ABSTRACT

Title of Document: INTEGRATED INPUT MODELING AND MEMORY MANAGEMENT FOR IMAGE PROCESSING APPLICATIONS

Fiorella Haim, Master of Science, 2005

Directed By: Prof. Shuvra Bhattacharyya
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies

Image processing applications often demand powerful calculations and real-time performance with low power and energy consumption. Programmable hardware provides inherent parallelism and flexibility making it a good implementation choice for this application domain. In this work we introduce a new modeling technique combining Cyclo-Static Dataflow (CSDF) base model semantics and Homogeneous Parameterized Dataflow (HPDF) meta-modeling framework, which exposes more levels of parallelism than previous models and can be used to reduce buffer sizes. We model two different applications and show how we can achieve efficient scheduling and memory organization, which is crucial for this application domain, since large amounts of data are processed, and storing intermediate results usually requires the use of off-chip resources, causing slower data access and higher power consumption. We also designed a reusable wishbone compliant memory controller module that can be used to access the Xilinx Multimedia Board's memory chips using single accesses or burst mode.

INTEGRATED INPUT MODELING AND MEMORY MANAGEMENT FOR IMAGE PROCESSING APPLICATIONS

By

Fiorella Geraldine Haim Hoffer

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2005

Advisory Committee:
Professor Shuvra Bhattacharyya, Chair
Professor Gang Qu
Professor Ankur Srivastava

Dedication

To Julián, Marion, Eddy, Guido. To my friends.

Acknowledgements

I would like to thank my advisor Shuvra Bhattacharyya for giving me the opportunity of being part of his research group and for his valuable guidance, and my committee members for their useful feedback. I'm very thankful to my lab-mates, the members of the DSPCAD group, who provide me important assistance in several occasions; however, I'm especially thankful to my friend Mainak Sen, joint developer of the HPDF/CSDF integration, for his help, advice and suggestions. I would also like to thank Juan Pablo Oliver, Julio Pérez, Sebastián Fernández and the other members of the Applied Electronics Group (Universidad de la República, Uruguay) for their advice and for always being available for my questions. I also want to thank the Fulbright Commission for giving me this great opportunity. Finally, I would like to thank the essential support of my friends and family (from Montevideo, Potomac, Vancouver, College Park, Bordeaux, London and many other cities) and the patient proofreading of Marion and Julián, my main supporter.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Image processing applications are becoming more and more elaborated in the sense that more processing calculations need to be performed in order to extract more information, but, at the same, applications that provide real-time analysis require higher speed. Moreover, other features such as low power and energy consumption need to be considered in a design. This is really clear when having autonomous (battery-powered) applications, but is also important for high performance applications that need to be able to dissipate this power. On the other hand, hardware designs are inherently parallel, and usually higher speeds and lower power consumptions can be achieved with hardware rather than with software designs. Consequently, demanding image processing applications are often implemented using hardware; in this work we focus on programmable hardware.

In order to be able to exploit eventual concurrency inherent to an application in the process of mapping it into the targeted parallel platform, its specification has to reflect this characteristic. If the application is specified using a sequential language, then it becomes hard to exploit concurrency. In this work we introduce a new modeling technique combining Cyclo-Static Dataflow (CSDF) base model semantics and Homogeneous Parameterized Dataflow (HPDF) meta-modeling framework, that exposes different levels of parallelism and is very suitable for specifying some image processing applications. We model two different applications and show how we can achieve efficient scheduling and memory organization.

A particular problem related to image processing applications is that they process large amounts of data. Storing intermediate results often requires the use of off-chip resources, which results in slower data access and higher power consumption. That is why

having an effficient memory organization is so crucial for this particular application domain.

In this study we focus on two image processing applications: a gesture recognition application and a motion detection one, and a particular hardware platform: the Xilinx Multimedia Board.

1.1 Related Work

There are different Model of Computation (MoC) suitable for exposing concurrency. In this work we concentrate on Dataflow Graphs, an account of other MoCs and in particular a description of different types of Dataflow graphs is given in Chapter 2.

In embedded systems, the access to external memory can be extremly costly in terms of time and power consumption, causing memory organization to be a critical issue for some applications. In [35] the authors introduce a method to efficiently allocate data into available memories for reconfigurable architectures. They assume a system with a CPU with its main memory connected through a system bus to an accelerator constituted by an FPGA and RAM chips. They start from a sequential description of an algorithm and try to detect data that are accessed repeteadly in arrays. First they give some guidelines to allocate data in the different hierarchical memory levels, then they develop a integer linear program to determine how to map data efficiently in the external chips.

Chapter 3 provides a background on image processing applications and Chapter 4 on FPGAs-based Boards.

1.2 Contributions of this Work

A central contribution of this work is demonstrating the integration of CSDF base model semantics into the Homogeneous Parameterized Dataflow HPDF meta-modeling framework. This integration, which was developed jointly with Mainak Sen, has both the advantages of HPDF which provides bounded memory, dynamic parameterization, and those of CSDF which offers the finer granularity, phased decomposition of actor execution. In this work we precisely specify how we integrated HPDF and CSDF and provide a method to find a valid schedule when such a schedule exists. Furthermore, we modeled two different image processing applications using HPDF/CSDF and integrating the input to the model, which exposed further levels of parallelism.

Another relevant contribution is our analysis on how different memory organization can affect the application's performance and energy consumption and more importantly we present a method to infer an efficient memory organization from a given application HPDF/CSDF modeling.

Finally, we also designed a reusable Wishbone memory controller core for the Multimedia Board that can be used to access the board's external memory chips.

1.3 Organization of the Thesis

In the first three chapters we study previous work; in the next chapter in particular we introduce Dataflow Graphs and present different types of Dataflows; in Chapter 3 we give an overview of different image processing algorithms and techniques; and in Chapter 4 we discuss the use of FPGAs and describe different board concepts, including the Multimedia Board. In Chapter 5 we formally specify HPDF meta-model for CSDF base actors and

give a method to find valid schedules when they exist, while in Chapter 6 we apply this modeling technique to two different image processing applications. Chapter 7 studies different memory organization schemes and shows how to infer an efficient memory organization given an HPDF/CSDF algorithm specification; Chapter 8 presents the experiments and results, and finally Chapter 9 concludes and introduces future directions to explore.

Chapter 2: Dataflow Representation of Algorithms

Having different teams work on the algorithm and its implementation in hardware is an usual practice. Algorithm signal engineers specify the algorithm in a formal language in order to avoid ambiguities. Thus, C language is often used to specify algorithms. However, since C is a sequential language, using it makes it very difficult for hardware engineers to find eventual inherent concurrency of the application. In this way, operations that could be done in parallel end up scheduled sequentially, causing a sub optimum hardware implementation.

In order to avoid these problems, many teams are specifying their algorithms using languages and tools that allow them to show the concurrency of the application. In this thesis we are going to focus on Dataflows, which are a model of computation (MoC) suitable to express the functional parallelism of an application. However, there are other MoCs that enable us to exploit concurrency of algorithms. For example, Celoxica's Handel C [19] is a subset of C (without pointers and floating point data types) extended with parallel constructs, created to provide C familiar programmers with a way to specify parallelism in an application to be ported in hardware. Another tool used to describe algorithms exposing their inherent parallelism is Compaan [22], which tries to compile programs in Matlab language into a concurrent representation.

In the following subsections we present Dataflow graphs, discuss some of their properties and then describe some particular types of Dataflows.

2.1 Dataflow Graphs

Digital signal processing, and image processing in particular, usually requires high speed and low power consumption. As it was previously stated, extracting concurrency present in sequential algorithms can become a very hard job. Dataflow then becomes an interesting way to specify algorithms exposing their intrinsic concurrency properties, facilitating the generation of an efficient parallel circuit synthesis.

A Dataflow Graph is a multidirected graph [29], where the vertices represent computation (actors), and the edges (arcs) represent the data communication among actors, implemented as FIFO queues. Each actor will fire when it has the required number of tokens in its input edges, producing a certain number of tokens in its output edges.

In this work we are going to use the following notation [29]. If V is the set of vertices and E is the set of edges, then each edge is an ordered pair (v_1, v_2) where $v_1, v_2 \in V$. An edge $e = (v_1, v_2), e \in E$ is said to be directed from v_1, the source vertex of e or $src(e)$, to v_2, the sink vertex of e or $snk(e)$

2.2 Properties of Dataflow Graphs

In general, dataflow graphs can represent any Turing machine. This makes the problems of deadlock and determining the maximum buffer sizes to implement the edges undecidable. Several restrictions can be imposed on general dataflows, generating new models that lose some of the descriptive power of general dataflows, but favoring, on the other hand, the possibility to analyze if the dataflow graph can be implemented with bounded

buffer memory size (consistency) and analyze deadlock ocurrence. In the following sections we present some particular restricted Dataflow Graphs.

2.3 Synchronous Dataflow Graph (SDF)

SDF was introduced in [16]. The main characteristic of SDF is that the number tokens produced and consumed by each actor when they fire are known a priori. This means that the system can be scheduled at compile time, avoiding the overhead of scheduling at runtime. Figure 1 shows an example from [29] of an SDF graph with actors A, B, C and two edges. The number of tokens produced as well as the number of tokens consumed are indicated on the edges. The edge (A, C) has a delay on it, the delays, or initial tokens, indicate that the data comes from previous firings of the precedent actor.

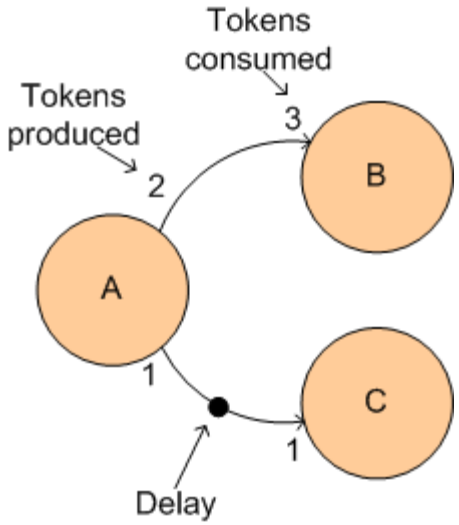


FIGURE 1. Example of a Syncronous dataflow graph

An SDF graph can be represented by its topology matrix Γ, that has one column for each vertex of the graph and one row for each edge. $\Gamma(i, j)$ represents the number of tokens produced in edge i by actor j. In our example, if we number actor A as 1, B as 2 and C as 3, and edges (A, B) as 1 and (A, C) as 2, we have that the topology matrix Γ is:

$$\Gamma = \begin{bmatrix} 2 & -3 & 0 \\ 1 & 0 & -1 \end{bmatrix} \quad \text{(EQ 1)}$$

The repetitions vector q for an SDF graph has length s equal to the total number of actors in the graph and is the smallest integer vector for which if each actor i is invoked a number of times equal to the ith component of q then the number of tokens in each edge of the SDF remains unchanged. If a connected SDF graph with s actors has consistent sample rates, it is guaranteed to have $rank(\Gamma) = s - 1$ and q can be found solving the following set of linear equations, also called balance equations:

$$\Gamma q = 0 \quad \text{(EQ 2)}$$

SDF is very well-suited to represent a wide class of digital signal processing algorithms. For instance, systems with decimators can be modeled in a natural way with SDF.

Also, an algorithm modeled as an SDF graph can afterwards be translated to hardware. For instance, in [36] synthesizable VHDL is generated from an SDF graph while in [25] synthesizableVerilog code is generated from an SDF graph.

Although SDF is suitable to represent systems as those previously described, systems where the amount of tokens generated or consumed by an actor are not known at compile time cannot be represented with this model.

2.4 Cyclo-Static Dataflow Graph (CSDF)

The CSDF model introduced in [3] is particularly suitable for representing applications that cyclically change their behavior. As with SDF, the number of tokens produced and consumed is known at compile time, but this number changes periodically. Figure 2

shows an example of a CSDF graph with three actors, A, B and C, and two edges, u and v, where $u = (A, B)$ and $v = (A, C)$. Actor A produces in edge u a number $x_A^u(i)$,

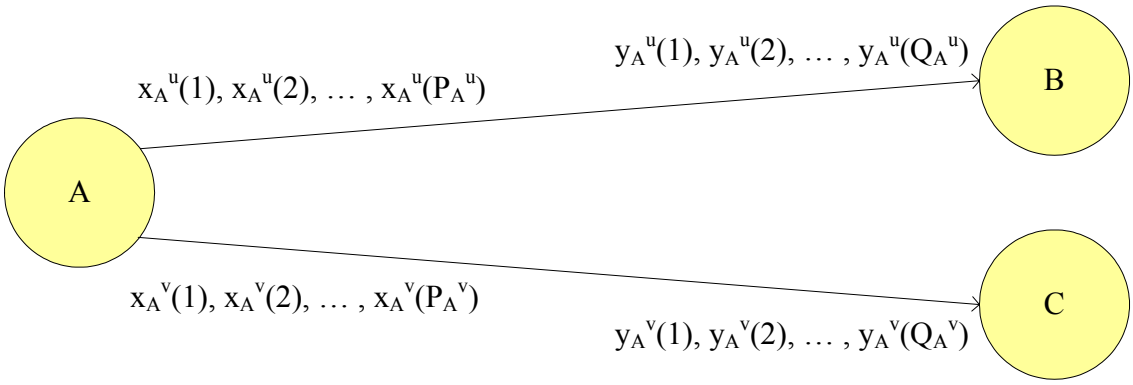


FIGURE 2. Example of a CSDF graph with three actors and two edges.

$(1 \le i \le P_A^u)$, of tokens every $(nP_A^u + i)^{\text{th}}$ time it is invoked, and similarly, in edge v a number $x_A^v(i)$, $(1 \le i \le P_A^v)$ of tokens every $(nP_A^v + i)^{\text{th}}$ time it is invoked. P_M^u is the period of the production sequence of actor M on edge u while Q_M^u is the period of the consumption sequence of actor M on edge u. The period P_M of actor M is defined as the least common multiple of all P_M^u and Q_M^u taken among all incoming and outcoming edges of actor M. Necessary and sufficient conditions for the existence of a static schedule are given in [3]. In [20] SDF and CSDF are compared, showing that CSDF actors can be transformed into SDF actors, and hence use SDF scheduling techniques. On the other hand, in certain applications, CSDF can expose more parallelism and this property would be lost during the transformation unless some multirate actors are included in the SDF graph.

2.5 Parameterized Dataflow Graph

As we discussed in the previous sections, SDF and CSDF are models for which a static schedule can be calculated. However, some signal applications, in particular those that have dynamic production or consumption of tokens, cannot be described using SDF or CSDF. In [2] a hierarchic parameterized dataflow modeling framework was introduced. This meta-model can be applied to different base dataflow graphs, in [2] formal semantics for parameterized synchronous dataflow (PSDF) are developed. PSDF can be considered an augmentation of the SDF model, incorporating parameterization and run-time management of parameter configurations. Formally, a PSDF graph is composed by PSDF actors and edges. The architecture of this framework is based on the decomposition of a specification (subsystem) in three different graphs: init, subinit and body, where the latter models the main functionality and the init and subinit graphs control the body behavior by configuring its parameters. In this way, the number of tokens produced or consumed by the actors can change dynamically.

2.6 Homogenous Parametrized Dataflow Graph (HPDF)

HPDF, proposed in [26], is, like PSDF, a meta-modeling technique that tries to model dynamicity. An HPDF subsystem is said to be homogeneous in two ways: first, the top level actors in an HPDF subsystem execute at the same rate; and second, reconfiguration across subsystems can be achieved without introducing hierarchy, although hierarchy can be used when desired. This meta modeling technique can be applied to composite actors that have SDF, CSDF or PSDF actors as their constituent actors.

In [26] the base actors were specified using SDF. Figure 3 shows an example of an HPDF graph. In this example actor A produces 2 tokens in edge (A, B) and n tokens in edge (A, C) and actor C fires after consuming n tokens from this edge. Actor B consumes the two tokens produced by A and after firing produces m tokens which are in turn consumed by D. A delay token is present in edge (A, C). It can be observed that we need to distinguish the tokens produced by A onto (A, C) and B onto (B, D) in different invocations. In [26] they implement an "end of packet" to know when an actor finishes producing the tokens corresponding to one invocation.

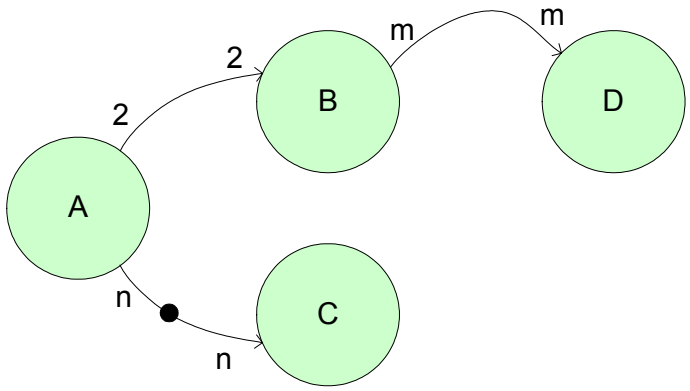


FIGURE 3. Example of an HPDF graph

In general, scheduling of an HPDF graph is very simple. In our example, for instance, a valid schedule is ABCD. Also, unlike PSDF, HPDF always executes in bounded memory when the component models do so as well.

Chapter 3: Image Processing Algorithms

Image processing is a continuously growing knowledge area. It has applications as diverse as medical research, automated manufacturing, space exploration and surveillance. In Section 3.1 we define basic concepts such as digital images and image processing. In Section 3.2 we present different techniques both for low and high-level image processing. Then, in Section 3.3 we give an overview of a particular application domain image processing algorithms, smart cameras, that have special time processing and sometimes energy consumption requirements. Finally, in Section 3.4, we describe two particular applications.

3.1 Basic Definitions

An image can be defined as a function $f(x, y)$, $f: \Re^2 \rightarrow \Re$, where x and y are real numbers representing spatial coordinates and the functional value f at each coordinate gives the grey level or intensity at that position [6]. A digital image has a finite number of discrete (x, y) coordinates and the function f can only take discrete finite values. Each element located at coordinates (x, y) is called *pixel*.

In color images each pixel needs to have at least three different components, each one being a coordinate in a color space. There are several color models, among them we can highlight the RGB and YUV models. RGB is an additive color model, in the sense that the colors red, green and blue are added to make other colors. This YUV model is the one used for analog television, Y is the intensity or luminance or luma, while U and V are the chroma or color difference components. Each color model can lead to different color spaces, for instance the RGB color model does not specify what red, green and blue are. A

color space is defined by the color model and the color map. An important color space defined in the ITU-R BT 601/656 standard is the YC_rC_b (sometimes also called YUV), which is a digital version of YUV. In this work we are going to use indistinctly the terms YUV and YC_rC_b in the understanding that when talking about digital image encoding we are referring to the YC_rC_b digital space. In order to convert the components of an RGB pixel to YUV components, we can use the following equations:

$$Y = 0.299R + 0.587G + 0.114B \quad \text{(EQ 3)}$$

$$U = -0.147R - 0.289G + 0.436B \quad \text{(EQ 4)}$$

$$V = 0.615R - 0.515G - 0.100B \quad \text{(EQ 5)}$$

$$R = Y + 1.140V \quad \text{(EQ 6)}$$

$$G = Y - 0.395U - 0.581V \quad \text{(EQ 7)}$$

$$B = Y + 2.032U \quad \text{(EQ 8)}$$

In Figure 4 a) a color RGB image with 300x225 pixels is shown, Figure 4 b) shows the grey level of image a) calculated using Equation 3.

FIGURE 4. Example of an RGB Color image (a) and its grey level (b). In (a) each pixel has 3 8-bit components, one for each color channel. In (b) the intensity was calculated for each pixel using Equation 3.

In [6] and in [23] the authors distinguish two main different purposes for image processing, the first one is to enhance an image for human perception, the second one is to extract information for further machine understanding of them. Although several processing techniques can be used for both situations, in particular for noise reduction and image enhancement, in this work we are going to focus on the latter category; some useful processing techniques targeted mainly for the first purpose can be found in [23].

In a broad sense, Image Processing includes all processing that has an image as its input. However, this definition overlaps with other areas such as Image Analysis, Computer Vision and Artificial Intelligence (AI). The borders between these areas are not strict and there is not a unique criterion to distinguish among them. For instance, in [4] the authors consider that low-level processing or image pre-processing is when the output of the processing stage is also an image, and they associate high-level processing to Computer Vision and AI if the output of the algorithm consists of other type of information extracted from the original image. On the other hand, in [6] the authors consider that since there are several simple image processing techniques (such as calculating the average intensity of an image) that produce outputs other than images, limiting image processing only to computations that yield another image is somewhat artificial. They define three categories: low-, mid- and high-level processing. Low-level processing is characterized by having images as inputs and outputs and consists of tasks such as noise filtering and image enhancement; mid-level processing usually has images as inputs and some of their attributes as outputs and it includes tasks such as identifying different objects; high-level processing usually consists in interpreting those attributes extracted from mid-level pro-

cessing. They then consider both low- and mid-level processing as belonging to the image processing domain, while extracting sense of this processing is associated to AI.

Low-level processing involves large amounts of data, thus requiring considerable power consumption and computational time. However, usually these tasks can be done concurrently at pixel or region levels; therefore, parallel processing can significantly reduce computational time [4]. In this work we focus on low-level and mid-level processing. In the next section we briefly introduce some typical low-level and high-level digital image processing problems along with some processing techniques.

3.2 Typical Image Processing Problems and Techniques Used

The first assumption that we make is that the image or video has been acquired with the best achievable quality. Thus, the processing techniques discussed in this section are applied to these digital images. For a discussion on optimizing the image acquisition refer to [23]. In the following subsection we first focus on low-level problems and then we discuss high-level ones.

3.2.1 Pre-processing Techniques

A typical low-level problem is removing noise from images. Depending on the type of noise present in a particular image, different techniques can be used [6], [23]. These techniques can be applied to grey level or color images, for simplicity we will mainly address techniques targeted for grey level image.

Random or gaussian noise can be removed with linear filters. The simplest linear filter is the 3x3 average filter where each pixel p is replaced by the averaged value of the

pixels belonging to the 3x3 square centered in p. This exploits spatial locality which means that close pixels will have similar values; time locality could also be exploited by averaging different consecutive frames. The size of the filter can be increased (i.e. 5x5, 7x7) and the weight assigned to each pixel in the box can be changed. However, some problems are associated with this kind of filter, for instance, the image can be blurred, in particular edges among different regions would tend to soften. In order to avoid these problems, non-linear filters can be used. One example of these filters is the 3x3 median filter. In this case, the value of each pixel p is replaced by the median of the pixels belonging to the 3x3 square centered in p. This is specially useful when in the presence of shot noise, also known as salt and pepper noise, where the small dots would disappear. Although now the blurring of the image is avoided, thin lines (compared to the size of the filter) may disappear.

Morphological filters such as erosion and dilation ones, are also very often used for image pre-processing. Dilation filters tend to fill structures that may have holes, while erosion filters tend to do just the opposite.

Finally, edge detection is also a complex and interesting low-level problem. Edge points contain a high grey value gradient, that means a high gray value difference in a local neighborhood [4]. Edge detectors perform either the first (gradient) or the second (gradient change) derivative of the intensity of an image in order to find where a region changes. Some well-known edge detectors are the Laplace and the Sobel Operators, in [4] pixel level parallel algorithms are given to detect edges using these operators.

3.2.2 High-level Processing Techniques

High-level images and video streams processing may consist in the ability to recognize objects or persons, expressions, gestures or motion. In order to appreciate the complexity of these tasks, we must be aware that very often, variations between images of the same face due to illumination and viewing direction are more significant than image variations due to change in face identity [7]. The application domains where it is important to solve these kind of problems includes surveillance, medicine and machine-human interfaces.

As it was previously stated, the techniques used for high-level processing are very close to machine learning and computer vision areas. Some tools used are Neural Networks, Support Vector Machines (SVM) and Hidden Markov Models (HMM). Neural networks which were inspired by the brain functionment, consist of a group of interconnected nodes, each one being a different processing element. There is a choice of several configurations for the connections (i.e. feedforward or recurrent, multilayered, self-organized), the processing elements (i.e. linear or sigmoid activation functions), and the learning processes (supervised versus unsupervised) [8]. SVM is another learning method generally used as a classifier. An introductory presentation along with some application examples, including a face detection application, can be found in [9]. HMM is a stochastic signal model where by observing the output of hidden states inferences can be made [21].

Each of the high-level problems presented here has its own particular difficulties to be overcome. For instance, in order to recognize faces, an intelligent system needs to separate accessories such as glasses, from distinctive features. On the other hand, it is a great challenge for the automatic system to recognize expression and gesture and to abstract them

independently of the face. For a human being, it is usually easy to detect a smile in a person, even if it comes from a total stranger and in spite of the fact that people smile in many different ways. Similarly, gestures can be very difficult for a machine to interpret, since big differences in positions can convey the same meaning, whereas subtle changes can have a deep impact on meaning. In spite of all these difficulties, machine learning techniques usually achieve good results in these applications.

3.3 Smart Cameras

Smart Cameras perform high-level processing of a scene in real-time [37]. We can identify different subtasks with increasing information extraction levels for the processing: first, pre-processing, which can consist of removing noise, improving the quality of the image, or even distinguish different regions in the image, such as finding skin tone regions [5]. This would be the low-level processing stage. Then the second group of tasks detects objects present in the frames and represents them conveniently; we can classify these tasks as mid-level processing. Finally, the third type of tasks performs high-level processing, analyzing the video stream and recognizing faces present on a database for instance or recognizing gestures independently of the identified person. Finally, the system may respond in a certain way, such as firing an alarm or transmitting or recording certain frames or some characteristics extracted from them. Figure 5 summarizes this classification of subtasks. An intelligent camera-based system can implement one or several high-level subtasks and execute different actions depending on the analysis and the concrete application.

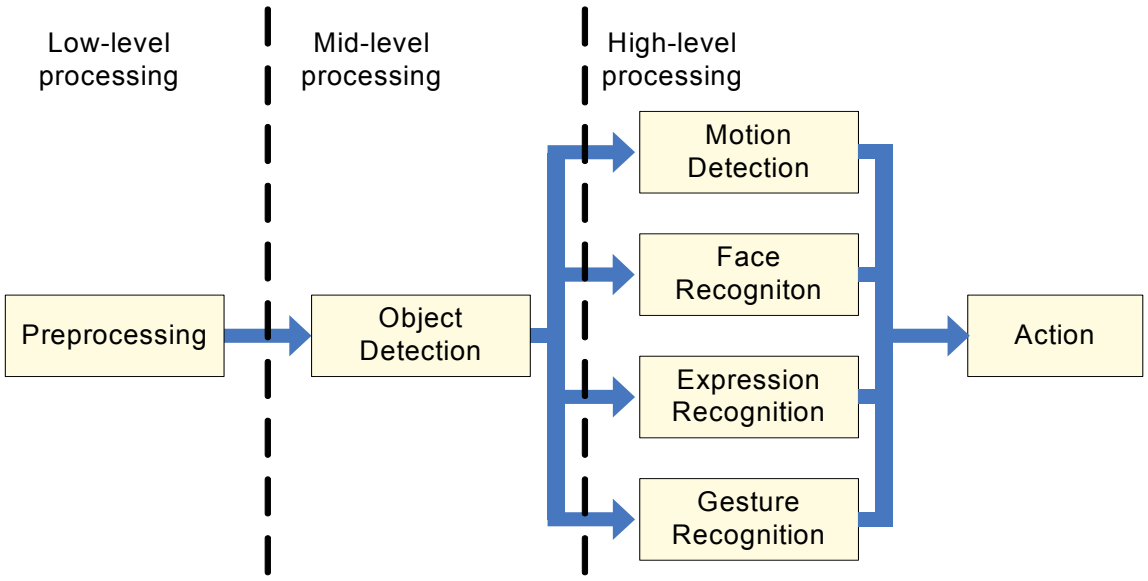


FIGURE 5. Example of different subtasks that can be present in a Smart Camera system

Smart Cameras then, have to be fast, processing frames at a high enough rate to have real-time information; the minimum acceptable rate is determined by the particular application requirements. Another usually desirable characteristic for these systems is that they have low energy consumption. There are two main reasons for this: the first one is to power them with batteries without having to replace them too frequently; but even if the system does not need to be powered with batteries, there is a second reason for choosing a low-energy design, which is the reduction in heat dissipation that this would entail.

3.4 Applications

In particular, for this work we studied two algorithms. The first one proposed by Lv and Wolf in [37] is a smart camera gesture recognition application. The second one is a motion detection algorithm proposed by Andrew Kirillov in [15]. In the following subsections we describe these algorithms.

3.4.1 Gesture Recognition Application

Figure 9 adapted from [37] shows the block diagram for the gesture recognition algorithm presented in [37]. We can see that there are two processing levels. The four blocks of

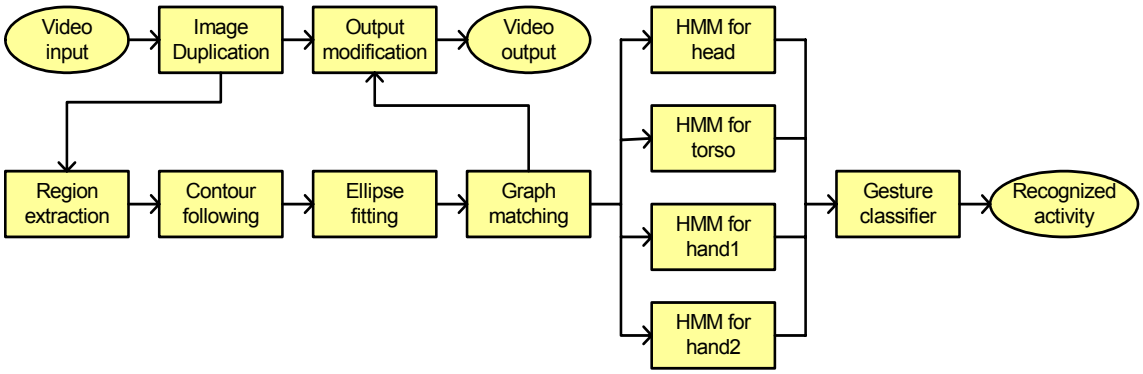


FIGURE 6. Block diagram of the gesture recognition algorithm from [37]

the low-level processing are Region extraction, Contour following, Ellipse fitting and Graph matching while the high-level processing consists of several parallel HMM processing blocks that evaluate the body's overall activity. In this work we focus on the low-level processing of this algorithm.

Region extraction assumes a YUV color model with the chroma components down-sampled by a factor of two to detect the skin areas of the input image. Contour following then uses 3x3 filters to follow the edges of the regions detected by Region. The next step is to find ellipses that fit the pixels belonging to the contours found previously. Parametric surface approximations are used to compute geometric descriptors for segments such as area, compactness, weak perspective invariants and spatial relationships. Finally, Graph matching uses a piecewise Bayesian classifier with the calculated ellipse parameters to compute feature vectors and then matches them to feature vectors of body parts (computed previously offline).

Figure 7 shows a 240x384 pixels color image used as frame input for Region, while Figure 8 shows the exact format of Region's input, a 480x384 pixels file consisting of the YUV components and image. The top part of Figure 8 is the Y component, in the middle

FIGURE 7. Original color input frame

we have the U and V downsampled components while the bottom part of the figure shows a replication of the first half of the Y component.

FIGURE 8. Region's input, YUV components of the input frame.

3.4.2 Motion Detection Application

The basic approach of the motion detection algorithm presented in [15] consists in comparing the new frames with a background frame stored. The difference for each pixel between two images is then compared to a threshold and finally an erosion filter is applied to it. These operations are performed to the grey scale images, so the first pre-processing

step would be to find the grey level component of the color RGB image. A block diagram description of this algorithm is shown in Figure 9.

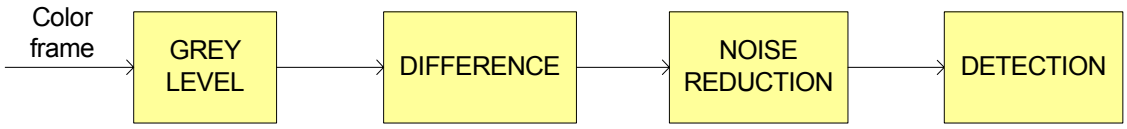


FIGURE 9. Block diagram for the motion detection algorithm

We created a grey function using Matlab Software whose input is a color image and its output is the grey level of this image, using Equation 3. Then a difference function was implemented. This function takes two grey frames and a threshold level as inputs, and outputs the difference at each pixel provided this difference is larger than the threshold level, otherwise, the difference for that pixel is assumed to be 0. To implement the erosion filter, we followed the morphological erosion filter for grey images presented in [13]. In this way, the value of each central pixel is the minimum value of its neighbors. The output image from the erosion function is finally added to the red channel of the background frame. All the scripts developed using Matlab can be found in Appendix A. Figure 10 shows an example of a four color frame sequence where the first one is considered to be the background..

Figure 11 shows the output of the different blocks for frame 3, using a threshold value of 15 and applying the erosion filter twice. Figure 12, on the other hand, shows the outputs for a threshold voltage of 60 for frame 4. The number of pixels that have different grey level between frames 1 and 2 after applying the erosion filter twice for a threshold value of 15 is 434 out of 124848 (less that 0.5%) while for a threshold value of 60 this number drops to 5 for the whole image.

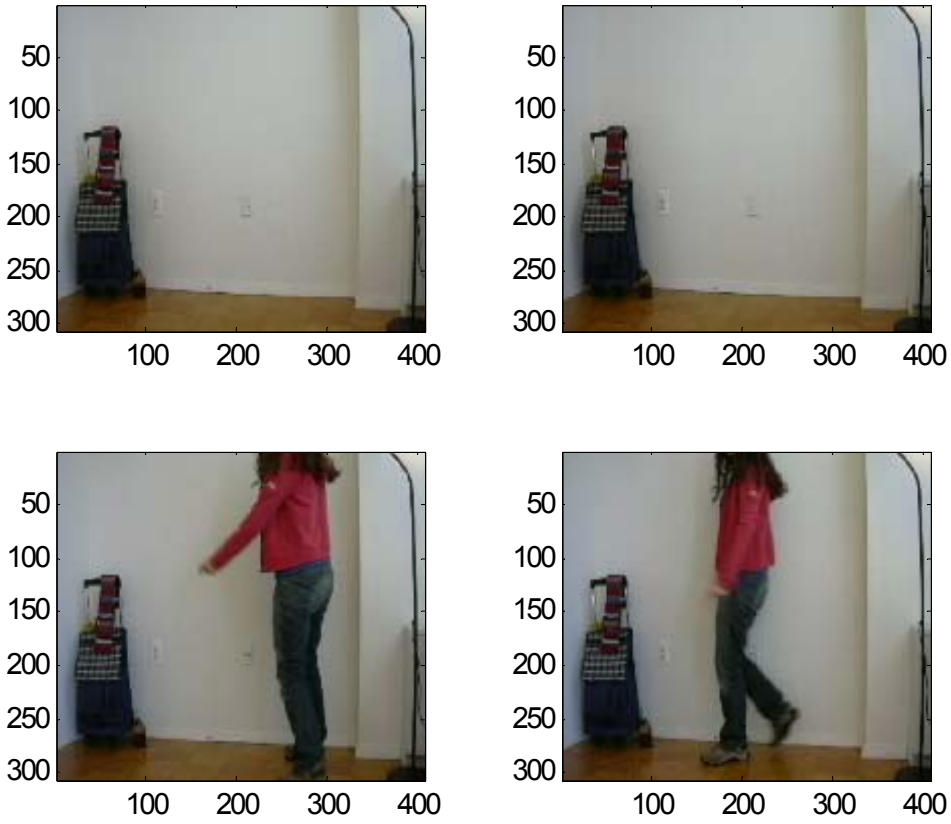


FIGURE 10. Four frames for the motion detection algorithm, the first one is considered the background.

The output of the detection block can be, for instance, the number of pixels of the current frame that differ from the background. Another possibility is to output a signal that indicates whether there is movement or not.

The setting of the threshold value would depend on the noise introduced by the camera, as well as the precision in shapes needed. In our example, a threshold value of 60 eliminates most of the shadows generated by the person moving. However, the erosion filter is still needed to eliminate the noise present in the right side of the frame. On the other hand, a threshold value of 15 would still be very efficient to detect movement since noise can be easily discriminate in the detection stage, i.e. establishing a certain percentage of

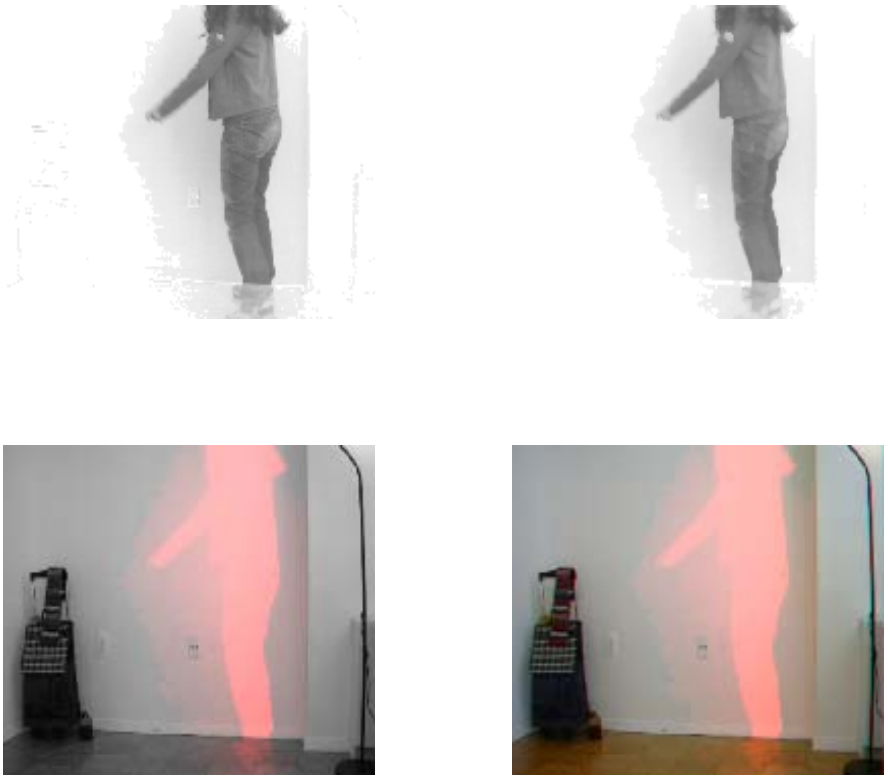


FIGURE 11. Processing frame 3: on top, outputs from the difference block with threshold = 15 and the erosion filter; bottom, detection of motion in red with grey and color background frame

pixels to differ as threshold for movement detection, but the shape of the moving object would be more diffuse.

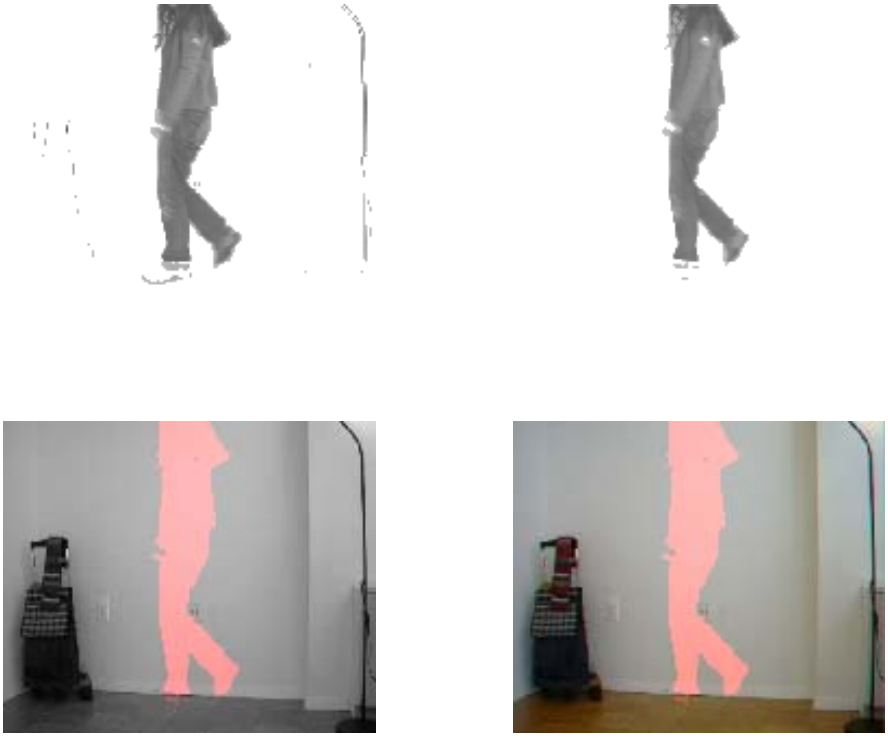


FIGURE 12. Processing frame 4: on top, outputs from the difference block with threshold = 60 and the erosion filter; bottom, detection of motion in red with grey and color background frame

Chapter 4: FPGA-based Boards for Digital Signal Processing

Algorithms in the DSP application domain can be implemented in different platforms. Microprocessors, general purpose ones but in particular DSP processors, are commonly used; for instance, in [26], the C64xx DSP processor family from Texas Instrument was targeted to implement the gesture recognition algorithm presented in [37]. Microprocessors provide high flexibility, since different algorithms can be programmed to make them perform different tasks. However, the processing speed that can be achieved using microprocessors is lower than the speed provided by custom-designed hardware for the same technology process fabrication. Application Specific Integrated Circuits (ASIC) are widely used when performance needs to be improved, usually when real-time processing is needed, or for low power or energy consumption systems.

FPGAs are programmable logic devices (PLD): they contain programmable hardware logic blocks that can be programmed to perform different logic functions. Moreover, the interconnection among these logic units can also be programmed and in this way, the same chip can implement different circuits depending on its programing. In general, each array element is a block consisting of some lookup tables (LUT) and registers interconnected. Figure 13 shows an example of a simple Logic Block. The design of each array element as well as their organization on chip depends on the vendor and on the FPGA family. In Section 4.3 the architecture of the Xilinx Virtex II family devices is studied in more detail.

For a given technology process, FPGAs are faster than microprocessors but slower than ASICs, and usually not as power-efficient as the latter. The cost of FPGAs, for small quantities, is less than the cost of ASICs, but this relationship is inverted for large quantities. That is why, traditionally, designs were prototyped in FPGAs and then implemented

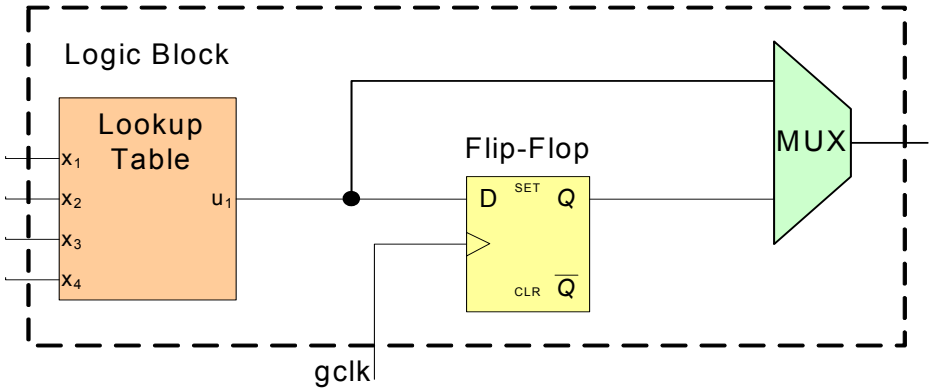


FIGURE 13. Example of a simple Logic Block consisting of a four-input lookup table and one register.

in ASICs for high production volumes. However, the gap in cost for large quantities is sometimes compensated by the time needed to translate and debug the design to ASICs, as shown in [28], where some FPGA families provide enough flexibility and performance results at a reasonable cost when compared to a similar process fabrication ASIC. Figure 14 summarizes the comparison between microprocessors, FPGAs and ASICs.

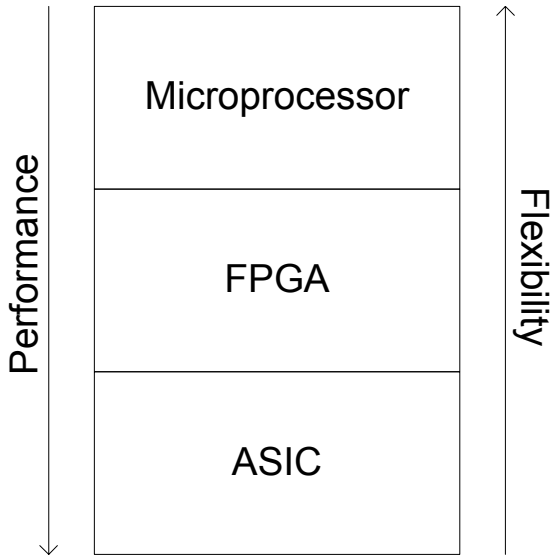


FIGURE 14. Comparison of Microprocessor, FPGAs and ASICs performance and flexibility.

In Section 4.1 we discuss the most relevant characteristics of FPGAs, as well as their main vendors; Section 4.2 presents different kinds of FPGA-based boards; Section 4.3

describes the characteristics of the Xilinx Multimedia Board and finally Section 4.4 introduces some design guidelines to build reusable blocks.

4.1 Classification of FPGA

We can distinguish FPGAs depending on their capacity, the number of user available pins, speed, and cost. In the following subsections we discuss these characteristics. Also, the main FPGA vendors are presented.

4.1.1 Capacity

The capacity of an FPGA gives an idea of the amount of logic that can be implemented with it. There are different metrics to determine the capacity. One of these metrics is “gate-counting”, which consists in establishing the number of two-input nand gates that would implement an equivalent circuit. Xilinx uses *logic cells* [32] to give an idea of the capacity. A logic cell is defined as a four-input LUT associated with a register, so that the output of the LUT may be connected to it. Figure 13 is an example of a logic cell. This metric does not take into account other resources that may be available in the array elements, such as multiplexers or RAM blocks, their *system gate* metric try to reflect these other resources as well. Altera has different metrics, its *logic elements* are analog to Xilinx’s logic cells.

4.1.2 Speed

The maximum frequency achievable in general depends on the design. FPGAs have different delay times associated to their logic implementation and usually within a family chips have different delays. Those chips having the smallest delays in a family are said to be the fastest, this information is reflected by the speed degree of the device.

4.1.3 User Pins

The number of pins for a FPGA chip varies widely. Xilinx FPGAs have from 100 to 1513 pins, depending on the family and the type of package. A large number of these pins, however, is used for power and ground connections, others are reserved, for instance to program the FPGA, and the rest can be used for input / output (I/O) purposes. A Virtex 4 chip with 1513 pins has 960 pins for I/O.

4.1.4 FPGA Vendors

The main companies are Xilinx and Altera, followed by Lattice Semi and Actel. In the second quarter of 2004 they had respectively 52%, 34%, 8% and 6% of the PLD market, according to [10]. The newest Xilinx FPGA family is Virtex 4, introduced in 2004 manufactured with a 90 nm triple-oxide process technology and a voltage as low as 1.2 V. These FPGAs have up to 200.000 logic cells. The newest FPGA family from Altera is Stratix II, manufactured on the Taiwan Semiconductor Manufacturing Company (TSMC) 1.2 V, 90 nm, 9-layer-metal, all-layer-copper, low-k dielectric process technology. The first samples were available in 2004. having up to 180K equivalent logic elements (LEs) and 9 Mbits of embedded memory. Both families can achieve speeds of 500 MHz.

4.2 Boards

FPGAs need to be connected to a power supply and a clock signal in order to work, and they also need to be connected to I/O devices. These circuits can be custom-designed and made for a given application, or bought out of the shelf. We can find boards for evaluation, for general purpose or domain-specific. Xilinx and Altera provide a wide range of evaluation boards for their chips as well as links to boards of other companies featuring

their chips. The boards may include only one FPGA or several of them with different characteristics, as well as other processing elements such as microprocessors or DSP processors. Other resources such as A/D converters or memory may also be on board. Usually they provide at least power and clock distribution and some practical input and output connectors. In Section 4.3 we present the Xilinx Multimedia Development Board.

4.3 Characteristics of Xilinx Multimedia Board

4.3.1 Overview

The Xilinx MicroBlaze and Multimedia Development Board is a platform for multimedia applications development. It has a Virtex II family FPGA, Video and TV input and output ports, communication ports such as RS-232 and ethernet, 10 Mbytes of static RAM and audio codecs, besides having power supply and system clock distribution. In the following subsections a description of the board's most relevant components for our application can be found. For a more complete description of the board, please refer to the board manual [31] and its schematics which are available online.

4.3.2 XCV2000 FPGA

A full description of the board's Xilinx Virtex II XC2V2000 FPGA can be found in its data sheet [30]. In this sub section we present its most relevant features, as well as a brief description of its architecture. For more information and a complete pin description of the chip please refer to [30].

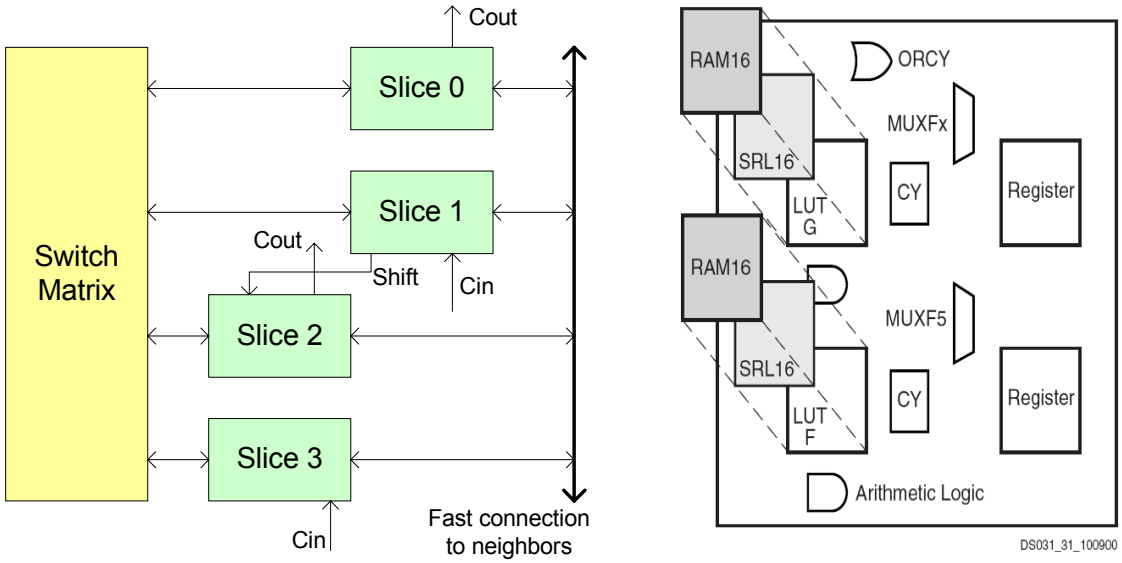
Virtex II family FPGAs were introduced in 2001 and are manufactured using 0.15 μm / 0.12 μm CMOS 8-layer metal process. The XC2V2000 FPGA has 24192 logic cells and 2 M system gates. It also has 1008 Kbits of Block RAM, organized in 56 different

blocks, each of size 18 Kbit, which is a total capacity of 126 Kbytes. In our case, the package is flip-chip fine-pitch ball grid array (BGA) with 1.00 mm pitches and dimensions of 31 x 31 mm. It has 896 pins, 624 of which are available for user input and output.

The device can be programmed by loading its configuration memory in 5 different ways: using slave-serial mode, master-serial mode, slave SelectMAP mode, master SelectMAP mode or Boundary-Scan mode (IEEE 1532). The configuration information can be optionally encrypted using Data Encryption Standard (DES) since a DES decryptor is available on-chip.

The chip has 2688 internal configurable logic blocks (CLBs), each CLB having four slices and two 3-state buffers. Each slice contains: two function generators (F and G), two storage elements, arithmetic logic gates, large multiplexers, wide function capability, fast carry look-ahead chain and horizontal cascade chain (OR gate). The function generators F and G can be configured as 4-input look-up tables (LUTs), 16-bit shift registers, or 16-bit distributed RAM memory. The two storage elements are either edge-triggered D-type flip-flops or level-sensitive latches. Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources. Figure 15 a) shows a Virtex 2 CLB composed by four slices, Figure 15 b) shows one of the slices structure.

The input and output blocks (IOBs) are programmable and have an optional single-data-rate or double-data-rate register. These registers can be D flip-flops triggered by either edge or level-sensitive latches. Several single-ended and differential standards are supported, including: LVTTL, LVCMOS (3.3V, 2.5V, 1.8V, and 1.5V), PCI-X (133 MHz



a) b)

FIGURE 15. Virtex II arquitecture: a) CLB organization. b) Slice components (figure from [30]).

and 66 MHz) at 3.3V, PCI (66 MHz and 33 MHz) at 3.3V, BLVDS (Bus LVDS), ULVDS, LDT, LVPECL.

4.3.3 Memory

One of the key features of this board is the five fully-independent banks of 512k x32 ZBT Synchronous Static RAM [24] with a maximum clock rate of 130 MHz. Although the memory devices support a 36-bit data bus and have a sleep input, pinout limitations on the FPGA prevent the use of the four parity bits as well as the low power consumption mode. The banks operate completely independent of each other, since the control signals, address and data busses and clock are unique to each bank with no sharing of signals between the banks. The byte writing capability is fully supported as is the burst-mode, in which the sequence starts with an externally supplied address.

4.3.4 Video input

The multimedia board accepts two different video input formats: Composite video and S-video (also known as Y/C video), and the signal can be either from a PAL or a NTSC source. Only one input can be processed by the board at a time, there are two switches to select the source type and the format, and this can also be configured from the FPGA through a I^2C protocol. The analog signal is then decoded by the ADS7581 chip, which outputs a 10 bit digital signal with YC_bC_r 4:2:2 CCIR601/CCIR656 format, as well as two clock lines. Figure 16 summarizes the video signal path in the Multimedia Board.

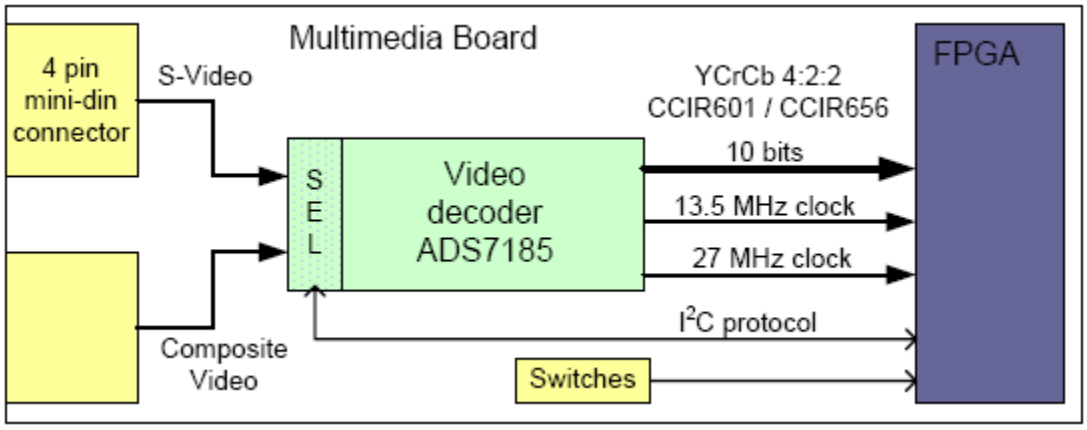


FIGURE 16. Video input to the FPGA

4.3.5 Serial Ports

Three serial ports are provided on board. A RS-232 port shares its connections to the FPGA with two PS2 ports for mouse and keyboard. In order to determine which ones are active, the board has two switches (Serial Port Selection switches) which provide four possible choices: disable all the ports, use RS-232 with handshake signals (CTS, RTS and CSR signals), use RS-232 without handshake and the PS2 keyboard port, or use the two PS2 ports.

4.4 Reuse Methodologies

Nowadays it is very difficult to develop a complex hardware project consisting of millions of gates from scratch. In [14] the authors state that casual reuse of code lowers the costs between 2 and 3 times, and that this rate is still insufficient to make competitive designs. Some guidelines are provided in their book, both to write reusable code and to be able to reuse third-parties' code. Although they focus on SoC ASICs, most of the RTL coding recommendations also are applicable to FPGA-targeted code. In this work we implemented some generic blocks taking into account these guidelines. In particular, the memory and video controller blocks for the Multimedia Board were designed to be easily reused in other projects. In the following subsections we present the Intellectual Property (IP) cores, a key concept for reuse and then we show different standards available to establish their interfaces.

4.4.1 IP Cores

IP cores, also called IP, IP blocks, macros and virtual components are design units that can reasonably be seen as stand-alone subcomponents of a complete design [14]. Having blocks which are already verified and tested available for a new design, enables the designer to build complex systems faster and in a more reliable way.

Several companies are selling IP cores, some of them are FPGA Companies, such as Xilinx and Altera, that provide IP cores targeted for their chips. Others provide generic IP cores; for instance, Mentor Graphics offers communication interfaces and microcontrollers as well as products in Ethernet, USB, Storage and PCI Express. ARM also offers several IP blocks, such as processors, memory, and cores for secure applications. Opencores [18] is a dynamic group of people interested in "developing hardware with a similar ethos

to free software movements". Several IP cores are freely available at their webpage, where they are classified in different categories such as communications, dsp, memory, microprocessor.

4.4.2 IP Cores Interface

In order to be reusable, IP cores need to have a common, standard, interface. There are several standards for interfacing IP cores. In this section we present three of them, each one recommended by different organizations.

The VSI Alliance (VSIA) recommends the Open Core Protocol (OPC) maintained by the OPC International Partnership (OPC-IP). This standard is available for members only. Non-members can have a royalty-free license but only for evaluation and research of the protocol.

The Advanced Microcontroller Bus Architecture (AMBA) specification [1] from ARM, an IP company established in 1990, is a widely used on-chip bus. This is the standard recommended in [14]. The AMBA specification defines three different buses and a test methodology for the interface. These buses are: the Advanced High-performance Bus (AHB), the Advanced System Bus (ASB) and the Advanced Peripheral Bus (APB). The first two are very similar, their bus cycles start and end in positive clock edges, and they are both recommended for high-performance system modules. The APB has bus cycles starting and ending in negative clock edges, and is recommended for low power applications, since its reduced interface complexity consumes less power.

As it is stated in its objectives, the AMBA specification targets embedded microcontroller products with one or more CPU or DSP processor. Although the specification is

practical for a microprocessor-based architecture, in a system without microprocessor cores, the standard turns out to be less convenient considering the overhead added.

The Wishbone interface [17] is the specification recommended by Opencores to interconnect cores in a chip (FPGA, ASIC, etc). This standard is not copyrighted, and is in the public domain. It may be freely copied and distributed by any means, and used for the design and production of integrated circuit components without having to pay royalties or other financial obligations. The standard includes read and write single cycles as well as block transfers. There are two types of cycles: unregistered and registered. Although the protocol clearly specifies these cycles, it also provides some tags that can be defined by the user, providing design flexibility. Also, in order to be Wishbone compliant, their documentation standard has to be followed. This is to ensure easy reuse of previously designed modules, specially if user tags are defined. The standard is independent from the hardware used to implement the circuit as well as from the testing and verification methods chosen. Among the IP cores that Opencores offers, several are Wishbone compliant, and can be used in a more complex design.

In this work we chose to follow the Wishbone standard for registered cycles. A more detailed description of the registered read and write cycles is presented in Chapter 7 along with the designed blocks.

Chapter 5: Proposed Modeling Technique: HPDF/CSDF

One important contribution of this work is demonstrating the integration of CSDF base model semantics into the HPDF meta-modeling framework. This integration, which was developed jointly with Mainak Sen, provides simultaneous application of the bounded memory, dynamic parameterization of HPDF and the finer granularity, phased decomposition of actor execution in CSDF. In this Chapter we describe how we integrated HPDF and CSDF. Furthermore, we present an example where a compact looped notation is introduced. Finally, we show how to find valid schedules for HPDF/CSDF and discuss their advantages.

5.1 Description

As it was presented in Section 2.6, the homogeneity requirement in HPDF is in the sense that data transfer across an edge (production and consumption) must be equal (but not necessarily constant or statically-known) across corresponding invocations of the source and sink actors. On the other hand, in CSDF a complete invocation of an actor involves execution of all of the phases in a fundamental period of the actor, as it was explained in Section 2.4. Integration of CSDF with HPDF allows the number of phases in a fundamental period to vary dynamically, and also allows the number of tokens produced or consumed in a given phase to vary dynamically. Such dynamic variation must adhere to the general HPDF constraint, however, that the total number of tokens produced by a source actor of a given edge in a given invocation (which, in the case of phased actors, means a given fundamental period) must equal the total number of tokens consumed by the sink in its corresponding invocation. Thus, for all positive n, the number of tokens

produced by the nth complete invocation of a source actor must equal the number of tokens consumed by the nth complete invocation of the associated sink actor.

For fundamental periods that involve dynamic token transfer, this can be accommodated by employing a special token that delimits the end of a fundamental period of a source actor. The source actor produces this special end-of-invocation (EOI) delimiter just after the end of each complete invocation. The HPDF restriction then requires the following:

Suppose that the sink actor of a dynamically parameterized HPDF edge e consumes the last token in its ith invocation (fundamental period of phases) at time $z_i(t)$. Then just after completing $\delta(e)$ more consumption operations after time $z_i(t)$, the sink actor will consume an EOI token, and it will not consume any EOI tokens before that. This pattern must hold for all positive integers i (i.e., all invocation indices); that is, after each complete sink invocation, the next EOI token is consumed after exactly $\delta(e)$ consumption operations. Furthermore no EOI token should be consumed during the first invocation $(i = 1)$ of the sink actor.

The above formulation is useful for precisely specifying how HPDF applies to dynamic parameterization of CSDF actors. The formulation can also be used to generate code for quasi-static schedules, and to verify consistency of HPDF specifications at run-time (i.e., to detect violations of HPDF behavior as soon as they occur).

5.2 Examples

Figure 17 shows an example of an HPDF/CSDF graph where we adapted the dataflow looped schedule notation (e.g., see [2]) to represent the tokens being produced and consumed in a fundamental period. This graph has four actors. Actor A has a cycle period

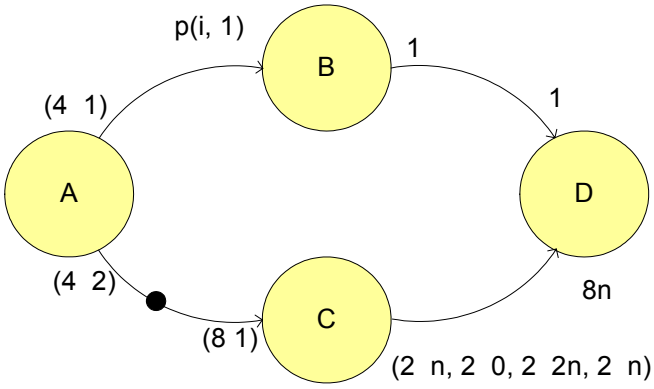


FIGURE 17. Example of an HPDF/CSDF graph

equal to 4; in each phase it produces one token in edge AB and 2 tokens in edge AC. Actor B has a parameterized number of phases $2p$ and also consumes a parameterized number of tokens during its odd phases. In this example, in order to remain homogeneous, the values that p can take in any invocation are limited to 1, 2, 3 and 4; Table 1 shows the different combinations of tokens consumed that can happen in any actor invocation. Actor C has 8 phases, in each phase it consumes 1 token from edge AC and produces a different parameterized number of tokens in edge CD depending on its phase: in the first two phases it produces n tokens, in the following two it does not produce any token, in the next two phases it produces $2n$ tokens and finally in each of the two remaining phases of

its invocation it produces n tokens. Actor D always consumes $8n$ tokens from edge CD. An initial delay is present in edge AC.

TABLE 1. Possible token consumptions for one invocation of actor B

p	tokens consumed in a fundamental period invocation
1	(3, 1)
2	(1, 1, 1, 1)
2	(0, 1, 2, 1)
2	(2, 1, 0, 1)
3	(1, 1, 0, 1, 0, 1)
3	(0, 1, 1, 1, 0, 1)
3	(0, 1, 0, 1, 1, 1)
4	(0, 1, 0, 1, 0, 1, 0, 1)

5.3 Scheduling

Due to the homogeneity property along edges for each invocation, we can have in a very simple way a consistent schedule, such as with HPDF; the only difference is that now the repetition number for each actor instead of being one equals the number of phases it has. Moreover, for a DAG, if we fire the actors a number of times equally to their repetition number in topological order we have a valid schedule.

Formally, let P_a be the least common multiple number of phases from all incoming and outgoing edges from actor a, then we say that the fundamental period of actor a is P_a. Now let's first prove that we have a consistent schedule, that is to say that if we fire each actor a of the graph P_a times then all the tokens produced are consumed, ending up only with the initial tokens in the buffers. In order to prove this statement, let's suppose that we have an extra token in an edge a_1a_2. That means that after a full invocation of actor a_1, m tokens were produced in the edge a_1a_2 but only $\delta(a_1a_2)+m-1$ were consumed by

actor a_2, leaving $\delta(a_1a_2)+1$ in that edge instead of just $\delta(a_1a_2)$. But this is in contradiction with the homogeneous property of the HPDF/CSDF graph. Therefore, if each actor a fires P_a (its repetition number) times we have a consistent schedule.

Furthermore, if now we traverse the graph in topological order firing each actor P_a times we also have a valid schedule.

Accordingly, in our example, a consistent and valid schedule would be:

$$(4A)(2pB)(8C)D \quad \text{(EQ 9)}$$

However, a more efficient schedule in terms of buffer requirements would be:

$$4(A(2C))(2pB)D \quad \text{(EQ 10)}$$

In the schedule represented by Equation 9 actor A fires four times producing eight tokens in edge AC, however in the schedule from Equation 10 C, each time A produces two tokens C consumes them. Therefore, in the first case a buffer of size 8 is needed while in the second case a buffer of size 2 is enough.

Furthermore, there can be other valid schedules as well, but we will always get a valid schedule in the proposed way, provided that such a schedule exists.

Chapter 6: Modeling Applications with HPDF/CSDF

In this chapter we show how to use HPDF/CSDF to model the gesture recognition application [37] that was described in Section 3.4.1. We focus on aspects that were not considered before [25], such as integrating the input to the model and exploiting further levels of parallelism without losing the modeling advantages introduced before. We also provide different schedules for this application in Section 6.3, whereas an analysis on how these different schedules can affect the memory organization of the system and consequently its performance and energy consumption is introduced in Chapter 7. Finally in Section 6.4 we show how to model the motion detection algorithm [15] that we described in Section 3.4.2.

6.1 Modeling the input

The input to the gesture recognition system comes from a video camera. In the algorithm, each frame of the video is assumed to have 384 x 240 pixels. Although each pixel has 3 components, the luminance Y, and the chrominances Cr and Cb, in this application the chroma components are downsampled as described in Section 3.4.1. In a first approach, we assume that the video output produces all $YCrCb$ components and that the downsampling is performed in Region. We model the input as a cyclo-static actor having 384 x 240 = 92160 = (s) phases, where each phase corresponds to a different pixel. Using the same adapted looped notation that we introduced in Section 5.2, the input (source) actor can be compactly represented as producing $(s\ 1)$ tokens in its fundamental period (1 token on each of s successive phases).

The static part of our system is now modeled as shown in Figure 18. The model cap-

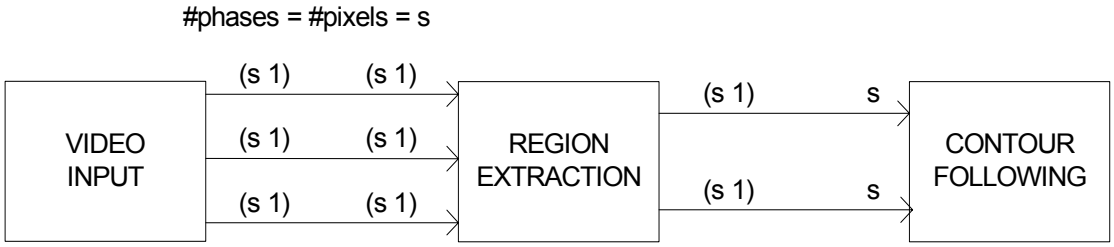


FIGURE 18. Model of the static part of the system

tures now the pixel-level parallelism present in Region, and also expresses the frame regularity of the algorithm — i.e., after s phases, we start processing a new frame. With this regular, fine granularity CSDF representation, we can explore implementations with different architectures that may exploit both the pixel and frame parallelism.

In particular, we take advantage of this representation of the algorithm for our implementation in two aspects. First, we observe that the application can be pipelined into five blocks, each one processing a different frame. The second improvement is in the memory organization and is discussed in detail in Chapter 7.

6.2 Modeling Dynamicity

Contour needs to wait until the whole frame is available to start executing. Although this is true for the worst case, most of the times Contour can fire prior to having the whole frame. In order to model this, we divide the behavior of Contour into two phases, as shown in Figure 19: the first one scans the image looking for a contour and continues until it finds the starting point of one, thus consuming X_i pixels, without producing any output tokens; the second phase follows the contour and finds all the contours that are overlapping with each other. Now instead of processing the whole image, it will only process the subimage

that goes from where a contour starts until all overlapping contours present are completed, thus consuming Y_i pixels. The output of this phase consists of k_i tokens. Each one of these output tokens is made up of a list of pixels belonging to a contour.

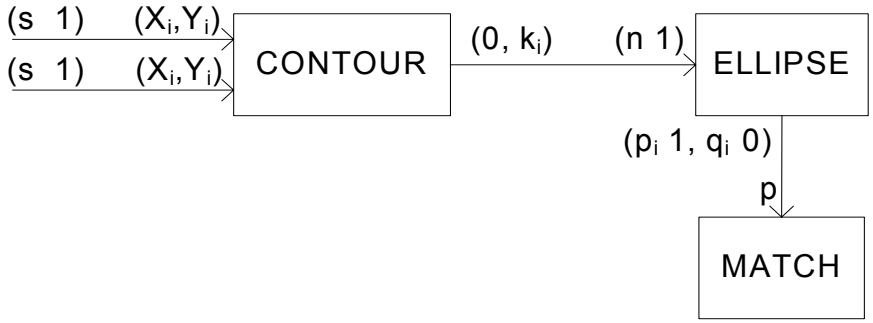


FIGURE 19. Model of the dynamic part of the system

The HPDF condition as previously developed is respected here since no matter how the processing of Contour gets broken down into phases (based on the actual input image), the total number of pixels consumed in a frame by Contour equals the number of tokens produced for that frame by Region. That is,

$$s = \sum_i (X_i + Y_i). \quad \text{(EQ 11)}$$

We model the input edge of Ellipse as a dynamically parameterized CSDF edge, capturing the dynamicity of Contour's production in the number of phases of Ellipse. The output of Ellipse is a set of parameters describing the ellipse whenever it can fit one to the contour, so for each of the n contours its output is either one token or zero token. The Match actor has to wait to have all the p ellipses available before it can execute. The HPDF condition is also respected here since match has to wait for all the Ellipses found, that is:

$$p = \sum_i p_i \text{ with } n - p = \sum_i q_i \quad \text{(EQ 12)}$$

It is easy to verify that the associated edges remain homogeneous in the sense of HPDF: an invocation of Contour will produce n tokens, based on the contours found in the current frame, while Ellipse will consume one token in each of its n phases (dynamically parameterized number of phases configured based on the pattern of EOI tokens on the edge). Similarly, Ellipse outputs tokens throughout n phases such that the sum of tokens over the phases is p, which is the number of tokens consumed by Match.

6.3 Scheduling

We can have different scheduling strategies depending on the implementation constraints that are most important. This is an advantage of dataflow modeling in general, and the utilization of HPDF enhances this advantage for the targeted class of applications. If data is passed between actors as vectors of different lengths, so that a static number of vector tokens whose lengths are dynamic are exchanged between a source and a sink, then we can have a very simple scheduler in place. In the HPDF/SDF model, if the edges $(\text{Contour}, \text{Ellipse})$ and $(\text{Ellipse}, \text{Match})$ receive and deliver one vector token each of length n and p respectively, and we consider a frame granularity, then a valid schedule of the graph would be

$$VRCEM. \quad \text{(EQ 13)}$$

We can apply this concept of variable length tokens to find the schedule for a general HPDF graph. However if more granularity is expressed in the model, as we have done by integrating HPDF and CSDF, and if we want to exploit this finer granularity specification,

we need to have a parameterized schedule and exchange data in a more fine-grained way. For our application, if it is modeled as in Figure 18 and Figure 19, a valid schedule that can be easily found exploiting the homogeneous property as shown in Section 5.3 would be:

$$(s\ V)(s\ R)(2I\ C)(n\ E)M. \quad \text{(EQ 14)}$$

In this schedule, the video input fires s times to provide the s pixels of a frame, while Region also fires s times, once per frame pixel. Contour fires $2I$ times, where I is the number of non-overlapping contours found in the current frame, since in its odd phases it searches for contours and in its even phases it follows the detected contours, and this happens I times in each frame. Ellipse fires n times, once for each contour and Match fires only once per frame. However, this basic schedule can be improved by grouping executions of Video and Region phases using the following modified schedule:

$$(s\ VR)(2I\ C)(n\ E)M. \quad \text{(EQ 15)}$$

Efficient quasi-static schedules of this form are enabled by the integrated HPDF/CSDF methodology that we have developed in this work. Moreover, buffer requirements decrease as it is shown in Chapter 8. Here, detection of EOI tokens as described before, can be used to control the loops whose iteration counts are based on dynamically parameterized CSDF structures.

6.4 Motion Detection Application

The motion detection algorithm [15] described in Section 3.4.2 can be modeled with 5 actors as shown in Figure 20, where Greylevel takes the grey level of the image, Difference performs the difference of the two images and compares it with the threshold, Noise

Reduction implements the erosion filter and outputs the contour of the moving object, and a Video actor was added to model the input. Count is a high-level processing actor that identifies and counts the number n of moving blocks in the frame and a following actor may take a decision depending on this number. We are going to focus on the low-level processing stage in the dashed box in Figure 20. We observe that both the grey scale computation and the difference with threshold have pixel level parallelism, which is captured by the model.

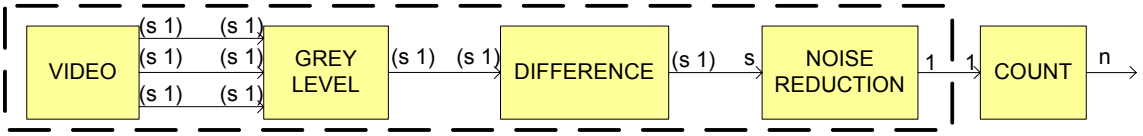


FIGURE 20. Modeling of the motion detection application

Using again the homogeneous property as shown in Section 5.3, a valid schedule for this application would be:

$$(sV)(sG)(sD)N \quad \text{(EQ 16)}$$

We observe that a more efficient schedule in terms of buffer size would be:

$$(s(VGD))N \quad \text{(EQ 17)}$$

Chapter 7: Memory Management

Image processing applications in general require large amounts of memory to implement buffers between actors. In several cases the amount of memory available on chip is insufficient, and consequently external memory has to be used. Accessing external memory can be very costly both in terms of performance and energy consumption and that is why having an efficient memory organization becomes so critical.

In this chapter we study different possible memory organizations that can be inferred from a given application HPDF modeling. In general, the amount of memory needed to store data produced by the actors of a HPDF model will depend on the input. However, some actors may consume or produce a fixed amount of data. For this study we focus on a particular image processing application: the gesture recognition algorithm [37] described in Section 3.4.1, and a particular hardware platform: the Xilinx Multimedia Board described in Section 4.3. First we analyze the memory requirements and possible memory configurations when using HPDF/SDF modeling for this application [26]. In Section 7.1 we discuss these requirements and configurations for a single frame [27] and in Section 7.2 we extend the discussion for a stream of frames by adding some input actor information to the model. Then, in Section 7.3 we propose a method to find a convenient memory organization using HPDF/CSDF to model the application. Finally, in Section 7.4 we design a reusable memory controller core for the Multimedia Board that can be used to access the board's external memory.

7.1 HPDF/SDF Modeling, Single Frame

In [27] the application is modeled at a frame level using HPDF/SDF, as shown in Figure 21. Region consumes three tokens and produces two, each one of them having 384x240 pixels. It can be observed that modules Region and Contour always process the whole image, even though Contour's output is data dependent. In the dataflow graph we have two tokens, each representing a whole image as the input needed for Contour. Contour needs to wait to have the whole images before starting with the processing, and will scan serially each image.

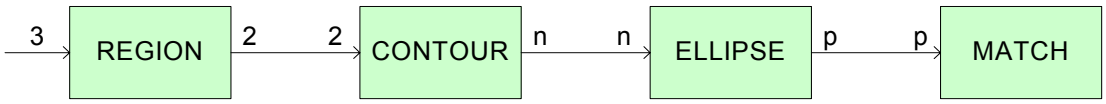


FIGURE 21. Modeling of the gesture recognition application with HPDF/SDF with frame granularity

Consequently, our static memory requirements, for each frame being processed, are such that we can store three input images and two output images for Region. These frame sizes are considerably big, in our case 384 x 240 pixels occupy 90 Kbytes, so for storing a total of five images we would need 450 Kbytes. Our Virtex II chip has 56 selectRAM blocks, each of them of 18 Kbit, giving a total on-chip storage capacity of 126 Kbytes. However, external on board memory is available (as described in Section 4.3 we have 5 RAM banks that can store 512 K words 32 bits long each), having a total on-board storage capacity of 10 Mbytes. The total amount of memory needed for image storing, 450 Kbytes, is then less than 5 % of the external memory capacity available on board. Yet, the organization of the images in the memory can dramatically change the number of memory access cycles performed and the number of banks used. These trade-offs also involve the total power and energy consumption.

Several strategies are possible for storing these images in the memory. The simplest one (Case 1) would be to store each of the five images in a different memory bank, using 90 K addresses and the first byte of each word. In this way, the five images can be accessed in the same clock cycle (Figure 22a). The HPDF/SDF model does not provide information about the existing pixel level parallelism, however, we observe that Region reads and writes the images always in the same order. Taking advantage of this fact we can minimize the number of memory banks used (Case 2). Thus, we can store the images in only two blocks, using each of the bytes of a memory word for a different image, and still access all the images in the same clock cycle (Figure 22b).

On the other hand, the best configuration in order to minimize the number of memory access cycles (Case 3) would be to store each image in a different bank, but using the four bytes of each memory word consecutively (Figure 22c). Other configurations are possible: for example (Case 4) we can have two images per bank, storing 2 pixels of each image in the same word (Figure 22d). Table 2 summarizes the number of banks and memory access cycles needed for each of these configurations.

Case 3 seems to be the most convenient memory organization form, the time associated to the images reading and writing is 69120 memory access cycles, and the total number of memory access cycles is also the lowest, 161280, which makes us assume that the power consumption will be lowest in this configuration though we are using all five memory banks. However, since Contour does not process pixels in a consecutive way, in order to use this configuration we may require a higher number of internal buffers. Figure 22 shows all the cases we discussed.

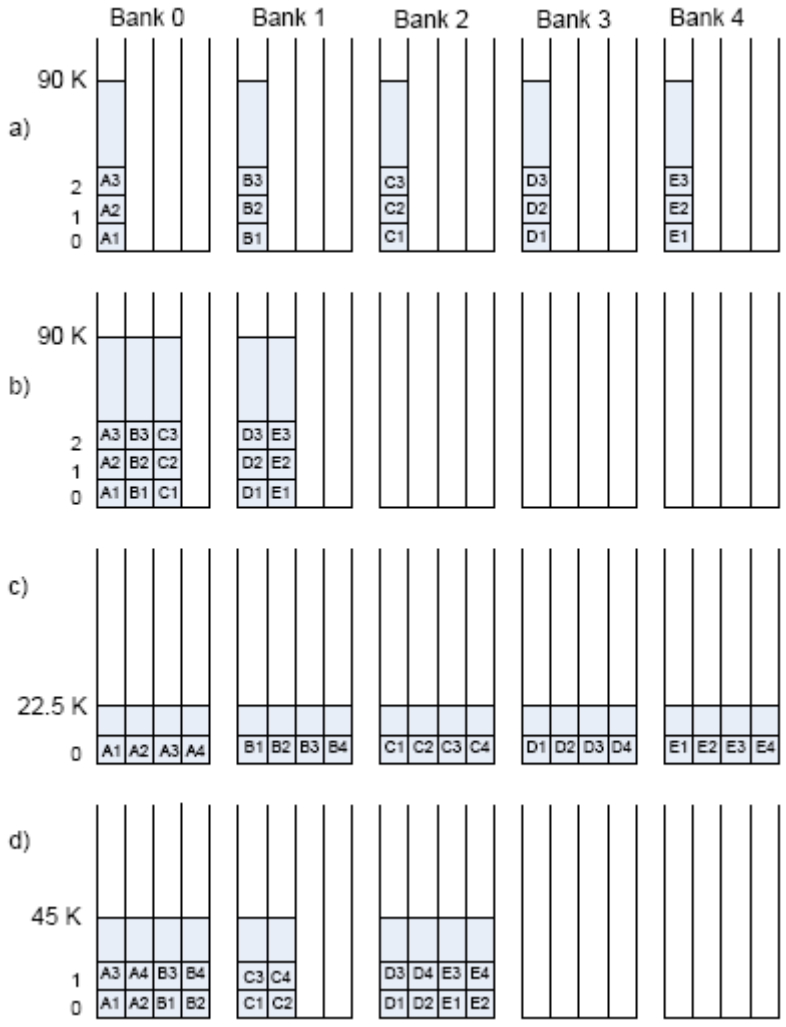


FIGURE 22. Possible memory organizations for a single frame. a) Case 1: Simple organization, each image in a different bank; b) Case 2: Minimum number of banks used. c) Case 3: Optimizing number of memory access cycles; d) Case 4: Another possibility

7.1.1 Power and energy consumption analysis

According to the memory datasheet, each bank consumes a maximum of 1.254 W when selected and 0.099 W otherwise. Assuming writing and reading cycles that last 3 clock periods (see design implemented in Section 7.4) and working at the maximum memory supported frequency (133 MHz), we can calculate E, the energy needed to write and read the images using Equation 18,

$$E = P \times \frac{C}{fmax} \times 3 \quad \text{(EQ 18)}$$

where P is the power consumed by the memory when selected, C is the total number of memory cycles and f_{max} is the maximum frequency supported by the memory chips. The factor of 3 can be reduced to 1 using burst mode for Region memory accesses.

This analysis assumes that Contour only reads memory location once, that is to say that internal buffers are available when needed. Table 2 shows the number of cycles required for each case analyzed as well as the energy consumed.

TABLE 2. Memory access cycles and energy needed for each configuration analyzed

Configuration	Banks used	Read cycles Region	Write cycles Region	Read cycles - Contour	Total non-overlapping cycles	Total number of cycles	Energy (mJ) at max freq
Case 1	5	92160x3	92160x2	184320x1	276480	645120	18.25
Case 2	2	92160x1	92160x1	184320x1	276480	368640	10.43
Case 3	5	23040x3	23040x2	46080x1	69120	161280	4.56
Case 4	3	46080x2	46080x1	92160x1	138240	230400	6.52

7.2 HPDF/SDF Modeling, Video Stream

With our model of the application, we have now two strategies to tackle the problem of having a stream of frames instead of a single frame as input. The first strategy would be to consider the execution and completion of the four stages of the algorithm serially before processing the next frame. However, the HPDF/SDF modeling shows frame level parallelism which we can exploit by pipelining the four modules, obtaining in this way a more efficient implementation. The second strategy would then consist in taking advantage of this property. The challenge is now to store the images in such a way that the different modules do not stall waiting for memory access. For instance, if Region is writing in a memory chip, Contour cannot access at the same time other location of the same chip

memory, since there is only a single data bus in each memory bank. It is clear then that the cases analyzed for the single frame are not well-suited for the multiple frames case.

Another aspect that we need to consider now is how we obtain in our hardware platform the inputs to Region, since we need to organize adequately the way the images are stored in memory. As explained in Section 4.3.4, the board video decoder output has the YC_bC_r 4:2:2 format, where the 4:2:2 ratio refers to the subsampling of the chroma components. It also provides two clocks, one at 27 MHz and the other at 13.5 MHz. Figure 23 explains how the three components of each pixel are serialized. Each of these component levels is coded with 10 bits. In our analysis we are only going to consider the eight most significant bits of the digitalization..

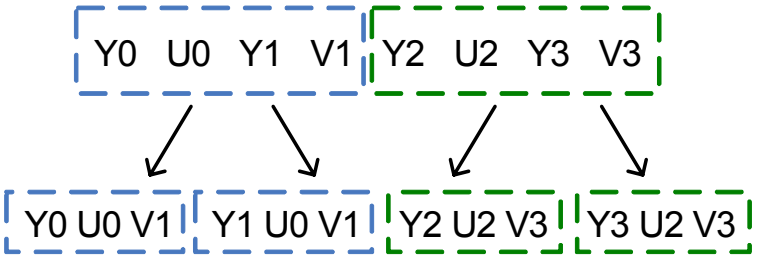


FIGURE 23. 4:2:2 YUV pixel description: the values of chrominance are subsampled by a factor of 2.

Figure 24 shows a suitable memory organization to solve these issues. The key idea is to receive each pixel as it is being received with the Video Acquiring module (V), that then stores the three images needed by Region in one of the external memory chips. This module will alternatively use banks 0 and 1 to store the images. While V is writing a new frame in bank 1, Region will be processing the previous frame stored in bank 0. Thus both modules can execute at the same time while working in different frames (pipeline). The same idea is applied for the Region-Contour interface: Region writes its output alternating banks 2 and 3, so Contour can process a frame while region is processing a new frame.

Contour's output is data dependent. However, most of the times the number of pixels belonging to the contours will be much smaller than the total number of pixels of a frame. Knowing this, we can use the internal selectRAM blocks to write different contours in them; in the rare cases where there would be more than 56 contours or the storage would not be enough, we can use other options such as stalling the pipeline or using external memory bank 4. The main advantage of this approach is that we can have several instances of ellipse working at the same time with different contours, and they would not have to wait to have all the contours to start working. Even though Match needs to wait for all the ellipses to be done before starting, ellipse will complete much faster in this way.

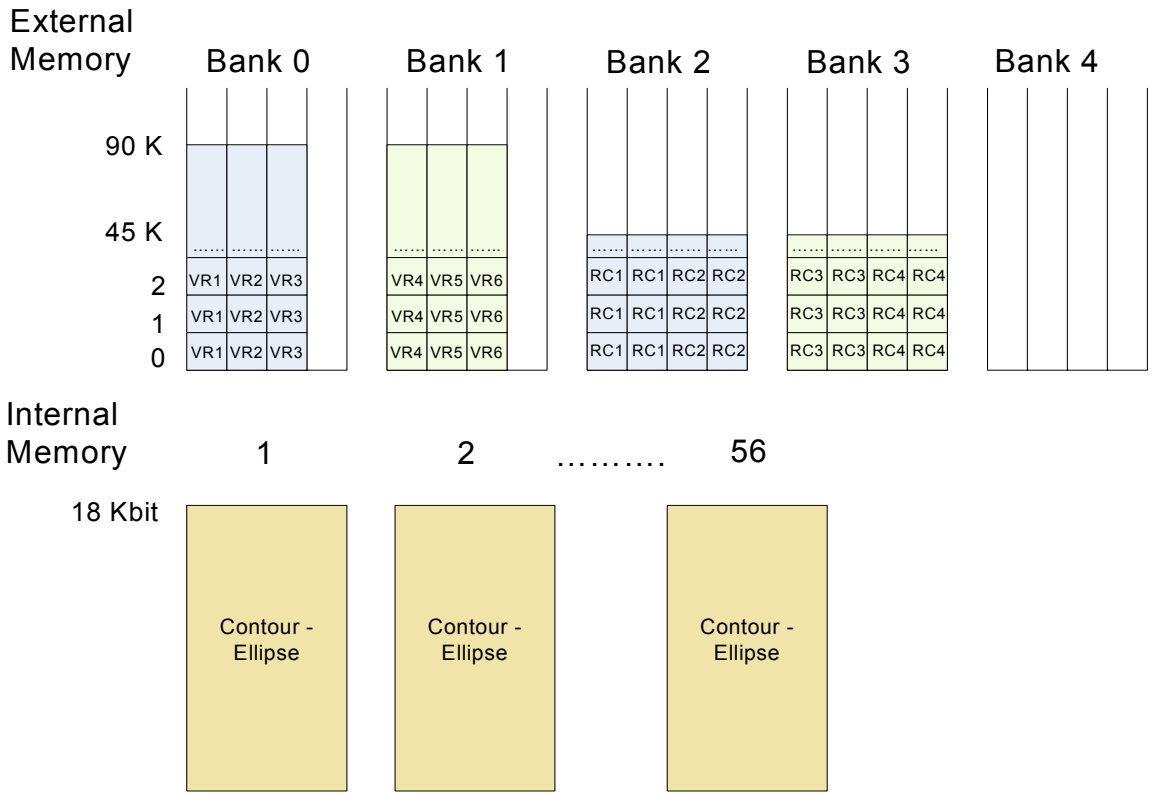


FIGURE 24. Memory organization to pipeline the algorithm when having a stream of frames: V writes the three images in blocks 0 and 1; Region reads from 1 and 0 and writes to banks 2 and 3; Contour reads from banks 3 and 2 and writes to internal selectRAM; Ellipse reads from internal selectRAM.

7.2.1 Power and energy consumption analysis

In order to estimate the energy needed to access the external memory per frame, we can also use Equation 18, where the total number of memory access cycles C is calculated adding the writing cycles of module V. In this way, for the memory organization proposed in this section we have a total of 90 K writing cycles from V, 90 K reading cycles from Region, 45 K writing cycles from Region and 90 K reading cycles from Contour, assuming again the availability of internal buffers for Contour. For the total number of accesses C = 315 K, the energy consumed by the external memory when selected, assuming maximum frequency, would be 9.12 mJ.

7.3 HPDF/CSDF Modeling

In the previous sections we observed that in this particular example the HPDF/SDF modeling had some limitations: for instance, although frame level parallelism was expressed by the model, pixel level parallelism also existing in the application was not explicitly shown. When analyzing different memory organizations we took advantage of this property, however, as shown in Chapter 6, HPDF/CSDF modeling of the application expresses both pixel level and frame level parallelism, as well as includes an input actor. In this section we find an efficient memory organization from the HPDF/CSDF modeling of the application as shown in Figure 18 and Figure 19.

For our hardware platform, we consider a given memory organization scheme to be the best from an energy consumption point of view if it is the one that minimizes the number of accesses to memory chips. This consideration can be justified by noticing that the power consumption of memory chips is more than ten times higher when accessed, so we want to minimize the total number of memory accesses. It could be argued that using dif-

ferent chips also requires more switching logic programmed in the FPGA and that therefore, minimizing the number of chips accessed would be also desirable. But, as it is shown in Chapter 8, the power overhead of a memory controller core is negligible.

Similarly, we consider a given memory organization scheme to be the best from a performance point of view if it minimizes the total number of non-overlapping accesses to memory chips, where two accesses are overlapping if they happen at the same time.

For a sequential frame execution of the algorithm (i.e., without pipelining) a configuration where we use all the possible memory bandwidth at every access is then the best possible memory organization in terms of performance, since it obviously minimizes the number of non-overlapping memory accesses. Interestingly, this configuration also minimizes the total number of memory accesses and in this way it is the one that consumes the least energy as well.

If we implement a pipelined architecture, however, using the whole bandwidth is not the best option anymore. Memory management can be better adapted in this case in the same way we did in Section 7.2, by swapping banks. The difference now is that we are modeling the application with HPDF/CSDF. Therefore, we can use the efficient schedule in Equation 15 to infer that Region can process each pixel as it is arriving from Video and thus, the whole Video output frames do not need to be stored before Region starts processing them. In this way, trying to maximize the memory bandwidth for each set of swapping banks being used, we proposed to store Region's output tokens in banks 0, 1, 2 and 3, swapping the first two with the latter pair as shown in Figure 25. With this scheme, the total number of memory accesses is 92160 per frame while the total number of non-over-

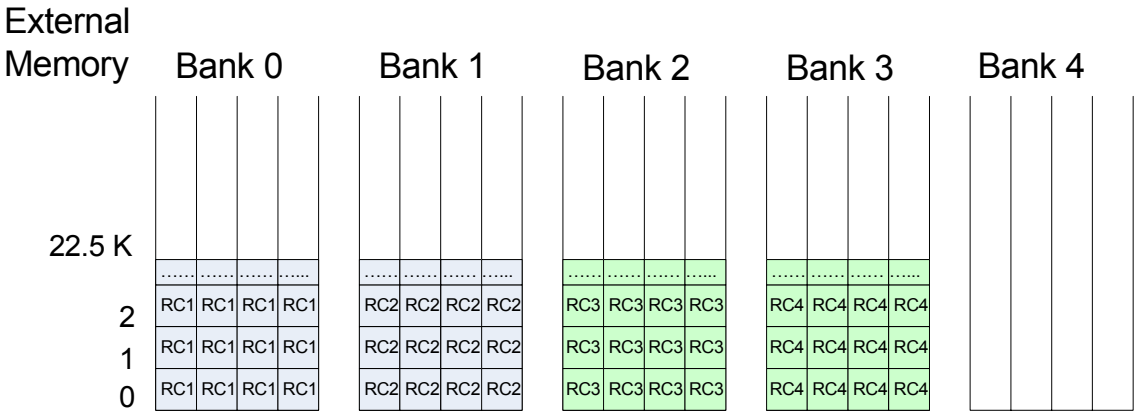


FIGURE 25. Memory organization inferred from the HPDF/CSDF modeling of the application: while region writes the four bytes of banks 2 and 3, Contour reads the previously written four bytes from banks 0 and 1.

lapping accesses for two pipelined frames is 23040 and the corresponding energy consumption is 2.66 mJ. A comparison with the other memory organization schemes discussed in this chapter is given in Chapter 8.

7.4 Memory Controller Core

In this section we briefly describe the Wishbone compliant slave memory controller core that we designed. Its Wishbone Datasheet is provided in Appendix D.

7.4.1 Input and Output Signals

The core interface signals can be classified in two groups: on one side the signals that are wired to the memory chip and implement the chip access cycles as specified in [24] and on the other side the signals that provide the Wishbone interface. A description of the Wishbone specification signals that we used in this design are described in Appendix C. Figure 26 shows the input signals on the left side and the output and bidirectional ones on the right side. All the signals communicating with the memory chip have a name starting with MEMORY_BANK0_ and a suffix _P or _N indicating if these signals are active high or low respectively. This notation was adopted from a Xilinx core example. The Wishbone

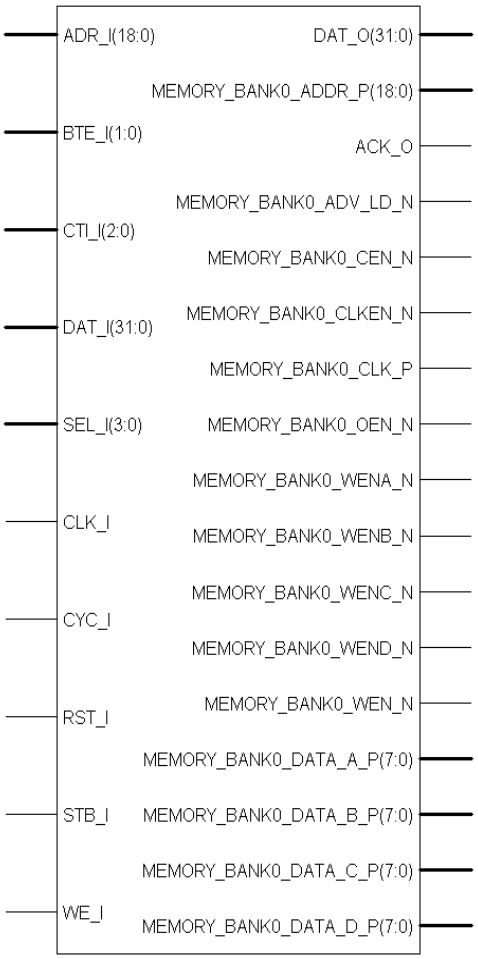


FIGURE 26. Input and Output signals from the wishbone memory controller designed

interface signals, on the other hand, are all active high and have a suffix _I when they are inputs and _O if they are outputs.

7.4.2 State Diagram

Figure 27 shows a simplified state diagram of our design, where inputs were omitted to enhance clarity and only possible transitions are shown. The states in dashed circles have to be present if we want to follow the guidelines from [14] (commented in Section 4.4) and have registered outputs. We implemented two designs, the first one did not have registered outputs; this feature was added to the second design to follow the guidelines for reusability.

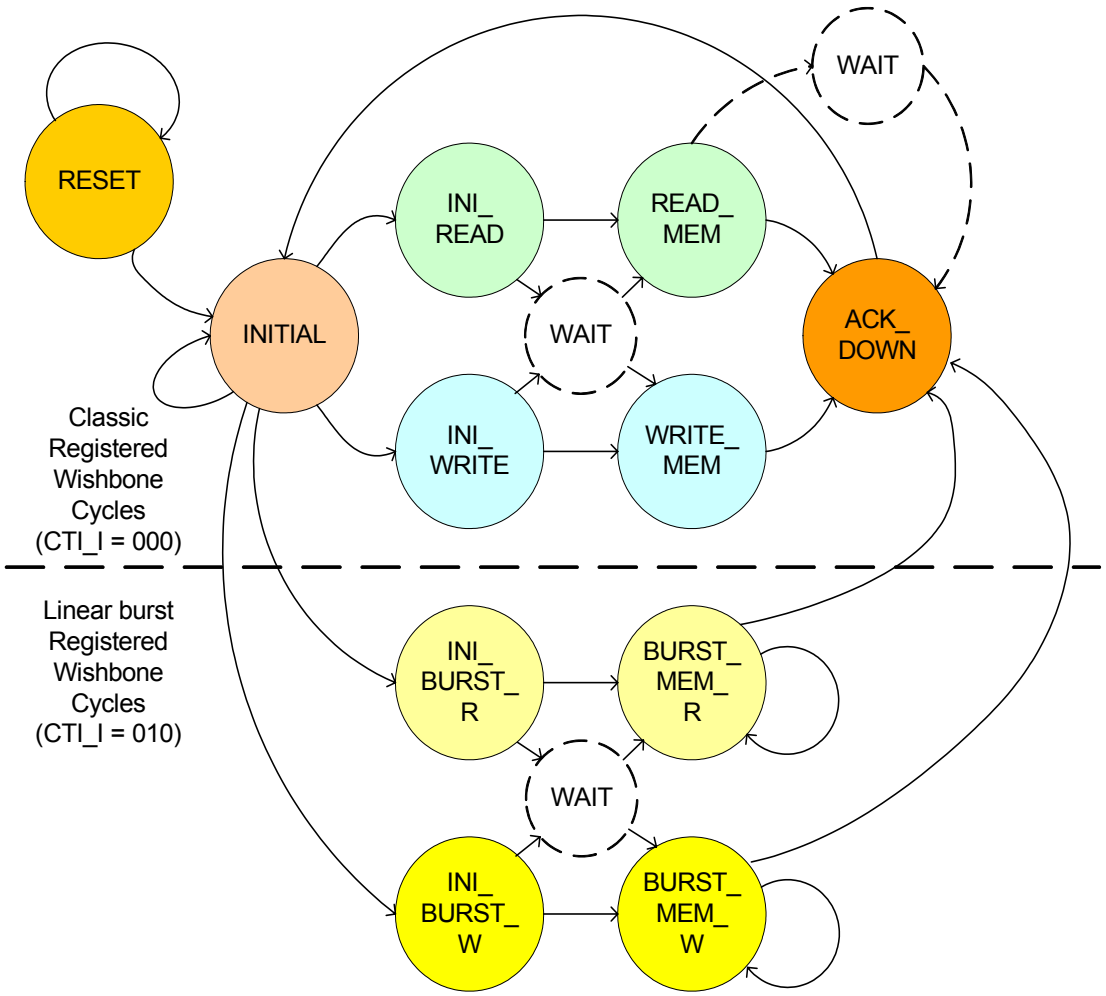


FIGURE 27. Simplified State Diagram for the Wishbone Memory Controller, the dashed states are needed for the registered design.

Our state machine has an asynchronous reset state, after a reset we always go to the initial state where we wait for the Master core to initiate a cycle and depending on the cycle type (CTI_I and WE_I) we either start a single or burst read or write cycle. After finishing the current cycle the handshake signal ACK_O is set to 0 and we return to the initial state, to wait for the next cycle. Simulations, synthesis results and descriptions of the experiments done on board are provided in Chapter 8.

Chapter 8: Experiments and Results

As we saw in the previous chapters, the new HPDF/CSDF modeling technique introduced in this work provides enough information to infer efficient schedules and memory organizations. In Section 8.1 we generalized the swapping banks method introduced in Section 7.2 and then in Section 8.2 we compare the performance and energy consumption achieved by using this method with other memory organization schemes. In Section 8.3 we describe the experiments performed with the memory controller core designed, as well as the simulations and synthesis results obtained using Xilinx ISE Project Navigator and Mentor Graphics Modelsim softwares. Finally Section 8.4 calculates the maximum frame rate provided by the board's video decoder chip.

8.1 Swapping Banks Method Generalization

In general, we swap banks so that an actor reads from a bank b data from frame i, while the preceding actor through edge e is writing in another bank data from frame $i+1$, where b and b' are "swapping banks" associated with edge e. In this way, an efficient memory organization consists of using half the banks available for each edge. An example with four memory banks is illustrated in Figure 28. .

8.2 Comparison of Memory Organization Schemes

In this section we compare different memory organization schemes for the gesture recognition algorithm [37] described in Section 3.4.1 and the hardware platform is the Xilinx Multimedia Board.

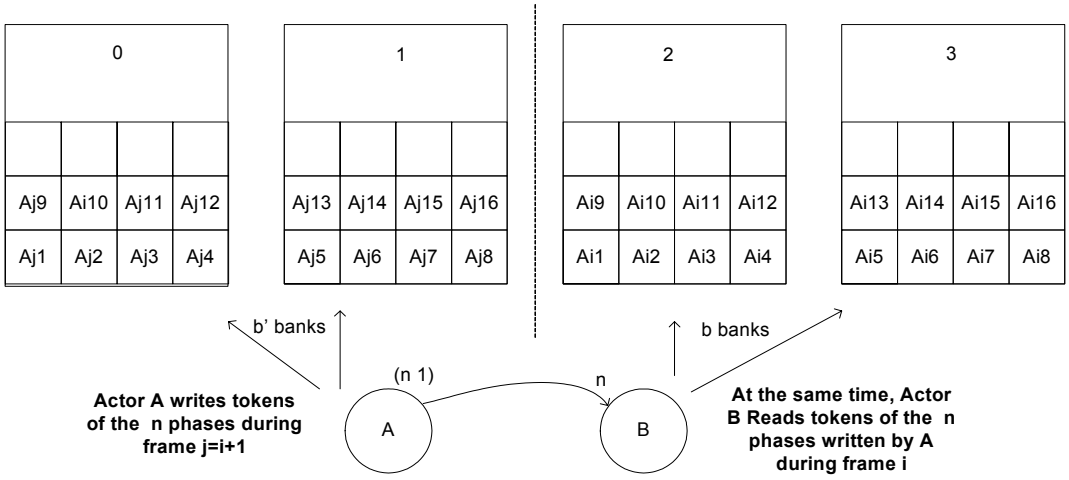


FIGURE 28. Example of swapping banks

In Table 3, we compare three cases with frames sized at 384x240 pixels. The first case considers an effective memory organization that can be achieved from the information given by the HPDF/SDF model, considering a single frame. In this case, Region needs to read the frame from memory and then write its output to memory from where Contour will read it next. An appropriate configuration is achieved by storing the three components of the frame in the first 22.5K addresses of three memory banks and storing Region's output in 22.5K addresses of the other two memory banks. The second case is an efficient memory organization for a single frame that exploits the pixel-level parallelism shown with the precise modeling of the input stream (Figure 18). Region does not need to have a whole frame stored in memory anymore, and a more effective memory configuration is accomplished when we write Region's output in the first 9216 addresses of the 5 banks.

Case 3 considers a stream of frames, extracting the frame level parallelism represented in the HPDF/CSDF model, where we swap between two pairs of banks to write Region's output in the first 23040 addresses in each case. In this case, at the same time that Contour is reading Region's output for frame i, Region is writing its output for frame

$(i+1)$.(Figure 28). Consequently, the buffer usage gets reduced from 184 Kb between Region and Contour to 3 bytes while the other edges still have the same worst case buffer size as the previous representation [25]. This worst case arises when there is only one body part filling the whole image. However, in a typical scenario, the buffer sizes will be reduced significantly compared to the HPDF/SDF model of [25]

TABLE 3. Results of optimized memory organization in different scenarios.

Scenario	Total # of access cycles	# of non-overlapping cycles	Energy (mJ)
Single frame w/R	161280	69120	4.56
Single frame	92160	18432	2.61
Stream	92160	23040 (2 frames pipelined)	2.66

8.3 Memory Controller

Both memory controllers implemented were simulated using ModelSim and tested on the Xilinx Multimedia Board. Figure 29 shows a simulation with two reading cycles followed by two writing cycles for the memory controller core that does not have output registers. As it was expected, since the Master requests or sends data until it has the answer we can have up to three clock cycles. Figure 30 shows a simulation with one reading cycle followed by a writing cycle for the memory controller that has output registers. In this case, the number of clock cycles since the Master requests or sends data until it has the answer can be up to five for a reading request and four for a writing task.

The Xilinx ISE Project Navigator synthesis results in terms of resources consumed are similar for both implementations. For the registered implementation the number of slices used is 101 out of 10752 (0 %), the number of Slice Flip Flops is 175 out of 21504 (0 %) and the number of 4 input LUTs used is 45 out of 21504 (0 %). Since those percent-

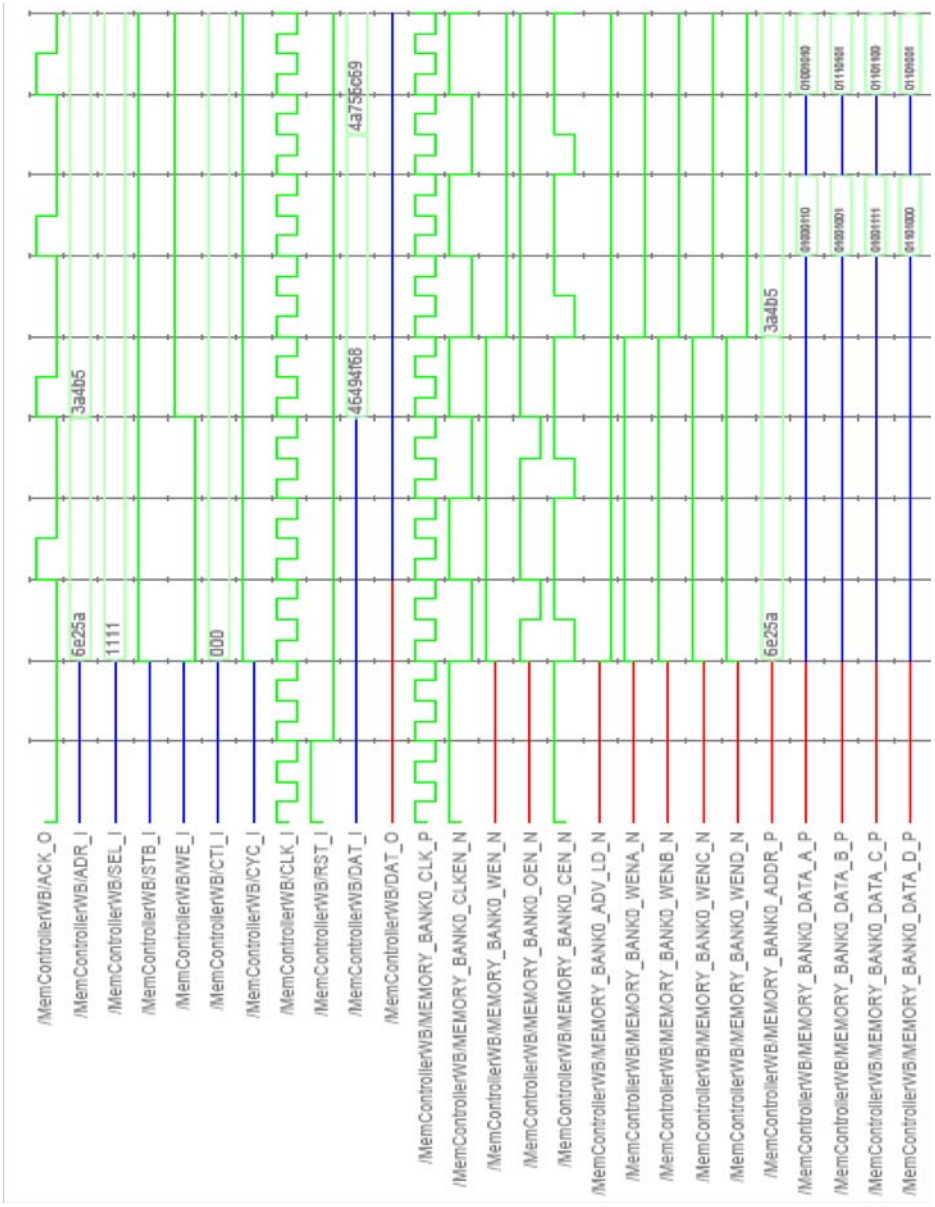


FIGURE 29. Simulation of two reading cycles followed by two writing cycles for the memory controller without output registers; each memory access cycle takes up to three clock cycles

ages are very low, the XPower analysis utility reports a low dynamic power consumption (79 mW for a 133 MHz clock frequency) compared to the 367 mW quiescient FPGA consumption. The maximum combinational delay found by the software in both cases is around 6.2 ns, that is to say a maximum operating frequency around 160 MHz, which is larger than the maximum operation frequency of the board's memory chips.

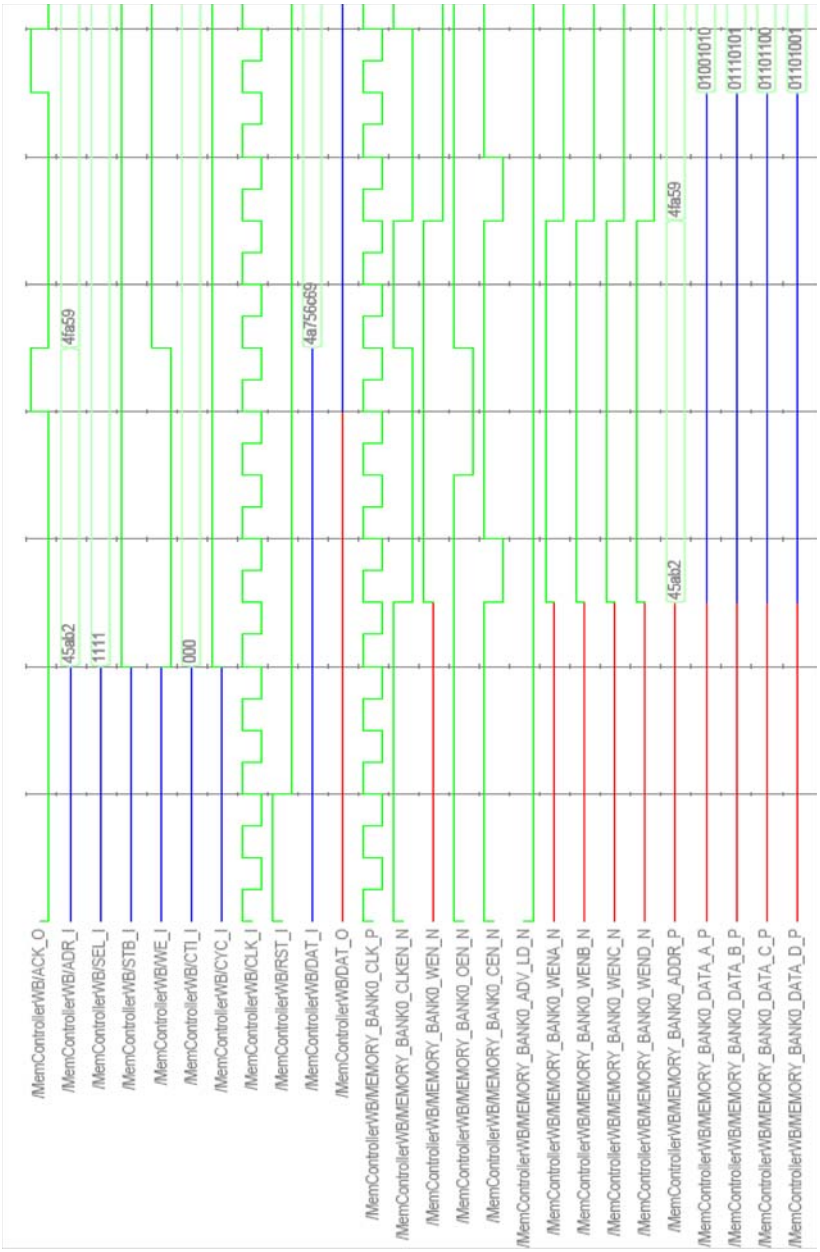


FIGURE 30. Simulation of a reading cycle followed by a writing cycle of the memory controller with output registers. It takes up to five clock cycles to complete a reading cycle and up to four for a writing cycle

8.4 Video Speed

As described in Section 4.3 the board's video decoder provides a new Y value for the pixels at a rate of 13.5 MHz. Equation 19 shows the relationship between the pixel and the frame rate.

$$F_{rate} = \frac{P_{rate}}{n^o pixels} \quad \text{(EQ 19)}$$

That is to say that the frame rate equals the pixel rate divided by the total number of pixels in a frame. In our case, the video input maximum rate is then approximately 146 frames per second. We notice that the pixel rate is almost ten times slower than the memory chips maximum operating frequency.

Chapter 9: Conclusions and Future Directions

In this thesis we studied different Dataflow models, we reviewed some image processing techniques and we described FPGA-based boards. Then we introduced a new modeling technique that combines HPDF meta-model and base actor CSDF modeling. We modeled two different image processing applications and showed that this new technique exposes more levels of parallelism than previous models, which can be exploited causing buffer sizes to reduce. Moreover, the model remains homogeneous, enabling us to find simple schedules. Furthermore, the CSDF firing granularity allows actors (static as well as dynamic) to fire without having to stall for all the tokens of a complete invocation to be produced, when this is not necessary.

We also provided a method to find valid schedules whenever they exist and a method to infer efficient memory organization from an application HPDF/CSDF modeling information.

Finally, we designed a reusable wishbone compliant memory controller module that can be used to access the Xilinx Multimedia Board's memory chips using single accesses or burst mode.

In the following sections we suggest some future work directions that could be interesting to explore.

9.1 Integrating DIF

The dataflow interchange format (DIF) [12], [11], is a textual language that captures the semantics of graphical design tools for DSP systems design. It is designed to be exported and imported automatically by these tools. DIF supports different kinds of data-

flow graphs such as SDF, CSDF and PSDF. It would be useful to have HPDF/CSDF also supported by DIF to take advantage of some of the utilities that are available in design tools.

9.2 Automatic Hardware Generation

Since our modeling technique exposes parallelism at different granularities, for instance in the application we modeled we had pixel level parallelism as well as frame level parallelism, it seems to be a very convenient way to specify an algorithm in order to produce automatic hardware generation. Taking into account previous work in synthesizable code generation from dataflow specifications [36], [25], [27], and the methods we proposed in this work to find valid schedules and efficient memory organization directly from the information given by the model, we consider that another interesting line of research would be to design a tool that automatically generates synthesizable hardware.

Appendix A - Matlab Scripts

In the following boxes we have the Matlab Scripts used to study the motion detection algorithm [15] described in Section 3.4.2.

```
y2 = gris(ori2);
y3 = gris(ori3);
y6 = gris(ori6);
y7 = gris(ori7);

res = difer(y2,y3,60);
erodo = erosion(res);
erodo = erosion(erodo);
figure
imshow(uint8(255-erodo));

R = uint8(double(ori1(:,:,1))+erodo);
I(:,:,1) = R;
I(:,:,2) = ori2(:,:,2);
I(:,:,3) = ori2(:,:,3);
figure
image(I);

J(:,:,2) = uint8(y2);
J(:,:,3) = uint8(y2);
J(:,:,1) = uint8(y2+erodo);
figure
image(J);

R = double(ori2(:,:,1));
G = double(ori2(:,:,2));
B = double(ori2(:,:,3));

Yori2 =  0.299*R + 0.587*G + 0.114*B;
Uori2 = -0.147*R - 0.289*G + 0.436*B;
Vori2 =  0.615*R - 0.515*G - 0.100*B;

R = Yori2 + 1.140*(Vori2);
G = Yori2 - 0.395*(Uori2) - 0.581*(Vori2);
B = Yori2 + 2.032*(Uori2);

ver(:,:,1) = uint8(R);
ver(:,:,2) = uint8(G);
ver(:,:,3) = uint8(B);
image(ver);
```

```
function res = difer(a,b,thre);

res = abs(a-b);
l = size(res);
for i=1:l(1)
   for j=1:l(2)
      if (res(i,j) < thre)
         res(i,j) = 0;
      end
   end
end
```

```
function Y = gris(M);

R = double(M(:,:,1));
G = double(M(:,:,2));
B = double(M(:,:,3));

Y = 0.299*R + 0.587*G + 0.114*B;
```

```
function sal = erosion(M);

l = size(M);
N = uint8(M);
for i = 2:(l(1)-1)
   for j = 2:(l(2)-1)
      if (N(i,j) ~= 0)
         vecinos = [N(i-1,j)
                    N(i+1,j)
                    N(i,j-1)
                    N(i,j+1)];
         M(i,j) = min(vecinos);
      end
   end
end

sal = M;
```

Appendix B - Multimedia Board Brief Tutorial

This appendix is a simple tutorial to rapidly become familiar with the Xilinx Multimedia Board and its programming environment, basic verilog knowledge is assumed. More advanced tutorials for general FPGA design and programming can be found in Xilinx's webpage. Some manuals also available online such as [30], [31], [33] and [34] provide more detailed information.

B.1 Creating a Design

In order to specify a design we use Xilinx's ISE software. First we need to create a new project in the ISE Project Navigator, selecting New Project from the file menu. A wizard will help us create the new project, the important thing is to select the board device which is xc2v2000 from virtex II family, ff896 package and -4 speed.

After creating the project we select the New source option in the Project menu. We can choose different types of files; in this example we are going to work with the Verilog Module type. Figure 31 shows the verilog code of our first design which consists only of connecting the user input switches to the user output leds.

```
module simple1(SW1,SW2,Led1,Led2);
   input SW1;
   input SW2;
   output Led1;
   output Led2;

assign Led1 = SW1;
assign Led2 = SW2;

endmodule
```

FIGURE 31. First design: controlling leds with switches.

B.2 Programming the FPGA chip

The following step is to assign package pins in the user constraints section. In [31] we can find which pins are connected to the leds and switches, Table 4 shows which pins were assigned to our design signals (always check your board manual before assigning pins).

TABLE 4. Pin numbers that are connected to leds and switches in the Multimedia Board

Signal Name	Pin number
Led1	B27
Led2	B22
SW1	D10
SW2	F14

Once the pins are assigned, we need to select the Synthesize, Implement Design and Generate Programming File options in this order. Then we connect Xilinx's kit power supply to the multimedia board, the MultiLINX Flying Lead Connector to the board and to the Parallel Cable IV Pod, power up the Parallel Cable IV, go to the section Configure Device (iMPACT), choose Boundary-Scan Mode and then Automatically connect to cable and identify Boundary-Scan Chain. After these settings are done, right click over the Xilinx xc2v2000 device and choose Program.

B.3 Other designs

The first design we introduced was combinational, but we can also have very simple sequential designs. In this design we switch each led alternatively, however, since the board clock frequency is 27 MHz, we need to implement a frequency divider in order to appreciate the changes. We divide the board master clock using a 22 bits counter, achieving in this way a 6.75 Hz switching frequency. The clock divider code shown in Figure 32 is an independent module instantiated by a top module. After the design is ready, we fol-

low the same previous steps for downloading the top module's programming bit stream to the board, the only difference is that now we also have to assign pin AH15 to our clock signal.

```
module clock_counter(MASTER_CLOCK, reset, slow_clock);
input MASTER_CLOCK;
input reset;
output slow_clock;

reg[21:0] clock;
assign slow_clock = clock[21:21];

always @(posedge MASTER_CLOCK or posedge reset) begin
   if (reset)
      clock <= 0;
   else
      clock <= clock + 1;
   end
end
endmodule
```

FIGURE 32. Clock division module

In order to use communication ports such as the RS-232 port, IP cores can be downloaded from [18]. A PC can communicate with the board serially through applications such as HyperTerminal, Terminal or Matlab, using an RS-232 straight cable where only pins 2 (Rx), 3 (Tx) and 5 (Ground) are needed. Furthermore, the board's serial port selection switches should be set properly, for instance in the XO position.

Appendix C - Wishbone Interface Specification Overview

This appendix compiles some Wishbone signals and cycles specified in [17] where the complete specification is provided.

C.1 Signals

All the Wishbone interface signals are active high and the '_I' or '_O' tags are attached to the signal names to indicate if they are inputs or outputs of a particular core. The specification defines signals for the syscon module, a master core and a slave core.

The syscon module generates signals CLK_O to coordinates all activities for the internal logic within the Wishbone interconnect and RST_O to force all Wishbone interfaces to restart.

Some of the signals that are common to master and slave interfaces are: CLK_I, the clock input coordinates all activities, all output signals are registered at the rising edge of [CLK_I] and all input signals are stable before its rising edge; DAT_I() and DAT_O(): (data input/output array) are used to pass binary data, their boundaries are determined by the port size, with a maximum port size of 64-bits (e.g. [DAT_I(63..0)]); RST_I is the reset input and forces the Wishbone interface to restart. Furthermore, all internal self-starting state machines will be forced into an initial state.

Some Master signals are: ACK_I: (acknowledge input), when asserted indicates the normal termination of a bus cycle; ADR_O(): (address output array) is used to pass a binary address; CYC_O (cycle output), when asserted indicates that a valid bus cycle is in progress, it is asserted for the duration of all bus cycles; SEL_O() (select output array) indicates where valid data is expected on the [DAT_I()] signal array during READ cycles,

and where it is placed on the [DAT_O()] signal array during WRITE cycles; STB_O: (strobe output) indicates a valid data transfer cycle; WE_O: (write enable output) indicates whether the current local bus cycle is a READ or WRITE cycle, it is negated during READ cycles, and is asserted during WRITE cycles.

The corresponding slave signals are: ACK_O that indicates the termination of a normal bus cycle, ADR_I(), CYC_I, SEL_I() and STB_I that indicates that the SLAVE is selected.

Additional signals for registered cycles: CTI_IO(): (Cycle Type Idenfier) Address Tag provides additional information about the current cycle. The master sends this information to the slave which can use this information to prepare its response for the next cycle. The other signal is BTE_IO() (Burst Type Extension) sent by the master to the slave, which provides additional information about the current burst. Table 5 describes the different CTI and BTE values.

TABLE 5. Cycle Type Identifiers (columns 1 and 2) and Burst Type Extensions (columns 3 and 4)

CTI_O(2:0)	Description	BTE_IO(1:0)	Description
000	Classic cycle	00	Linear burst
001	Constant address burst cycle	01	4-beat wrap burst
010	Incrementing burst cycle	10	8-beat wrap burst
011	Reserved	11	16-beat wrap burst
100	Reserved		
101	Reserved		
110	Reserved		
111	End-of-Burst		

C.2 Wishbone Registered Cycles

Figure 33 shows an example of a Wishbone classic registered reading cycle.

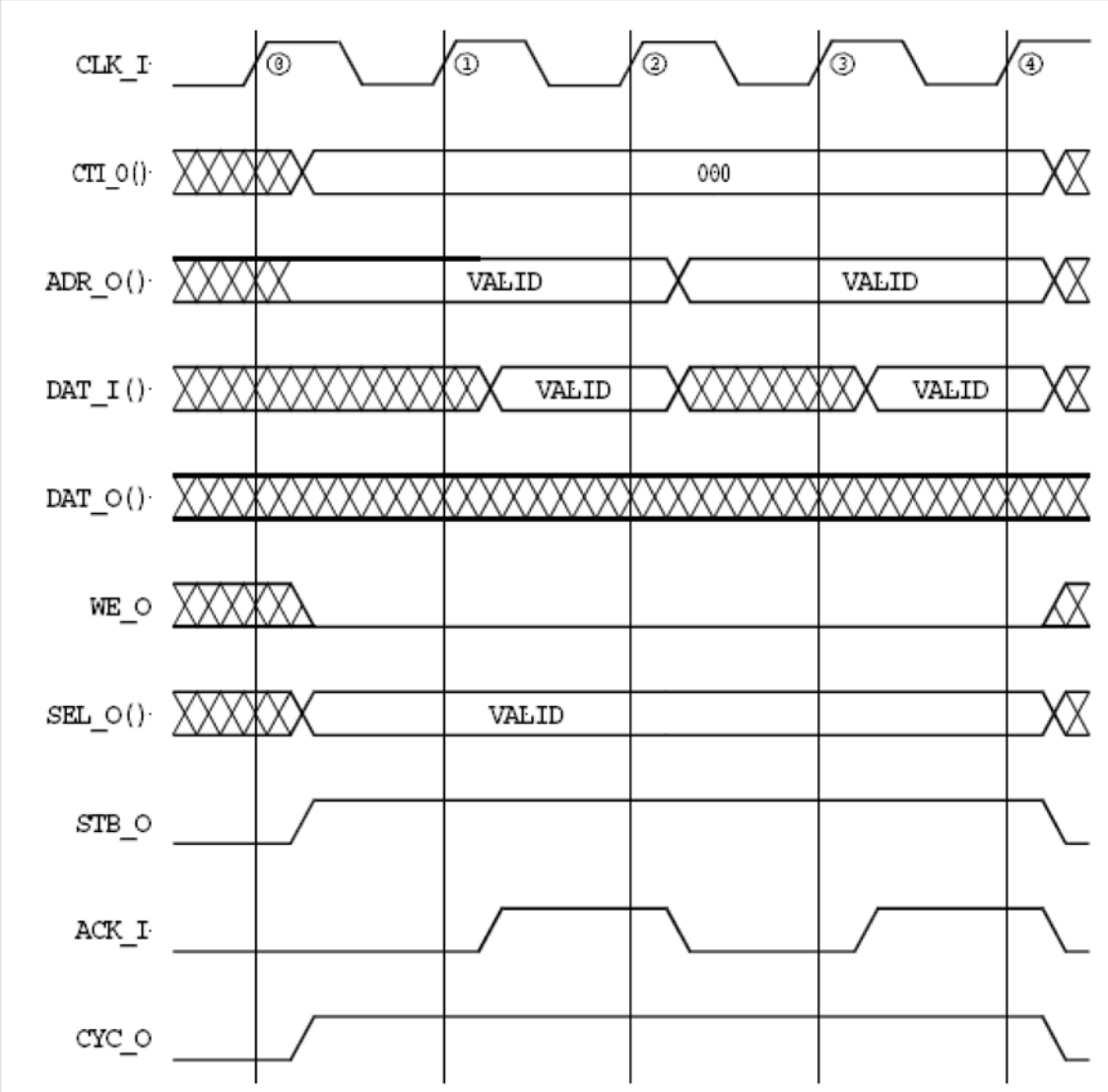


FIGURE 33. Wishbone classic registered reading cycle

Appendix D - Wishbone Static Memory Controller Datasheet

TABLE 6. Revision 1.0 - Wishbone Memory Controller Core Datasheet

Type of interface:	slave
Port size:	32-bit
Granularity:	8-bit
Maximum operand size:	32-bit
Data Organization:	little endian
Interface signals:	
	ACK_O
	ADR_I (18:0)
	SEL_I(3:0)
	STB_I
	WE_I
	CTI_I
	BTE_I -- only linear burst (00) supported. Classic Wishbone is provided by default
	CYC_I
	CLK_I
	RST_I
	DAT_I(31:0)
	DAT_0(31:0)

References and Bibliography

[1] ARM Limited. *AMBA Specification* (Rev 2.0). 1999.

[2] Bhattacharya, B.; Bhattacharyya, S.S., "Parameterized dataflow modeling for DSP **sys**tems", *IEEE Transactions on Signal Processing*, Vol.49, Iss.10, Oct 2001. pp. 2408-2421

[3] G. Bilsen, M. Engels, R. Lauwereins, J. A. Peperstraete, "Cyclo-Static Data Flow", in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, Vol.5, Iss., 9-12, pp. 3255-3258, May 1995

[4] T. Bräunl, S. Feyrer, W. Rapf and M. Reinhardt, *Parallel Image Processing*, Springer-Verlag Berlin Heidelberg, 2001.

[5] Chai, D.; Phung, S.L.; Bouzerdoum, A, "A Bayesian skin/non-skin color classifier using non-parametric density estimation", in *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, Vol.2, Iss., 25-28 May 2003. Pages: II-464- II-467 vol.2

[6] R.C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd Ed Prentice Hall, 2002.

[7] Guodong Guo, Li, S.Z., Kapluk Chan, "Face recognition by support vector machines, Automatic Face and Gesture Recognition", in *Proceedings. Fourth IEEE International Conference on*, Vol., Iss., 2000 Pages:196-201.

[8] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice Hall, 2nd edition, 1998.

[9] Hearst, M.A.; Dumais, S.T.; Osman, E.; Platt, J.; Scholkopf, B. "Support vector machines. Intelligent Systems and Their Applications", *IEEE [see also IEEE Intelligent Systems]*, Vol.13, Iss.4, Jul/Aug 1998. Pages:18-28

[10] Jack Horgan, "FPGA Direction" *EDACafe Weekly.* August 9, 2004.

[11] C. Hsu and S. S. Bhattacharyya, "Dataflow interchange format version 0.2", Technical Report UMIACS-TR-2004-66, Institute for Advanced Computer Studies, University of Maryland at College Park, November 2004. Also Computer Science Technical Report CS-TR-4624.

[12] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, "DIF: An interchange format for dataflow-based design tools", in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pp. 423-432, Samos, Greece, July 2004.

[13] Z. Hussain. *Digital Image Processing Practical Applications of Parallel Processing Techniques*, Ellis Horwood Limited, 1991.

[14]Michael Keating, Pierre Bricaud, *Reuse Methodology Manual for System-on-a-chip designs*, Third Edition. Kluwer Academic Publishers, 2002.

[15] Andrew Kirillov, “Motion Detection Algorithms”, The code project, Sep 2005

[16] E. A. Lee, D.G. Messeschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing”, *IEEE Transactions on Computers*, vol. C-36 no 1, January 1987, pp. 24-35.

[17]OpenCores Organization, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, September 2002.

[18] OpenCores Organization webpage: http://www.opencores.org/

[19]I. Page, “Constructing Hardware-Software Systems from a Single Description”, Journal of VLSI Signal Processing, v.12, 1996, Pages 87-107.

[20] T. M. Parks, J. L. Pino and E. A. Lee, “A comparison of Synchronous and Cyclo-Static Dataflow”, in *Proceedings of Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, Vol.1, Iss., 30 Oct-1 Nov 1995. pp. 204-210 vol.1

[21] L.R. Rabiner, “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition”, in *Proceedings of the IEEE*, Vol.77, No. 2, Feb 1989, pp. 257-286.

[22] E. Rijpkema, B. Kienhuis and E. F. Deprettere, “Compilation from Matlab to Process Networks”, Presented at the *Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)*, October 1-3 1999, Washington.

[23] J. C. Russ, *The Image Processing Handbook*, CRC Press, Inc., 1992.

[24]Samsung Electronics CO., LTD. *512Kx36 & 1Mx18-Bit Pipelined NtRAM™ datasheet*. 2.0 Feb. 2001.

[25]M. Sen and S. S. Bhattacharyya, “Systematic exploitation of data parallelism in hardware synthesis of DSP applications”, in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. V-229-V-232, Montreal, Canada, May 2004.

[26]M. Sen, S. S. Bhattacharyya, T. Lv, and W. Wolf, “Modeling image processing systems with homogeneous parameterized dataflow graphs”, in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. V-133-V-136, Philadelphia, Pennsylvania, March 2005.

[27]M. Sen, I. Corretjer, F. Haim, S. Saha, J. Schlessman, S. Bhattacharyya, and W. Wolf. “Computer Vision on FPGAs: Design Methodology and its Application to Gesture

Recognition", presented at *The First IEEE Workshop on Embedded Computer Vision*, 2005, San Diego.

[28]S. Sharp, "FPGAs Can Be an Effective Alternative to Mask Gate Arrays", *Xcell Journal Online*, Vol 30. Oct 1998. pp. 6-7.

[29]S. Sriram, S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, Inc., 2000

[30] Xilinx, Inc., *Virtex-II Platform FPGAs: Complete Data Sheet,* DS031, (v3.3) June 24, 2004.

[31] Xilinx, Inc., *MicroBlaze and Multimedia Development Board User Guide*, UG020 (v1.0) August 2002.

[32] Xilinx, Inc., An alternative Capacity Metric for LUT-Based FPGAs., Application Brief. XBRF 011 Feb 1997.

[33] Xilinx, Inc. JTAG Programmer Guide, Version 3.1i.

[34] Xilinx, Inc. Parallel Cable IV, MultiPRO and MultiLINX Quick Start Guide

[35]M.Weinhardt and W.Luk, Memory access optimisation for reconfigurable systems IEE Proc.-Comput. Digit. Tech., Vol. 148, No. 3, May 2001.

[36]M. C. Williamson, "Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications", Ph.D. dissertation, University of California, Berkeley, 1998.

[37]W. Wolf, B. Ozer, T. Lv, "Smart cameras as embedded systems", *IEEE Computer Magazine* Vol 35, Iss 9, Sept 2002, pp. 48-53.