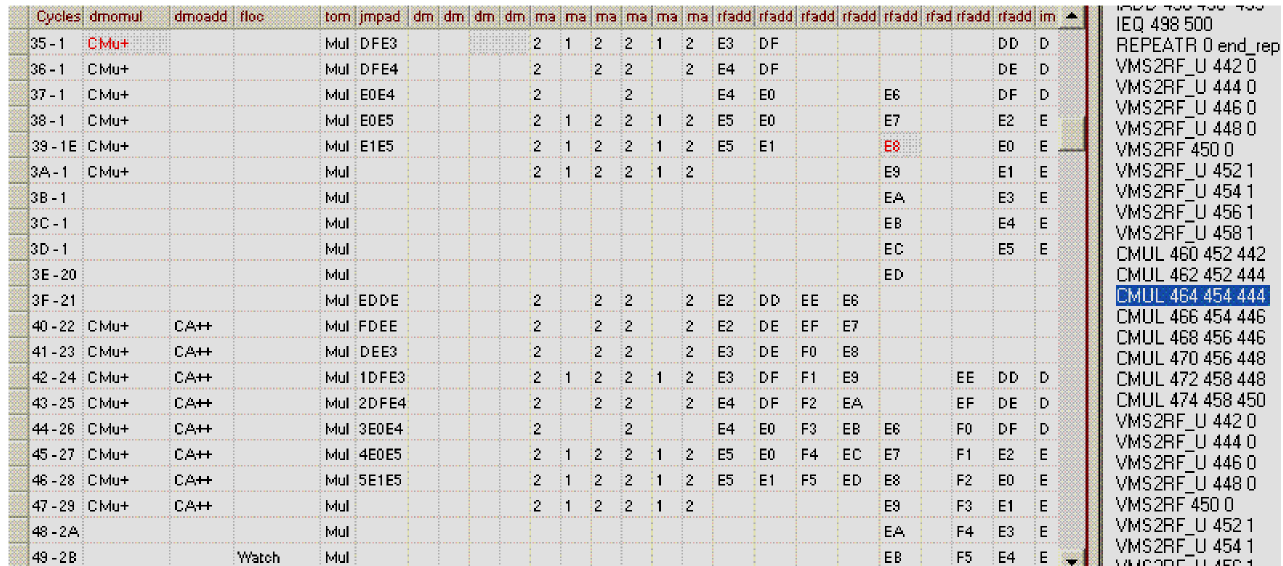


Figure 17 - PMA profile unrolled (8) + software pipelining

We can se on the left the algorithm in practice accesses, in the same way, the same part of memory. Exactly the consecutive memory cells used in the loop once unrolled for 8 operations.

Obviously the reduction on memory occupation impacts on the computation time and thus on performance. Despite the software pipelining, the code is not as parallel as in the previous step, while the memory occupation of the FIR filter function is almost the half.



Cycles	dmomul	dmoadd	floc	tom	jmpad	dm	dm	dm	dm	ma	ma	ma	ma	ma	ma	rfadd	rfadd	rfadd	rfadd	rfadd	rfad	rfadd	rfadd	im
35 - 1	CMu+			Mul	DFE3					2	1	2	2	1	2	E3	DF						DD	D
36 - 1	CMu+			Mul	DFE4					2		2	2		2	E4	DF						DE	D
37 - 1	CMu+			Mul	E0E4					2			2			E4	E0			E6			DF	D
38 - 1	CMu+			Mul	E0E5					2	1	2	2	1	2	E5	E0			E7			E2	E
39 - 1E	CMu+			Mul	E1E5					2	1	2	2	1	2	E5	E1			E8			E0	E
3A - 1	CMu+			Mul						2	1	2	2	1	2					E9			E1	E
3B - 1				Mul																EA			E3	E
3C - 1				Mul																EB			E4	E
3D - 1				Mul																EC			E5	E
3E - 20				Mul																ED				
3F - 21				Mul	EDDE					2		2	2		2	E2	DD	EE	E6					
40 - 22	CMu+	CA++		Mul	FDEE					2		2	2		2	E2	DE	EF	E7					
41 - 23	CMu+	CA++		Mul	DEE3					2		2	2		2	E3	DE	F0	E8					
42 - 24	CMu+	CA++		Mul	1DFE3					2	1	2	2	1	2	E3	DF	F1	E9			EE	DD	D
43 - 25	CMu+	CA++		Mul	2DFE4					2		2	2		2	E4	DF	F2	EA			EF	DE	D
44 - 26	CMu+	CA++		Mul	3E0E4					2			2			E4	E0	F3	EB	E6		F0	DF	D
45 - 27	CMu+	CA++		Mul	4E0E5					2	1	2	2	1	2	E5	E0	F4	EC	E7		F1	E2	E
46 - 28	CMu+	CA++		Mul	5E1E5					2	1	2	2	1	2	E5	E1	F5	ED	E8		F2	E0	E
47 - 29	CMu+	CA++		Mul						2	1	2	2	1	2					E9		F3	E1	E
48 - 2A				Mul																EA		F4	E3	E
49 - 2B			Watch	Mul																EB		F5	E4	E

Figure 18 - VLIW scheduler unrolled (8) + software pipelining

Here it's possible to see how the unrolling of eight affects the VLIW scheduling, causing a reduction of parallelism.

Assembler-code of whole mAgic code

As a final step, to really understand the effects of optimization, it's possible to write the whole code for mAgic in assembly, including the signals initialization loops. Doing this way, it's almost achieved the theoretical number of cycles in computing the filter.

In this case the code is fully unrolled and software pipelined.

The new profile is the following:

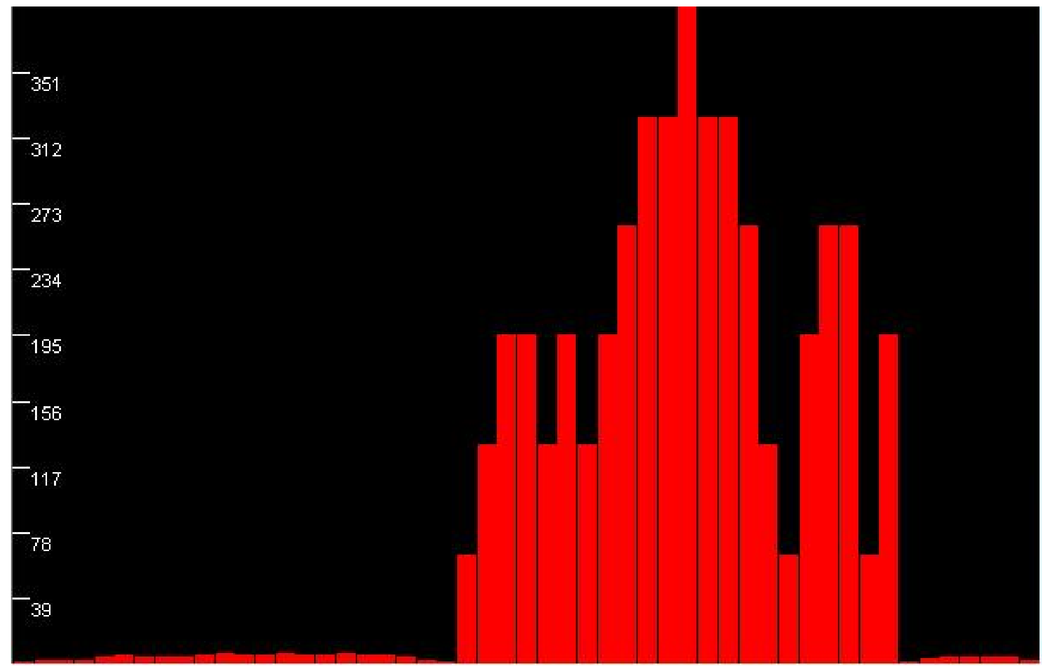


Figure 19 - PMA profile of whole mAgic code

The final result is great; the area of the histogram is considerably reduced from the first implementation of the C-code. The result in performance is somewhat unbelievable, and the memory occupation is also reduced.

Naturally the level of parallelism is almost maximum, as it's possible to see in the following schedule:

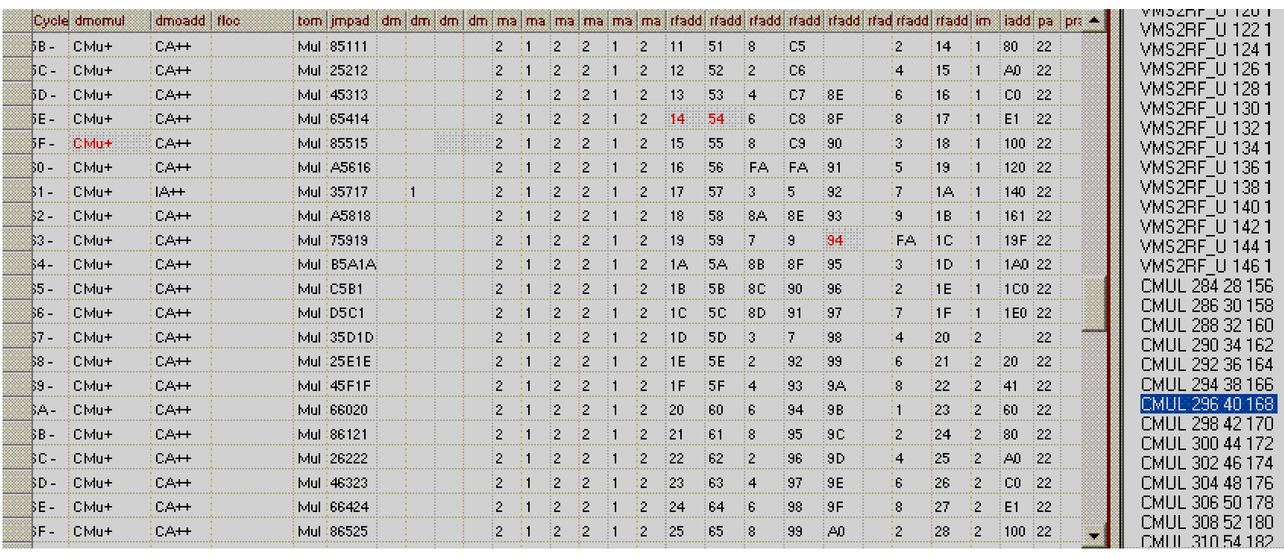


Figure 20 - VLIW scheduler of whole mAgic code

No more significant optimization can be done without spending too much time finding a better solution at algorithm level. However in next session will be clearer how much the optimizations improve the performance in each step.

Analysis

First of all, to make a correct performance evaluation, it's necessary to compute the theoretical number of clock cycle the function has to spend. This calculus it's indeed easy, in fact it's enough to multiply the number of times the external loop and the inner loop are executed. In practice, to compute the FIR filter for 64 samples, the mAgic has to compute 65*64 = 4160 operations, theoretically one per cycle. Obviously, it's impossible to reach this performance, since actually the number of operations it's greater. In the previous calculus, in fact, we did not take care about the loop instructions, like the condition, the initialization and the increment of the counter.

However the optimizations allow reaching a very similar performance, in which the mAgic can almost compute each loop iteration in one clock cycle!

Legend:

- FIR1 = C-code
- FIR2 = fully unrolled
- FIR3 = Register dependencies removed
- FIR4 = Software pipelining + unrolled (8)
- FIR5 = Whole function unrolled + software pipelining

		FIR5	*FIR4*	FIR3	FIR2	FIR1
	Cycle count	4721	*8105*	5791	17562	64689
FIR5	4721		*1,7167*	1,2266	3,72	**13,7**
FIR4	*8105*			*0,7145*	*2,1668*	***7,9813***
FIR3	5791				3,0326	**11,1706**
FIR2	17562					**3,6835**
FIR1	64689					

Table 4 - Optimization steps: number of cycles

In the previous table seems to be something strange, in fact the FIR4 optimization does not have a direct positive effect on performance. But, if you remember what it was said

before, this optimization was made only for reduce memory occupation, as shown in the following table:

		FIR5	*FIR4*	FIR3	FIR2	FIR1
	Memory occupation	51	*59*	101	129	110
FIR5	51		*1,16*	1,98	2,52	**2,16**
FIR4	*59*			*1,71*	*2,19*	***1,86***
FIR3	101				1,28	**1,09**
FIR2	129					**0,85**
FIR1	110					

Table 5 - Optimization steps: memory occupation

Now it's possible to realize that the reduction of memory occupation during FIR4 optimization it's more relevant than the ones before.

Finally is shown the importance of each optimization wrt theoretical cycles (4160).

	cycles	% of theoretical
FIR5	4721	**113 %**
FIR4	8105	**195 %**
FIR3	5791	**139 %**
FIR2	17562	**422 %**
FIR1	64689	1555 %

Table 6 - Optimization steps: wrt theoretical number of cycles

Conclusion

This project highlights the importance in doing optimizations in the code for embedded systems. The better performance we want to achieve, the more effort is necessary. But this work doesn’t mean whenever we produce code we have to do optimization: actually a large set of function are already prepared and optimized by the producers of the processors, because they can exploit better than anybody the architecture of the system. In this case, mAgic has in its library several mathematical functions, some audio and frequency-specific functions, and so on.

So, the utility of this project wasn’t create a good code, optimized for future use because mAgic already has it; instead, the goal of the whole work is to better understand the architecture of a DSP processor which use VLIW just like mAgic, and after that, to exploit the specific functionality to improve the performance.

Indeed the final result is very relevant, and lets us understand the powerfulness of embedded systems and dedicated programming.

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)
- FIR Filter FAQ - http://www.dspguru.com/info/faqs/firfaq.htm
- Finite Impulse Response – http://en.wikipedia.org/wiki/Finite_impulse_response
- Chiedi all'esperto – http://www.vialattea.net/esperti/php/risposta.php?num=8783
- "Experimental Modal Analisys" – Peter Avitabile, University of Massachusetts Lowell, Lowell, Massachusetts
- "Misure di Vibrazione e Analisi Modale" – Rivola Alessandro, Univeristà degli studi di Bologna