TITLE: Migration of a stereoscopic camera system from Matlab to the TMS320DM814x DaVinci platform

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: Joan Fité Sànchez

DIRECTOR: Antoni Gelonch

SUPERVISOR: Albert Puig

DATE: July 9th 2012
Títol: Migration of a stereoscopic camera system from Matlab to the TMS320DM814x DaVinci platform

Autor: Joan Fité Sànchez

Director: Antoni Gelonch

Data: 9 de juliol de 2012

Resum

El manteniment del ferrocarrils es produeix de nit en ambients sorollosos. És necessari un sistema que previngui els accidents laborals mentrés les vies dels trens són restaurades.

La tesina del màster es centra en la migració d’un sistema estereoscòpic de càmeres desenvolupat en Matlab a un sistema encastrat. Un sistema estereoscòpic de càmeres és capaç de mesurar la distància a un objecte utilitzant les imatges de dos càmeres diferents separat a una distància coneguda.

El sistema encastrat utilitza la placa TMS320DM814x. La plataforma inclou un ARM Cortex A8 i la DSP C674x de coma flotant que el fa perfecte per sistemes digitals de video. Per dur a terme aquesta migració s’ha utilitzat el C6ACCEL que interactua entre el DSP i el ARM. El C6ACCEL té varis funcions implementades però també permet crear de noves.

L’ autor de la tesina ha optimizat una part dels algoritmes de Matlab d’un temps inicial de 8 segons fins a 94,7 ms.
Overview

The maintenance of the railways is done at night in a noisy environment. A system is needed to prevent the occupational accidents while the train tracks are restored.

The master thesis is focused on the migration of a stereoscopic camera system developed in Matlab to an embedded environment. A stereoscopic camera system is capable of measuring the distance of an object using the images of two different cameras knowing the separation between them. If an object is detected inside a specific area with a distance lower than a threshold, an alarm is generated.

The embedded system used is the TMS320DM814x DaVinci board. The platform includes an ARM Cortex A8 and a floating point DSP C674x that makes it suitable for digital video systems. To execute the migration, the C6EZACCEL would be used. It interacts between the ARM and the DSP that has its own library. The C6EZACCEL has several functions implemented but also allows to create new ones.

The author of the master thesis has optimized a part of the Matlab algorithms from an initial execution time of 8 seconds to 94,7ms.
INDEX

INTRODUCTION .................................................................................................................. 3

CHAPTER 1. WORK ENVIRONMENT ................................................................................. 4
1.1. TMS320DM814X ..................................................................................................... 4
   1.1.1. Features ................................................................................................. 5
   1.1.2. ARM Cortex-A8 .................................................................................... 6
   1.1.3. DSP C674x ........................................................................................... 6
1.2. Personal Computer ................................................................................................. 7

CHAPTER 2. C6EZACCEL .............................................................................................. 8
2.1. Advantages and disadvantages ............................................................................ 8
2.2. Devices supported .............................................................................................. 10
2.3. DSP kernel ......................................................................................................... 11
   2.3.1. Function IDS ....................................................................................... 12
   2.3.2. Extended Input Arguments ................................................................... 12
   2.3.3. Memory management .......................................................................... 13
2.4. Using C6EZAccel in an ARM application .......................................................... 14

CHAPTER 3. TECHNIQUES TO REDUCE THE EXECUTION TIME ...................... 16
3.1. Algorithms ........................................................................................................... 16
3.2. TI DSP Libraries and intrinsic operations ........................................................... 19
3.3. Software Pipelining ........................................................................................... 20
3.4. Compiler ............................................................................................................. 21

CHAPTER 4. STEREOSCOPIC CAMERA SYSTEM VISION .................................... 22
4.1. Stereo vision camera vision system .................................................................... 22
4.2. Block diagram ..................................................................................................... 23
   4.2.1. Transforma ............................................................................................ 24
   4.2.2. Block search .......................................................................................... 25
   4.2.2.1. Edge search ...................................................................................... 26
   4.2.2.1.1. Gaussian filter ............................................................................. 26
   4.2.2.1.2. Spline interpolation .................................................................... 27
   4.2.2.1.3. Derivative filter .......................................................................... 27
   4.2.2.1.4. Logic block .................................................................................. 27
   4.2.2.2. Lineal interpolation .......................................................................... 28
   4.2.3. Search of candidates ............................................................................. 28
   4.2.4. Disparity analysis ................................................................................... 29
## CONTENTS OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 1.1</td>
<td>Block diagram of a DaVinci Processor</td>
<td>4</td>
</tr>
<tr>
<td>Fig. 1.2</td>
<td>TMS320DM814x [1]</td>
<td>6</td>
</tr>
<tr>
<td>Fig. 1.3</td>
<td>High level C6000 DSP Architecture [1]</td>
<td>7</td>
</tr>
<tr>
<td>Fig. 2.1</td>
<td>Schematic of the intercommunication between the DSP and ARM [2]</td>
<td>8</td>
</tr>
<tr>
<td>Fig. 2.2</td>
<td>Processor load of the SoC as function on the running mode [2]</td>
<td>9</td>
</tr>
<tr>
<td>Fig. 2.3</td>
<td>Block Diagram view of calling multiple DSP kernels in C6Accel [2]</td>
<td>10</td>
</tr>
<tr>
<td>Fig. 2.4</td>
<td>Structure of an ID function [3]</td>
<td>12</td>
</tr>
<tr>
<td>Fig. 2.5</td>
<td>Schematic of a DSP function call [3] [4]</td>
<td>14</td>
</tr>
<tr>
<td>Fig. 3.1</td>
<td>Comparison of lineal execution (a) and pipeline execution (b)</td>
<td>20</td>
</tr>
<tr>
<td>Fig. 4.1</td>
<td>Schematic of an stereoscopic camera system</td>
<td>22</td>
</tr>
<tr>
<td>Fig. 4.2</td>
<td>Images from the master and slave cameras</td>
<td>23</td>
</tr>
<tr>
<td>Fig. 4.3</td>
<td>Block diagram of the whole stereoscopic camera system</td>
<td>23</td>
</tr>
<tr>
<td>Fig. 4.4</td>
<td>Correction of a biconvex distorsion</td>
<td>24</td>
</tr>
<tr>
<td>Fig. 4.5</td>
<td>Before (a.) and after (b.) of the biconvex correction distortion</td>
<td>25</td>
</tr>
<tr>
<td>Fig. 4.6</td>
<td>Block diagram of the Block Search</td>
<td>25</td>
</tr>
<tr>
<td>Fig. 4.7</td>
<td>Block diagram of the Edge Search</td>
<td>26</td>
</tr>
<tr>
<td>Fig. 4.8</td>
<td>Input and output images of the Gaussian filter</td>
<td>27</td>
</tr>
<tr>
<td>Fig. 4.9</td>
<td>Output image of the Lineal Interpolation</td>
<td>28</td>
</tr>
<tr>
<td>Fig. 4.10</td>
<td>Disparity image generated with the disparity map</td>
<td>29</td>
</tr>
<tr>
<td>Fig. 5.1</td>
<td>Evolution of the execution time</td>
<td>31</td>
</tr>
<tr>
<td>Fig. 5.2</td>
<td>Call sequence of the functions</td>
<td>34</td>
</tr>
</tbody>
</table>
CONTENTS OF TABLES

Table 1. Main features of the TMS320DM814x .................................................. 5
Table 2. List of devices able to run the C6EZAccel [3] ................................. 11
Table 2. Codification of the category of function [3] .................................... 12
Table 3. 1 Clock cycles of each .M unit .......................................................... 16
Table 5. 1 Percentage of optimization ............................................................ 31
Table 5. 2 Errors on each function due to the migration ................................. 32
Table 5. 3 Execution time of each function in the different stages ............... 33
Table 5. 4 Functions on the call sequence ..................................................... 33
Table 5. 5 Execution of each call sequence option ......................................... 34
INTRODUCTION

The master thesis is involved in the Argus product (Active Reverse Guide System) of the IDNEO Technologies where the author has been working for a year collaborating in the designing of the hardware as well of the software.

Argus, a railway project, is a stereoscopic camera system able to detect any object in front of it in a certain distance. The system is composed by two cameras that are continuously acquiring images and it is installed in the railway engine.

Argus prevents the occupational accidents while the train tracks are restored. The maintenance of the railways is done at night in a noisy environment. The developed system advises with a buzzer and/or a led when an object is inside or cross the train tracks. The system needs to process the images on real-time, so the execution time of the algorithms become important.

The master thesis is focused on the optimization of the algorithms to reduce the execution time. As a complex project, it is developed by different departments and people. The algorithms have been designed in the Matlab software by the image processing technicians. The author of the report was the responsible to migrate and optimize a part of these algorithms from the Matlab environment to an embedded one.

In Chapter 1, we explain the TMS320DM814x, a TI evaluation board composed by a DSP and an ARM processor. We use an evaluation board to optimize the algorithms while the hardware department finishes the designing of the embedded system. The features of the computer used to evaluate the execution time of the Matlab algorithms are also explained. Chapter 2 contains information of the C6Accel API that allows the memory share between the ARM and the DSP and the parallel working.

The techniques used in the algorithms to reduce the execution time are presented in the Chapter 3. In Chapter 4, we explain the Argus system. We divide the Argus system in three main blocks. The first one gets the images from both cameras, detects the edges and interpolates the images. The second block processes the edges and determines if they are common in both images or not. The third block makes a disparity analysis and determines if inside a specific region there is an object or not. Although the report is centered in the optimization of the first block, the author also collaborates in the other ones.

The results of the optimization are presented in Chapter 5, which includes an evolution of a function from the Matlab environment to the embedded system. We calculate the execution time of each function in the ARM and DSP of the block 1 and decide the best call sequence to reduce even more the time. To sum up, Chapter 6 details the targets reached and future work lines.
CHAPTER 1. WORK ENVIRONMENT

The Chapter 1 shows the work environment of the project. The digital media processor is explained with its main features and also an explanation of the personal computer used to test the Matlab algorithms.

1.1. TMS320DM814X

A powerful processor is needed to analyze the images coming from the cameras of the system on real-time. One of the best solutions in the market is the TMS320DM814x manufactured by Texas Instruments.

The TMS320DM814x is a SoC (System on Chip) of the TI DaVinci family that includes an ARM Cortex A8, a floating point DSP C674x and several user interfaces like Gigabit Ethernet, USB and Parallel Camera Interface among others. It is ready for low power consumption and has a graphics accelerator that allows applications with a high image quality. A display subsystem is also incorporated to connect with external display. The schematic Fig. 1. 1 below shows the main blocks of the DaVinci Processors.

![Fig. 1. 1 Block diagram of a DaVinci Processor.](image-url)
1.1.1. Features

A list of the main features of the ARM and DSP is shown below. Further information of the TMS320DM814x can be found at the Texas Instruments site [1].

Table 1. Main features of the TMS320DM814x

<table>
<thead>
<tr>
<th>Possible applications</th>
<th>Image processing (video security, video conferencing, video phones...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Systems</td>
<td>Linux, Android, DSP/BIOS. In our case, we use the Angstrom Linux.</td>
</tr>
<tr>
<td>DSP</td>
<td>C674x</td>
</tr>
<tr>
<td>DSP Instruction Type</td>
<td>Floating and Fixed Point</td>
</tr>
<tr>
<td>DSP maximum frequency</td>
<td>750 MHz</td>
</tr>
<tr>
<td>ARM CPU</td>
<td>ARM Cortex-A8</td>
</tr>
<tr>
<td>ARM maximum frequency</td>
<td>1GHz</td>
</tr>
<tr>
<td>Video Acceleration</td>
<td>1 HDVICP (450 MHz)</td>
</tr>
<tr>
<td>Video capability</td>
<td>Decode, Encode, Multi-Channel, Transcode, Multi-format.</td>
</tr>
<tr>
<td>TI Audio codecs</td>
<td>AAC-LC/HE, MP3, G.711</td>
</tr>
<tr>
<td>TI Video codecs</td>
<td>H.264 BP/MP/HP, MPEG-4, MPEG-2, JPEG/MJPEG</td>
</tr>
<tr>
<td>Video resolution</td>
<td>1080p</td>
</tr>
<tr>
<td>On chip L1 Cache</td>
<td>64 KB (ARM), 64KB (DSP)</td>
</tr>
<tr>
<td>On chip L2 Cache</td>
<td>512KB (ARM) 256 KB (DSP)</td>
</tr>
<tr>
<td>General Purpose Memory</td>
<td>16-bit GPMC, LPDDR, NAND flash, NOR flash, Pseudo-SRAM, SRAM</td>
</tr>
<tr>
<td>DRAM</td>
<td>2x32-bit (DDR2-800, DDR3-800, LPDDR-400)</td>
</tr>
<tr>
<td>Peripherals</td>
<td>USB (2), EMAC, PCIe, SATA, MMC/SD (3), CAN (2), UART(6), I2C (4), McASP(6), SPI(4), LCD (2), HDMI.</td>
</tr>
<tr>
<td>IO Supply (V)</td>
<td>1.8-3.3V</td>
</tr>
<tr>
<td>Operating temperature</td>
<td>0-90°C</td>
</tr>
</tbody>
</table>

All the algorithms have been tested in a TMS320DM814x board with the following settings:

- Operative System: Arago Project (Linux)
- ARM frequency: 400 MHz
- DSP frequency: 500 MHz.
- Bootloader: NTFS
1.1.2. **ARM Cortex-A8**

The processor ARM Cortex A8 is based on the ARMv7 architecture and is able to run with a frequency up to 1GHz with a single core. The low power consumption, less than 300mW, makes it suitable for power-optimized applications as mobile phones. The ARM does not support floating-point but can emulate it. This emulation causes a decrease of the performance.

1.1.3. **DSP C674x**

The DSP C674x belongs to the new generation floating-point DSP C6000 with 64 general-purpose 32 bit registers and eight functional units as the Fig. 1. 3 represents.

The eight functional units are:
- .D (.D1 and .D2) handles data load and store operations.
- .S (.S1 and .S2) handles shift or compare operations.
- .M (.M1 and .M2) handles multiply operations.
- .L (.L1 and .L2) handles logic and arithmetic operations.
The device has a complete set of optimized development tools, including an efficient C compiler and an assembly optimizer.

The C6000 devices are able to execute up to eight instructions per cycle, improving the performance of typical DSP by ten. An instruction packing is also included reducing the code size and power consumption as the Appendix C shows. The different types of data-support (8/16/32 bits) provide efficient memory for a variety of applications.

Specifically the C674x includes these main additional features:
- Each multiplier can perform two 16x16 or four 8x8 multiplications every clock cycle.
- The common instructions (AND, ADD, LD, MPY) have a 16 bit version to reduce the code size.
- Hardware support for single-precision (32 bit) and double-precision (64 bit).
- Use advance VLIW to achieve high performance through increased instruction-level parallelism.
- Fully pipelined branches.

### 1.2. Personal Computer

The algorithms have been done in the technical software Matlab R2009b. The computer used to run them is:

- Processor: Intel Core Duo CPU P8600 @ 2.4GHz
- RAM: 4GB
- OS: Windows 7 32bits

All the Matlab results show in this report has been obtained with the computer listed above.
CHAPTER 2. C6EZAccel

One of the main problems of a digital processor system is the memory management. The C6EZAccel, developed by Texas Instruments (TI), is an API that simplifies the management of shared memory between the ARM and DSP.

C6EZAccel consists of a DSP algorithm that follows the algorithm interface XDAIS (eXpress DSP Algorithm Interoperability Standard) of the TI improving the execution performance of the algorithms. The C6EZAccel API abstracts the cache management, address translation and parameter passing of the ARM to DSP.

![Fig. 2. 1 Schematic of the intercommunication between the DSP and ARM [2]](image)

2.1. Advantages and disadvantages

The C6EZAccel is an easy application interface with simple invoking ARM APIs to DSP functionalities. Consequently to this simplicity of use, the learning curve is reduced and the time to market of the application is also reduced.

Texas Instruments also develops optimized libraries for DSP like the Math API, Signal Processing API or even the Image API. Although the users can develop their own libraries, more libraries are expected in a closer future. Moreover, all the APIs and algorithms implemented are independent from the SoC target platform, thus the code is portable to various platforms.

One of the most important advantages is that C6EZAccel allows asynchronous and synchronous modes. The asynchronous mode enables parallel processing
on DSP and ARM, reducing the execution time of the application. As we can see in the Fig. 2. 2 below, the processor load is optimized with asynchronous calls.

![Fig. 2. 2 Processor load of the SoC as function on the running mode [2]](image)

On the other hand, the main disadvantage is the C6EZAccel overhead. The overhead is the additional time included by the C6EZAccel for the interaction between the ARM and the DSP. This additional time comes from:

- Address translation from ARM-side (virtual address) to DSP-side (physical address).
- Transitioning execution from ARM to DSP-side processing.
- ARM side cache invalidation of buffer passed from the ARM to the DSP.
- Invalidating cache of the buffers so the DSP manages the right data. In case of large data buffers, this process is so slow.
- Activating, processing, desactivating the C6EZAccel algorithm on the DSP.
- Writing back the cache of the buffers so the ARM sees the right data.
- Transitioning execution from DSP to ARM-side processing.
- Address translation back from DSP-side physical to ARM-side virtual.

The average time of the steps described before is approximately 6ms. This overhead make no satisfactory performance on small data buffer sizes. The best way to solve this problem is to make a single call to a DSP function with all the parameters needed. Once we are in the DSP side, then we invoke all the functions before come back to the ARM side.
Fig. 2.3 Block Diagram view of calling multiple DSP kernels in C6Accel [2]

The previous schematic (Fig. 2.3) shows two possible ways to run an application from the ARM and DSP. The Application 1 makes “n” C6Accel calls from the ARM to the C6Accel API, adding “n” overhead to the execution time of the application. On the other hand, the Application 2 is more efficient than the Application 1. The Application 2 makes a single call to a DSP kernel and then inside the DSP makes several calls to functions. The waste time for the interaction between the ARM and the DSP is reduced with the Application 2.

In our case, we choose the way of Application 2.

2.2. Devices supported

There are different two-core heterogeneous SoC processors from Texas Instruments that support the C6EZAccel. The table below shows these devices, specifically the ARM+DSP ones which run Linux on the ARM; the yellow one is the selected for the project.
2.3. DSP kernel

The DSP has implemented an algorithm with several kernels ready to be invoked from the ARM with the C6Accel API, but maybe the user needs to create his/her own kernels. Instead of using the implemented ones of TI, we decide to develop most of them.

The C6Accel provides an algorithm with a large number of kernels. The algorithm implements the XDAIS interface (eXpress Dsp Algorithm Interface Standard), helpful to create and delete instances of algorithm. Moreover, the algorithm also implements the iUniversal interface that provides a description of the functions and arguments to be used by the algorithm.
2.3.1. Function IDS

Each kernel of the XDAIS algorithm has a unique function ID that the codec can identify. The list of the functions ID is in the header file iC6Accel_ti.h at the C6Accel directory (ti\sdo\codecs\C6Accel). This function ID is composed by 32 bits:

<table>
<thead>
<tr>
<th>Vendor Field for customer added custom kernel</th>
<th>Field to identify type of DSP function</th>
<th>Field for function ID for kernel to be executed on DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 2. 4 Structure of an ID function [3]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first field (8 bits) is used to identify the vendor ID that in the case of TI is 0x01. The next field (8 bits) identifies the category of function being executed as the Table 2. 2 below indicates.

Table 2. 2 Codification of the category of function [3]

<table>
<thead>
<tr>
<th>XX01XXXX</th>
<th>DSP Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX02XXXX</td>
<td>IMAGE Library</td>
</tr>
<tr>
<td>XX03XXXX</td>
<td>MATH Library</td>
</tr>
<tr>
<td>XX04XXXX</td>
<td>ANALYTICS Library</td>
</tr>
<tr>
<td>XX05XXXX</td>
<td>MEDICAL Library</td>
</tr>
<tr>
<td>XX06XXXX</td>
<td>POWER CONTROL Library</td>
</tr>
<tr>
<td>XX07XXXX</td>
<td>AUDIO SPEECH Library</td>
</tr>
</tbody>
</table>

The third field (16 bits) is reserved to identify the kernel that is to be executed on the DSP.

2.3.2. Extended Input Arguments

The extended input arguments structure carries information such as the number of functions being called in that particular API call and an array of function structures listed below.

```c
iC6ACCEL_InArgs(
    Int size;
    Int Num_fxns;
    FXN_struct fxn[MAX_FXN_CALLS];
) CODEC_InArgs;
```
struct FXN_struct{
    XDAS_Int32 Fxn_ID;
    Int Param_ptr_offset;
}FXN_struct;

Each one of the function has its own parameter structure defined in the iC6ACCEL_t.h interface header file. An example of a parameter structure is shown below where InArrID1 and OutArrID1 are integer buffers and Col is an integer scalar.

struct IMG_median_3x3_8_Params{
    unsigned int InArrID1;
    unsigned int OutArrID1;
    int Col;
}IMG_median_3x3_8_Params;

All vectors to be passed to the functions are passed using the inBuffs and outBuffs of the Universal process API calls. Each function parameter structure contains IDs that are responsible to identify the correspondence between the inBuffs and outBuffs shared and the function argument.

### 2.3.3. Memory management

As we said before, the application code runs on the ARM while the codecs run on the DSP. The DSP and ARM exchange information using a shared memory in the DDR2 defined in the CMEM module.

The DSP processes buffers and data aligned contiguously in memory while the ARM can work on fragmented buffers thanks to its MMU (Memory Management Unit).

The best way to create a aligned contiguously memory is using the function Memory_alloc() provided by TI. The function defines memory blocks on the memory region defined in the CMEM module (by default the start address is 0x96C00000 and a maximum size of 20MB). If the programmer uses other functions to reserve memory, as for example “malloc”, the C6Accel could not be able to translate between the physical to virtual address, making impossible the interaction between ARM and DSP [5].
2.4. Using C6EZAccel in an ARM application

The implementation of a new DSP function involves different files as the schematic Fig. 2. 5 shows. In this section, we briefly explain the content of each file. We do not include any one of the files where the functions ID are defined. In the Appendix A is included all the steps that the user has to follow to success when a new DSP function is defined.

![Diagram showing the different files and their contents](image)

**Fig. 2. 5 Schematic of a DSP function call [3] [4]**

The appmain.c file is the responsible to allocate in memory all the buffers, to create the c6accel handle and of course to invoke the functions. The main code cannot invoke a DSP function directly. We need to call what we call “C6accel function”. They are defined in the c6accelw.c

The c6accelw.c file contents all the C6Accel functions. A C6Accel function adapts all the parameters to fit in the iUniversal framework implemented in the XDAIS. The C6Accel functions always have the same structure: the c6accel handle, input variables and output variables. The input variables are encapsulated in the input buffers while the output variables in the output buffers. There are up to fifteen input buffers and others fifteen for the output buffers. All the buffers need to be defined and assigned to one of these fifteen. Even if we want to return a single value, we need to spend a buffer of one position. Once we define the buffers, we must assign the function ID and call the universal_process (c6accel.c).
The c6accel.c file includes the function universal_process that is the one that discerns between the different libraries. The algorithm applies a mask to the function ID to determine which one of the libraries should call.

In the library file (c6accel_stauplib.c), we apply another mask to determine the function to call. All the parameters are checked before we invoke the DSP function.

The DSP function file (Function1.c) runs the entire implemented algorithm. The DSP function must be defined as void, they cannot return a value. All the C6accel functions returns an error code, helpful to determinate what is the cause of a possible error.

- **_EOK (0):** No error happens.
- **_EFAIL (-1):** Error appears when invoke the Codec Engine interface. This error often is caused if the buffers/vectors being passed to the codec are not assigned as contiguous memory. It can also occur if the application makes an asynchronous call to C6EZAccel when there is already a pending asynchronous call.
- **_PARAMFAIL (-6):** Error due to invalid parameters. This error occurs when the parameters passed do not satisfy the parameter specifications of the underlying kernel.
- **_FXNIDFAIL (-7):** Error on function ID passed. This error appears when the application passes a wrong function ID to the codec. In such case, please verify if the function ID being passed that is defined in the application codec interface header file iC6accel_ti.h.

If the DSP returns a value different from zero, the C6Accel will understand that an error has happened and it will not allocate the data in the buffers.
CHAPTER 3. TECHNIQUES TO REDUCE THE EXECUTION TIME

There are different techniques to optimize your algorithms and reduce the execution time of the application.

First of all we explain techniques to be applied in the algorithms that are useful in all the architectures (not only for the C6000 family). Each one of them has a numeric example to show the real effect when they are applied in an algorithm. A second block of techniques relative how the libraries and intrinsic instructions of TI could reduce the execution time. The software pipelining concept is also explained to improve the efficiency of the application.

The chapter is concluded with a description of the compiler used by TI and which is the best "level of optimization" option to achieve the best performance.

3.1. Algorithms

One of the first optimizations in a system is in the manner that the algorithms are implemented. The strategies presented below are efficient when the algorithms are invoked so many times, as for example to manipulate an image of 752x480 (360.960 pixels) stored in an array of 8 bits.

- Use the properly size of variables for each case. The programmer often declares an integer variable as int (32b), neglecting if a short (16b) or a char (8b) variable is enough. The Table 3.1 shows the required clock cycle by the .M for each operation. For example if we want to implement an algorithm that multiplies the pixel value by a constant, we obtain the next execution times (only is taken into account the multiply operation):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operations/Clock cycles</th>
<th>Execution time of the example (f=500MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32bx32b</td>
<td>1</td>
<td>721,92 us</td>
</tr>
<tr>
<td>16bx32b</td>
<td>1</td>
<td>721,92 us</td>
</tr>
<tr>
<td>16bx16b</td>
<td>2</td>
<td>360,96 us</td>
</tr>
<tr>
<td>8bx8b</td>
<td>4</td>
<td>180,48 us</td>
</tr>
</tbody>
</table>
• Reutilization of common expressions and avoid unnecessary intermediate variables like the example below shows:

\[
\begin{align*}
  \text{CODE A} & : & a &= b+c \\
  & & d &= b+c-e \\
  & & f &= d+e \\

  \text{CODE B} & : & a &= b+c \\
  & & d &= a-e \\
  & & f &= a
\end{align*}
\]

In the CODE A, the CPU executes four adds and three assignments or loads. On the other hand, in the CODE B, the CPU only executes two adds and two loads (we can exchange f to a in CODE B). If this piece of code is executed in each pixel of our image (752x480), the CODE B executes 721,920 adds less than CODE A and 360,960 loads. If all the variables are char (int8) and taking into account that in one cycle of clock the system can add up to 4 operations and the DSP is running with a frequency of 500MHz then:

Cycles of clock A\(=\frac{4\times 752\times 480}{4} + \frac{3\times 752\times 480}{4}\)\(=1\,443\,840 \rightarrow 2.88\,ms\)

Cycles of clock B\(=\frac{2\times 752\times 480}{4} + \frac{2\times 752\times 480}{4}\)\(=902\,400 \rightarrow 1.80\,ms\)

• Reorganize the operands of an operation. Mathematically, there is no difference between the CODE A and CODE B below. If we look careful what the system does in both blocks some differences appears.

\[
\text{CODE A} \quad a = b\times c\times (d+e)
\]

The “b” variable is moved to a register and then is multiplied by the “c” variable. The result is storage in a temporarily register. Then “d” is loaded into a register and added to “e”. The temporarily register is multiplied by the result of the add between “d” and “e”. The last step is to copy the result of this multiplication to the “a” variable.

\[
\text{CODE B} \quad a = (d+e)\times b\times c
\]

First “d” is loaded into a register and then is added to “e”. The result of the adding is multiplied by “b” and the result of this operation is multiplied by “c”. The last step is to copy the result to the “a” variable.

The CODE A needs four loads to a register, two multiplies and one addition. On the other hand, CODE B needs only two loads, two multiplies and one addition.
• Change multiplications/division by adds and shifts as for example:

\[
\begin{align*}
2^*x & \rightarrow x+x \text{ or } x<<1 \\
4^*x, 8^*x & \rightarrow x<<2, x<<3 \\
x/4 & \rightarrow x>>2
\end{align*}
\]

• Extract the maximum operations from inside the loop to outside it as the example below shows:

```
for (i=0; i<cols; i++)
  for (j=0; j<row; j++)
    array[i*row+cols]=...;
```

CODE A

```
for (i=0; i<cols; i+=row)
  for (j=0; j<row; j++)
    array[i+cols]=...;
```

CODE B

In the CODE A, we increment the “i” variable by 1 in each iteration. Then we multiple this variable by the number of rows when we access into the cell array. On the other hand, CODE B is more efficient because it avoid a multiply in each iteration, incrementing “i” by row instead of by 1.

For example this technique was used in the derivative filter (explained at 4.2.2.1.3) obtaining an important reduction of the execution time from 5,4 ms to 0,2 ms.

• Hardcoded functions. The typical function in digital processing is the convolution between a signal and a filter. Usually the programmer develops a function where the signal and filter are passed as parameters. Using this strategy, the programmer obtains a flexible function that could be invoked in different system. Otherwise, this is not the optimized one.

In a typical convolution function, the system has to access to the memory and load each cell of both vectors (signal and filter). The best option, If the coefficients of the filter are fixed and its length is not too big, is to use define directives instead of a vector as the example below shows.

In code A at the main code, we declare a buffer called “filter” of 5 positions and we invoke the convolution function (conv). This function needs the input buffer or also called signal, the filter and the output as parameters. Inside the function, the system has to load the cell of the signal and the filter for each iteration and then load this result to the corresponding cell of the output.
On the other hand, in code B we do not need to declare a filter at the main code. We invoke the convolution function (conv) which only requires the input buffer and the output buffer. Inside the function, the system has only to load the cells of the input buffer. The coefficients of the filter are hardcoded inside the function.

Code A is flexible while Code B spends less execution time.

```
CODE A

short filter[5]={12,4,6,4,12};
void conv (short *s, short *f, short *o)
{
    ....
    for(i=x;i<len;i++)
        o[i]=s[i-2]*f[0]+ s[i-1]*f[1]+....s[i+2]*f[4];
    ....
}

CODE B

void conv (short *s, short *o)
{
    ....
    for(i=x;i<len;i++)
        o[i]=s[i-2]*F0+ s[i-1]*F1+.... s[i+2]*F4;
    ....
}
```

### 3.2. TI DSP Libraries and intrinsic operations

Texas Instruments provides highly optimized DSP software libraries for the C6000 family of devices. These libraries incorporate many standard math, image or audio functions. The programmer could use these optimized libraries, reducing the execution time and the developing time too.

On the hand, the C6000 family has intrinsic operations that are useful to optimize the code, allowing the programmer to directly invoke C600 assembly operations. Some examples of the intrinsic operations are presented below:

- `amem2, amem4, amem8` → Loads and stores 2/4/8 bytes of memory.
- `mpy, mpyus, mpysu` → Multiplies the 16 LSBs of one variable by the 16 LSBs of a second variable.
The Appendix B shows the whole list of the intrinsic instructions of the C6000 family of devices. [9]

3.3. Software Pipelining

The software pipelining is a technique used to optimize loops so that multiple iterations of the loop are executed in parallel.

In the lineal execution, an instruction waits until the previous one has finished. Once all the instructions of an iteration are done in a loop, then the next iteration can start to execute again the routines. An example of lineal execution is presented at Fig. 3.1 (a).

In the pipeline execution, the instructions of a loop are arranged so that once the first instruction of the iteration 1 is done, the system starts to execute in parallel the second instruction of the iteration 1 and the first instruction of the iteration 2. The Fig. 3.1 (b) shows an example of pipeline execution.

In our example, each iteration of the loop is composed by five instructions (A, B, C, D, E) and we consider that one instruction spends one clock cycle. In the lineal execution we need 25 clock cycles to do five iterations of the loop. Otherwise, the pipeline execution only spends 9 clock cycles.

![Fig. 3.1 Comparison of lineal execution (a) and pipeline execution (b)](chart.png)

The pipeline can introduce delay slots. The cycles that the CPU has to wait on are the delay slots of the instructions; for example a load instruction spends four cycles of clock while a single precision floating point multiplication spends three
cycles of clock. When is executed the load instruction, the CPU has to wait four cycles of clock until it is completed to start the next instruction. Also this technique is not useful when a current instruction is dependent on the result of a previous iteration. (e.g A2 was dependent of the result of E1).

3.4. Compiler

The compiler is the responsible to translate the C code into a machine language. The powerful the compiler is, the more optimized is the code. The compiler used has different options to achieve a better level of optimization.

The optimization level (–opt_level) which can get a value between 0 and 3. Each one of the optimization level is explained below.

- **-opt_level=0 (-o0)**
  - Allocates variables to registers.
  - Eliminates unused code.
  - Simplifies expressions and statements.

- **-opt_level=1 (-o1)**
  - Performs all the –o0 optimizations.
  - Performs local copy/constant propagation.
  - Removes unused assignments.
  - Eliminates local common expressions.

- **-opt_level=2 (-o2)**
  - Performs all the –o0 and –o1 optimizations.
  - Performs software pipelining.
  - Performs loop optimizations.
  - Eliminates global unused assignments.
  - Performs loops unrolling.

- **-opt_level=3 (-o3)**
  - Performs all the –o0, –o1 and –o2 optimizations.
  - Removes all functions that are never called.
  - Simplifies functions with return values that are never used.
  - Reorders functions declarations.

We use the third level of optimization to achieve the better performance. [7][10]
CHAPTER 4. STEREOSCOPIC CAMERA SYSTEM VISION

In chapter 4, the stereoscopic camera vision system is explained with a briefly description of each block. The reader will appreciate how the system works synthetically, without entering into technical knowledge or large descriptions.

As we said at the beginning, the purpose of the project is to migrate algorithms from a Matlab environment to an embedded environment. These algorithms have not been implemented in Matlab by the author of the report. The author is the responsible to the optimization since the images are acquired until the search candidates block.

4.1. Stereo vision camera vision system

A stereoscopic camera system is a system conformed by two cameras capable of measure the distance to an object using their images. The cameras are displaced horizontally from one another and they are used to obtain two views on a scene like the human vision like the Fig. 4. 1 below shows.

Knowing the separation between them and processing the images, we can obtain an exact distance to an object. In this project, if the stereoscopic camera vision system detects any kind of object in a specific region, a buzzer alarm and/or a led will be activated.
We call the left camera as master and the right camera as slave. We can see in the Fig. 4. 2 below an example of a raw image got from the master camera (a) and other from the slave camera (b). The resolution of the image is 752*480.

![Images from the master and slave cameras](image)

4.2. Block diagram

The block diagram of the system is shown at Fig. 4. 3 that synthesize the different processing steps required to determinate the distance to an object.

![Block diagram of the whole stereoscopic camera system](image)

The images from the master camera and slave camera suffer a biconvex distortion due to the lens that needs to be corrected. The block called transforma gets the image and corrects the distortion of the lenses.
Then the block search looks for the edges in the image. Continuously, we find the search candidates that look for the common edges in both images generating a disparity map. The last block, disparity analysis, processes the disparity map and generates a signal alarm if an object is found inside a region.

Each one of the blocks is described below.

### 4.2.1. Transforma

The images taken from the camera has a biconvex distortion that needs to be corrected. The phenomenon is shown in the Fig. 4.4 below. In a biconvex distortion, the theory straight lines are curbed. The distortion is corrected using generated maps. The generated maps are individually for each camera, so we have a map for the master and another one for the slave.

![Correction of a biconvex distortion](image)

Briefly explaining, the pixel map indicates where a pixel from the original image has to be allocated in the final image without distortion. The original pixel is multiplied by some constants and then is moved to its new position in the final image.

The Fig. 4.5 below shows an example of how a biconvex distortion is corrected thanks to the transforma block. In the image acquired directly to the camera, the straight lines are curved (a), but after the transforma block these lines would become straight (b). The black areas are due to that zones are not pointed for the pixel map.

The output image of the transforma block keeps the same size of the distortioned image of the camera (752x480).
Migration of a stereoscopic camera system from Matlab to the TMS320DM814x Davinci platform

4.2.2. Block search

The block search is composed by the edge search block and a linear interpolation. This block is executed twice in every loop, once for the master image and other one for the slave image. In the next subsections a description of both blocks can be found.

Fig. 4. 6 Block diagram of the Block Search
4.2.2.1.  Edge search

The edge search block is the responsible block to find the edges of the image. An edge is detected when there is change in the saturation value. In the Fig. 4. 7 below, the reader could see the block diagram of the edge search composed by a Gaussian filter with line decimation, a spline interpolation and a double derivative block. At the end, the logic block generates an image with the edges.

![Fig. 4. 7 Block diagram of the Edge Search](image)

4.2.2.1.1.  Gaussian filter

A Gaussian filter is a filter that smoothes an image by a Gaussian function. In image processing is used to reduce the image noise and reduce the details. The visual effect of this technique is a small smooth blur of the image.

Mathematically, a Gaussian filter is a convolution of the image with a Gaussian function and acts as a low pass filter, reducing the image’s high frequency components. The equation of a Gaussian function in two dimensions is at (4.1) equation.

\[
G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2 + y^2}{2\sigma^2}}
\]

(4.1)

Once the standard deviation is fixed, the coefficients of the filter can be found easily. The dimensions of the filter implemented are 3x3. Moreover, we decimate the lines of the image by a factor of 5 to reduce the computational cost.

The resolution of the output image is 752x96. The Fig. 4. 8 shows the effects of the Gaussian filter.
4.2.2.1.2. Spline interpolation

To detect the edge is needed to know exactly where the transition is done. Thus, we need extra pixels to determine when this transition is done thanks to the next two blocks (derivative filter).

Instead of the linear interpolation, we use the cubic spline interpolation. The benefits of this interpolation resides that the value of the new pixel is determinated by the ten neighbors pixels of its line. The value of the pixel is obtained through the balanced value of these ten pixels.

The new size of the output image is 2254x96.

4.2.2.1.3. Derivative filter

The derivative filter is used to know exactly the pixel transition. We concatenate two derivative filters. The first derivative is useful to know the maximums and minimums. If we apply again a derivative filter over the output of the first derivative filter, knowing it as second derivative, we are able to detect inflexion points.

4.2.2.1.4. Logic block

The logic block gets the first derivative filter and the second derivative filter to generate an image with the edges. We apply a filter to each one of the derivative signals (the 1st and the 2nd derivative). If both outputs of these filters are higher than a threshold, the pixel of the output image gets the value of the one from the first derivative filter. If not, the value of the pixel is zero.

The size of the image remains equal to the previous blocks (2254x96).
4.2.2.2. **Lineal interpolation**

A lineal interpolation of the original image after the correction of the distortion is needed to validate the common edges of the master and slave. We decide to use a lineal interpolation instead of the spline interpolation as we did before.

![Fig. 4. 9 Output image of the Lineal Interpolation](image)

The pixel value of the output image is generated through the two pixels close to this one. The new image (displayed in Fig. 4. 9) has a resolution of 2254x480.

4.2.3. **Search of candidates**

Once we applied the block search in both master and slave image, we analyze their outputs in the Search of Candidates block. Briefly explain this block looks for common edges of both images and stores information as the distance of a first and a second closer maximum of an edge and the value of this maximum or also called disparity.

The Fig. 4. 10 shows the disparity image from the disparity map of the original image got it from the camera. As we can see the railways are well detected and the distance to the objects is encoded with the pixel value. The further is the object, the darkness is the pixel.
4.2.4. Disparity analysis

The disparity analysis is done to determine if an object is inside a region and activate an alarm. The cameras are allocated in a height of 2.5 m, so the point of view is not parallel to the ground plane. The first step to do in this block is to find the ground plane. It will allow us to change the cameras point of view through image processing.

Once the perspective is changed, we can define a zone of interest of this image and process if this box is empty or not. If the number of found edges inside the box is higher than a threshold, an alarm will be generated.

Fig. 4. 10  Disparity image generated with the disparity map.
CHAPTER 5. RESULTS OF THE MIGRATION

The results of the implementation from Matlab to the embedded system are presented below. First of all, we provide an example of the evolution of the time execution of a function. The reader could see the importance of the implemented techniques in an algorithm. Then the final results are presented.

5.1. Evolution of the execution time in an algorithm

The most common function in signal processing is the convolution between two signals. Following this criteria, we choose the Gaussian filter algorithm as example of how the execution time can be decreased. The Gaussian filter algorithm, as the other functions, has been implemented first in Matlab. Matlab is a powerful mathematical software that can operate arrays easily. It is able to generate a C code from the Matlab code. However, this code is not efficient and the best mode to achieve the optimization is writing again algorithm in C.

The size of the image has changed while the report of the master thesis was written. These modifications are done to obtain a better performance of the system. The department responsible for the development of the algorithms in Matlab decides that the best resolution is 752x480. However, the results shown in this section are obtained with the same size of the image: 1153x923.

The Fig. 5. 1 displays the evolution of the execution time in the Gaussian filter. We can appreciate that the C code generated by Matlab spends 192,7 ms. The code generated by Matlab has some features that makes it not suitable for our application:

- Declares all the variables as double (64 bits).
- Makes a copy of the array to be sure that the size of the array is the required one and avoid memory access problems.
- Checks the dividend in each division to avoid a division by zero.

Once we see the poor efficiency of the C code of Matlab, we decide to write again the algorithm. We start the implementation of the Gaussian filter in the ARM side where the function has as parameters the input image, the filter and the output image. We obtain a considerable reduction from 192,7 ms to 154 ms. As the coefficients of the filter are constant, we decide to hardcoded them. This technique reduces the spend time in memory access.

At this point the arrays are defined as integer 32b (int) when the maximum possible value after the filter fits in integer 16b (short). Using the properly size of the variables, the execution time is reduced until 68,8 ms.
After that we implement the function in the DSP side obtaining an important reduction of the spend time by the function. Moreover if we use compiler intrinsic instructions, the execution time decreases until 18,2 ms.

![GAUSSIAN FILTER](image)

**Fig. 5. 1 Evolution of the execution time**

The optimization level achieved until the “Assembler” point (18,2 ms) respect to the first version of the algorithm “C Matlab” (192,7) is about 91%. The Table 5. 1 presents the percentage of optimization in each step respect to the C code generated by Matlab.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>19,82%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard code filter</td>
<td>25,22%</td>
</tr>
<tr>
<td>Int 16</td>
<td>64,30%</td>
</tr>
<tr>
<td>DSP</td>
<td>89,88%</td>
</tr>
<tr>
<td>Assembler</td>
<td>90,56%</td>
</tr>
</tbody>
</table>

The Gaussian filter is concatenated with decimation, so we decide to implement the decimation inside the Gaussian filter. At the end, the execution time of the Gaussian filter is 8,7 ms. Notice that the Matlab code has not implemented the decimation inside the algorithm.
5.2. Errors due to the migration

The fact to migrate from the Matlab environment to the TMS320DM814x causes some errors. The Matlab code rounds the float values while the system truncates it. It provokes a number of erroneous pixels that need to be checked to determine if the algorithm is affected or not. Although all the buffers and variables are defined as integers, there are operations that can introduce errors like the divisions or shifts.

We have calculated the error on each block processig the same input buffer in Matlab and in the embedded system. We get the output buffer for each function implemented in the system and then we compare to the Matlab output.

We define an erroneous pixel as the pixel that its value in Matlab differs from more than one its value in the TMS320DM814x. The Table 5. 2 presents the percentage of error in each function. We can observe that the error in each function alone is negligible, but if we concatenate the different blocks the error rises to 2 %. However the system is unaffected by this percentage of error.

<table>
<thead>
<tr>
<th>Function</th>
<th>Nº of pixels</th>
<th>Erroneous pixels</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss Filter</td>
<td>72.192</td>
<td>0</td>
<td>0,0%</td>
</tr>
<tr>
<td>Interp. Horizontal</td>
<td>216.384</td>
<td>30</td>
<td>0,014%</td>
</tr>
<tr>
<td>DX filter</td>
<td>216.384</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Arith. Block</td>
<td>216.384</td>
<td>303</td>
<td>0,140%</td>
</tr>
<tr>
<td>Block search</td>
<td>216.384</td>
<td>4.370</td>
<td>2,020%</td>
</tr>
<tr>
<td>Interp. Full</td>
<td>1.081.920</td>
<td>16</td>
<td>0,001%</td>
</tr>
</tbody>
</table>

5.3. Final results

The execution time of each function of the block search (Gauss filter, interp. Horizontal, DX filter, Arith. Block) and the interpola full is detailed in the Table 5. 3. We can see the different development stages, from the Matlab environment to the DSP. The yellow cells indicate which side of the system (ARM or DSP) is selected to run each function. The average optimization level obtained is above 93% respect to the C Matlab code.
Table 5.3 Execution time of each function in the different stages

<table>
<thead>
<tr>
<th>Function</th>
<th>Matlab (ms)</th>
<th>C Matlab (ms)</th>
<th>ARM (ms)</th>
<th>DSP (ms)</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transforma</td>
<td>1.536</td>
<td>483.5</td>
<td>35,41</td>
<td>36.2</td>
<td>93.09%</td>
</tr>
<tr>
<td>Gauss filter</td>
<td>252</td>
<td>69.4</td>
<td>5.8</td>
<td>3.1</td>
<td>95.53%</td>
</tr>
<tr>
<td>Interp. Horizontal</td>
<td>621.20</td>
<td>107.2</td>
<td>6.9</td>
<td>4.7</td>
<td>95.62%</td>
</tr>
<tr>
<td>DX filter</td>
<td>920.3</td>
<td>10.1</td>
<td>5.4</td>
<td>0.2</td>
<td>98.02%</td>
</tr>
<tr>
<td>Arith. Block</td>
<td>394</td>
<td>32.6</td>
<td>14.02</td>
<td>1.5</td>
<td>95.40%</td>
</tr>
<tr>
<td>Interp. Full</td>
<td>3.360</td>
<td>80.9</td>
<td>9.4</td>
<td>15.3</td>
<td>81.09%</td>
</tr>
</tbody>
</table>

The call sequence is composed by three blocks (see Table 5.4) that are invoked twice. As we said in the Chapter 2, the C6Accel adds an overhead in each call to the DSP. When we run the block search in the DSP, we have to add an overhead of 5.8ms to the execution time.

Table 5.4 Functions on the call sequence

<table>
<thead>
<tr>
<th>Function</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transforma</td>
<td>35.41 (ARM)</td>
</tr>
<tr>
<td>Block Search</td>
<td>15.5 (DSP)</td>
</tr>
<tr>
<td>Interp. full</td>
<td>9.4 (ARM)</td>
</tr>
</tbody>
</table>

Once we choose the running side for each function, we can decide the strategy to obtain the maximum performance. As we comment in the chapter 2, the C6Accel allows a parallel working of both sides of the system. We execute the block search (gauss filter, interpolation horizontal, DX filter and arithmetic block) in the DSP and the interpolation full in the ARM.

There are three different options of call sequence for our application as the Fig. 5.2 shows. The first two sequences (a, b) execute first the algorithm of the master camera (orange) and wait until it has finished to start the algorithm of the slave camera (blue). The main difference between them is that in a) the system is set to work in synchronous mode, while b) in asynchronous mode. In synchronous mode, the ARM waits until the DSP has finished its function. On the other hand, in asynchronous mode, the ARM and DSP run in parallel. The third option is to combine the slave and master image algorithm to obtain the best optimization in the call sequence.

The third option call sequence starts acquiring the images of both cameras and correcting the camera distortion of the master. After that, we invoke the block search of the master that runs in the DSP and the transforma function of the slave at the same time. The system is set to wait until the DSP finishes its algorithm to start the next block: interpola full of the master. After that at the same time, the interpola full of the master and the block search of the slave...
image starts. The interpola full of the slave image is also executed while the block search of the slave image is running.

![Diagram](image)

**Fig. 5. 2 Call sequence of the functions**

The faster call sequence is the third option, as we can observe in Table 5. 5, with an execution time of 94,7 ms. This sequence can be done because the block search and the interpola full are independent functions.

**Table 5. 5 Execution of each call sequence option**

<table>
<thead>
<tr>
<th>Call sequence</th>
<th>Execution time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option a)</td>
<td>120,62</td>
</tr>
<tr>
<td>Option b)</td>
<td>101,57</td>
</tr>
<tr>
<td>Option c)</td>
<td>94,7</td>
</tr>
</tbody>
</table>

The running time of the system with the C Matlab in the ARM side for the call sequence is 1,587 s (793,8 ms for each camera). On the other hand, after the optimization, the execution time of the system is only 94,7 ms for both cameras. The level of optimization achieved is 94,03%. If we compare with the Matlab environment (8,003 s for each camera), the level optimization achieved is 98,81%.
CHAPTER 6. CONCLUSIONS

6.1. Reached targets

The target of the project has been the migration from the Matlab world to an embedded system of a stereoscopic camera system. This implementation has been done with the TMS320DM814x evaluation board of Texas Instruments. We have decided to use this board for the framework agreement between Texas Instruments and IDNEO. The board is equipped with an ARM and DSP that are easily intercommunicate through the C6Accel API. This API allows the memory share between them, parallel working and creates an easy codec interface to call functions running in the DSP from the ARM.

The main problem of an image processing system is the execution time of the system. The idea of the Argus project is provide a safety tool to the railway employees. The execution time has been reduced with the different methods presented in the report. The methods used algorithms techniques that try to minimize the operations or memory accesses in each iteration of the loop. Others important tools to optimize the system are the software pipeline that is provided by the C6Accel and the powerful compiler that reduce and reorganize the code.

The objective of the master thesis was to reduce the execution time since the images are captured to the edges are detected. The C code generated by Matlab spends 1,587 s, while the code implemented by the author spends only 94,7 ms, that means an optimization of 94,03%. If we compare with the execution time in Matlab (8.003 ms), the optimization level increases up to 98,8%.

The problems found during the master thesis have been solved with the TI literature ([7], [8], [9], [10]) and the TI forum ([6]). The main problems are related to the memory management and the interaction between the DSP and ARM.

6.2. Future work lines

A huge part of the code has been optimized but there is still code to optimize. Nowadays we are working on the optimization of the other algorithms to achieve a real time stereoscopic camera system. Now the system is able to detect and generate an alarm (buzzer and/or a led) if an object is placed in a determined zone. In a closer future, the Argus will split the detection zone in three zones to discern better the dangerous level. We are also working in the objects classification to determine if the object detected is a person or not.

The last step will be the implementation of the system in a passengers train.
REFERENCES

[1] Features of the TMS320DM814x


[4] ARM user documentation
URL: http://processors.wiki.ti.com/index.php/C6Accel_ARM_user_Documentation

[5] Default memory map
URL: http://processors.wiki.ti.com/index.php/EZSDK_Memory_Map

URL: http://e2e.ti.com/

[7] TMS320C6000 Optimizing Compiler

[8] TMS320C64x+DSP Image/Video Processing Library

[9] TMS320C674x DSP CPU and Instruction Set
URL: http://www.ti.com/lit/ug/sprufe8b/sprufe8b.pdf

[10] Introduction to TMS320C6000 DSP Optimization
URL: http://www.ti.com/lit/an/sprabf2/sprabf2.pdf
APPENDIX A. IMPLEMENTATION OF THE C6ACCEL IN THE MAIN CODE

A.1. ARM main code

When a user wants to integrate the C6EZAccel in his/her ARM application should follow some steps or rules.

- First of all, we have to include in the Appmain all the files needed for the Codec Engine:

```c
#include <ti/sdo/ce/Engine.h>
#include <ti/sdo/ce/CERuntime.h>
#include <ti/sdo/ce/osal/Memory.h>
```

- The next step is to include the C6EZAccel application codec header file.

```c
#include "../c6accelw/c6accelw.h"
```

- The user has to declare a C6accel Handle

```c
C6accel_Handle hc6accel=NULL;
```

The C6accel_handle is a handle to the C6Accel Object defined as:

```c
C6accel_Object{
    Engine_Handle hEngine;
    UNIVERSAL_Handle hUni;
    E_CALL_TYPE callType;
} C6accel_Object;
```

The object carries the Engine Handle and the iUniversal Handle required for the current instance. The parameter E_CALL_TYPE defines the working mode of the device. If the parameter takes the value ASYNC, the ARM and DSP will work simultaneously. In the other hand, if it takes the value SYNC, the ARM will wait until the DSP finished its processes.
The engine name and alg name has to be defined with the same name as we configured the .cfg file. The default name by the alg name is c6accel.

```
#define ENGINENAME "ti8148"
#define ALGNAME "c6accel"
```

The follow instruction initializes the codec engine runtime.

```
CE_Runtime_init();
```

After the initialization of the codec engine runtime, the user generates the C6accel handle.

```
hC6accel=C6accel_create(engineName, NULL, algName, NULL);
```

The arguments of the function are described below:

1. engName= Engine Name
2. hEngine= Engine Handle
3. algname=Algorithm name
4. hUniversal= Universal handle

The function returns a C6Accel Handle from the Engine Handle and Universal handle.

At this point, the user can make calls to kernels in the codec using API calls and develop its own functions or kernels. After the main code, the user has to delete the c6accel handle and free all the buffers created in it.

```
if(hC6)
    c6accel_delete(hC6);
```
We attach an example of a main code:

```c
/* CODEC ENGINE INCLUDES */
#include <ti/sdo-ce/Engine.h>
#include <ti/sdo-ce/CERuntime.h>
#include <ti/sdo-ce/osal/Memory.h>

/* SYSTEM TIME INCLUDE */
#include <sys/time.h>

/* C6ACCEL AND LIBRARIES INCLUDE */
#include "../c6accelw/c6accelw.h"
#include "../../../src/C6accel_ti_daplibFunctionCall.h"
#include "../../../src/C6accel_ti_mathlibFunctionCall.h"
#include "../../../src/C6accel_ti_opencvlibFunctionCall.h"
#include "../../../src/C6accel_ti_imglibFunctionCall.h"

/* DECLARE A C6ACCEL HANDLE */
C6accel_Handle hC6accel = NULL;

/*DEFINE OF THE ENGINENAME AND ALGNAME*/
#define ENGINENAME "ti8148"
#define ALGNAME "c6accel"

Int main (Int argc, char *argv[])
{
    /* CREATE THE MEMORY PARAMS */
    Memory_AllocParams memParams = Memory_DEFAULTPARAMS;

    /* CODEC ENGINE INITIALIZATION */
    CE_Runtime_init();

    /* GENERATE THE C6ACCEL HANDLE */
    hC6accel = C6accel_create(ENGINENAME, NULL, ALGNAME, NULL);

    /*DEFINE A CACHED AND CONTIGUOUS MEMORY TO GET THE BEST PERFORMANCE */
    memParams.flags = Memory_CACHED;
    memParams.type = Memory_CONTIGHEAP;

    /* CREATE THE BUFFERS FOR USE BY ALGORITHMS */
    Buffer1 = Memory_alloc(framesize, &memParams);
    if (Buffer1 == NULL)
        goto end;
    else
        Memory_cacheWbInv(Buffer1, framesize);

    /* CHECK FOR FAILURE */
    if(hC6==NULL)
        goto end;

    /****************************************************/
    /* USER CODE WITH THE CALLING DSP FUNCTION */
    /****************************************************/
```
A.2. Implementing a new DSP kernel

We will explain how to create your own DSP kernel with an example step by step. The example kernel is a function that multiplies two arrays element by element and save the result in a third array.

1. First of all we write the algorithm of the function. In our case:

```c
void mult2array(short * array1,
                 short *array2,
                 short n_elements,
                 short *result){
    short i=0;
    for(i=0;i<n_elements;i++)
        result[i]=array1[i]*array2[i];
}
```

The code has to be included in the DSP source files (/dsp/alg/src) with the name mult2array.c

2. Once the code is implemented, the user needs to define a unique function ID. We declare an ID to our function in the next two files:

   a. iC6accel_ti.h: In this file is set the main function ID and the parameters of the function. We define the function ID as:

```
#define MATH_MULT2ARRAY 0x02030001
```

Taking into account the Table 2.2, we have defined our function with an independent vendor code of TI and as a
math function. Then we also have to define the parameters of the function as list below:

```c
typedef struct DSP_mult2array_Params
{
    unsigned int array1_InArrID1;
    unsigned int array2_InArrID2;
    int nelements;
    unsigned int result_OutArrID1
}DSP_mult2array_Params;
```

The buffers are defined as a pointer to its first memory position. The memory address is composed of 16 bits.

b. C6accel.h: The file must main consistency with the function ID of the a) file. We set the function ID as:

```c
#define F_MULT2ARRAY 0x00000001
```

The function defined as MATH_MULT2ARRAY ID is used to determine the library. Once inside the library, the F_MULT2ARRAY ID is needed to determine which function has to invoke.

3. The next step is to insert the case statement for the kernel. We include the case statement in the math library (C6accel_ti_mathFunctionCall.c).

```c
int C6ACCEL_TL_mathlibFunctionCall(void *pFnArray,int fxnID,XDM1_BufDesc *inBufs, XDM1_BufDesc *outBufs)
{
    Int32 fxnidMsk= FXN_ID_MASK;

    // MATHLIB functions

    switch ( fxnID & fxnidMsk)
    {
        case(FUNCTION1_FXN_ID):
        {
            /* ………………*/
        }
        break;

        case(F_MULT2ARRAY_FXN_ID):
        {
```
// Create the parameter structure of the function
DSP_mult2array_Params *DSP_mult2array_paramPtr;
DSP_mult2array_paramPtr = pFnArray;

// Check if the parameters are correct
If( (DSP_mult2array_paramPtr->array1_InArrID1>INBUF15)
   | ( DSP_mult2array_paramPtr->array2_InArrID2>INBUF15)
   | ( DSP_mult2array_paramPtr->nelements) < 0
   | ( DSP_mult2array_paramPtr->result_OutArrID1)>OUTBUF15))
   return(UNIVERSAL_EPARAMFAIL);

// Call the implemented function
else
   mult2array/ (short*) inBuFs->descs[DSP_mult2array_paramPtr->array1_InArrID1].buf,
   (short*) inBuFs->descs[DSP_mult2array_paramPtr->array2_InArrID2].buf,
   DSP_mult2array_paramPtr->nelements,
   (short*) outBuFs->descs[DSP_mult2array_paramPtr->result_OutArrID1].buf);

} 
break;

return(0);

4. The step 4 consists of creating the C6Accel ARM side API call. The below code should be added to the "c6accelw.c" file. The API contains a C6accel handle argument in addition to all the arguments of the original DSP kernel call. All the C6Accel APIs return error codes, so the return values are always integers. In our case we have to include:

int C6accel_DSP_mult2array(
    C6accel_Handle hC6accel,
    short* restrict ptr_array1,
    short* restrict ptr_array2,
    int npoints,
    short* restrict ptr_result
)
{
    XDM1_BufDesc inBufDesc;
    XDM1_BufDesc outBufDesc;
    XDAS_Int32 InArg_Buf_size;
    IC6Accel_InArgs *CInArgs;
    UNIVERSAL_OutArgs uniOutArgs;
    Int status;

/* Define pointer to function parameter structure */
DSP_mult2array_Params *fp0;
XDAS_Int8 *pAlloc;

ACQUIRE_CODEC_ENGINE;

/* Allocate the InArgs structure as it varies in size*/
InArg_Buf_size= sizeof(Fxn_struct)+
sizeof(DSP_mult2array_Params)+
sizeof(CInArgs->size)+
sizeof(CInArgs->Num_fxs);

pAlloc = (XDAS_Int8 *)Memory_alloc(InArg_Buf_size,
&wrapperMemParams);

CInArgs= (IC6Accel_InArgs *)pAlloc;
uniOutArgs.size    = sizeof(uniOutArgs);

/* Set up buffers to pass buffers in and out to algorithm */
inBufDesc.numBufs  = 2;
outBufDesc.numBufs = 1;

/* Fill in input/output buffer descriptor parameters and manage ARM cache*/
CACHE_WB_INV_INPUT_BUFFERS_AND_SETUP_FOR_C6ACCEL(ptr_array1,0,npoints*sizeof(short));
CACHE_WB_INV_INPUT_BUFFERS_AND_SETUP_FOR_C6ACCEL(ptr_array2,1,npoints*sizeof(short));
CACHE_INV_OUTPUT_BUFFERS_AND_SETUP_FOR_C6ACCEL(ptr_result,0,npoints*sizeof(short));

/* Initialize the extended InArgs structure */
CInArgs->Num_fxs = 1;
CInArgs->size    = InArg_Buf_size;

/* Set function Id and parameter pointers for first function call */
CInArgs->fxn[0].FxnID     = MATH_MULT2ARRAY_FXN_ID;
CInArgs->fxn[0].Param_ptr_offset = sizeof(CInArgs->size)+sizeof(CInArgs->Num_fxs)+sizeof(Fxn_struct);

/* Initialize pointers to function parameters */
fp0 = (DSP_mult2array_Params *)((XDAS_Int8*)CInArgs + CInArgs->fxn[0].Param_ptr_offset);

/* Fill in the fields in the parameter structure */
fp0->array1_InArrID1 = INBUF0;
fp0->array2_InArrID2 = INBUF1;
fp0->result_OutArrID1 = OUTBUF0;
fp0->npoints= nelements;

/* Call the actual algorithm */
if (hC6accel->callType == ASYNC)
{
    /* Update async structure */
    if (c6accelAsyncParams.asyncCallCount!=0)
    {
        status = UNIVERSAL_EFAIL;
        printf("Async call failed as %d are still pending\n");
    }
}
else{
    /* Context Saving */
    c6accelAsyncParams.asyncCallCount++;
    memcpy(&(c6accelAsyncParams.inBufs), &inBufDesc, sizeof(XDM1_BufDesc));
    memcpy(&(c6accelAsyncParams.outBufs), &outBufDesc, sizeof(XDM1_BufDesc));
    memcpy(&(c6accelAsyncParams.inArgs), CInArgs, sizeof(UNIVERSAL_InArgs));
    memcpy(&(c6accelAsyncParams.outArgs), &uniOutArgs, sizeof(UNIVERSAL_OutArgs));

    /* Asynchronous Call to the actual algorithm */
    status = UNIVERSAL_processAsync(hC6accel->hUni, &inBufDesc, &outBufDesc, NULL, (UNIVERSAL_InArgs *)CInArgs, &uniOutArgs);
}

else{
    /* Synchronous Call to the actual algorithm */
    status = UNIVERSAL_process(hC6accel->hUni, &inBufDesc, &outBufDesc, NULL, (UNIVERSAL_InArgs *)CInArgs, &uniOutArgs);

    /* Free the InArgs structure */
    Memory_free(pAlloc, InArg_Buf_size, &wrapperMemParams);
}

RELEASE_CODEC_ENGINE;
    return status;
}
## APPENDIX B. COMPILER INTRinsics INSTRUCTIONS

Interfacing C and C++ With Assembly Language

<table>
<thead>
<tr>
<th>C/C++ Compiler intrinsic</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int _abs (int src);</code></td>
<td>ABS</td>
<td>Returns the saturated absolute value of src</td>
</tr>
<tr>
<td><code>int _add2 (int src1, int src2);</code></td>
<td>ADD2</td>
<td>Adds the upper and lower halves of src1 to the upper and lower halves of src2, and returns the result. Any overflow from the lower half add does not affect the upper half add.</td>
</tr>
<tr>
<td><code>ushort &amp; _u lim (void *ptr);</code></td>
<td>LHU</td>
<td>Allows aligned loads and stores of 2 bytes to memory. The pointer must be aligned to a byte boundary. For 256-bit _asmixh corresponds to different assembly instructions than when used with other C6000 devices; see Table 7-5 for specifics.</td>
</tr>
<tr>
<td><code>const ushort &amp; _umixw (const void *ptr);</code></td>
<td>LHUU</td>
<td>Allows aligned loads of 2 bytes from memory. The pointer must be aligned to a two-byte boundary. For 256-bit _asmixh corresponds to different assembly instructions than when used with other C6000 devices; see Table 7-5 for specifics.</td>
</tr>
<tr>
<td><code>unsigned &amp; _umixm (void *ptr);</code></td>
<td>LEW</td>
<td>Allows aligned loads and stores of 4 bytes to memory. The pointer must be aligned to a four-byte boundary. For 256-bit _asmixh corresponds to different assembly instructions than when used with other C6000 devices; see Table 7-5 for specifics.</td>
</tr>
<tr>
<td><code>const unsigned &amp; _umixm (const void *ptr);</code></td>
<td>LEWU</td>
<td>Allows aligned loads of 4 bytes from memory. The pointer must be aligned to a four-byte boundary. For 256-bit _asmixh corresponds to different assembly instructions than when used with other C6000 devices; see Table 7-5 for specifics.</td>
</tr>
<tr>
<td><code>double &amp; _umixmm (void *ptr);</code></td>
<td>LEW/LDW, STW/STW</td>
<td>Allows aligned loads and stores of 8 bytes to memory. The pointer must be aligned to an eight-byte boundary. For 256-bit _asmixh corresponds to different assembly instructions than when used with other C6000 devices; see Table 7-5 for specifics.</td>
</tr>
<tr>
<td><code>const double &amp; _umixmm (const void *ptr);</code></td>
<td>LDW</td>
<td>Allows aligned loads of 8 bytes from memory. The pointer must be aligned to an eight-byte boundary. For 256-bit _asmixh corresponds to different assembly instructions than when used with other C6000 devices; see Table 7-5 for specifics.</td>
</tr>
<tr>
<td><code>unsigned _clz (unsigned src2, unsigned data, unsigned castl);</code></td>
<td>CLR</td>
<td>Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by data and castl, respectively.</td>
</tr>
<tr>
<td><code>unsigned _clz (unsigned src2, int src1);</code></td>
<td>CLR</td>
<td>Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by the lower 10 bits of src1.</td>
</tr>
<tr>
<td><code>__int40_t _ afl (double src);</code></td>
<td>Reinterprets double register pair src as an __int40_t stored as a register pair.</td>
<td></td>
</tr>
<tr>
<td><code>longlong _ afl (double src);</code></td>
<td>Reinterprets double register pair src as a long long register pair.</td>
<td></td>
</tr>
<tr>
<td><code>int _ ext (int src2, unsigned data, unsigned castl);</code></td>
<td>EXT</td>
<td>Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; data and castl are the shift left and shift right amounts, respectively.</td>
</tr>
<tr>
<td><code>int _ ext (int src2, int src1);</code></td>
<td>EXT</td>
<td>Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; the shift left and shift right amounts are specified by the lower 10 bits of src1.</td>
</tr>
<tr>
<td><code>unsigned _axru (unsigned src2, unsigned data, unsigned castl);</code></td>
<td>EXTU</td>
<td>Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by an unsigned shift right; data and castl are the shift left and shift right amounts, respectively.</td>
</tr>
<tr>
<td><code>unsigned _axru (unsigned src2, int src1);</code></td>
<td>EXTU</td>
<td>Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by an unsigned shift right; the shift left and shift right amounts are specified by the lower 10 bits of src1.</td>
</tr>
<tr>
<td><code>unsigned _ flo (float src);</code></td>
<td>Reinterprets the bits in the float as an unsigned. For example: _ flo (1.0) returns unsigned int with all bits set.</td>
<td></td>
</tr>
<tr>
<td><code>unsigned _ flo (double src);</code></td>
<td>Returns the high (odd) register of a double register pair.</td>
<td></td>
</tr>
<tr>
<td><code>unsigned _ hli (longlong src);</code></td>
<td>Returns the high (odd) register of a long long register pair.</td>
<td></td>
</tr>
<tr>
<td><code>double _ field (unsigned src2, unsigned src1);</code></td>
<td>Builds a new double register pair by reinterpreting two unsigned values, where src2 is the high (odd) register and src1 is the low (even) register.</td>
<td></td>
</tr>
<tr>
<td><code>float _ flo (unsigned src);</code></td>
<td>Reinterprets the bits in the unsigned as a float. For example: _ flo (0x00000000) = 1.0</td>
<td></td>
</tr>
<tr>
<td><code>longlong _ flo (unsigned src2, unsigned src1);</code></td>
<td>Builds a new long long register pair by reinterpreting two unsigned values, where src2 is the high (odd) register and src1 is the low (even) register.</td>
<td></td>
</tr>
<tr>
<td><code>unsigned _ lmbd (unsigned src1, unsigned src2);</code></td>
<td>LMBD</td>
<td>Searches for a leftmost 1 or 0 of src2 determined by the LSB of src1. Returns the number of bits up to the bit change.</td>
</tr>
<tr>
<td><code>unsigned _ lo (double src);</code></td>
<td>Returns the low (even) register of a double register pair.</td>
<td></td>
</tr>
<tr>
<td>C/C++ Compiler Intrinsic</td>
<td>Assembly Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>unsigned _loli(long long src);</td>
<td></td>
<td>Returns the low (even) register of a long long register pair.</td>
</tr>
<tr>
<td>double _floid(_int32_t src);</td>
<td></td>
<td>Interprets the _int32_t register pair src as a double register pair.</td>
</tr>
<tr>
<td>double _tfeid(long long src);</td>
<td></td>
<td>Interprets the long long register pair src as a double register pair.</td>
</tr>
<tr>
<td>int _mexpf (int src1, int src2);</td>
<td>MPY+</td>
<td>Multiplies the 16 LSBS of src1 by the 16 LSBS of src2 and returns the result. Values can be signed or unsigned.</td>
</tr>
<tr>
<td>int _mexpfz (unsigned src1, int src2);</td>
<td>MPYH</td>
<td>Multiplies the 16 LSBS of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.</td>
</tr>
<tr>
<td>int _mexpfh (int src1, unsigned src2);</td>
<td>MFPY</td>
<td>Multiplies the 16 LSBS of src1 by the 16 LSBS of src2 and returns the result. Values can be signed or unsigned.</td>
</tr>
<tr>
<td>int _mexpfhz (unsigned src1, int src2);</td>
<td>MFPYH</td>
<td>Multiplies the 16 LSBS of src1 by the 16 MSBS of src2 and returns the result. Values can be signed or unsigned.</td>
</tr>
<tr>
<td>int _mexpfl (int src1, int src2);</td>
<td>MFPYH</td>
<td>Multiplies the 16 LSBS of src1 by the 16 MSBS of src2 and returns the result. Values can be signed or unsigned.</td>
</tr>
<tr>
<td>int _mexpflz (unsigned src1, int src2);</td>
<td>MFPYHL</td>
<td>Multiplies the 16 MSBS of src1 by the 16 LSBS of src2 and returns the result. Values can be signed or unsigned.</td>
</tr>
<tr>
<td>int _mexpfhfl (int src1, int src2);</td>
<td>MFPYH</td>
<td>Multiplies the 16 MSBS of src1 by the 16 MSBS of src2 and returns the result. Values can be signed or unsigned.</td>
</tr>
<tr>
<td>unsigned _norm (int src);</td>
<td>NORM</td>
<td>Returns the number of bits up to the first non-redundant sign bit of src.</td>
</tr>
<tr>
<td>unsigned _normi (_int32_t src);</td>
<td>NADD</td>
<td>Adds src1 to src2 and saturates the result. Returns the result.</td>
</tr>
<tr>
<td>int _rad (int src1, int src2);</td>
<td>SAT</td>
<td>Converts a 45-bit long to a 32-bit signed int and saturates if necessary.</td>
</tr>
<tr>
<td>unsigned _set (unsigned src2, unsigned csrc, unsigned sdst);</td>
<td>SET</td>
<td>Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by csrc and dst, respectively.</td>
</tr>
<tr>
<td>unsigned _seti (int src2, int src1);</td>
<td>SET</td>
<td>Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by the lower ten bits of src1.</td>
</tr>
<tr>
<td>int _expa (int src1, int src2);</td>
<td>SADD</td>
<td>Multiplies src2 by src1, left shifts the result by 1, and returns the result. If the result is 0x80000000, saturates the result to 0xFFFFF.</td>
</tr>
<tr>
<td>int _expaf (unsigned src1, unsigned src2);</td>
<td>SHL</td>
<td>Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result.</td>
</tr>
<tr>
<td>int _eu (int src, int src2);</td>
<td>SUB</td>
<td>Subtracts src2 from src1, saturates the result, and returns the result.</td>
</tr>
<tr>
<td>unsigned _sub (unsigned src1, unsigned src2);</td>
<td>SUBC</td>
<td>Conditional subtract divide step.</td>
</tr>
<tr>
<td>int _eub (int src1, int src2);</td>
<td>SUB</td>
<td>Subtracts the upper and lower halves of src2 from the upper and lower halves of src1, and returns the result. Borrowing in the lower half subtract does not affect the upper half subtract.</td>
</tr>
</tbody>
</table>
APPENDIX C. COMMERCIAL BROCHURE OF THE SYSTEM

The Appendix C shows the commercial brochure of the system used to promote it in conventions.

ARGUS (Active Reverse GUide System)

ARGUS está basado en tecnologías de visión. Ayuda a prevenir accidentes durante las operaciones de mantenimiento y obra alertando obstáculos cercanos y identificando su forma y medidas, evitando el arranque en caso de detección de peligro.

ARGUS ofrece un alto nivel de fiabilidad y cobertura de área, pudiendo discriminar objetos incrementando el nivel de seguridad. Colocado en la parte alta del frontal o de la parte trasera de la máquina, ARGUS lo ve todo, pudiendo no tener ángulos muertos.

El área de detección de ARGUS se puede dividir en distintas zonas de seguridad que a su vez se dividen en diferentes niveles, completamente ajustables de acuerdo a las necesidades del cliente. Es posible definir la "Zona de parada" y la "Zona de aviso".

El cliente puede ajustar el área de detección, incluyendo la anchura exacta en este punto.

* PROYECTO FINANCIADO POR CDTI.
Especificaciones ARGUS

Especificaciones Generales

Fuentes de alimentación de 24V, 48V (según modelo)
5 paneles de salida programables especializados
5 luces de entrada programadas
Alarma acústica
Luz de estado

Características Principales

Zona de detección completa y configurable
Detección de objetos
Extracción de objetos muertos
Control automático

Certificaciones

Certificación CN 50155

Unidad de Control

Ubicada en la cabina, procesa los datos que suministra la unidad del sensor y envía al operator un comunicado visual y acústico. Puede interactuar con el control de la máquina o con señales externas a través de 5 señales programables opcionales.

Dimensiones:
278mm (ancho) x 230mm (alto) x 91mm (alto)
Peso: 4.33 kg

Unidad del Sensor

Colocada en el frente o parte superior de la máquina, recoge información de los objetos dentro de su alcance, transmite esta información a la unidad del control.

Dimensiones:
225mm (ancho) x 225mm (alto) x 97mm (alto)
Peso: 4.40 kg