

# A METHODOLOGY TO DEPLOY APPLICATIONS ON THE DUAL-CORE OMAP PLATFORM

SAULO O. D. LUIZ\*, JAYARAMA S. SANTANA\*, GENILDO DE M. VASCONCELOS\*, ANGELO PERKUSICH\*, ANTÔNIO M. N. LIMA\*, MARCOS R. A. MORAIS\*

\*Laboratory of Embedded Systems and Pervasive Computing, Academic Unit of Electrical Engineering, Center of Electrical Engineering and Informatics, Federal University of Campina Grande 10105, 58.109-900 Campina Grande, PB, BRAZIL

Emails: saulo@dee.ufcg.edu.br, jayaramasantana@yahoo.com, genildo@dee.ufcg.edu.br, perkusich@dee.ufcg.edu.br, amnlima@dee.ufcg.edu.br, morais@dee.ufcg.edu.br

**Abstract**— In this article, a methodology to develop applications for the OMAP 161x platform is introduced. This platform is composed of an ARM9 general purpose processor and a TMS320C55x fixed-point Digital Signal Processor (DSP). At first the mathematical formulation of the application is discussed, and is verified with a high level language using floating-point and fixed-point arithmetic. The verified solution is then implemented using floating-point arithmetic in C language, and then in fixed-point arithmetic. This last implementation is analyzed using a simulator for the target DSP, in this case a C55x simulator. Then, a user interface application that runs on the ARM processor is developed. Such application is then integrated with the DSP application through an ARM-DSP inter-processor communication mechanism, named DSP Gateway. These steps were applied for a case study, an adaptive Wiener bi-dimensional image filter. The introduced methodology allows students to acquire the necessary knowledge to develop applications targeted to the OMAP platform.

**Keywords**— OMAP161x platform, digital signal processing, inter-processor communication, DSP application development process.

## 1 Introduction

Embedded systems used in communications and multimedia such as digital video cameras, DVD players/recorders, portable multimedia equipment, etc. rely on computationally intensive digital signal processing algorithms which would be impracticable without nowadays' high-performance and low-power Digital Signal Processors. The OMAP (Open Multimedia Application Platform) dual-core platform, which is composed of an ARM (Advanced Risk Machine) and a DSP (Digital Signal Processor), allows the implementation of digital signal processing in the low power consuming DSP-side, while the general purpose applications can run on the ARM-side.

The development of applications to the OMAP platform involves several challenges, such as communication between the processors, implementation of algorithms based on fixed-point arithmetic, memory usage limitations and specific knowledge of the hardware architecture. Thus, it is important to develop a procedure for the deployment of applications on the OMAP platform.

Useful information about the OMAP 161x architecture can be found in Instruments (2005). Specific information about the TMS320C55x digital signal processor architecture, software development tools, assembly language programming, are available in the real-time DSP textbook Kuo and Lee (2001).

An application deployment process to the OMAP platform is described in this article. For the development of the DSP-side application, the following software tools are used: MATLAB, Vi-

sual C++ and Code Composer Studio (CCS) for the C55x processors. The mathematical formulation of the digital signal processing application is performed, followed by their implementation in a high level language (MatLab) using floating-point and fixed-point arithmetic. In order to serve as a preceding step to the fixed-point implementation in CCS, C programs in Visual C++ are developed using floating-point and fixed-point.

The ARM runs Embedded Linux Operating System compiled for the OMAP platform. The ARM-side application is developed using the Scratchbox toolkit Movial (2005) toolkit for cross-compilation on a PC running Linux to provide a user interface and proper communication with the DSP-side.

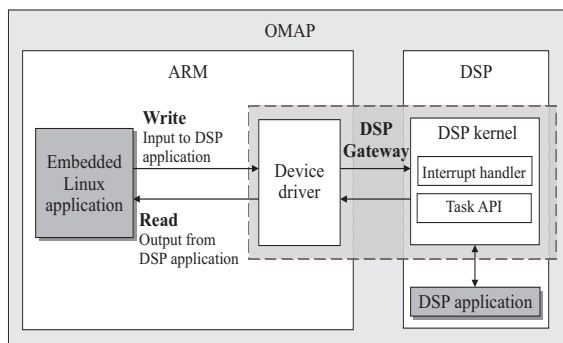


Figure 1: Overview of the ARM-DSP communication process

The communication process between the ARM and DSP is made possible by DSP Gateway, which is composed of a Linux device driver on the ARM-side and a DSP-side kernel library. Since

version 2.6.6, Linux kernel officially supports DSP Gateway for OMAP 15XX, 16XX, 1710, 5910, and 5912. An overview of the elements involved in the ARM-DSP communication is illustrated in figure 1 and the procedures for the deployment of an application to the OMAP platform is illustrated in figure 2.

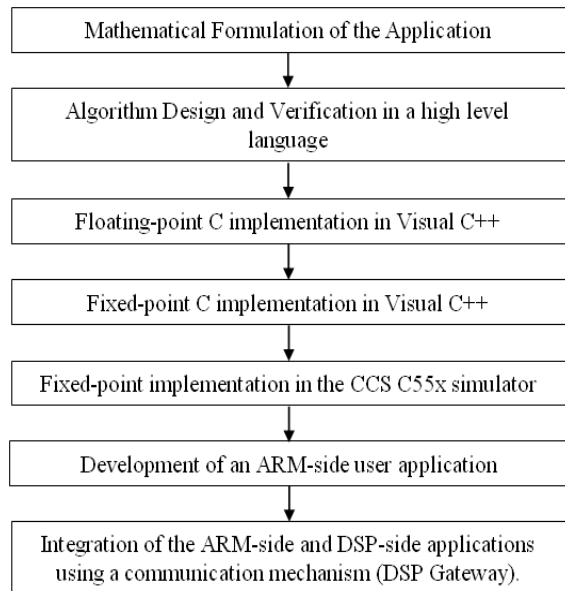


Figure 2: Procedures for the deployment of an application to the OMAP platform

The following sections are organized as follows. Section 2: mathematical formulation of the application; section 3: algorithm design and verification in a high level language; section 4: implementation of the algorithm in C language using floating-point and fixed-point arithmetic; section 5: port of the code to the DSP; section 6: development of an application in the ARM-side and integration with the DSP-side using DSP Gateway.

## 2 Mathematical Formulation of the Application

As an illustration of the mathematical description of the application to be implemented, the case study of the Wiener adaptive filter is presented in this section.

This filter is used in order to reduce the additive white Gaussian noise present in the acquisition of a digital bi-dimensional signal captured by the image sensors used in digital cameras.

The Wiener digital filter is adaptive, i.e., the filtering of the image varies according to the signal's statistics (mean and variance) in a local neighborhood. This filter is based in the minimization of the squared error between the original image  $f(n,m)$  and its estimation  $\hat{f}(n,m)$  and it is supposed that the noise is additive and White.

This filter is described mathematically according to equations 1, 2 and 3 Chan (1984).

$$\mu(n,m) = \frac{1}{NM} \sum_{i,j \in \eta} a(i,j) \quad (1)$$

$$\sigma^2(n,m) = \frac{1}{NM} \sum_{i,j \in \eta} a^2(i,j) - [\mu(n,m)]^2 \quad (2)$$

$$\hat{f}(n,m) = \mu(n,m) + \frac{\max(0, \sigma^2(n,m) - \nu^2)}{\sigma^2(n,m)} \cdot (a(n,m) - \mu(n,m)) \quad (3)$$

Where  $\hat{f}(n,m)$  is the filter's output,  $a(i,j)$  is the filter's input (noisy image),  $\nu^2$  is the variance of the noise,  $\mu(n,m)$  and  $\sigma^2(n,m)$  are the mean and local variance, respectively in  $\eta$ , i.e, a  $N \times M$  local neighborhood (window), around the pixel  $(n,m)$ .

When the noise is not known a priori,  $\nu^2$  can be calculated as the mean of all local variances  $\sigma^2(n,m)$ .

## 3 Algorithm Design and Verification in a High Level Language

### 3.1 Floating-point arithmetic

The use of a high level language (in this case, MatLab) makes the implementation and the algorithmic verification easier. Moreover, the use of floating-point arithmetic yields higher precision in the calculations.

The representation of a number in floating-point with single precision, following the IEEE (Institute of Electrical and Electronics Engineers) notation, uses 32 bits. The first bit is used for the signal, the following 8 bits represent the exponent and the remaining 23 bits represent the mantissa. The mantissa is a real number in the range  $[1, 2)$  and it is stored as a binary number in negative powers of 2. The exponent is an integer in the range -126 to 127 biased with 127. The conversion of a number in floating-point notation to a real value can be done according to 4.

$$RealValue = (-1)^{signal} \cdot Mantissa \cdot 2^{Exponent - Bias} \quad (4)$$

The advantage of the implementation using floating-point notation is that very small and very large numbers can be represented and without much loss of precision. For example, the single precision floating-point notation can represent numbers in the range  $1.175494351 \cdot 10^{-38}$  to  $3.402823466 \cdot 10^{+38}$ . Thus, special considerations regarding overflow, underflow and saturation are generally not necessary (W. Gan, 2006). For high precision applications, the double precision

floating-point notation can be used, in which a number is represented using 64 bits (1 for signal, 11 for the exponent and 52 for the mantissa).

For the implementation using floating-point, a 24-bit bitmap image was used as the input bi-dimensional signal.

The 24-bit bitmap image format represents each pixel (picture element) in the RGB (Red, Green, Blue) format. Each color component is specified by 8 bits, thus each pixel is composed by 24 bits. Each color component's is in the range 0 to 255 and the Wiener filter is applied individually to each color component.

### 3.2 Fixed-point arithmetic

The implementation in MatLab using fixed-point arithmetic facilitates the simulation and verification of occurrences of overflow/underflow in the algorithm for a calculated Q notation.

Concerning the manipulation of data in devices with fixed-point architecture, the following numeric representation should be used:

$$m2^{-e} \quad (5)$$

Where  $m$  is the mantissa and  $e$  is the exponent.

To increase the precision in the calculations, we use the transformation of the numbers to the Q format. This format is based on a pre-multiplication and pos-division of the numbers involved in the operations.

The designation of the Q format representation deals with a tradeoff between precision and memory use. With the use of data types of 16 bits, for example, there will be a smaller memory usage. However we may compromise the precision of the calculations. On the other hand, if we use data types with more bits, for example, 32 or 40 bits, much more precise calculations will be done at the cost of higher memory utilization, which may be critical in some applications for embedded systems.

Let  $x$  be a given number in floating-point which will be converted to fixed-point using a multiplication and saved in a variable. Consider that this variable has a data type which supports a maximum absolute value  $N_{max}$ , then:

$$N_{max} = 2^Q \cdot x \quad (6)$$

Where,  $N_{max}$  is the variable's maximum absolute value,  $Q$  is the corresponding Q format representation and  $x$  is maximum original number in floating-point.

Applying the base-2 logarithm in the above equation 6, the maximum value of  $Q$  can be determined, which corresponds to a maximum precision without the occurrence of overflow. This can be written as follows:

Variables	Labels
$N=M$	window
$\frac{1}{NM}$	invsqwindow
$\sum_{i,j \in n} a^2(i,j)$	mean
$\sigma^2(n,m)$	var
$\hat{f}(n,m)$	output

Table 1: Labels for the variables used

$$\log_2 N_{max} = Q + \log_2 x \quad (7)$$

$$Q = \log_2 N_{max} - \log_2 x \quad (8)$$

This conversion should be applied to the variables in floating-point before performing mathematical operations with them, having in mind to use the corresponding arithmetic operations in fixed-point.

### 3.3 Case Study: Port of the Wiener Filter to Fixed-point

To port the Wiener filter algorithm from floating-point to fixed-point, we should analyze equations 1, 2 and 3. The variables of these equations are labeled according to table 1. Considering the C55x DSP core's 16-bit architecture, unsigned 16 bits variables are used, except for the results of intermediate operations where unsigned 32 bits variables are applied to gain precision on fixed-point calculations. In the output variable, we will use a signed 32 bits variable.

The Wiener filter is analyzed using fixed-point arithmetic for windows of size 3x3 and 5x5. These values have been selected in order to limit the range of the variables. Thus, this range can be identified a priori.

The worst case of the variable *invsqwindow* occurs when *window* is equal to 3, since  $\frac{1}{NM} = \frac{1}{window \cdot window}$  is greater when *window* = 3. Applying equation (8) to *invsqwindow*, the maximum Q format representation that doesn't cause overflow is Q19. Thus,  $invsqwindow_{max} = \frac{1}{3 \cdot 3} \cdot 2^{19} = 58254$ .

The sum variable is represented in *Q0* since this avoids the divisions involved in normalizing the inputs  $a(i,j)$ , which is in the range 0 to 255.

All the operations utilizing *invsqwindow* and sum should be tested so as to verify that the representation of *invsqwindow* in Q19 will not cause overflow. For the calculation of the mean 1, *invsqwindow* and sum are multiplied and the result will be stored in an auxiliary unsigned variable *sumAux* of 32 bits, therefore it is needed to observe that:

$$sum_{max} \cdot invsqwindow_{max} \leq 2^{32} - 1$$

The maximum value of *sum* occurs when *window* = 5, therefore  $sum_{max} = 255 \cdot (5 \cdot 5) = 6375$ .

It is verified that:





