Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Una nueva metodología para co-diseño de sistemas embebidos centrados en procesador usando FPGAs

Tesis presentada para obtener el título de Doctor
de la Universidad de Buenos Aires
en el área Ciencias de la Computación

Sol Pedre

Directora: Dra. Patricia Borensztejn

Director Asistente: Dr. Elías Todorovich

Lugar de Trabajo: Departamento de Computación, Facultad de Ciencias Exactas y Narurales, Universidad de Buenos Aires.

Buenos Aires, 2013

Una nueva metodología para co-diseño de sistemas embebidos centrados en procesador usando FPGAs

Hoy en día, los sistemas embebidos son partes vitales de equipos de comunicaciones, sistemas de transporte, plantas de energía, electrónica de consumo, robótica entre muchos otros. Su amplio campo de aplicación y las crecientes complejidades de sus diseños torna esencial la propuesta de nuevas metodologías, lenguajes y herramientas. El objetivo de esta tesis doctoral es contribuir al campo del co-diseño hardware/software de sistemas embebidos.

Primero, presentamos el co-diseño de un sistema embebido de control aplicando el flujo de diseño tradicional, que combina procesadores y circuitos integrados (ICs): el desarrollo de un nuevo mini-robot llamado ExaBot. Luego, introducimos un flujo de diseño tradicional para Field Programmable Gate Arrays (FPGA), y lo aplicamos a un problema de sensado remoto: procesar video infrarrojo en tiempo real en un UAV (Unmanned Aerial Vehicle). Finalmente, de la observación de las dificultades en experiencias anteriores, y analizando las tendencias y tecnologías actuales, proponemos una nueva metodología de co-diseño para sistemas embebidos centrados en procesador usando FPGAs. Este es un creciente y novedoso campo de los sistemas embebidos: durante 2011, tanto Xilinx como Altera (los dos fabricantes mas grandes de FPGAs) lanzaron nuevas familias de chips que combinan potentes procesadores ARM con lógica programable de bajo consumo.

El objetivo de la nueva metodología de co-diseño es lograr soluciones embebidas de tiempo real, utilizando aceleración por hardware, pero con un tiempo de desarrollo similar al de proyectos de software. Para ello, combinamos metodologías y herramientas bien establecidas del mundo del software, como Diseño Orientado a Objetos, UML, y programación multi-hilos, con nuevas tecnologías del mundo del hardware, como herramientas semi-automáticas para síntesis de alto nivel. La metodología propuesta fue aplicada a un algoritmo de localización de múltiples robots en un sistema de visión global. La solución embebida final procesa 32 imágenes de 1600×1200 píxeles por segundo, logrando una aceleración de $16\times$ con respecto a la solución de software más optimizada, con un 43% de incremento en área pero un 92% de ahorro de energía.

A new co-design methodology for processor-centric embedded systems in FPGA-based chips

Embedded systems are nowadays vital parts of communication equipment, transportation systems, power plants, consumer electronics, robotics among many others. Their vast field of application and the growing complexities of their designs turn the proposal of new methodologies, languages and tools essential. The goal of this thesis is to make such contributions in the field of hardware/software co-design of embedded systems.

First, we present the co-design of a control embedded system applying the traditional flow in which processors and off-the-shelf Integrated Circuits (ICs) are combined: the development of a mini-robot called ExaBot. Secondly, we introduce a traditional Field Programmable Gate Array design flow, and apply it to a remote sensing application that processes real-time video from an infrared camera on an UAV (Unmanned Aerial Vehicle). Finally, from the observation of difficulties in previous experiences and analyzing current technologies and trends, we propose a new co-design methodology for processor-centric embedded systems in FPGA-based chips. This is a growing and novel field of embedded systems: during 2011, both Xilinx and Altera (the two leading FPGA vendors) launched new chip families that combine powerful ARM processor cores with low-power programmable logic.

The goal of the proposed co-design methodology is to achieve real-time embedded solutions, using hardware acceleration, but with development time similar to that of software projects. For this, well-established methodologies and tools from the software domain, such as Object Oriented Design, Unified Modeling Language or multithreaded programming, are combined with new techniques from the hardware world, like semi-automatic high level synthesis tools. The proposed methodology was successfully applied to a multiple robot localization algorithm in a global vision system. The final embedded solution processes 1600×1200 pixel images at 32 frames per second, achieving a $16\times$ acceleration with respect to the most optimized software solution, with a 43% increase in area but a 92% energy saving.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

An embedded system is a system designed to perform one or few dedicated functions and that is embedded into a larger device [8, 9]. Common characteristics include efficiency in terms of energy, cost and weight; reliability since they are often components of critical systems; and the need to meet real-time constraints. Nowadays, embedded systems are vital parts of communication equipment, transportation systems, power plants, consumer electronics, robotics among many others [10].

Most embedded systems include off-the-shelve ICs (Integrated Circuits) for specific functions together with one or many processor elements, i.e., microcontrollers, microprocessors or DSPs (Digital Signal Processors). The design of systems that have software running in processors interacting with hardware modules is called hardware/software co-design. Already in 2004, over 90 % of embedded systems included some kind of processor [11], and from the 9 billion processors manufactured in 2005, 98% was used in embedded systems [12]. According to the 2012 Embedded Market Survey conducted by EEtimes and Embedded magazines [13], 97% of the 1,700 consulted embedded engineers were using at least one processor in their designs. This makes research in co-design methodologies and tools a key field in the embedded system domain.

However, some systems require massive data processing with real-time constraints that cannot be met with this standard approach. Examples include digital signal processing methods such as image, video or audio processing, and their applications to robotics, remote sensing, consumer electronics among many other fields. In these cases, solutions include the use of Field Programmable Gate Arrays (FPGAs) or the design and implementation of ASICs (Application Specific Integrated Circuits). These approaches take advantage of the inherent parallelism of many data processing algorithms and allow to create massive parallel solutions. They also allow tailored hardware acceleration, e.g., with particular memory access patterns or bit tailored multipliers/adders. ASICs provide the best solution in terms of performance, unit cost and power consumption. FPGAs are designed to be configured by a designer after manufacturing— hence "field-programmable". The ability to update the functionality after shipping, partial re-configuration of a portion of the design, and the lower non-recurring engineering costs and shorter time-to-market compared to an ASIC design, offer advantages for many applications. According to the 2012 Embedded Market Survey, 35% of the surveyed engineers are currently using FPGAs in their designs [13].

For many applications designing the entire system in FPGAs or hardware is not the most practical solution. Even the most data intensive processing methods frequently contain sequential sections that are easier implemented in processors. These hardware/software co-designed solutions try to combine the best of both software and hardware worlds, making use of the ease of programming a processor while designing tailored hardware accelerator modules for the most time-consuming sections of the application. This not only accelerates the resulting system compared to the processor solution, but also allows savings in energy. The inclusion of processor cores embedded in programmable logic has made FPGAs an excellent platform for these approaches. During 2011, the two major FPGA vendors (Xilinx and Altera) announced new chip families that combine powerful ARM processor cores with low-power programmable logic [14, 15]. While FPGA vendors have previously produced devices with on-board processors, the new families are unique in that the ARM processor system, rather than the programmable logic, is the center of the chip [16]. This strengthens the growing trend towards co-designed processor-centric solutions in FPGA-based chips [17]. According to the 2012 Embedded Market Survey, 37 % of the engineers that do not use FPGAs in their current designs confirmed that this trend will change their minds [13].

The novelty of this approach together with its potential in the embedded system world makes academic research in hardware/software co-design in FPGA-based chips an important field. The main problem to tackle is time-consuming development. The rising complexity of these applications make it difficult for designers to model the functional intent of the system in languages that are used for implementation such as C or HDLs (Hardware Description Languages). Moreover, the difficulty of programming FPGAs in HDL is pointed out by engineers as an important reason for not using FPGAs [13]. This poses a strong need for methodologies, languages and tools that reduce development time and complexity by raising the abstraction level of design and implementation [18] [19]. Advances have been made in high-level modeling using specific Unified Modeling Language (UML) profiles to simplify design. Also, much work is being done in high-level synthesis tools, which translate constructs in C/C++ to HDL to simplify hardware implementation.

However, there is still much research needed in co-design methodologies, languages and tools so that the recent combination of powerful processors with programmable logic can raise to its full potential.

1.2 Research goal

The goal of this thesis is to make a contribution in the field of hardware/software co-design of embedded systems. We expect that the developed work will help reduce design and implementation effort in an important field of embedded systems design, at a time when the growing complexities of these designs make the need for new methodologies, languages and tools vital.

The particular research goals in this thesis are:

1. the study of traditional co-design flows using processors and off-the-shelf ICs, and their application to the co-design of an embedded system with real-time, power consumption and size requirements.
2. the study of traditional design flows using FPGAs and their application to the design of an embedded system that require massive data processing with real-time constraints
3. the proposal of a new co-design methodology for a significant class of embedded systems: processor-centric embedded systems with hardware acceleration in FPGA-based chips. The new methodology will be focused in reducing design and implementation effort, integrating methodologies, languages and tools from both the software and hardware domain.

1.3 Outline of the thesis

In this thesis we present designs in each of the mentioned fields of embedded systems and we propose a new co-design methodology for processor-centric embedded systems with hardware acceleration in FPGA-based chips.

Chapter 2 is devoted to the co-design of a control embedded system applying the traditional flow in which processors and off-the-shelf ICs are combined: the development of a mini-robot called ExaBot [20]. This system has stringent real-time, power consumption and size requirements, providing a case study for traditional co-design flows. The main goal for pursuing this task was to obtain a low-cost robot that could be used not only for research, but also for outreach activities and education. In this sense, neither the commercially available research robots nor the commercially available educational robots were considered a suitable solution. Six ExaBot robots are currently in use in the Laboratorio de Robótica y Sistemas Embebidos of the FCEN-UBA. They have been used for educational robotics activities for high school students, research experiments in mobile robotics, and education in graduate and undergraduate university courses.

In chapter 3 we introduce a traditional FPGA design flow and apply it to a remote sensing application [21]. The application processes real-time video from an infrared camera on an UAV (Unmanned Aerial Vehicle) in order to find the location and spatial configuration of the hot spots present in each frame. The proposed method successfully segments the image with a total processing delay equal to the acquisition time of one pixel (that is, at video rate). The processing delay is independent of the image size. This real-time massive data processing was possible because the algorithm was designed for parallel FPGA implementation, and it was fully implemented in programmable logic and hardware ICs.

In chapter 4 we propose a new co-design methodology for processor-centric embedded systems with hardware acceleration in FPGA-based chips [22, 23, 24]. The aim of the methodology is to achieve real-time embedded solutions using hardware acceleration, but with development times similar to software projects. To reduce the development time, well established methodologies, techniques and languages from the software domain are applied, such as Object-Oriented Paradigm design, Unified Modeling Language and multithreaded programming. Moreover, to reduce hardware coding effort, semi-automatic C-to-HDL translation tools and methods are used and compared. As a case study, we use a robust algorithm for multiple robot localization in global vision systems. This algorithm integrates an e-learning robotic laboratory for distance education that allows students from all over the world to perform experiments with real robots in an enclosed arena. The co-designed implementation of this algorithm following the proposed methodology shows the usefulness of the methodology for embedded real-time massive data processing applications.

In chapter 5 conclusions and future work are outlined.

Chapter 2

Embedded systems using processors and ICs

ExaBot, a new mobile mini-robot

Most embedded systems include off-the-shelve ICs (Integrated Circuits) for specific functions together with one or many processor elements, i.e. micro-controllers, microprocessors or DSPs (Digital Signal Processors). The design of systems that have software running in processors interacting with hardware modules is called hardware/software co-design. Already in 2004, over 90 % of embedded systems included some kind of processor [11], and from the 9 billion processors manufactured in 2005, 98% were used in embedded systems [12]. According to the 2012 Embedded Market Survey conducted by EEtimes and Embedded magazines [13], 97% of the 1,700 consulted embedded engineers were using at least one processor in their designs. This makes research in co-design methodologies and tools a key field in the embedded system domain.

This chapter is devoted to the co-design of a control embedded system applying the traditional flow in which processors and off-the-shelve ICs are combined: the development of a mini-robot called ExaBot [20]. This system has stringent real-time, power consumption and size requirements, providing a challenging case study for traditional co-design flows. The main contributions in this chapter are:

- The adaptation of traditional co-design flows in which processors and off-the-shelve ICs are combined to the autonomous robotics field. The particular co-design flow is explained and the development of the robot following its different stages is shown.
- The design, construction and testing of the ExaBot robots. The main goal for pursuing this task was to obtain a low-cost robot that could be used not only for research, but also for outreach activities and educa-

tion. In this sense, neither the commercially available research robots nor the commercially available educational robots were considered a suitable solution. Six ExaBot robots are currently in use in the Laboratorio de Robótica y Sistemas Embebidos of the FCEN-UBA. They have been used for educational robotics activities for high school students, research experiments in mobile robotics, and education in graduate and undergraduate university courses.

This chapter is organized as follows: section 2.1 offers a short introduction to mobile robotics and the reasons for choosing this case study. Section 2.2 presents the co-design and testing methodology followed. The development of each stage to construct the ExaBot is presented in section 2.3. Section 2.4 comments several successful research, education and outreach activities carried out with the ExaBot. Finally, section 2.6 shares some conclusions.

2.1 Introduction

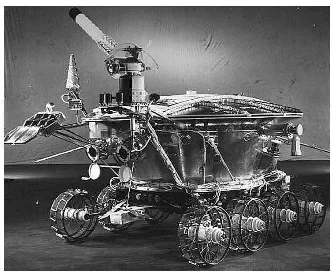
Robotics has achieved its greatest success to date in the world of industrial manufacturing[1]. Robot arms, or manipulators, comprised a US$ 5,7 billion market in 2010, totaling 1,035,000 units installed in factories around the world [25]. Yet, for all of their successes, these commercial robots suffer from a fundamental disadvantage: lack of mobility. A fixed manipulator has a limited range of motion, that depends on where it is bolted down. In contrast, a mobile robot would be able to travel throughout the manufacturing plant, flexibly applying its talents wherever it is most effective [26].

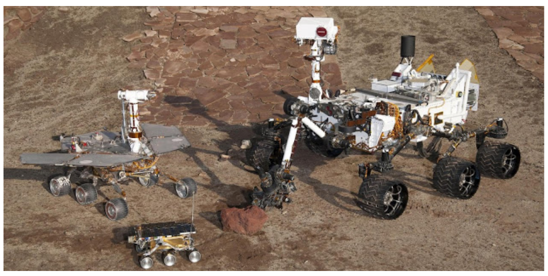
Mobile robots can be found in many fields. One of the most important applications is their use in hostile or inhospitable environments for human beings. For example, using mobile robots is nowadays the only viable option of exploring other planets. From the Soviet Lunokhod 1[2] that landed on the Moon in November 17, 1970 to the recent Curiosity that landed on Mars, a long line of mobile robots have been developed to handle the extreme conditions of outer space. In dangerous and inhospitable environments on Earth, such teleoperated systems have also gained popularity (see Fig. 2.1).

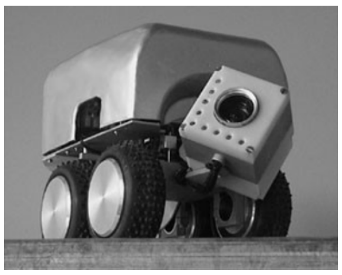
Other commercial robots operate not where humans cannot go but rather share space with humans in human environments. In 2010, about 2.2 million service robots for personal and domestic use were sold, for a value of US$ 538 million [27]. So far, service robots for personal and domestic use are mainly in the areas of domestic robots, which include vacuum cleaning, lawn-mowing,

[1]Industrial robots are defined by ISO 8373 as “An automatically controlled, reprogrammable, multipurpose manipulator programmable in three or more axes which may be either fixed in place or mobile for use in industrial automation applications.”

[2]The first mobile robot which landed on any celestial body


(a) Lunokhod


(b) NASA Rovers


(c) Air-duct Robot

(d) M510

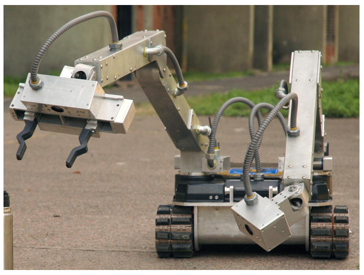

(e) KonaBot

Figure 2.1: Robots for hostile environments. Space exploring rovers: **a)** The soviet Lunokhod, **b)** NASA's Spirit, Sojouner and Curiosity. On Earth: **c)** Airduct inspection robot **d)** iRobot's 510 PackBot used at Fukushima Daiichi Nuclear Power Station in Japan **e)** KonaBot Robot for bomb control constructed at our lab

pool cleaning, toy robots and hobby systems. Educational robots are also a growing field, being the Lego Mindstrom the most popular kits for educational robotics activities [28]. For a good idea of the amount, price and availability of service robots see [29].

(a) Roomba

(b) LawnBott

(c) Genibo

Figure 2.2: Service Robots: **a)** Roomba 790 vacuum cleaning **b)** KA LawnBott LB1200 Spyder Lawnmower **c)** Genibo pet robot

Although mobile robots can be found in many fields as already discussed, achieving a fully autonomous robot is still a major task in robotics. The fundamental question is: how can a mobile robot move unsupervised through real-world environments to fulfill its tasks? Research into high-level questions of cognition, localization, and navigation can be performed using research robot platforms that are tuned to the laboratory environment. Various mobile robot platforms are available for programming, ranging in terms of size and terrain capability. The most popular research robots are those of Adept MobileRobots and K-Team SA (see Fig. 2.3). However, many times this commercial robots do not quite fit the necessary characteristics for particular tasks, and are difficult to adapt since they have proprietary software and hardware.

For example, Khepera [30] is a mini (around 5.5 cm) differential wheeled mobile robot that is developed and marketed by K-Team Corporation. The basic robot comes equipped with two drive motors and eight infrared sensors that can be used for sensing distance to obstacles or light intensities. It is very popular and widely used by over 500 universities for research and education. However, Khepera robot serves only for indoor small environments and although several extensions can be added, it is very limited when modifications to its sensing or programming capabilities are needed.

Another example of a well-known commercial mobile robot for research is the Pioneer 2-DX and its successor Pioneer 3-DX [31]. They are popular platforms for education, exhibitions, prototyping and research projects. These robots are quite bigger than the Khepera (more than 10 times) and have a computer integrated into a single Pentium-based EBX board running Linux. This processor unit is used for high-level communications and control functions. For locomotion, the Pioneer robots have two wheels and a sonar ring as range sensors. They can be used for both indoor and outdoor environments. Lots of accessories such as new sensors and actuators can be purchased from Adept MobileRobots manufacturer. Nevertheless, because of its size, Pioneer robots need a large workspace to move around and its weight makes it unsuitable to be transported around easily and tedious to be operated by a single human.

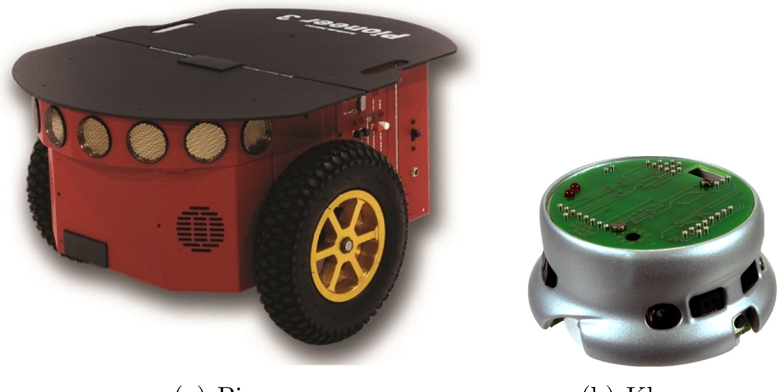

(a) Pioneer (b) Khepera

Figure 2.3: Research Robots: **a)** Adept's Pioneer 3-DX robot **b)**K-Team Khepera II mini robot

Besides the above disadvantages, the main drawback of these commercial mobile robots is their cost. For instance, a basic Pioneer robot costs approximately \$5,000 dollars, and a basic Khepera robot costs \$4,000. It is very difficult for Latin American research labs and universities to afford these costs and hence this severely limits the possibilities of buying or upgrading these robots, and even more for multi-robot systems. The maintenance of commercial robots can also be very hard for developing countries. If some component of the robot breaks it is not easy to purchase the replacement, it could take a lot of time due to shipping, and this in turn may delay planned experiments. On the other hand, although available educational robots are much cheaper, their capacities are far from enough for research activities.

Moreover, robotics is peculiar in that solutions to high-level challenges are most meaningful only in the context of a solid understanding of the low-level details of the system. Hence, and keeping in mind that there is no robotics without robots, for a Robotics Lab the design and development of the robot itself is a goal. Finally, in countries such as Argentina, the development of technology and technological proficiency is also a goal by itself.

These issues are the main motivations for developing our own low-cost mobile robot: the mobile robot ExaBot. Our main goal was *to obtain a low cost robot that could be used not only for research, but also for outreach activities and education.*

In order to build a mobile robot, the basic questions of locomotion and sensing must be first addressed. For this, it is necessary to design the hardware and software of an embedded system that can control the actuators and sensors of the robot. In this sense, a robot is a special case of a co-designed embedded system with real-time, space and power consumption restrictions.

2.2 Co-Design Flow

In this section, we present the design and testing methodology followed during the development of the ExaBot. This is a traditional co-design flow applied to the particular field of autonomous robots. Although this was the adhoc methodology for this design, it can be argued that such a methodology can be followed to obtain timely designs for other robots as well. In Fig. 2.4 the main stages of this flow can be seen together with the general stages of traditional co-design flows based in processors and ICs.

In the following subsections, some details of each stage is presented.

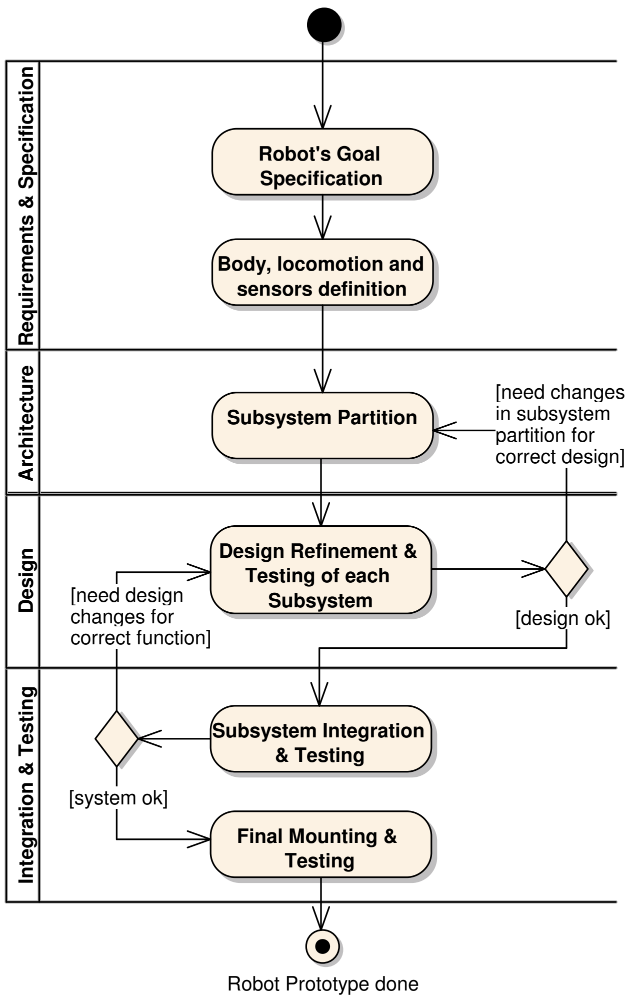

Requirements & Specification
Architecture
Design
Integration & Testing
Robot's Goal Specification
Body, locomotion and sensors definition
Subsystem Partition
[need changes in subsystem partition for correct design]
Design Refinement & Testing of each Subsystem
[need design changes for correct function]
[design ok]
Subsystem Integration & Testing
[system ok]
Final Mounting & Testing
Robot Prototype done

Figure 2.4: Co-design flow for the ExaBot. The horizontal swimlines show the equivalent stages from traditional general co-design flow.

2.2.1 Goals Specification. Body, locomotion and sensors definition

The first stage is to define what the robot should be able to do, that is specify which are the goals of the robot and which applications are targeted. The goals define the requirements for the robot.

These requirements have to be taken into account when defining the body, locomotion system and sensors of the robot. Some questions to answer are the following: How big does the robot need to be? How much weight should it be able to carry? What type of locomotion is needed? What sensing capabilities? What processing power? Other considerations include topics like how much reconfiguration is needed and what is the required battery autonomy.

To resolve this issues correctly, extensive research in perception, locomotion, processing units and mechanical issues is needed.

2.2.2 Partition in subsystems

In this stage, the whole system is divided in subsystems according to the elements to be controlled (e.g. actuator control, sensor control, etc). A preliminary evaluation of the required hardware and software is done. Some guiding questions are: Will cpu-like processing elements like microcontrollers be used? What other ICs will be possibly needed? Which communication buses can be used? For this step, extensive research on robotic solutions to similar goals is needed, together with reading data-sheets of possible sensors, actuators and ICs.

2.2.3 Design Refinement & Testing of each subsystem

In this stage, each subsystem is refined and tested. Of course, while doing this refinement, it may be necessary to modify the original subsystem partition. The following things need to be taken into account for each subsystem:

- *Hardware*: what ICs are needed? will some processor be used? what adhoc analog and digital circuits are needed? For each element to control (sensor or actuator) take into account their needed voltage supply and current consumption (maximum, minimum and typical). This defines if a driver will be needed and will affect the battery and power stage requirements.
- *Software*: what type of control is needed for each element? (e.g. open-loop or closed loop, linear or not, frequency of control, etc). Detailed information of these topics can be found in Chapter 1 of Chen's book "Analog and Digital Control System Design" [32]. Particular programming techniques for the selected processor need to be studied.
- *Prototyping*: design of a prototype circuit and prototype software.
  - Design prototype schematics. Test all important capabilities with circuit subsets using perfoards, breadboars, home-made Printed Circuit Boards (PCB). Use similar (or the same) ICs as will be used in the final board. *Tools and techniques*: Circuit capture program (e.g, OrCad or DipTrace), techniques to make "'home-made"' PCBs, soldering, cable construction.
  - Program Software for this prototype board. *Tools and techniques*: Depend on the processor in use (e.g, MPLAB tool for MicroChip microcontrollers).

  - Electrical test using particular signals and code to test paths. Functional Tests of the software and hardware modules. *Tools and techniques*: Debuggers (e.g, MPLAB) for some software parts; tester, oscilloscope for "'real-time"' software and complete-hardware parts.

2.2.4 Subsystems integration & Testing

In this stage, the subsystems need to be integrated so that the functionality of the complete system can be tested. This stage may require changes in the design and implementation of particular subsystems, hence influencing the previous stage. Some steps in this stage are:

- *Communication Protocol Refinement:* Define the network protocol, packet formats, frequency, etc. If possible, communicate a couple of the prototype boards to test preliminary integration.
- *Power Subsystem:* Taking into account all ICs, sensors, actuators defined for each subsystem and the battery autonomy requirements, calculate the needed battery and power stages.
- *Design, Print and Test* the complete electrical circuit:
  - Design the schematic integrating the schematics for prototype boards. Extend to cover the full capabilities and the final ICs. *Tools:* Diptrace or similar.
  - Route the final PCB. Include physical space restrictions and all final footprints. Take into account all circuit design rules (e.g. path width depending on current, path distance, buses routing, etc). For details see chapter 12 of Horowitz's book "The art of electronics" [33].
  - Automatic and peer review checks to prevent bugs in the circuit.
  - Board production.
  - Electrical Test using tester, oscilloscope and particular code for path testing. Fix all the physical/electrical bugs possible, and if there is an unfixable bug, iterate to produce a bug free board.
- *Software*
  - Program the final code for each subsystem separately. In each subsystem, program and test each capability separately and integrate one by one. Debug and test this code (MPLAB, oscilloscope, tester).
  - Program the communication protocol. Program and test each layer separately and integrate. Debug and Test.
  - Integrate each subsystem one by one. Integration tests.

2.2.5 Final mounting

In this stage, all the sensors and actuators need to be mounted in the robot's body. This also means setting in the control board, battery, all the cables and connect everything to get the final robot.

2.3 Case Study: ExaBot development

2.3.1 Goal Definition

The goal of the ExaBot is to have one single robotic platform that allows the following:

1. *Research:* The research activities are focused in autonomous navigation, mainly indoors.
2. *Outreach:* The outreach activities are based in programming simple behaviors in the robot (obstacle avoidance, line-following, maze-solving).
3. *Undergraduate education:* The existing undergraduate courses concern mostly on vision-based navigation.
4. *Low cost:* The robot should be low cost compared to its commercial counterparts. We aimed at a robot that is ten times cheaper than the khepera or the Pioneer robots.

2.3.2 Body, sensors and locomotion definition

Since the robot has a wide application spectrum (research, outreach and education) the main requirement is

Easy Reconfigurability: The robot should support many different sensors and processing units, and can be easily reconfigured with a particular subset of them for a given activity.

In the next subsections, the particular choices for locomotion, body and sensors are explained. However, it is important to keep in mind that these are very tightly coupled decisions: not any body can produce any locomotion or carry any amount of sensors.

Locomotion

A mobile robot needs locomotion mechanisms that enable it to move unbounded throughout its environment. But there are a large variety of possible ways to move, and so the selection of a robot's approach to locomotion is an important aspect of mobile robot design. In laboratories, there are research robots that can walk, jump, run, slide, skate, swim, fly, and, of course, roll. Most of these locomotion mechanisms have been inspired by their biological counterparts. There is, however, one exception: the actively powered wheel is a human invention.

Biological systems succeed in moving through a wide variety of harsh environments. Therefore it can be desirable to copy their selection of locomotion mechanisms. However, replicating nature in this regard is extremely difficult. In general, legged locomotion requires higher degrees of freedom and therefore greater mechanical and control complexity than wheeled locomotion. Wheels, in addition to being simple, are extremely well suited to flat ground. However, even when choosing wheels, there are still many options on type, number and wheel arrangement that result in particular forms of locomotion. Details on these schemes can be found in Chapter 2 of Siegwart's book "Introduction to Autonomous Mobile Robots" [7], together with a good overview of legged locomotion and the introductory concept to locomotion used in this section. More detailed information can also be found in Chapters 7 to 11 of Braunl's book "Embedded Robotics - Mobile Robot Design and Applications with Embedded Systems" [34].

Research in locomotion schemes is not one of the ExaBot´s goals. Keeping in mind the KISS[3] principle of engineering, the simplest solution that can achieve the needed locomotion can be chosen. This robot will move in controlled environments with mostly flat floors (all goals refer to indoor applications). In order to fulfill navigation tasks (Goal 1), it will need to move forward, backwards, and turn in it´s place. However, there is no need for moving sideways. The simpler solution is to use the widely known differential drive with two wheels that cover the whole chassis (so in-place turning can be achieved). Moreover, since in some situations a harsher environment can be encountered, it is desirable to get caterpillars instead of wheels. This locomotion mean can go over small obstacles and traverse softer floors, giving a small advantage that can come in handy when working with high school students, and also for controlled outdoors experiments.

In conclusion, the locomotion for the ExaBot will use two wheels or caterpillars with a two-motor control to achieve differential drive.

[3]Keep It Simple, Silly

Body

The body should be rugged enough to support being handled by middle and high school students (Goal 2), small enough to be transported around easily for those activities but also big enough to support many sensors and different processing power (Reconfigurability Requirement), and on top of all that, low cost (Goal 4). Moreover, from the locomotion point of view, a two wheeled/caterpillar chassis is preferred. Since research in mechanical issues is not a goal for this robot, the simplest solution that meets the goals and requirements is preferred. Hence, a pre-built mechanical kit was selected: the Traxster Kit [35]. This kit is light (900 gr) and medium sized (229 mm length $\times$ 203 mm wide $\times$ 76 mm height), so it is transportable and can accommodate multiple sensors and processing units. It has two caterpillars, each connected to a direct current motors (7,2 V and 2 Amp) with built-in quadrature encoders (see following section), accommodating the required locomotion scheme.

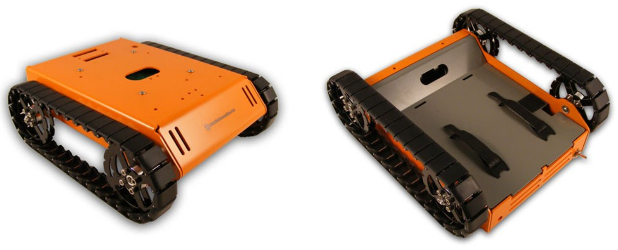


Figure 2.5: The Traxster mechanical kit

Sensors

One of the most important tasks of an autonomous system of any kind is to acquire knowledge about its environment. This is done by taking measurements using various sensors and then extracting meaningful information from those measurements. Sensors can either be Proprioceptive or Exteroperceptive.

- *Proprioceptive* sensors measure values internal to the system (robot); e.g. motor speed, wheel load, robot arm joint angles, battery voltage.
- *Exteroceptive* sensors acquire information from the robot's environment; e.g. distance measurements, light intensity, sound amplitude. Exteroceptive sensor measurements are interpreted by the robot in order to extract meaningful environmental features.

Of course, the discussion about which sensors are exteroperceptive or propioceptive is sometimes fuzzy.

General Classification (typical use)	**Sensor or Sensor System**	**Proprio/ Extero**	**A/P**
Tactile sensors (detect physical contact or closeness; security switches)	Contact Switches	EC	P
	Optical barriers	EC	A
	Non-contact proximity	EC	A
Wheel/motor sensors (wheel/motor speed and position)	Brush encoders	PC	P
	Potentiometers	PC	P
	Synchros, resolvers	PC	A
	Optical encoders	PC	A
	Magnetic encoders	PC	A
	Inductive encoders	PC	A
	Capacitive encoders	PC	A
Heading sensors (orientation in relation to fixed reference frame)	Compass EC	P	
	Gyroscopes	PC	P
	Inclinometers	EC	A/P
Ground-based beacons (localization in a fixed reference frame)	GPS	EC	A
	Optical or RF beacons	EC	A
	Ultrasonic beacons	EC	A
	Reflective beacons	EC	A
Active ranging (reflectivity, time-of-flight, and geometric triangulation)	Reflectivity sensors	EC	A
	Ultrasonic sensor	EC	A
	Laser rangefinder	EC	A
	Optical triang. (1D)	EC	A
	Structured light (2D)	EC	A
Motion/speed sensors (speed relative to fixed or moving objects)	Doppler radar	EC	A
	Doppler sound	EC	A
Vision-based sensors (visual ranging, segmentation, object recognition)	CCD/CMOS camera(s)	EC	P
	Visual ranging packages	EC	P
	Object tracking package	EC	P

Table 2.1: Classification of the most useful sensors for mobile robot applications (table taken from [7])

Table 2.1 provides a classification of the most useful sensors for mobile robot applications.

When choosing sensors, a criteria to compare performance is needed. The main four performance characteristics to take into account are: range, resolution, linearity of the transfer function and frequency (how many sensor readings per second). Moreover, other characteristics need to be noticed for the next steps, mainly voltage of operation; maximum, minimum and typical consumption; if the output signal is analog or digital. Note that there are other important performance characteristics to take into account that cannot be easily measured in a lab and depend on the environment the sensor will work. Examples are sensitivity, cross-sensitivity, error or accuracy. The introductory concepts to sensors in this section are taken from Chapter 4 of Siegwart's book [7] and Chapter 2 of Braunl's book [34]. Much more information about different sensors and all the definitions used in this section can be found in these books.

Exteropercetive Sensors

Goal 1 establishes that the robot will be used for research in autonomous navigation, mainly indoors. Moreover, the low-cost goal also has to be taken into account. From table 2.1, sensors usually used for navigation are ground-based sensors, range-finding sensors and vision-based sensors. The most common ground-based sensor is GPS, but it is not suitable for indoor navigation. The other methods include costly beacons and only work in the particular place these beacons have been set up (e.g. based on radio waves [36], multi-camera systems [37] or similar technology). Hence, no ground-based sensors were included.

From the range-finding sensors, lasers are the most precise and reliable, however their cost exceeded by several times the intended final cost of the robot. The infrared optical triangulation range finders are cheap, short-range punctual sensors (less than a meter). Ultrasonic range finders are non-punctual, long range (several meters) sensors, and more expensive. Hence, the ExaBot was given a ring of 8 Sharp GP2D120 [38] IR range finders and a Devanatech SRF05 sonar [39].

The final sensors used for navigation are vision-based sensors or cameras. Vision-based robotics is a growing field. Cameras can be very cheap and give a lot of information about the environment. There is considerable research into image processing, trying to extract meaningful features from images to perform many high-level operations (object recognition, localization, navigation among others). Also, combining cameras with structured light allows to have sensors that provide 3D data. Hence, a camera was also included. Since camera technology evolves very fast, support for an USB camera was included but the exact camera was not decided. This also covers the undergraduate education goal (Goal 3).

Finally, Goal 2 establishes that the outreach activities are based in programming simple behaviors in the robot (obstacle avoidance, line-following, maze-solving). For these, two cheap extra sensors were included: line-following and bumpers.

Proprioceptive Sensors

The proprioceptive sensors are not directly related to the goals of the robot but more as a service for the control needed for those goals. The main proprioceptive sensors were included to have a better control over the motors to achieve an accurate differential drive.

Wheel quadrature encoders (built-in in the Traxster kit motor) were included to measure the movement of the motor. An encoder is an electromechanical device that converts the angular position of a shaft to an analog or digital signal. Incremental encoders provide information about the motion of the shaft, that is, how many counts of encoders have passed since the last time the sensor was checked. Quadrature encoders provide also information about the direction of the rotation.

Also, an Allegro ACS712 [40] current consumption sensor for the motors and battery voltage sensors were included. Both are analog sensors, in which the output voltage signal of the sensor is proportional to the current consumption and the battery level accordingly.

Final considerations

With these sensor choices, the design of the ExaBot covers five of the seven types of sensors mentioned in table 2.1, only missing heading sensors and ground-based beacons. This means that the requirement for many sensors that can be taken off at any time is covered. To improve this even further, a new requirement arises:

Sensor Expansion Requirement: try to have exported expansion ports so that new sensors can be added in the future.

This is to be kept in mind when choosing microcontrollers, DSPs or microprocessors and mapping the pins to the different functions. Most of the time, the pins of these embedded controllers can have many different functions, so the mapping can be done having as a goal to maximize the pins that are left free with interesting functions to export.

Table 2.2 summarizes the Exabot's sensors together with their characteristics.

Computational power

The different goals of the ExaBot pose very different computational power needs. Many research activities such as vision-based algorithms are usually computationally demanding, while most of outreach experiences can be done with very simple programs. Hence, the main drive in deciding on the computational power of the ExaBot was again the easy Reconfigurability Requirement. The processing power was divided in two levels: low level processing units for sensor and motor control, and a high level processing unit for more complex algorithms. This high level processing unit was thought so it could be easily removed or replaced. This will be explained in detail in following sections.

Sensor	range	precision	linear	freq.	A/D
GP2D120	4-30 cm	A/D conversion	no	26 hz	A
SRF05	1cm-4mts	3-4 cm	yes	20 hz	D
linefol	1-12mm	NI	no	NI	D
bumpers	1-10mm	NA	no	NA	D
3D camera	NA	640x480 pixels	NA	30fps	D
encoders	NA	624 ppsr	yes	NA	D
ACS712	-20-20A	100mV/A	yes	80kHz	A
battery	dep. on Rs	A/D conversion	yes	NA	A

Table 2.2: Characteristics of the ExaBot sensors. NA: Not Applicable, NI: Not Informed, Rs: Resistance

2.3.3 Partition in Subsystems

At this point, all the actuators, sensors and the mechanical body are established. Hence, it is the moment to think on what hardware and software will be necessary to control them all. First, a general idea of what subsystems are needed is presented.

From the previous section, three different subsystems can be easily identified: sensor control, motor control and high-level control. In this division, proprioceptive sensors (encoders, motor consumption and battery level) should indeed be part of the motor control, since they are included as sensors to implement better motor control algorithms. This three main control subsystems create the requirement for at least two other subsystems: a power subsystem and a communication protocol to interconnect them.

A good way to start refining each subsystem is to think of what processing unit (if any) is appropriate for each subsystem. There are mainly three types of CPU-based processing elements that are used in embedded systems: embedded microprocessors, microcontrollers and DSPs (Digital Signal Processors). Embedded microprocessors are usually used for general purpose programming or high-level programming. They usually include hardware support for multiple function calls, interrupts and for Operating Systems; pins to connect to external memory, and also several I/O for embedded applications. Microcontrollers are used mainly for control in embedded systems. They include a CPU, smaller than that of a microprocessor, generally with no support for OS and limited support for function calls and interrupts. They also include on-chip program and data memory, and very good and varied amount of I/Os to connect directly to the element to be controlled. DSPs are a special branch of microprocessors tuned for Digital Signal processing, usually featuring special SIMD (Single Instruction Multiple Data) instructions, multiple MACs (Multiply and Accumulate) and floating point units integrated in the datapath.

These are used for processing signals that require large amount of mathematical operations to be performed quickly and repeatedly over an incoming data stream (for example in audio or video processing).

For motor and sensor control, the most logical and common choice are microcontrollers, since the main function of these CPU-based elements is control. There are thousands of different microcontroller types in the world today, made by numerous manufacturers. One of the most used is the Microchip PIC. Microchip has several PIC families: each family has many devices all sharing the same CPU core but with different combination of peripherals and memory sizes. At the moment the microcontrollers for the ExaBot were chosen, the PIC18F was the most advance family from Microchip. The devices from this family are low-cost, self-contained, 8-bit, Harvard structure, pipelined, RISC, single accumulator (the Working or W register), with fixed reset and interrupt vectors. A very important feature of this family is that it is targeted to be programmed in C, unlike its predecessors that were largely programmed in assembler.

For the high level control, the main idea is to provide a processing unit to control the whole robot in order for it to be autonomous, and with enough processing power for some research related algorithms such as artificial intelligence or robotics vision, among others. In this case, since this processing unit will carry general purpose programming, the most suitable CPU-based element is an embedded processor. This processor can also provide the connection for the camera (that is otherwise difficult to control with a microcontroller) and wi-fi connection. Having this in mind, an embedded ARM 9 PC104 of 200 Mhz TS-7250 [41] was included. This embedded PC has 2 USB ports, a serial port, an Ethernet port and several General Purpose I/O. It runs Linux Kernel 2.26, so that drivers to use the USBs for a webcam or a Wi-Fi key can be easily obtained. Moreover, a full cross-compiler chain is available so it can be easily programmed. It consumes at maximum 400 mA, quite a reasonable consumption for a processor with those characteristics.

However suitable this embedded processor was in 2008 when the ExaBot was first designed, it was clear that this would change in time. Hence, the ExaBot was designed so the PC104 can either be present or not, and can be easily replaced. In this manner, the processing power of the ExaBot also follows the Reconfigurability Requirement, and not only can change depending on the application, but also as more powerful and smaller embedded processors or processing units are launched into the market. As can be seen in section 2.4, this particular property has come in handy. This reconfigurability poses requirements for the communication subsystem and in the integration phase, since all hardware and communication protocols need to be ready to work without the PC104 or with another type of high-level control.

The power and communication subsystems do not need special CPU-based elements. The power subsystem will be mostly analog power circuitry to attain the required regulated voltages. The communication subsystem will be implemented in each of the communicating elements (that is, the microcontrollers and the microprocessor). This poses the need to look into the communications at this moment, to see if a suitable communication protocol is available in the selected PIC family and embedded processor. If not, the family decision needs to be redefined. Most of the devices from the PIC18F family include at least two communication modules: a Synchronous Serial Port module that can implement Inter-Integrated Circuit (I^2C) or Serial Peripheral Interface (SPI) protocols, and an Universal Synchronous/Asynchronous Receiver/Transmitter (USART) module. Since in normal operation, the communication needed is between one master (the embedded microprocessor) and several slaves (the microcontrollers), the most suitable protocol from these is the SPI. This bus is exactly a one master-multi slave protocol, being the simplest of the three available protocols that complies with the needed communication pattern. The PC104, as many other embedded processors, include this communication protocol, but it is not as widespread as the USART protocol for other type of high-level processing units. Hence, following the Reconfigurability requirement, the implementation of an USART interface needs to be also studied. This will be further discussed in the communication subsystem refinement.

The inclusion of CPU-based programmable devices in the design posses the need for an additional subsystem: the programming subsystem. That is, all the electronics and connectors so that the microcontrollers can be easily programmed even when already soldered in the final board. This is fundamental for the Reconfigurability requirement: if extra sensors or other high-level controlling units are to be included, the programming in the microcontrollers will probably need updating.

In the following subsections, each subsystem is designed and tested independently, complying with the Subsystem Refinement stage. The hardware and software needed to control the elements in each subsystem is devised. Of course, these are tightly coupled decisions, so they will be discussed jointly.

Microcontroller Programming Guidelines

A key issue when programming the control of a robot is that all sensor data and commands need to be real-time and synchronized, since the robot moves in the real world. If commands or sensing are delayed, or happen at unknown intervals, there is no way of knowing if the conditions have already changed and hence the commands are no longer useful or even harmful. Microcontrollers have special features to accommodate this type of real-time synchronized control: the combination of different hardware modules for different peripherals, interrupts and timers. For a good introduction to microcontroller programming guidelines, see chapters 3 and 6 of Wilmshurst's book "Designing

Embedded Systems with PIC Microcontrollers" [42].

2.3.4 Subsystem Refinement: Motor Control

The goal of this subsystem is to control the two DC (Direct Current) motors in the ExaBot. Since the control of each motor is independent and symmetrical, the control of only one motor will be first discussed.

Hardware

A direct current motor has only two wires: Vcc and Gnd. Depending on the way they are connected, the motor moves one way or the other. Moreover, the speed of the motor depends on the Voltage supply and it´s torque on the current. In the ExaBot, the motors need 7,2V for maximum speed and consume 2 Amp for maximum torque. The basic control idea is to achieve two things: run the motor in forward and backward directions and modify its speed.

In order to run the motor forward or backwards without having to reconnect the cables to Vcc and GND alternatively, an H-bridge is generally used. An H bridge is an electronic circuit that enables a voltage to be applied across a load in either direction. However, the PIC cannot drive the motor directly: a typical PIC pin can drive up to 15 mA at 5V, and the motors consume up to 2A at 7,2V. Hence, some kind of driver is needed. In the ExaBot a L298 [43] driver was included. This driver implements an H-bridge and can work with up to 46V and drive up to 4A.

In order to modify the motor's speed avoiding analog power circuitry Pulse Width Modulation (PWM) can be used. PWM utilizes the fact that mechanical systems have a certain latency. Instead of generating an analog output signal with a voltage proportional to the desired motor speed, it is sufficient to generate digital pulses at the full system voltage level (in this case 7,2V). These pulses are generated at a fixed frequency. By varying the pulse width in software (see Fig. 2.6), the equivalent or effective analog motor signal is changed, and therefore the motor speed is controlled. The motor system behaves like an integrator of the digital input impulses over a certain time span. The quotient t_{on}/t_{period} is called the "pulse–width ratio" or "duty cycle"

To achieve this PWM control, the most simple way is to program it in some kind of CPU-like programmable device. As already stated, the PIC18F microcontroller family was chosen for both this subsystem and the sensor control subsystem. In this case, the chosen microcontroller of the PIC18F family must have a PWM module. Moreover, the proprioceptive sensors that this subsystem must control are quadrature encoders, current sensors and battery level

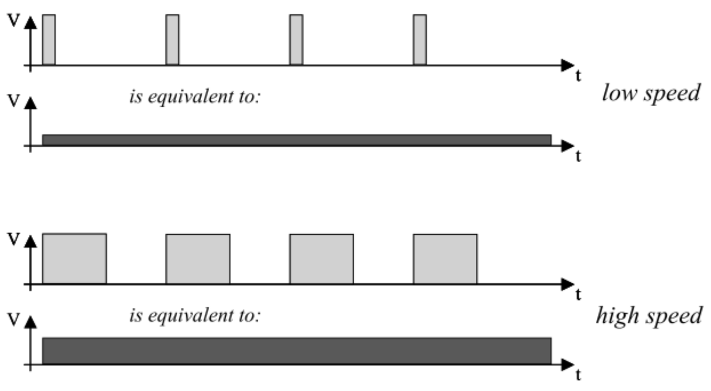


Figure 2.6: Pulse Width Modulation

sensors. For quadrature encoders, at least two digital pins are needed for their two outputs. However, as these are widely used sensors, several microcontrollers include Quadrature Encoder Interface (QEI) modules that interpret by hardware the increment and direction of the wheel, so such a module is desired. Finally, current and battery sensors are both analog signals, so an ADC (analog Digital Converter) module and at least two analog pins are needed.

If one PIC is to be used to control both motors, each module needs to be duplicated and the whole programming to control both motors have to fit in a single PIC. For simplicity reasons, and since the velocity and direction control of the motors is independent from each other, one PIC was used for each motor. From the PIC18F family, the PIC18F2431 [44] complies with all the required modules: it has been design specifically for motor control.

At this point, all the fundamental hardware for motor control has been defined. However, an extra safety measure was also included in the motor control. The output voltage of the ACS712 indicating motor current consumption is also used to implement a fault circuit in order to override all PWM output and hence stop the motors if error conditions happen. The problem arises when a motor is jammed for some reason. In that case, the motor consumes all the available current from the battery, and if not turned off, surely something will be burnt. To prevent this from happening, the sensed current inputs a LM319 voltage comparator [45] that is set so a 3 Amp threshold is not surpassed. If that threshold is passed, the Fault pin of the PIC is driven low, and the Fault module overrides the PWM output by hardware, hence stopping the motor. Figure 2.7 shows a diagram of the main ICs and connections on this subsystem.

A final remark needs to be done about the battery sensor: this sensor, differently from all the others, is not an off-the-shelve item. A resistive divisor was implemented to get the battery voltage to the 0-5V range so it could input the PIC directly.

For more information on DC motors and their low-level control, see chapter

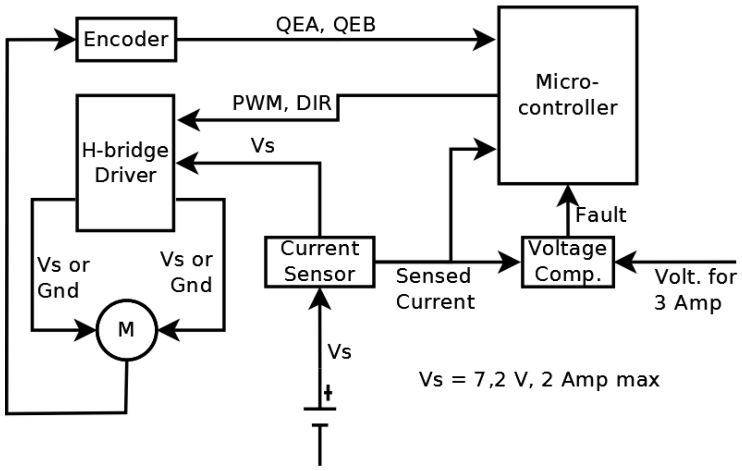


Figure 2.7: Motor subsystem diagram

3 of Braunl's book [34]. For a good introduction to microcontrollers, their difference to microprocessors and an overview of Microchip PIC families, see chapter 1 of Wilmshurst's book [42].

Software

Motor Control

The most simple control for the velocity and direction of the motor is to use PWM. For the desired velocity, the appropriate duty is calculated by simple rule of three cross-multiplication and then set to the PWM hardware module in the PIC18F2431. The problem with this approach is the lack of feedback: in this scheme, the actual motor speed cannot be obtained. This is important, because supplying the same analog voltage (or equivalent, the same PWM signal) to a motor does not guarantee that the motor will run at the same speed under all circumstances. For example, with the same PWM signal, a motor will run faster when free spinning than under load. In order to control the motor speed, feedback from the motor shaft encoders is needed. Feedback control is called "closed loop control", as opposed to "open loop control" such as PWM.

The most commonly used closed loop controller is the Proportional Integrative Derivative controller (PID controller). A PID controller calculates an "error" value as the difference between a measured process variable and a desired set-point. The controller attempts to minimize the error by adjusting the process control inputs. In the case of motor control, it tries to minimize the difference of the actual motor speed from the desired motor speed by adjusting the duty cycle (i.e., the PWM signal).

The PID controller algorithm involves three separate constant parameters, and is accordingly sometimes called three-term control: the proportional, the integral and derivative values, denoted P, I, and D. Heuristically, these values can be interpreted in terms of time: P depends on the present error, I on the accumulation of past errors, and D is a prediction of future errors. The weighted sum of these three actions is used to adjust the process via a control element; in the case of the control of a motor using PWM, the duty cycle (see Fig. 2.8).

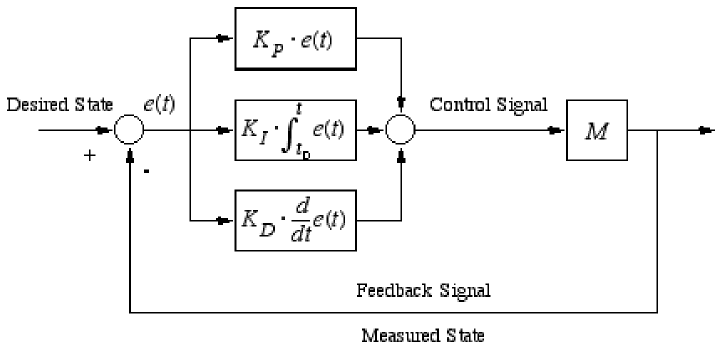


Figure 2.8: Proportional Integrative Derivative Control

In this control, the Proportional, Integrative and Derivative constants K_p, K_i and K_d need to be adjusted empirically since they depend on the particular motor to be controlled. For the ExaBot, these parameters were tuned using the Ziegler-Nichols method.

For a good overview on PID control including guidelines on how to adjust the PID constants, see chapter 4 of Braunl's book [34]. For complete details on the control theory and mathematics behind PID, see chapter 14 of Chen's book [32]. PID control and Ziegler-Nichols method is also explained in chapter 10 of Ogata's book “Modern Control Engineering” [46].

The PIC hardware modules used in motor control are the PWM and the QEI, together with interrupt logic. The QEI module is configured in Position Mode and hence automatically increments- or decrements- a counter with each hardware encoder pulse, depending of the direction of the motor. This counter can be read and reset from software. The basic four things to configure in the PWM module are: period, duty, direction and fault logic. The PWM in the ExaBot is configured to have a frequency of 1,225 Khz (period of 0.81 ms), to generate an interruption once per period and have the fault logic enabled. The duty and direction are controlled by the PID and are set every period according to the last speed and direction command received, and the readings from the encoder counter of the QEI. In Fig. 2.9 a sequence diagram can be found with the interactions between the different modules to achieve motor control.

The interrupt for the PID control is the only interrupt set at high priority.

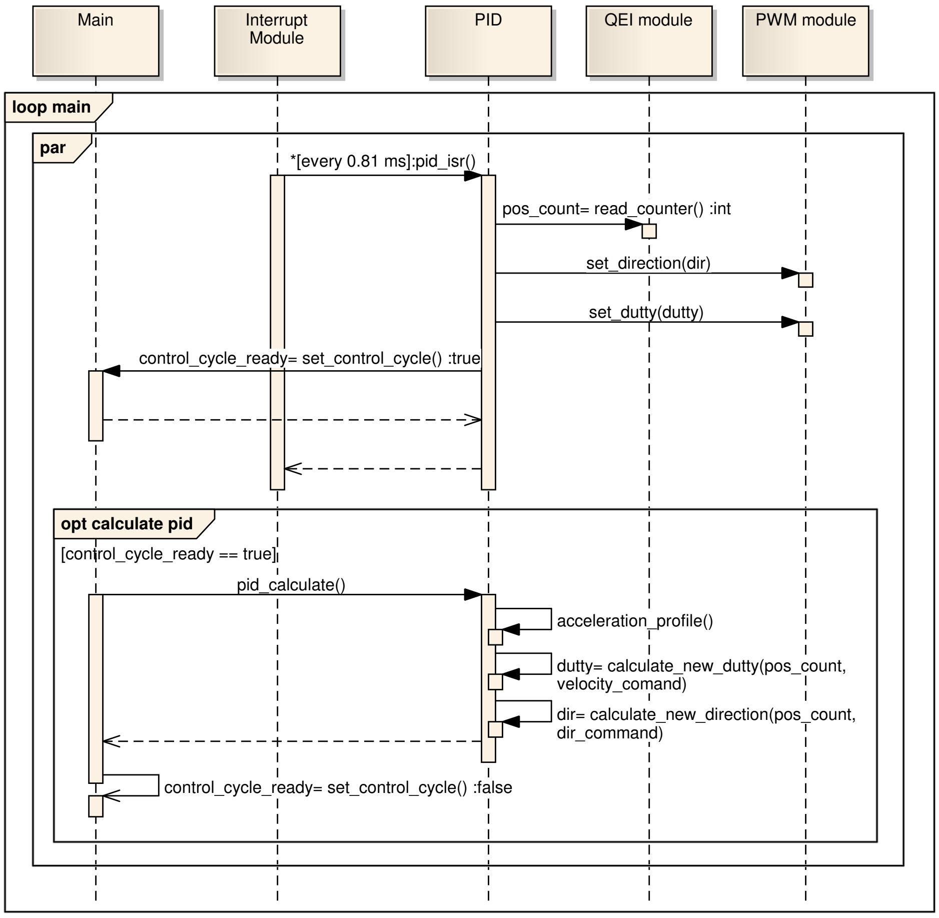

Main
Interrupt Module
PID
QEI module
PWM module
loop main
par
*[every 0.81 ms]:pid_isr()
pos_count= read_counter() :int
set_direction(dir)
set_dutty(dutty)
control_cycle_ready= set_control_cycle() :true
opt calculate pid
[control_cycle_ready == true]
pid_calculate()
acceleration_profile()
dutty= calculate_new_dutty(pos_count, velocity_comand)
dir= calculate_new_direction(pos_count, dir_command)
control_cycle_ready= set_control_cycle() :false

Figure 2.9: Sequence diagram showing the interaction of the different modules to achieve the PID control

This is done to try to ensure that the control will be done as timely as possible.

For extra details see the PIC18F2431 data sheet [44], chapter 17 for PWM module and chapter 16 for QEI module.

Battery Sensor Control

The basic idea of the battery sensor is to warn the ExaBot user that the battery is running low, and hence, the robot should be turned off and the battery should be changed. The battery sensor is implemented with a resistive divisor, tuned to take the raw voltage output of the battery (i.e., before regulating) to a 0-5V range. Hence, the battery sensor control takes as input an analog voltage between 0-5V, that represents the real output voltage of the battery multiplied by a constant α given by the resistive divisor. The battery control is implemented using the 10-bit analog/Digital Converter module of the PIC. This module works with a timer: when the timer wraps around, it launches an ADC conversion. The timer was configured for the maximum possible period, since the battery is depleted slowly[4].

Since both the resistive divisor and the ADC converter are linear, we can easily calculate the real output voltage of the battery V_{out} from the 10-bit digitalized value V_{dig}, knowing the maximum voltage of the battery V_{max} and the constant of the resistive divisor α using the three-cross multiplication rule:

$$\begin{aligned} V_{max} \times \alpha &\longrightarrow 2^{10} \\ V_{out} &\longrightarrow V_{dig} \end{aligned}$$

Hence, to calculate V_{out}, equation 2.1 can be used:

$$V_{out} = \frac{V_{dig} \times (V_{max} \times \alpha)}{2^{10}} \tag{2.1}$$

The battery sensor subsystem checks if V_{out} is under a defined threshold, and if that happens, it turns on a led informing the user that the battery is too low to continue using. For details on the ADC module, see chapter 20 of the PIC18F2431 data sheet [44].

Motor Consumption and Fault Control

Although all the hardware necessary for motor consumption control is included in the board, this control is not implemented in the current software version. The required software for this control is similar to the one for the battery control, since it is an analog value. However, instead of turning a led on, it would be an extra input in motor control. How to take this information into account in motor control requires extra thought.

The Fault control does not require extra software: it is just a configuration of the PWM module. If it is set to take into account Fault control, when the Fault pin is driven low, all the PWM outputs are driven to zero by hardware, hence stopping the motor.

Prototyping

Before integrating this subsystem into the complete system and making the final PCB, its most important parts need to be prototyped. Prototyping is a time consuming and expensive procedure, since it requires printing boards, buying ICs and components, soldering, building cables, etc. However, it is a vital step to prevent designs erros from entering into the final circuit version

[4]Since the timer that works with the ADC converter is Timer5 (a 16 bit timer) and the PIC runs with a 10 Mhz clock; the maximum possible period is $2^16 \times \frac{1}{10^6} = 0.00655$ seconds. This period is far enough for battery control.

that can make the board useless. Hence, it is fundamental to prototype at least the critical sub-circuits. In this subsystem, that means that motor control must be prototyped, while the battery level and motor consumption circuits can have more relaxed testing. Apart from the already mentioned ICs and circuits, in designing the prototype (or final) schematic the needed electronics for each IC to work needs to be taken into account. This information can be found in the data sheet of the IC, generally there is an application section with typical operating circuits.

There are different techniques for prototyping, depending on the extent of the needed circuit. These include the use of breadboards, stripboards, perfboards, home-made PCBs and even industrially made PCBs. A breadboard (also called protoboard) is a construction base that is solderless. Components are plugged into the board, so when the prototype is done, it can be dismounted and the board can be reused. Stripboards are characterized by a 0.1 inch (2.54 mm) rectangular grid of holes, with wide parallel strips of copper cladding running in one direction all the way across one side of the board. A Perfboard is also a thin, rigid sheet with holes pre-drilled at 0.1 inch intervals across a grid. However, instead of strips of copper, this board has round or square separate copper pads in each hole (see Fig. 2.10). For these prototype boards, only Dual-In-Line (DIP) ICs can be used.

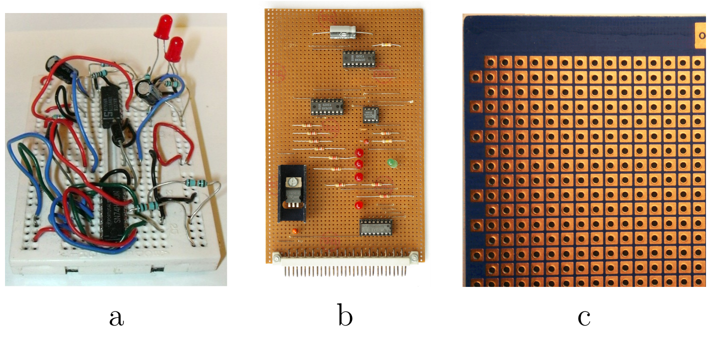


Figure 2.10: Prototyping boards: **a)** Breadboard, **b)** Stripboard, **c)** Perfboard

If the circuits are more complex, then home-made or industrial PCBs are needed. For this type of prototyping circuits, the schematic needs to be captured and the PCB routed. There are several proprietary and open-source Electronic Design Automation (EDA) tools for PCB design, varying in complexity, capabilities, and also cost. In this project, CadSoft's EAGLE[47] was used for prototype boards and Novarm's DipTrace [48] for the final board. This is a mid-range Schematic Capture and PCB Layout software that although is proprietary, has a free version for designs of up to 4 layers and 500 pins. To manufacture a home-made PCB, a subtractive method is generally used that removes copper from an entirely copper-coated board, leaving copper only in the IC footprints or paths.

For the motor subsystem prototype, simple industrial PCBs were used. These boards were adapted from prototype boards used for another robot development in the lab (the KonaBot). They include a PIC18F2431 DIP microcontroller and a L293 [49] H-bridge driver. The L293 driver is identical to the L298 driver used in the final version of the motor subsystem but for smaller motors (it can drive up to 1 Amp instead of 2 Amp). Hence, the system was tested with a 2224R006SR Faulhaber motor [50], a small motor that is combined with quadrature encoders. For the prototyping stage, this is not a problem since the electrical circuit and the complete code can be tested, only the PID constants need to be adjusted afterward for the real motor. The prototype board also includes a small power stage using a LM7805 [51] regulator to guarantee stable 5V for all the logic. Tests with the ExaBot motor were also conducted, but at reduced speed and with no load to not surpass the 1A limit.

The PIC programming was done using C language, Microchip's C18 compiler for PIC18F family and Microchip's MPLAB [52] programming and debugging environment. For debugging, a Tektronix TDS 1002 oscilloscope was also used when the real time works of the signal needed to be seen, for example to monitor the relationship between the PWM signals and the motor movement. In Fig. 2.11 a picture of the prototype board for the ExaBot's motor subsystem can be seen.

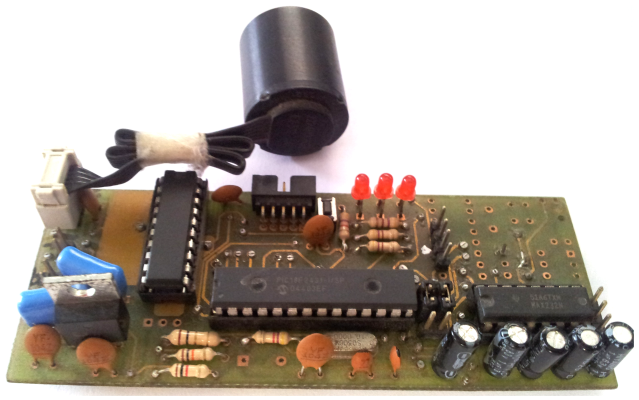


Figure 2.11: Prototype board for the Motor control with Faulhaber motor connected. It includes a PIC18F2431, a L293 driver in the upper left and a LM7805 in the bottom left. The other IC is a MAX232 for USART.

2.3.5 Subsystem Refinement: Sensor Control

This subsystem needs to control eight infrared range finders, one sonar, two line-following and two bumpers.

Hardware

In this subsection, the needed hardware to control each type of sensor will be described.

Infrared range finders The chosen infrared range finders are the Sharp GP2D120. As can be seen in the data sheet, this sensor has three pins: Vcc, Gnd and Vout. The timing diagram shows that measures are given every 38 ± 9.6 ms after power is supplied in the Vcc pin (see Fig. 2.12) [38].

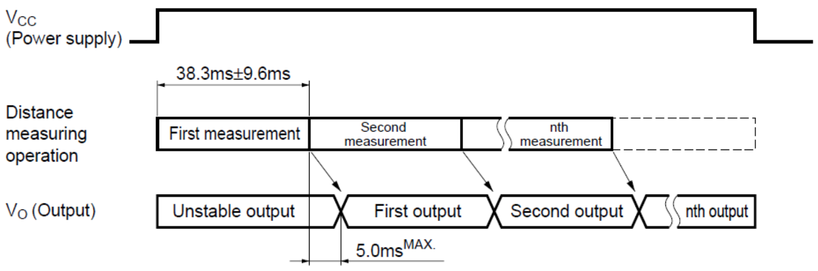


Figure 2.12: Infrared range finder timing diagram.

The output is an analog voltage between 0 and 3.1V that is a function of the measured distance (see Fig. 2.13).

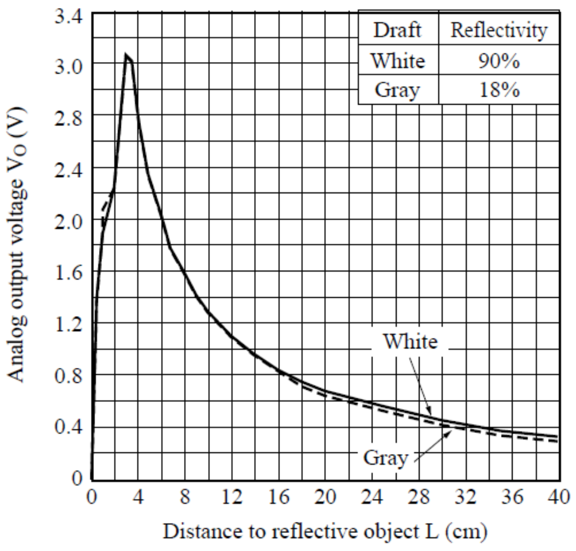


Figure 2.13: Infrared range finder analog output voltage vs distance to reflective object.

In order to get the distance measurements, the analog output has to be sensed in the middle of the output period (when the signal is stable), digitalized and then- using the inverse of the transfer function- translated to distance in cm. The simplest way to achieve this algorithm is to program a microcontroller, a device of the PIC18F family. The requirements for this part are that the microcontroller has one timer- to trigger the range finders and sense the output at the appropriate time-, 8 digital pins to trigger each range finder, 8 analog pins to get the output voltage of each range finder and at least one analog/Digital Converter (ADC) module to digitalize the analog output.

Moreover, since the typical consumption of the range finders is 45 mA and the typical PIC pin can only drive 15 mA, the digital trigger pin cannot drive the range finder directly. Hence, a driver is needed. A simple way to create such a driver is to use a transistor working as a switch. The transistor enables current to go from the battery (emitter) to the range finder (collector) when the trigger pin (base) goes low.

Sonar The SRF05 sonar has four pins: Gnd, Vcc, Trigger Input and Echo Output. First, a 10μs pulse to the trigger input needs to be provided to start the measuring. When the sonar sends the ultrasonic burst, it raises the echo output until it receives an echo from some object. The echo line is therefore a pulse whose width is proportional to the distance to the object. By timing the pulse it is possible to calculate the range in centimeters: if the width of the pulse is measured in us, then dividing by 58 will give the distance in cm. If nothing is detected then the SRF05 will lower its echo line anyway after about 30mS. The sonar can be triggered every 50 ms (i.e, 20 times per second). Figure 2.14 shows the timing diagram for the sonar.

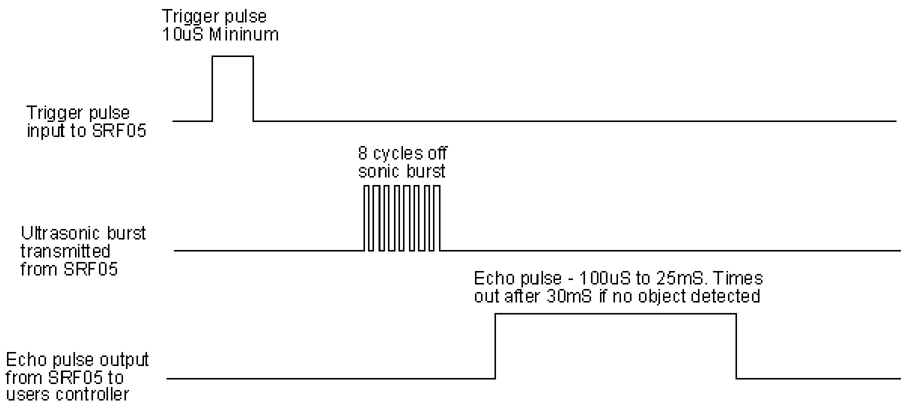

Trigger pulse
10uS Mininum
Trigger pulse input to SRF05
8 cycles off sonic burst
Ultrasonic burst transmitted from SRF05
Echo pulse - 100uS to 25mS. Times out after 30mS if no object detected
Echo pulse output from SRF05 to users controller

Figure 2.14: Sonar Timing Diagram

The requirements for the microcontroller in this case are two digital pins (one for the trigger and another for the echo), a timer to produce the 10 μs trigger and some way to measure the time the echo line is up. Several PICs have a special Capture module to do exactly this, so one of such modules is required. In this case, the trigger and the Vcc are different lines, so although the Sonar can sink up to 30 mA, these don't need to be driven by the PIC pin: they are drawn directly from the battery through the Vcc line. Hence, no extra driver is needed.

Line-following and Bumpers The two line following have three pins:

Vcc, Gnd and Out. The Out line has a digital one (5V) when it is seeing a white line, and a digital zero (0V) otherwise. The bumpers have only two pins: when it is pressed, it joins both pins. Hence, they way to make it work is to connect one of the pins to Vcc (likewise, Gnd) and the other one to the PIC pin with a pull-down (likewise, pull-up). In this manner, when the bumper is not pressed the PIC line has a digital zero and when it is pressed, the line has a digital one.

To get a reading from this sensors, one way is to connect the out pin to an interrupt-on-change pin in the microcontroller. As it is clear from the name of the pin, this hardware module in the microcontroller generates an interruption when the input value changes. However, it is hard to get a microcontroller with 4 such pins. Another possibility is to poll the value of the pins at a regular interval. In this manner, the requirement for the microcontroller is to have 4 digital inputs, 2 digital outputs for the trigger of the line-following (the bumpers are simply a switch so they don´t need a trigger), and a timer to trigger the polling.

The total requirements for the microcontroller (only to control the sensors) are: 8 analog pins, 16 digital pins, 2 timers, an ADC module and a Capture module. From the PIC18F family, the PIC18F4680 device features all this, so it is a clear candidate. This is a 44 pin microcontroller, differently from the 28 pin PIC18F2431 used for motor control. Figure 2.15 shows a diagram of the main ICs and connections on this subsystem.

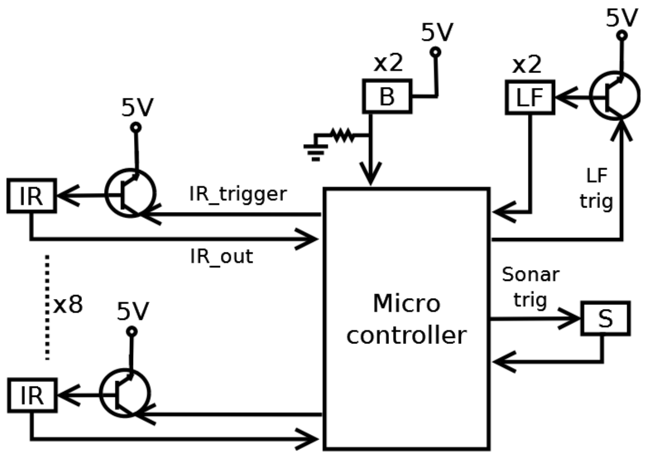


Figure 2.15: Sensor subsystem diagram

Software

An important feature of sensor control is that each sensor can be turned on or off by software. There are special high-level commands so that only the sensors that are going to be used for the particular application are sensed. This follows the Reconfigurability Requirement, enabling also important savings in energy- and hence in battery time-, always a sensible issue for an autonomous robot.

Infrared range finder Control Following the timing diagram in Fig. 2.12, the output of the range finder is stable for sure between $38.3 + 9.6 + 5 = 52.9$ms and $(38.3-9.6+0)*2 = 57.4$ ms after it has been first triggered. Hence, a timer needs to be set so that each range finder is put to A/D conversion 56 ms after it has been triggered. Since the PIC18F4680 has only one ADC module, a round-robin algorithm was implemented to capture the value of each range finder at a time. The timer was set to 28 ms, so that two range finders are on at all times and each range finder is ready for capture after two timer interruptions. This yields a throughput of one new range finder value every 28 ms. When the range finder is ready for capture, the ADC module starts the conversion, and interrupts the microcontroller when it finishes digitalizing the value. At that time, the result is stored and the range finder is turned off. In the current code version, the translation between the digitalized range finder value and the distance in cm is implemented in the high-level processing unit, following the transfer function of Fig. 2.13. It is important to note that, in the case of infrared range finders, the ability to turn off unused sensors ensures that the range finders that are in use are sensed more frequently.

For programming details see the PIC18F4680 data sheet [53], chapter 11 for TMR0 module, chapter 19 for the ADC module and chapter 9 for Interrupts.

Sonar Control The sonar trigger is controlled by the TMR1 timer module. This timer is first set to 100 μs to ensure the trigger line of the sonar is raised for that time. Then it is set to 50 ms to ensure the CCP module has captured a new echo value, and that the sonar is ready for a new trigger as specified by the sonar data sheet.

The sonar capture is controlled by the Capture/Compare/PWM (CCP) module. This module in its Capture mode works with an associated timer, in this case TMR3. The module captures the TMR3 value and produces an interrupt when a specified change in the capture pin occurs (for example, falling or raising edge). In this case, the capture pin of the CCP is connected to the echo line of the sonar. Figure 2.16 presents a sequence diagram with the interactions between the different PIC modules to achieve sonar control. For programming details see the PIC18F4680 data sheet [53], chapter 12 for TMR1, chapter 14 for TMR3 module, chapter 15 for CCP module and chapter 9 for Interrupts.

Line Following and Bumper Control The Out lines of the two line-following and the two bumpers are connected to digital pins in the PIC. The software to control these sensors is simply polling: if they are on, once in every main control cycle, each pin is polled to check the corresponding sensor value.

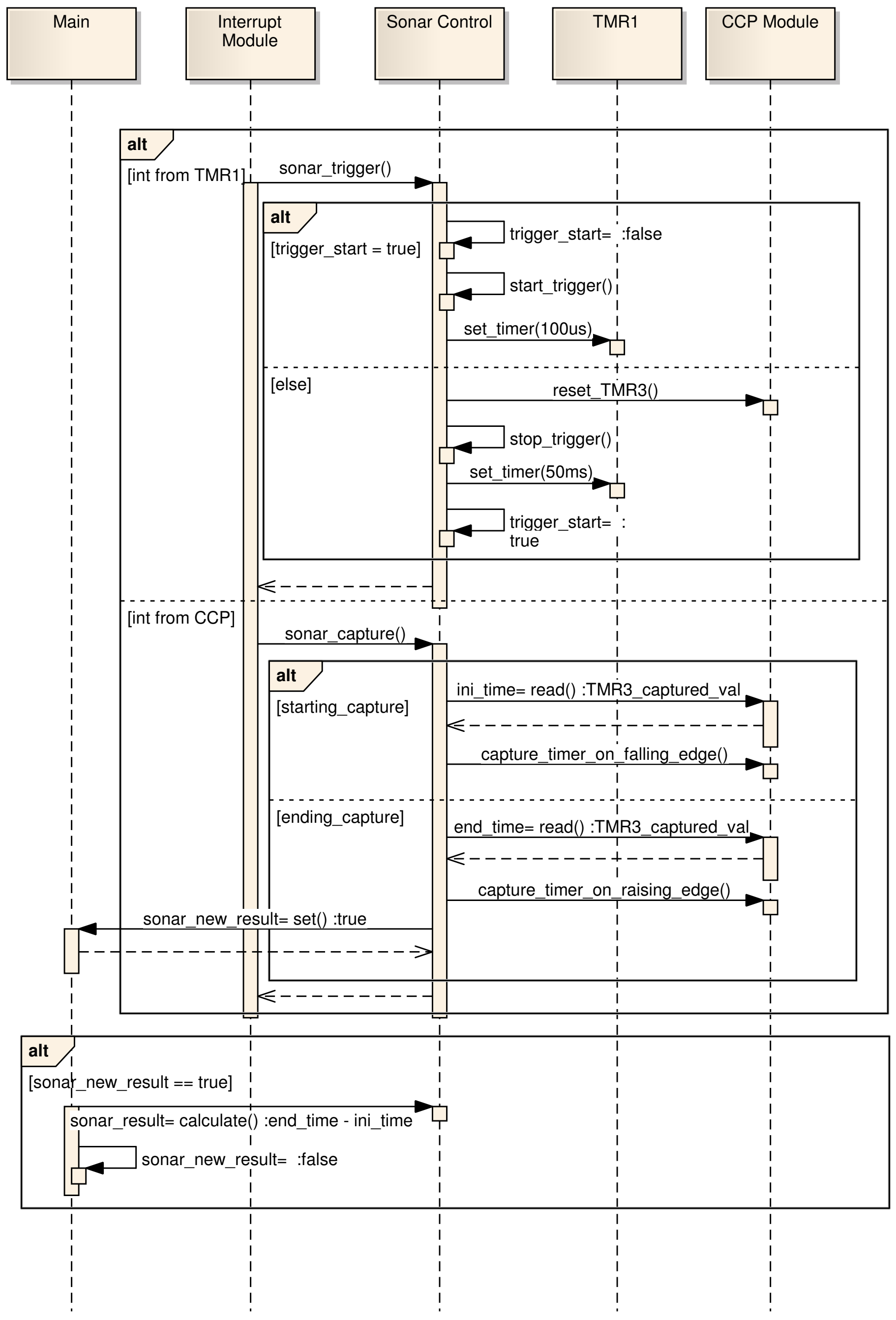


Figure 2.16: Sequence diagram showing the interaction of the different modules to achieve Sonar control

Prototyping

To prototype the sensor subsystem, a simple industrial PCBs was adapted from a prototype board used for a previous robot development in the lab (the

FenBot). This include a PIC18F248 DIP microcontroller, a similar controller as the PIC18F4680 that includes all the necessary modules to test (CCP, ADC and timers), but it is smaller (only 28 pins) and hence simpler to route. In this board, the complete range finder acquisition was tested with one range finder (so one transistor was included), and the round-robin ring algorithm with 3 triggers. The sonar, line-following and bumper code was also tested, as well as the SPI communication (as described in section 2.3.7). In this case, the prototype board did not include a power subsystem, so the 5V were regulated externally using a Kenwood PR18-12 regulated DC power supply. As in the motor subsystem, the PIC programming was done using C language, Microchip's C18 compiler and MPLAB environment, and the oscilloscope was used for debugging real time signals. In Fig. 2.17 the prototype board for the ExaBot's sensor subsystem can be seen.

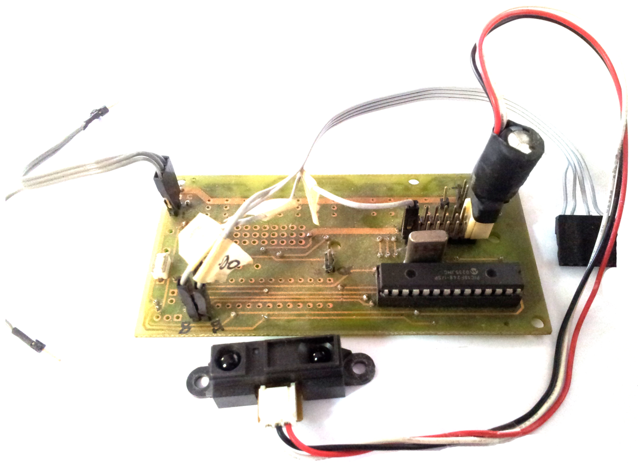


Figure 2.17: Prototype board for the Sensor control with a GP2D120 range finder connected. The cables for SPI communication can be seen in the right, the ones for Vcc and Gnd in the left.

2.3.6 Subsystem Integration

In the following subsections, the integration of the previous subsystems and the development and testing of the final embedded system is described following the methodology of subsection 2.2.4. First the three service subsystems are described: Communication in subsection 2.3.7, Programming in subsection 2.3.8 and Power in subsection 2.3.9. Finally, subsection 2.3.10 describes the final design and test of the PCB.

2.3.7 Subsystem Integration: Communication

In this subsection, the communication in the two main processing configurations is described, i.e., with or without the embedded PC104.

Hardware

The SPI bus has 4 wires: SS (Slave Select), MOSI (Master Out Slave In), MISO (Master In Slave Out) and SCLK (Slave Clock). Fig. 2.18 shows the typical connections between one master and three slaves. The master always starts the communication, selecting the slave it wants to talk to by driving the appropriate SS pin low and generating the communication clock in the SCLK pin. The SPI bus is full-duplex: the SPI master gets one bit from the slave at the same time it transmits one. Moreover, this is the only way a master can get information from the slave, by transmitting some data.

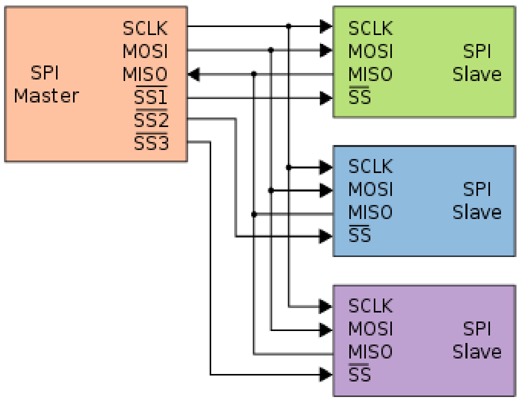


Figure 2.18: Typical configuration of SPI bus for one master and three slaves

Configuration with the PC104

The configuration shown in Fig. 2.18 is exactly how the PC104 (master) is connected to the three PICs (slaves). At first glance, nothing more than the appropriate wires between the pins are needed from the hardware point of view. However, there are some details. One problem is that the PC104 uses 3.3 V in the SPI pins, while the PIC uses 5V. This is not a big problem with the communication from the PC104 to the PIC- since the logic 1 in almost all PIC pins is from 2.5V up, and hence 3.3V is read as a logic 1[5]. However, it can be a serious problem in the communication from the PICs to the PC104, since its pins are not prepared for higher than 3.3V voltages. Hence, a CD4050B [54] IC was included to convert the 5V of the PIC MISO lines into 3.3V.

[5]A "cute" detail is that just so happens that the SDI (or MOSI) pin in the PIC18F2431 is one of the very few pins in which this is NOT true, so a pull-up had to be included to take the voltage slightly up for the communication to work. This was discovered after the production of the PCBs, so an adhoc pull up had to be added to the final board.

Another problem is that the SS pin of the PIC18F4680 is also the analog pin AN4. The issue is that the analog pins cannot be selected independently of the rest. For example, in order to use AN5, AN1-4 must be set as analog pins. Hence, overriding AN4 to be used as SS pin would mean that only AN1-3 can be used. This is not enough analog pins to control the 8 range finders and 2 line-following. To solve the problem, we used the SS pin as AN4, and connected the slave select wire to the external interrupt pin (INT0). An extra module and interrupt had to be configured and programmed, and hence the slave SPI code in the PIC18F4680 is different from the one in the other PICs.

Finally, the reset line of all three PICs are driven by the PC104, so that the PICs are taken out of reset after the PC104 is ready to start communication.

Configuration without the PC104

As already stated, it is desirable that the PC104 may be replaced by some other processing unit or taken away completely. This poses the need to put in all the electronics so that one of the three PICs can be the master of the SPI communication. Moreover, this PIC needs to communicate to an external processing unit by a more common communication protocol, such as USART. In the PIC18F2431, the SPI pins overlap with the USART pins, so both cannot be implemented at the same time. Hence, the PIC18F4680 takes up the master spot in this configuration, implementing both the SPI master communication and the USART communication.

Several jumpers need to be included to reconnect the MOSI/MISO, Slave Select and Reset lines. When the PIC18F4680 is a slave, its communication output line is connected to the input line of the master. When it is the master, it's output line needs to be connected to the input line of the other PICs. The slave select and reset lines suffer a similar fate, since they need to be driven by the PIC18F4680 instead of the PC104. For the RS232 communication, although the PIC has a module to produce the appropriate USART protocol, a MAX232 [55] driver was included to translate between the -11,+11V RS232 voltage range and the PIC's 0-5V voltage range.

Software

The developed communication protocol follows the Open System Interconnection (OSI) model, implementing some of its seven layers. In the master of the communication, Physical, Data Link, Network and Application layers are implemented. Since the slaves never start the communication, the network layer is not needed and hence not implemented in the slaves.

Configuration with the PC104

Code in the SPI Master (PC104)

Physical layer: The SPI physical module in the PC104 implements two independent receive and transmit FIFOs that can be configured to hold up to eight words of 1 or 2 bytes each. It can be configured to work with interruptions or polling. The Slave Selects are managed separately through the Digital Input Output pins. Three DIO pins are connected to the three PIC slaves.

Link layer: The link layer packet sizes have an upper limit that is given by the SPI physical buffers, which can hold up to eight words of 1 or 2 bytes. Since the SPI physical modules in the PICs can hold up to one byte, the SPI words were set to 1 byte long. The packet size is of course reconfigurable. In the current code it is set to four bytes.

Network layer: The network protocol is implemented in the PC104 and manages the reception and sending of packets to the three slaves. For each slave it stores two buffers: one for received packets and another one for packets that need to be sent. The dispatching routine implements a round-robin algorithm that runs every 10 ms, sending and receiving packets from each slave at a time. Figure 2.19 shows the interactions between network, link and physical layers to implement this dispatching routine. The net layer exports read_packet() and send_packet() functions that actually copy packets from/to the corresponding reception and sending buffers. The network layer packet formats can be seen in Table 2.3.

Application layer: A library is implemented to abstract the SPI and network communication details from a user application. For one, the library implements the decoding of sensor data and encoding of commands, following the network layer packet format (see Table 2.3), and using the read_packet() / send_packet() functions exported by the network layer. Since to receive data from a slave the SPI master needs to send data, the library thread regularly sends a packet to every slave -containing commands or dummy data- to get an answer (e.g., to get the latest sensor readings). This application library, called libexabot, exports the following functions: exa_initialize(), exa_deinitialize(), exa_set_motors(), exa_set_sensors(), exa_reset_odometry(), exa_receive(), exa_IR_distances(), exa_sonar_distance(). The last two functions implement the transformation of raw range finder and sonar measures to distance in cm as explained in subsection 2.3.5.

External communication In this configuration, communication with external PCs is done by Wi-Fi or cabled Ethernet. In both cases, the User Datagram Protocol (UDP) was used. The preference of UDP over TCP for robot control from an external PC comes from the fact TCP is connection-oriented and hence provides reliable ordered delivery of the stream of packets (even if that means retransmitting and buffering packets). In contrast, UDP does not guarantee every packet is received, but sends packets as they are received and does not buffer or retransmit any information. As already discussed, in robot control the timing of the commands and the return information is vi-

From	To	Information	packet format
PC104	Sensor PIC	turn sensor on	[0x01, sensor, U,U]
		turn sensor off	[0x02, sensor, U,U]
Sensor PIC	PC104	Sensor Info	[sensor, val1, val2, val3]
PC104	Motor PIC	set velocity	[0x10, vel, U,U]
Motor PIC	PC104	odometry	[0xf0, delta,countH, countL]

Table 2.3: Network layer packet formats. This can be extended if new sensors are added. sensor: 0x00 : three left range finders, 0x01: three middle range finders, 0x02, three right range finders, 0x03:sonar, 0x04: line-following, 0x05: bumper

tal. Receiving all commands together and late as would happen with a TCP connection is far worse than eventually loosing some command because the UDP connection does not guarantee packet delivery. If the robot receives ten commands together, it will execute them all but in a late moment when the environment conditions may have already changed, making those commands useless or harmful. The UDP protocol is implemented using sockets, shared memory for the receiving and sending buffers and two threads: one for receiving and a main thread that sends data back if required.

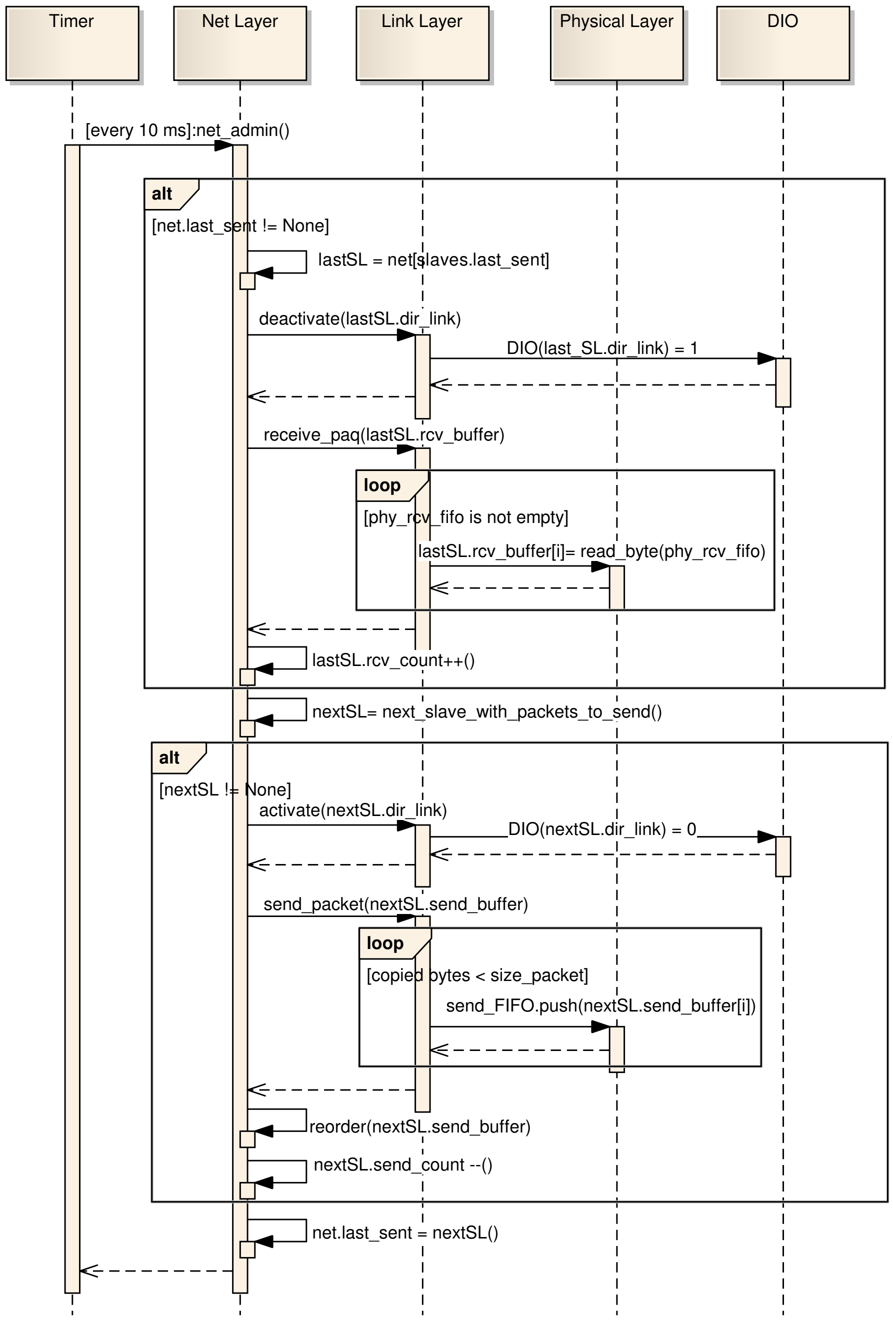
In total, three threads are running in the PC104 to complete communication: one for the complete OSI stack of SPI, one for the UDP receive, and one for the UDP send.

Code in the SPI Slaves

In the PICs, only the physical, link and application layers are implemented. In this case, the Physical SPI module has only a one-byte buffer for both incoming and outcoming data, an hence lifts an interrupt when each byte is received.

Figure 2.20 shows a state machine diagram with typical interactions between the different layers to implement the slave protocol. The state diagram shows the different states in the Link layer, including three of its internal variables (receive and send buffers, and next byte to send). The transitions are given by the Physical layer interruptions when a byte is received `PHY_SPI_ISR`; and the calls from the Application layer to the two exported Link layer functions: `SPI_send_packet` and `SPI_receive_packet`. In any state, the application layer can call these link layer functions: when these transitions are not explicitly drawn, the functions return false, and no transition is performed (i.e., no packet is copied from/to the link buffers).

In this state diagram, the reception starts when there is a packet to send. If there is no packet to send, then only dummy bytes are sent (depicted as `D` in the diagram). Also, if a reception starts before a previous packet has been read by the application, then the old packet is lost. That is why after receiving


Timer
Net Layer
Link Layer
Physical Layer
DIO
[every 10 ms]:net_admin()
alt
[net.last_sent != None]
lastSL = net[slaves.last_sent]
deactivate(lastSL.dir_link)
DIO(last_SL.dir_link) = 1
receive_paq(lastSL.rcv_buffer)
loop
[phy_rcv_fifo is not empty]
lastSL.rcv_buffer[i]= read_byte(phy_rcv_fifo)
lastSL.rcv_count++()
nextSL= next_slave_with_packets_to_send()
alt
[nextSL != None]
activate(nextSL.dir_link)
DIO(nextSL.dir_link) = 0
send_packet(nextSL.send_buffer)
loop
[copied bytes < size_packet]
send_FIFO.push(nextSL.send_buffer[i])
reorder(nextSL.send_buffer)
nextSL.send_count --()
net.last_sent = nextSL()

Net Layer structures:

```
struct Slave {
   uchar send_buffer[MAX_PACKS*SIZE_PACK];
   uchar send_count; //of packets
   uchar rcv_buffer[MAX_PACKS*SIZE_PACK];
   uchar rcv_count; // of packets
   unsigned char dir_link; // DIO for the SPI Slave Sel.
};
```

```
struct Net {
    slave slaves[SLAVES_QTY];
    uchar lastSent;
};
```

Figure 2.19: Sequence diagram showing the interactions between the different OSI layers to implement the network protocol dispatching routine.

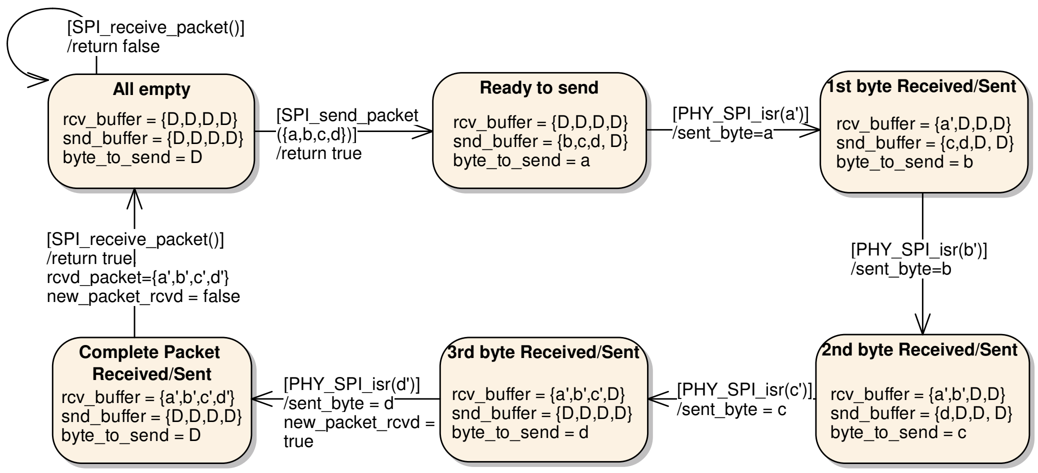


Figure 2.20: State Diagram showing a typical interaction of Phisycal, Link and Application layers to send and receive a packet

the fourth byte of a packet, the link layer asserts the `new_packet_rcvd` flag. This flag performs the synchronization with the application layer: it is checked in the main loop of each PIC. When it is asserted, the `SPI_receive_packet` is called; and a new packet of data is sent using the `SPI_send_packet`.

There is one type of packet for the motor PICs, and six for the sensor PIC (see Fig. 2.3). The sensor PIC packets are sent back in turns; and only the packets corresponding to the turned on sensors are sent back. Hence, if only the sonar is turned on, then all the packets from the sensor PIC include sonar values. Moreover, the last sensor data is sent.

Configuration without the PC104

When the PC104 is absent, the code for the SPI master is implemented in the Sensor control PIC. This implementation follows the same link, application and network layer algorithms (i.e., the packet is four bytes long, the commands are the same, and the round-robin network dispatching routine is the same). The changes reside in the physical layer- since the SPI physical module is different in the PIC and in the PC104-, and in the way it is programmed In the physical layer, the main difference is that the PIC has only one byte of physical buffer (for both reception and sending), and the PC104 has two separate reception and sending FIFO buffers. Since the link and network algorithms in the master are unchanged, the implementation in the slaves (i.e, the other two PICs) is also unchanged.

External communication In this configuration, communication with external PCs is done by cabled RS232. The Sensor PIC receives commands to turn on/off sensors and change motor speed, and redistributes those commands using the SPI network. Regularly, it forms a packet with all the data from the motors and sensors, and sends that packet through the RS232 connection.

Prototyping

To prototype the communication subsystem, the prototype boards for the sensor and motor subsystem and the PC104 were connected. In order to test with the PC104, a small perfboard with a DIP CD4050B and two connectors was created to solve the 3.3V to 5V needed conversion. As in the other subsystems, the PIC programming was done using C language, Microchip's C18 compiler and MPLAB environment, and the oscilloscope was used for debugging real-time signals. The PC104 was programmed in C using the available cross-compilers. In Fig. 2.21 the connected prototype boards for the ExaBot's communication subsystem can be seen.

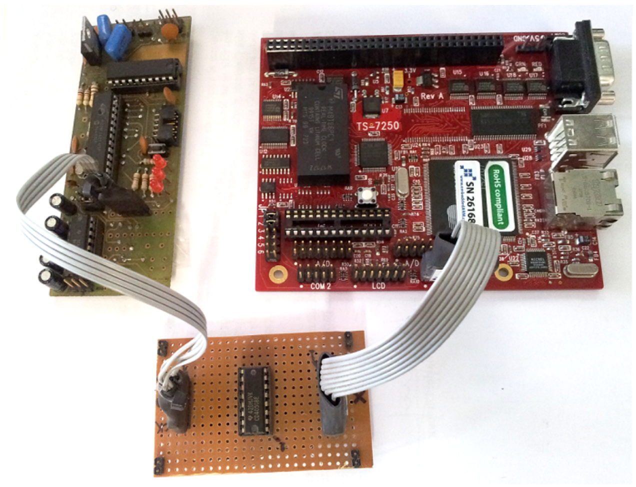


Figure 2.21: Prototype boards for communication subsystem. In this configuration, the prototype board of the Motor Control is connected to the PC104 through the perfboard with the CD4050B IC.

2.3.8 Subsystem Integration: Programming

The programming pins of each PIC are exported in a RJ45 connector. In this manner, each of the three pics can be easily programmed in the final board. There are not many things to note about this subsystem. Since the Microchip programmer needs to manage the PIC resets when programming, a switch was included for each PIC reset line to switch the control between the programmer and the SPI master. There is no special software for this subsystem. The existence of this subsystem is fundamental for the multiple code changes in the ExaBot, due to its reconfigurable characteristics.

2.3.9 Subsystem Integration: Power

Once every sensor, actuator, IC and analog logic is defined, the exact requirements for the battery and the power subsystem can be calculated. The first conclusion from the previous sections is that there is need for two different regulated voltages: 7,2V for the motors and 5V for the ICs and sensors. Hence, there are two separate batteries and power stages. The batteries used are rechargeable Lithium-Ion cells[6]. Each battery cell provides between 3,6V and 4,2V, and its cappacity is 1,9 AmpH.

Motor power subsystem

Each motor needs regulated 7,2V and sinks a maximum of 2 Amp. To obtain the regulated voltage, a minimum of 3 cells in series is needed. This also covers the required current. To regulate to 7,2V for this kind of current, a LM350[56] regulator IC was used. This is a 3.0 Amp adjustable output (between 1.2V and 33V) positive voltage regulator. Each motor has its own regulator, and the adjustment to 7.2V is achieved with a variable resistance in the regulator circuit. Moreover, each motor has its own 3 Amp fuse in the event that the motor consumes more than the specified current.

Logic and sensors power subsystem

The logic and the sensors require regulated 5V. Since we are using a linear voltage regulator, this means that the battery needs at least 2 cells in series. The amount of cells in parallel depend on the consumption of all the components. In table 2.4, typical and maximum consumption for all ICs (including the PC104) and sensors can be found. Initially, we built batteries with only two cells (in series), since the calculated consumption was below the 1,9 AmpH cell capacity and this meant that the battery would last more than an hour. However, lately we are building batteries with 4 cells (two in parallel and two in series), to get extra autonomy for longer experiments. A LM323 voltage regulator [57] was included. This is a 3 Amp, 5V positive voltage regulator, that is enough for the calculated consumption. A fuse was also included.

[6]Recently Lithium-ion polymer batteries have also been used.

Component	typical	max	total typ.	total max.
PC104	450	450	450	450
Sensor Subsys				
8 GP2D120	33	50	264	400
1 SRF05	4	15	4	15
1 PIC4680	23	40	23	40
1 MAX232	8	10	8	10
Total	-	-	*316*	*465*
Motor Subsys				
2 L298	50	70	100	140
2 ACS712	10	13	20	26
2 PIC2431	23	40	46	80
Total	-	-	*166*	*246*
Power Subsys				
2 LM350	3.5	10	7	20
1 LM323	12	20	12	20
Total	-	-	*19*	*40*
Estimated Total			***951***	***1201***

Table 2.4: Typical and Maximum consumption rates of the main(i.e., most consuming) ICs and sensors to estimate the 5V battery requirements. All measures in mA. PICs running at 40 Mhz.

2.3.10 Subsystem Integration: Design and Test of the final PCB

Final Schematic Design

The final schematic was captured using Novram's Diptrace, integrating the schematics for each of the subsystems and making the final changes.

An important feature in the final design is the inclusion of sensor expansion ports. Although all the pins in the PIC18F4680 are used up, the two PICs that control the motors have several pins that are not used. In the final circuit, the pins needed for motor control were planned in those PICs in order to maximize the possible applications of exported pins. In this way, the ExaBot has two expansion ports, each one exporting the following:

- one analog pin (to connect any analog sensor like infrared range finders, light sensors, etc)
- one CCP pin (Capture/Compare/PWM module like the one used to control the sonar).
- a PWM pair (Pulse Width Modulation module, like the ones used to

control the motors).

- the INT0 pin (external interruption pin).
- GND and Vcc.

These expansion ports may prove a very powerful tool to add sensors and functionality to the ExaBot. The programming needed to control these added sensors can be done with the exported programming ports for each PIC18F2431

The final schematic for the complete board can be found in appendix A. This schematic has six sheets. The motor subsystem can be seen in sheets `motor_1`, `motor_2` and `motor_flt`; the sensor subsystem can be seen in sheet `sensores`, the communication subsystem in sheets `con_pc104` and `sensores`, the power subsystem in sheet `power`, and the programming subsystem in sheet `con_pc104`.

Final PCB routing

The complete schematic is exported to the PCB routing program and the final circuit is routed. The correct footprint for each final IC has to be used. Although there are several footprint libraries, many ICs do not have available footprint so these have to be hand drawn respecting the exact package dimensions. These can be found in the corresponding data sheet.

The space restrictions, the sensors and actuators spatial configuration need to be taken into account. Also, all the routing rules need to be followed. Some of these are: width of the path depending on the current, minimum distance between paths, or maximum angle in the paths. Other rules depend on the function of the signals, for example, all the paths that are in a bus need to have roughly the same length so that the signals arrive approximately at the same time (e.g., the clock and data paths in the SPI bus). Also heat dissipation needs to be thought, for example with ground coating over unused parts of the board or with extra dissipation for over heating ICs (such as the voltage regulators). For details on this type of rules, see chapter 12 of [33]. The final routed board for the ExaBot has two layers and can be seen in appendix A.

Board production and testing

Before sending the board to production, it was peer reviewed and several bugs were found. After production, all the ICs, resistances, transistors and capacitors were soldered. Then, special code was programmed to electrically test each pin of the board. In that phase, several bugs were found in the final circuit, luckily these bugs could be fixed and none required an additional iteration of the board. However, these bugs form a list of known issues, that need to be fixed if a new iteration of boards is produced.

Software

The complete software has already been described in the subsystem refinement sections. At this point the complete software can be tested in the final board. For this, the code for each subsystem is tested separately, and in each subsystem every module is programmed and tested separately. Then, every module is integrated one by one testing that the previous capabilities are still valid. To program the code, Microchip's MPLAB environment and C18 compiler was used. For debug, apart from the PIC debugging capabilities, the Textronik oscilloscope and tester were also used.

2.3.11 Final Mounting

In the final mounting stage, the chassis was modified to mount all sensors, the final board and batteries. Moreover, all the cables were done. In Fig. 2.22 three configurations of the final ExaBot can be seen.

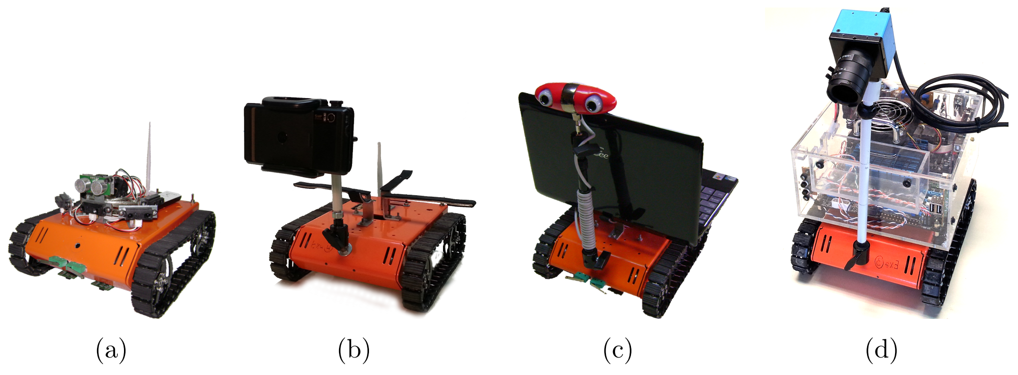


Figure 2.22: Four different configurations of the ExaBot: **a)** with all the sensors and PC104 **b)** with a smart phone, **c)** with a netbook and a 3D Minoru Camera, **d)** with an embedded Mini-ITX board and a FireWire Camera

2.4 Applications

Different configurations of the ExaBot were used in several applications fulfilling all the goals the ExaBot was designed for. In this section we briefly comment some of them.

2.4.1 Research: Autonomous visual navigation

The main research activities with the ExaBot are related to autonomous visual navigation, both with monocular or 3D cameras.

An important work presents a new real-time image based monocular path detection method. It does not require camera calibration and works on semi-structured outdoor paths. The core of the method is based on segmenting images and classifying each super-pixel to infer a contour of navigable space. To achieve real-time computation necessary for on-board execution in mobile robots, the image segmentation is implemented on a low-power embedded GPU. For these experiments, the ExaBot was configured to use an embedded Mini-ITX board (AT5ION-T Deluxe) -that includes a 16 core NVIDIA GPU- as a main processing unit, and a firewire camera (model 21F04, from *Imaging Source*) as the only exteroperceptive sensor (see Fig. 2.22, image **d**). The PC104 was taken away, and the AT5ION-T connects through RS232 to the PIC18F4680 as explained in section 2.3.7. This work was published in the proceedings of the 18th Iberoamerican Congress on Pattern Recognition, CIARP 2012 [58], and in the Journal of Real-Time Image Processing [59].

The same configuration of the ExaBot was used as the platform for all the experiments in a completed PhD Thesis entitled "Vision-based mobile robot system for monocular navigation in indoor/outdoor environments"[60]. The thesis proposes a hybrid method for navigation that combines the aforementioned segmentation-based navigation method to follow paths, and a landmark-based navigation method to traverse open areas.

Another work presents the use of disparity maps for obstacle avoidance using two cameras. In this work, a common notebook or netbook is used as a main processing unit and a cheap Minoru 3D USB webcam as the only exteroperceptive sensor. The notebook connects through cabled Ethernet to the PC104, although the PC104 can be taken away if an appropriate convertor between USB and RS232 is provided for the notebook (see Fig. 2.22, image **c**). This work was published in the VII Jornadas Argentinas de Robótica, JAR 2012 [61]. An extension of this work using elevation maps was presented in the IEEE RAS Summer School on Robot Vision and Applications 2012 [62].

Two undergraduate students are doing their thesis on monocular visual odometry using as a main processing unit- and as the camera sensor- an Android-based smart-phone. This again presents a different configuration of the ExaBot (see Fig.2.22, image **b**). In this case, the Android cellphone connects through wi-fi to the on-board PC104. This is an ongoing work, the thesis will probably be presented during April 2013.

Moreover, an undergraduate student is currently working on integrating a SICK TIM310 laser range finder to perform laser-based navigation. Finally, another student is integrating a compass sensor to perform sensor fusion with the encoders and achieve a better localization estimation than pure encoder-based odometry. These works will be integrated in further autonomous navigation experiments.

2.4.2 Outreach

The ExaBot is also used in Educational Robotics courses, talks and visits. Educational Robotics proposes the use of robots as a teaching resource that allows inexperienced students to approach fields other than specifically robotics. Over the last years, Educational Robotics has grown substantially in elementary and high-school classrooms and also in outreach experiences to interest students in STEM[7] undergraduate programs. A key problem is to have an adequate easy-to-use interface between inexpert public and robots. Hence, one of the developments of the lab is a new behavior-based application for programming robots, specially the ExaBot. Evaluation data on the application show that over 90% of students find it easy to use. In Fig. 2.23 two snapshots of this programming interface can be found.

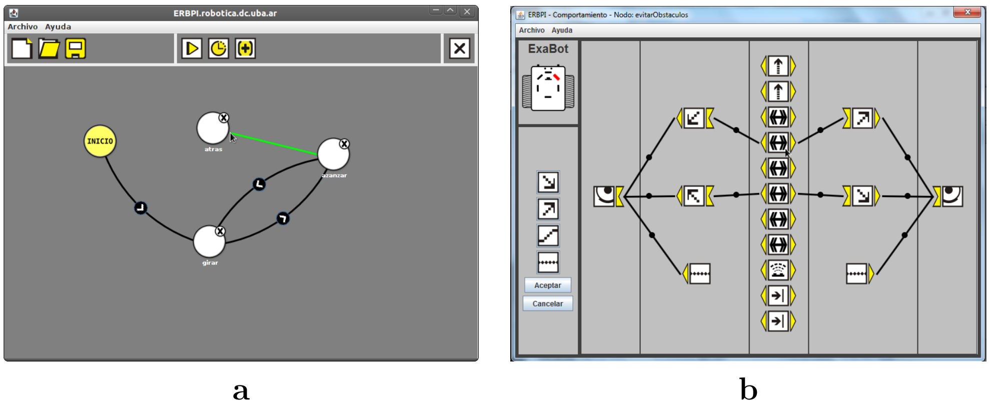


Figure 2.23: Easy Robot Behavior-Based Programming Interface: **a)** Main view. The upper panel shows the main control functions: *left:* new, open, save. *center:* play, new timer, new counter.*right:* close, **b)** A screen shot of the Braitenberg view. The left panel shows the robot schema and the transfer functions that can be used (top-down: inhibitory, excitatory, broken, constant). In the center of the work canvas all the sensors are shown (top-down: 2 line-followings, 6 infrared telemeters, 1 sonar, 2 bumpers). The wheels are shown on each side of the work canvas. The programmed behavior is a simple explorer that moves around at constant speed and can avoid obstacles

Robotic-centered courses and other outreach activities were designed and carried on. In the last years, three eight-week courses, five two-days courses, more than ten one-day workshops and talks were taught to different high school students with the programming interface and several ExaBots. Also, the lab participated in several big expositions such as ExpoUBA or INNOVAR. These

[7]Science, Technology, Engineering and Mathematics

activities are part of a comprehensive outreach program conducted by the Exact and Natural Science Faculty of University of Buenos Aires, Argentina (FCEN-UBA). Statistical data show that since 2009 over 35% of new students at the FCEN-UBA had participated in some outreach activity, showing a significant impact of these activities in student enrollment at STEM-related careers.

For this work, the ExaBot was configured with the PC104 as the main processing unit and all the exteroperceptive sensors described in section 2.3.2: IR telemeters, sonar, bumpers and line-following (see Fig. 2.22, image **a**) These works resulted in one graduate thesis, and two papers: one in the 4th International Conference on Research and Education in Robotics [63] and a more complete one in the IEEE Transactions on Education Journal [64].

2.4.3 Undergraduate Education

The ExaBot is also used in undergraduate and graduate courses of the Departamento de Computación, FCEN-UBA. In particular, it is used in the Robotics Vision course[8]. The course covers several topics on monocular 2D images and two cameras 3D images. Then, several algorithms for robot autonomous navigation are programmed and tested using the ExaBot. This course has been taught in two semesters during 2011 and 2012, and it is currently being taught. The ExaBot was also used in short lessons in other courses such as Organización del Computador I, mainly to show microcontroller programming guidelines.

2.5 Publications

In this section, the publications about the ExaBot and the papers using the ExaBot are summarized.

We have published one journal article describing the ExaBot architecture, and also two short papers in conferences:

- **2010** *A mobile mini robot architecture for research, education and popularization of science*, Sol Pedre, Pablo de Cristóforis, Javier Caccavelli and Andrés Stoliar, Journal of Applied Computer Science Methods, Guest Editors: Jacek Zurada and Pablo Estevez, vol 2, no 1 pp 41-59, ISSN 1689-9636.

- **2009** *ExaBot: a mini robot for research, education and popularization of science*, Pablo De Cristóforis, Sol Pedre, Javier Caccavelli and Andrés

[8]see the course webpage http://www-2.dc.uba.ar/materias/visrob/

Stoliar. Poster in VI Latin American Summer School in Computational Intelligence and Robotics - EVIC2009, Santiago, Chile, December 2009. Third price in poster contest.

- **2008** *Exabot: un robot para divulgación, docencia e investigación*, Pablo De Cristóforis, Sol Pedre y Juan Santos. Short communication in V Jornadas Argentinas de Robótica – JAR08, Bahía Blanca, Argentina, November 2008.

The ExaBot was also used as a robotic platform for experiments in two journal papers, three conference papers and one poster:

- **2013** *Real-time monocular image based path detection*, Pablo de Cristóforis, Matías Nitsche, Tomáš Krajník and Marta Mejail. Journal of Real Time Image Processing, Springer Berlin Heidelberg, ISSN 0018-9162. In press, to appear may-june 2013.

- **2013** *A Behavior-Based approach for educational robotics activities*, Pablo de Cristóforis, Sol Pedre, Matías Nitsche, Thomas Fischer, Facundo Pessacg, Carlos Di Pietro, vol. 56, number 1, pp 61-66, IEEE Transactions on Education, ISSN 0018-9359.

- **2012** *Stereo vision obstacle avoidance using disparity and elevation maps*, Taihú Pire, Pablo de Cristóforis, Matías Nitsche and Julio Jacobo Berlles, IEEE RAS Summer School on Robot Vision and Applications, Santiago, Chile. December 2012. (Poster)

- **2012** *Real-Time On-Board Image Processing Using an Embedded GPU for Monocular Vision-Based Navigation*, Matías Nitsche and Pablo de Cristóforis, 18th Congress of Iberoamerican Pattern Recognition, CIARP 2012, Buenos Aires, Argentina. Lecture Notes in Computer Science, Springer Berlin Heidelberg, vol 7441, pp 591-598.

- **2012** *Evasión de obstáculos en tiempo real usando visión estéreo*, Taihú Piré, VII Jornadas Argentinas de Robótica - JAR12, Olavarría, Argentina, November 2012.

- **2011** *A new programming interface for Educational Robotics*, Javier Caccavelli, Sol Pedre, Pablo de Cristóforis, Andrea Katz and Diego Bendersky, 4th International Conference on Research and Education in Robotics, EUROBOT 2011. Prague, Czech Republic. Communications in Computer and Information Science, Springer Berlin Heidelberg, vol 161, pp 68-77, ISSN 1865-0929.

2.6 Conclusions

In this chapter, we described the co-design of a control embedded system applying the traditional flow in which processors and off-the-shelve ICs are combined: the development of the mini-robot ExaBot. This system has stringent real-time, power consumption and size requirements, providing a challenging case study.

To successfully design and build the robot prototype, the traditional co-design flow was adapted to the autonomous robotics field. The resulting flow has six stages, starting from the robot's goal definition, going through subsystem partition and refinement, all the way to the final mounting. All this stages were applied to develop the robot prototype, and are described in detail in this chapter. The main goal for pursuing this task was to obtain a low-cost robot- i.e., ten times cheaper than commercially available research robots- that could be used not only for research, but also for outreach activities and education. In this sense, neither the commercially available research robots nor the commercially available educational robots were a suitable solution.

The main requirement to achieve a low cost robot that could be used for such diverse fields was that the robot is highly reconfigurable. Hence, the ExaBot was designed with many sensors and built-in sensor expansion ports. Also, the high level processing unit is reconfigurable. So far, the ExaBot was used in four main configurations in our lab: with the PC104 and most sensors for Educational Robotics activities; with a netbook/notebook and the Minoru Camera; with an embedded GPU and different cameras; and with an Android-based smart phone.

Six ExaBot robots are currently in use in the Laboratorio de Robótica y Sistemas Embebidos of the FCEN-UBA. They have been used for educational robotics activities for high school students, research experiments in mobile robotics, and education in graduate and undergraduate university courses. In the Educational Robotics field, robotic-centered courses and other outreach activities were designed and carried on. In the last years, three eight-week courses, five two-days courses, and more than ten one-day workshops and talks were taught to different high school students, using the ERBPI programming interface developed in our lab and several ExaBots. The ExaBot is also used in graduate and undergraduate courses at the Departamento de Computación, FCEN-UBA. Finally, the ExaBot is used for research activities at our lab, mainly in vision-based autonomous navigation. It has been used to make experiments in one finished and two ongoing PhD thesis; and in three finished and two ongoing graduate thesis. In this manner, the ExaBot was the platform used for experiments in two journal articles, three conference articles and one poster.

Chapter 3

Embedded Systems using FPGAs

Real-time hotspot detection

Some embedded systems require massive data processing with real-time constraints that cannot be met with the standard microprocessor and IC approach. Examples include digital signal processing methods such as image, video or audio processing, and their applications to robotics, remote sensing, consumer electronics among many other fields. In these cases, solutions based on Field Programmable Gate Arrays (FPGAs) or the design of particular ASICs (Application Specific Integrated Circuits) are common. These approaches take advantage of the inherent parallelism of many data processing algorithms and allow to create massive parallel solutions. They also allow tailored hardware acceleration, e.g., with particular memory access patterns or bit tailored multipliers/adders. ASICs provide the best solution in terms of performance, unit cost and power consumption. FPGAs are designed to be configured by a designer after manufacturing- hence "field-programmable". The ability to update the functionality after shipping, partial re-configuration of a portion of the design, and the lower non-recurring engineering costs and shorter time-to-market compared to an ASIC design, offer advantages for many applications. According to the 2012 Embedded Market Survey, 35% of the surveyed engineers are currently using FPGAs in their designs [13].

In this chapter we introduce a traditional FPGA design flow and apply it to a remote sensing application that requires massive data processing with real-time constraints [21]. The main contributions in this chapter are:

- The introduction of a traditional FPGA design flow, derived from the analysis of design flows presented in several foundational FPGA books. A short comparison with those design flows is also presented. Also, pros and cons of this type of design flows are discussed, based on bibliographical research and our own experience.

- The proposal of a novel algorithm to segment real-time video from an infrared camera on an UAV (Unmanned Aerial Vehicle) in order to find the location and spatial configuration of hot spots present in each frame. To process each pixel, the algorithm only uses the information of the previous line of pixels, enabling a parallel implementation that can process the image as it is being acquired to meet tight real-time constraints.

- The implementation of this algorithm following the presented traditional FPGA design flow. The FPGA solution takes the analogical output of an IR camera, process the frames in real-time and sends the information of found hotspots in Ethernet packets. The implemented solution successfully segments the image with a total processing delay equal to the acquisition time of one pixel (that is, at video rate). The processing delay is independent of the image size. Sizing equations are presented, and timing, area and power results are discussed.

This chapter is organized as follows. Section 3.1 introduces FPGAs, including their field of application and importance, while section 3.2 presents a description of FPGA basic architecture. Section 3.3 introduces the traditional FPGA design flow. In section 3.4, this flow is applied to obtain a real-time hot spot detection using FPGA for the UAV application. Finally, discussion and conclusions are presented in section 3.5.

3.1 Introduction

The concepts of this section are mostly taken from three important books of FPGA design: Wayne Wolf's "FPGA-Based Design" [6], Clive Maxfield's "The Design Warrior's Guide to FPGAs" [1] and Cofer and Harding's book "Rapid System Prototyping with FPGAs" [2]. Other works are cited as required.

3.1.1 What is an FPGA?

Field programmable gate arrays (FPGAs) are digital integrated circuits (ICs) that contain configurable (programmable) blocks of logic along with configurable interconnects between these blocks. Design engineers can configure (program) such devices to perform a tremendous variety of tasks. The "field programmable" portion of the FPGA's name refers to the fact that its programming takes place "in the field" (as opposed to devices whose internal functionality is hardwired by the manufacturer). This may mean that FPGAs are configured in the laboratory, or it may refer to modifying the function of a device resident in an electronic system that has already been deployed in the outside world.

3.1.2 Why are FPGAs of interest?

Some embedded systems require massive data processing with real-time constraints that cannot be met with the standard microprocessor and IC approach. In these cases, solutions based on Field Programmable Gate Arrays (FPGAs) or the design of ASICs (Application Specific Integrated Circuits) are common.

Nowadays FPGAs contain million of logic gates and can be used to implement extremely large and complex functions that previously could be realized only using ASICs. The cost of an FPGA design is much lower than that of an ASIC (although the ensuing ASIC components are much cheaper in large production runs). At the same time, implementing design changes is much easier in FPGAs, and the time-to-market for such designs is much faster. Thus, FPGAs make a lot of small, innovative design companies viable because- in addition to their use by large system design houses- FPGAs facilitate "Fred-in-the-shed"-type operations. This means they allow individual engineers or small groups of engineers to realize their hardware and software concepts on an FPGA-based test platform without having to incur the enormous nonrecurring engineering (NRE) costs or purchase the expensive toolsets associated with ASIC designs.

These advantages are of particular interest in countries like Argentina. Nowadays, Argentina has no capabilities for manufacturing complete ASICs, and the initial cost for such designs make them prohibitively for small start-ups. FPGAs offer the possibility for national small companies to design and implement products that are adequate for our reality, or that could even replace imported products (given appropriate state policies). These not only include possibilities for design companies, but also for functional and formal verification start-ups that could verify those designs (and others). There are already some companies in Argentina developing with FPGAs, such as INVAP [65]. INVAP [65].

3.1.3 What are FPGAs used for?

When they first arrived on the scene in the mid-1980s, FPGAs were largely used to implement glue logic, medium-complexity state machines, and relatively limited data processing tasks. During the early 1990s, as the size and sophistication of FPGAs started to increase, their big markets at that time were in the telecommunications and networking arenas, both of which involved processing large blocks of data and moving that data around.

Later, toward the end of the 1990s, the use of FPGAs in consumer, automotive, and industrial applications underwent a very big growth. FPGAs were (and still are) often used to prototype ASIC designs or to provide a hardware platform on which to verify the physical implementation of new algorithms. However, their low development cost and short time-to-market made them to increasingly find their way into final products.

By the early-2000s, high-performance FPGAs containing millions of gates became available. Some of these devices feature embedded microprocessor cores, high-speed input/output (I/O) interfaces, and the like. The end result is that today's FPGAs can be used to implement just about anything, including communications devices and software-defined radios; radar, image, and other digital signal processing (DSP) applications; all the way up to system-on-chip (SoC) components that contain both hardware and software elements.

FPGAs are currently growing into three major market segments: ASIC and custom silicon, Digital Signal Processing, and physical layer communication chips. Furthermore, FPGAs have created a new market in their own right: reconfigurable computing.

By 2010, FPGAs comprised a 4 billion dollar market, with leading applications in communications and industry [66]. Fig. 3.1 shows the distribution of FPGA market by end applications [67].

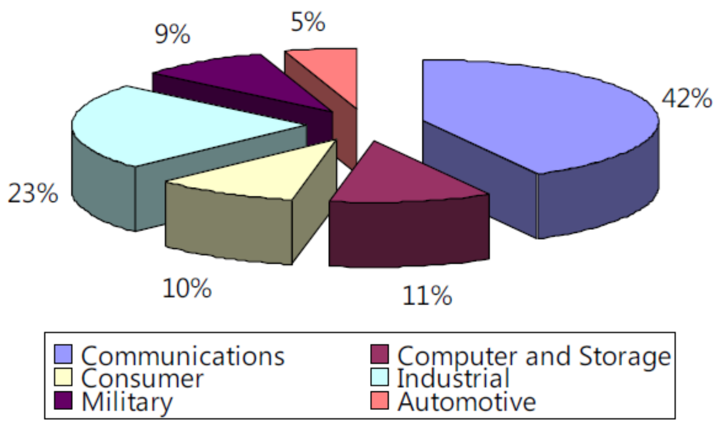


Figure 3.1: FPGA market by end application

Moreover, FPGA vendors offer particular families for different end application. For example, Xilinx offer families and solutions for applications in Aerospace and Defense, ASIC Prototyping, Audio, Automotive, Consumer Electronics, Data Center and several others (See [68]).

3.2 FPGA Basic Architecture

FPGA devices are based on a number of common configurable structures. While there are minor and major variations in the implementation of these structures between manufacturers and device families, the structures are common to almost all mainstream FPGA devices. The fundamental FPGA structures are:

- Logic Blocks
- Routing Matrix & Global Signals
- I/O Blocks
- Clock Resources
- Memory
- Multipliers, Adders, DPS blocks.
- Advanced Features (e.g, hard embedded processors).

In following sections we briefly describe this structures. Most of the concepts of this section are taken from [1] and [2].

3.2.1 Logic Blocks

FPGA logic blocks may have different architectures within different families, even if they are from the same manufacturer. Each manufacturer tends to call the lowest-level FPGA logic block by different names, e.g. logic cell (Xilinx) or logic element (Altera). A logic block will typically contain one or more N-input look-up tables (LUTs) along with one or more flip-flops, signal routing muxes, control signals and carry logic. In the advanced FPGA families, the internal architecture of a logic block is often quite complicated.

Each LUT element can implement any Boolean function with N or fewer inputs. A majority of the implementations of LUT architectures have four inputs; and allow LUTs to be also used as distributed memory or shift registers. Figure 3.2 shows a simplified Xilinx Logic Cell.

In order to support higher levels of functionality, logic blocks may be grouped together by the manufacturer, forming a larger structure. Some example names for these combined logic block groups are: tile, configurable logic block (CLB) and logic array block (LAB). Fig. 3.3 shows a simplified array of Xilinx's CLBs.

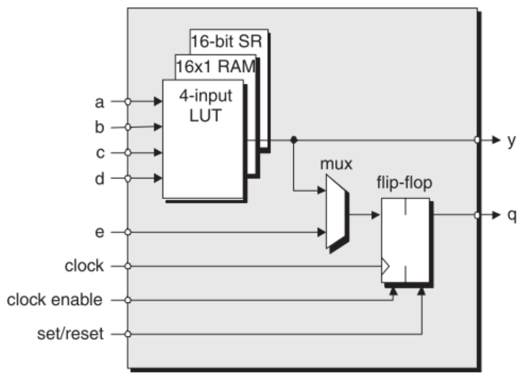


Figure 3.2: Simplified Xilinx Logic Cell (figure taken from [1])

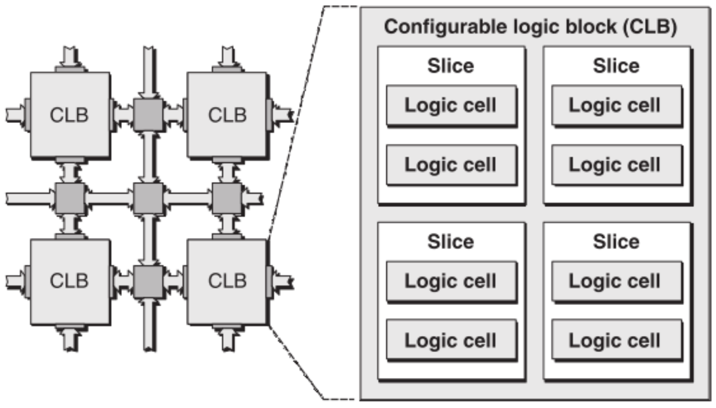


Figure 3.3: Simplified Xilinx CLB, comprising 4 Slices of 2 Logic Cells each. (figure taken from [1])

3.2.2 Routing Matrix & Global Signals

The fundamental routing elements for an FPGA are the horizontal/vertical routing channels and programmable routing switches. The function of the horizontal and vertical routing channels is to provide a connection mechanism between routing switches. The routing switch is programmable and can provide either 180- or 90-degree routing path. Figure 3.4 shows a typical routing matrix.

Another mechanism that FPGA employs for connecting both switches and CLBs is carry chain logic. Carry chain logic is commonly used to build large efficient structures for implementing arithmetic functions within the general logic fabric. Most manufacturers have also implemented global low-skew routing resources. These resources are typically limited in quantity and should be reserved for high-performance and high-load signals. Global routing resources are often used for clock and control signals, which tend to be both high-performance and high-fanout.

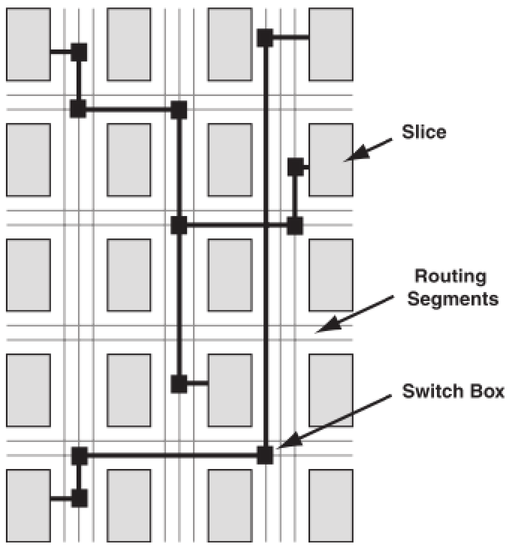


Figure 3.4: FPGA routing signal (figure taken from [2])

3.2.3 I/O Blocks

The ring of I/O banks surrounding the array of CLBs is used to interface the FPGA device to external components. I/O block (IOB) is a common term used to describe an I/O structure. An IOB includes input and output registers, control signals, muxes and clock signals. Fig. 3.5 shows a generic I/O block

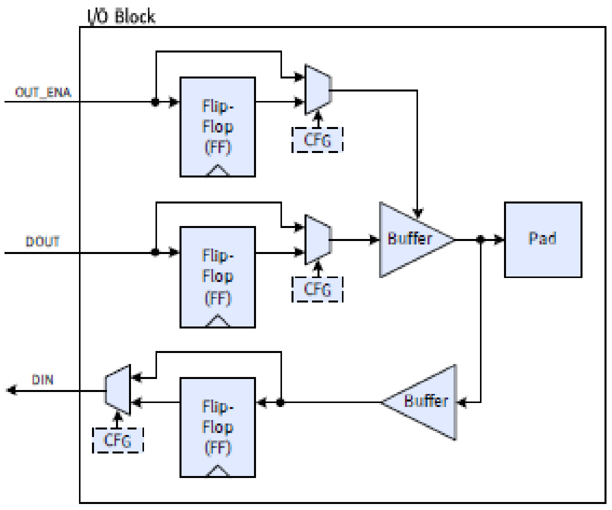


Figure 3.5: I/O block

In order to interface to different types of logic, I/O Blocks are highly configurable. Possible configurations include direction (input, output, bidirectional), data rate (SDR, DDR, SERDES), I/O standard (single-ended, differential, referenced, etc.) and I/O voltage (1.2 V to 3.3 V for single-ended standards).

3.2.4 Clock Resources

Clock Tree

All of the synchronous elements inside an FPGA need to be driven by a clock signal. Such a clock signal typically originates in the outside world, comes into the FPGA via a special clock input pin, and is then routed through the device and connected to the appropriate elements. Fig. 3.6 shows a simple clock tree.

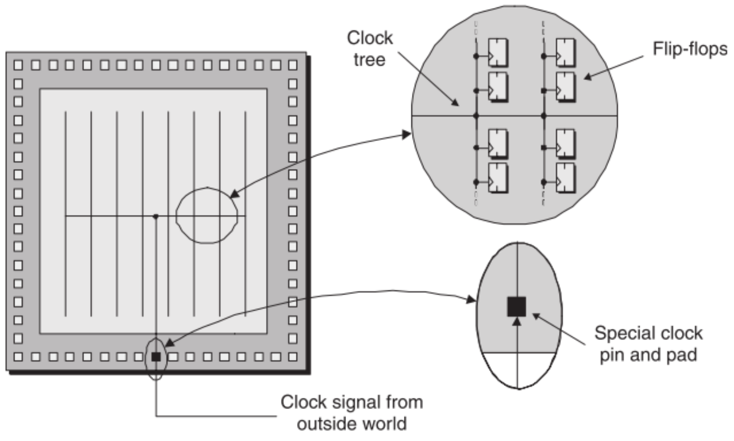


Figure 3.6: A simple clock tree (figure taken from [1]).

The clock tree is implemented using the special low-skew tracks and is separate from the general-purpose programmable interconnect. Multiple clock pins are available (unused clock pins can be employed as general-purpose I/O pins), and there are multiple clock domains (clock trees) inside the device.

Clock Managers

Instead of configuring a clock pin to connect directly into an internal clock tree, that pin can be used to drive a special hard-wired function (block) called a clock manager that generates a number of daughter clocks (Figure 3.7).

Clock Managers are used to generate daughter clocks with frequencies that are derived by multiplying or dividing the original signal; and/or by phase-shifting it. They are also used to correct clock signals, for example by removing jitter or correcting the skew from generated daughter clocks.

3.2.5 Embedded Memory

A lot of applications require the use of memory, so FPGAs include relatively large blocks of embedded RAM called e-RAM or block RAM. Depending on

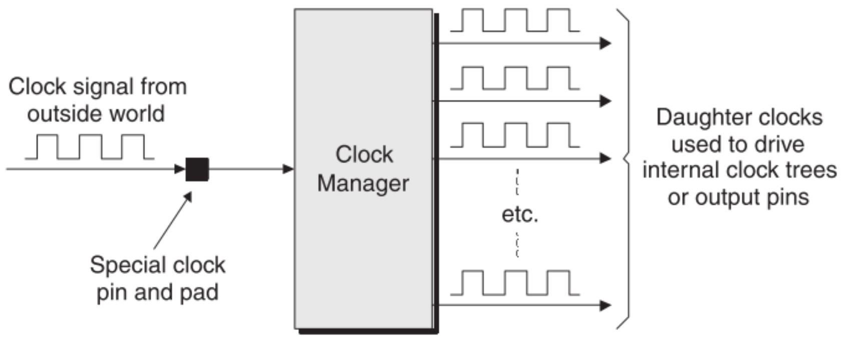

Clock signal from outside world
Special clock pin and pad
Clock Manager
etc.
Daughter clocks used to drive internal clock trees or output pins

Figure 3.7: Clock Manager (figure taken from [1]).

the architecture of the component, these blocks might be positioned around the periphery of the device, scattered across the face of the chip in relative isolation, or organized in columns. Each block of RAM can be used independently, or multiple blocks can be combined together to implement larger blocks. These blocks can be used for a variety of purposes, such as implementing standard single- or dual-port RAMs, first-in first-out (FIFO) functions, state machines, etc. Fig. 3.8 shows the structure of a memory block.

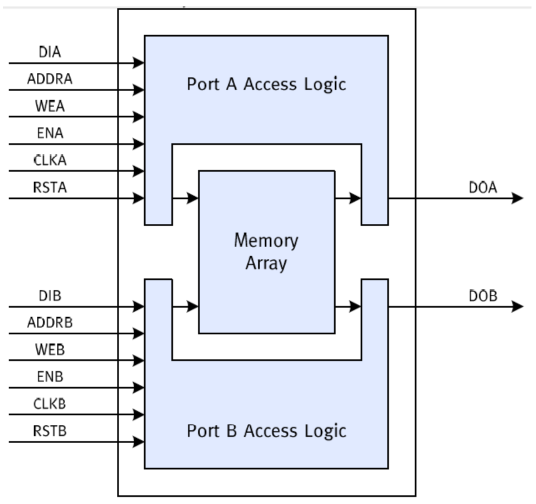


Figure 3.8: Embedded memory block.

3.2.6 Multipliers, Adders, DPS blocks

Some functions, like multipliers, are inherently slow if they are implemented by connecting a large number of programmable logic blocks together. Since these functions are required by a lot of applications, many FPGAs incorporate special hardwired multiplier blocks. These are typically located in close proximity to the embedded RAM blocks because these functions are often used in conjunction with each other.

Similarly, some FPGAs offer dedicated adder blocks. One operation that is very common in DSP-type applications is called a multiply-and-accumulate (MAC). This function multiplies two numbers together and adds the result to a running total stored in an accumulator. Nowadays' FPGA offer DSP blocks that typically implement 18×18 bit multiplier, 48 + 48 bit adder/accumulator, pre-adders for symmetric FIR filters and that can be dynamically configured and highly pipelined (Figure 3.9).

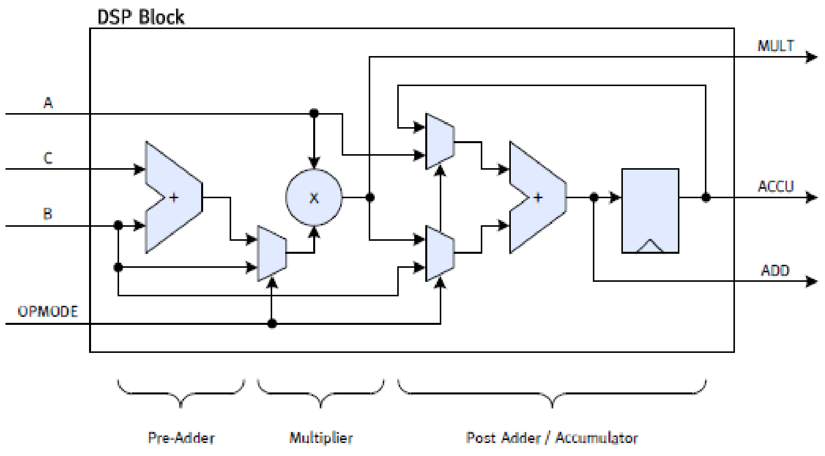


Figure 3.9: Digital Signal Processing blocks

3.2.7 Advanced features

As FPGA devices and architectures continue to evolve, certain advanced structures will be implemented in significantly different ways by different manufacturers. Often these advanced FPGA structures and features are targeted toward very specialized applications and technology specialties. Examples include enhanced clock features, specialized Intellectual Property (IP) cores or advanced I/O standards and protocol support.

Of special interest to this thesis are embedded processors, as referred in next chapter. FPGAs may contain one or more embedded microprocessors. These processors may be hard or soft processors. A hard microprocessor core is implemented as a dedicated, predefined block. As opposed to embedding a microprocessor physically into the fabric of the chip, it is possible to configure a group of programmable logic blocks to act as a microprocessor. These are called soft cores.

3.2.8 The complete picture

Fig. 3.10 shows a generic FPGA architecture including the described structures.

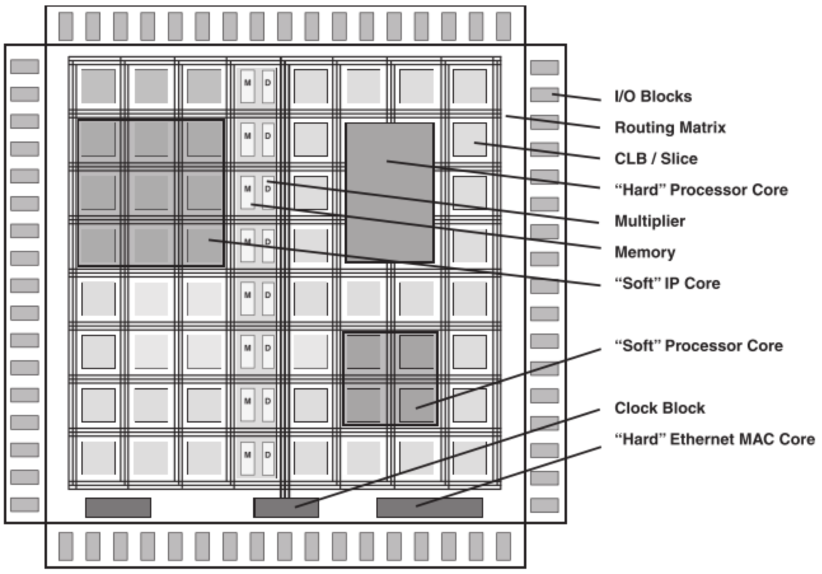


Figure 3.10: Generic FPGA architecture (figure taken from [2])

3.3 Design Flows for FPGA

In this section, a standard FPGA design flow is presented. The concepts of this section are taken from these books: Cofer and Harding's "Rapid System Prototyping with FPGAs" [2], Chu's "FPGA Prototyping by VHDL Examples"[3], Maxfield's "The Design Warrior's Guide to FPGAs" [1], Wolf's "FPGA-Based System Design" [6] and Kilts' "Advanced FPGA Design" [5]. The equivalence of the presented flow with the exact ones showed in those books is presented at the end of this section. It is important to point out that this is a generic FPGA design flow, it may vary according to the particular application area.

The high-level design phases associated with FPGA design include requirements, architecture, implementation, and verification. The following descriptions provide some detail on the tasks which must be completed during the primary design phases specified:

- **Requirements Phase** Define and detail the required functionality, interfaces, performance, and design margin. Develop and maintain a design requirement specification.

- **Architecture Phase** Partition design into functional blocks, allocate functionality, and performance requirements. Define system architecture and design hierarchy. Determine which design components will implement required functionality.
- **Implementation Phase** Code the design using a Hardware Description Language (HDL), and then synthesize and place and route the design to the particular FPGA. Several simulation steps can also be performed in this stage to test the design.
- **Verification Phase** The design file that defines the state of every configurable element within the FPGA is "downloaded" to the part. This process is also referred to as "configuration'. Using external test equipment and access to internal nodes, the design's functionality and real-world performance is verified.

In the next subsections two main phases of this flow are described: the Architecture and Implementation Phases.

3.3.1 Architecture Phase

In this phase, the general architecture of the system is designed. The key concept in this phase is hierarchichal design.

Hierarchical design is a standard method for dealing with complex digital designs. It is commonly used in programming: a method is written not as a huge list of primitive statements but as calls to simpler methods. Each method breaks down the task into smaller operations until each step is refined into a method simple enough to be written directly. This technique is commonly known as **divide-and-conquer**- the method's complexity is conquered by recursively breaking it down to manageable pieces.

In digital circuits, implementing an informed subsystem partition in a hierarchichal manner is an important method to increase a design's ability to absorb change, and decrease development time and complexity. In rapid development, individual design subsystems can be developed concurrently by isolated groups of specialists. This requires the development of individual blocks designed to specific requirements, allowing independent development and verification. Ideally, the modules should be designed so they are highly independent of one another. This can isolate risks associated with functional implementation and allow design modules to be updated with less disruption to the rest of the design implementation. With a modular design approach, the design integration phase can also progress more smoothly if the individual modules' functionality and interfaces have been correctly defined, implemented and tested.

At this stage is important to consider a range of potential design implementation approaches. High-level and detailed FPGA system block diagram and functional level block diagrams, including appropriate interface details, are very useful. It is important to keep in mind that decisions made at this stage might influence requirements, and also that implementation details of the next phase might also change architecture. It is usual that partition is done taking into account possible implementations of the modules down the line.

Critical elements in a modular design are timing and design block interfaces.The selection of design boundaries can have significant effects on placement and routing and overall compilation efficiency. The following is a list of partitioning considerations to keep in mind:

- Group related functional blocks, or blocks that have many signals in common. They have the greatest effect on the place-and-route process.
- Separate portions of the design that have different design goals (performance, area, etc.). This allows the designer to apply appropriate synthesis directives to specific design blocks.
- Where possible, divide groups along boundaries where signals are registered. Avoid assigning a boundary across combinatorial logic, since this can interfere with logic optimization.

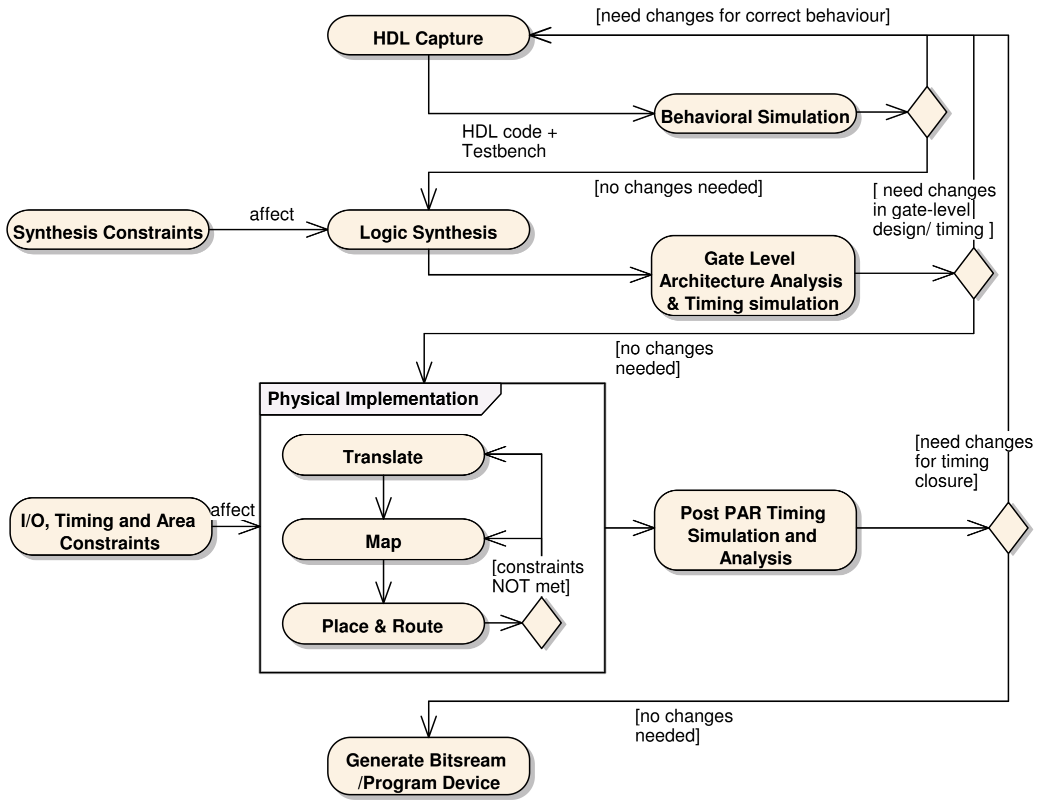
3.3.2 Implementation Phase

In this phase, each module of the architecture design is refined, implemented, synthesized and simulated in an iterative manner until the module is functionally correct and meets all the timing and performance goals. Then, all the modules are integrated and tested. Fig. 3.11 shows the stages in this phase.

HDL Capture

The design is captured using a Hardware Description Language, special languages used to describe digital circuits. The functionality of a digital circuit can be represented at different levels of abstraction (Fig. 3.12).

The lowest level is the switch level, which describes the circuit as a netlist of transistor switches. A slightly higher level of abstraction is the gate level, which describes is as a netlist of primitive logic gates and functions. Both switch-level and gate-level netlists may be classed as **structural** representations.



HDL Capture
[need changes for correct behaviour]
Behavioral Simulation
HDL code + Testbench
[no changes needed]
[need changes in gate-level design/ timing]
Synthesis Constraints
affect
Logic Synthesis
Gate Level Architecture Analysis & Timing simulation
[no changes needed]
Physical Implementation
Translate
Map
Place & Route
[constraints NOT met]
I/O, Timing and Area Constraints
affect
Post PAR Timing Simulation and Analysis
[need changes for timing closure]
[no changes needed]
Generate Bitsream /Program Device

Figure 3.11: Activity diagram of the Implementation phase of FPGA design flow

The next level of HDL abstraction is **functional** representations. This includes the description of functions using Boolean equations and also *Register Transfer Level* (RTL) representations. In RTL descriptions, the clocked behavior of the design is expressly described in terms of data transfers between storage elements in sequential logic (which may be implied) and combinatorial logic (which may represent any computing or arithmetic-logic-unit logic). RTL descriptions are said to be technology-independent (retargetable to different device families), however, *the architecture implied by the description is fixed* [1].

The highest level of abstraction supported by traditional HDLs is known as **behavioral**, that include constructs such as processes or loops. This usually does not imply specific timing. Hence, Behavioral descriptions are usually *architecture-independent*: several RTL circuits can implement the same behavioral description, but with different timing, area and power implications. Synthesis to gates, from a description at this level of abstraction, requires sophisticated tools. To influence the design implementation made by synthesis tools, synthesis constraints can be applied. However, there is only so much a synthesis tool with appropriate constraints can do: to ensure a particular

[1]A detail on this is that some synthesis constraints can change the final implied architecture, e.g., the Register balancing constraint that is used to meet design timing requirements by moving the placement of Boolean logic functionality across register boundaries

synthesized architecture, HDL software changes may be needed.

As a final comment, HDLs have a dual nature: they are not only used for implementation but can also be used for simulation. Hence, there are constructs in HDL which are syntactically correct but are not *synthesizable* into a design that can be placed in a targeted FPGA. The two most commonly used HDLs are Verilog and VHDL.

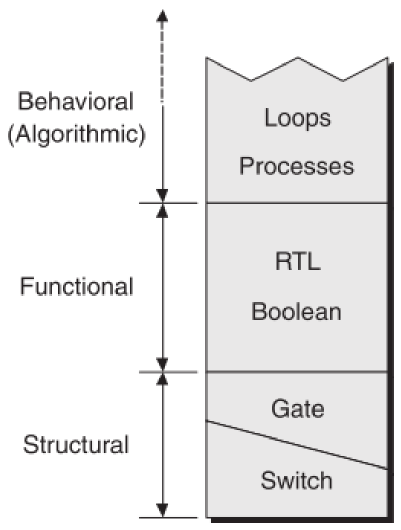


Figure 3.12: VHDL and Verilog levels of abstraction

HDL coding guidelines and patterns

Coding guidelines and patterns in HDL are very important because they impact performance. One reason for this is that, although they might be logically equivalent, different RTL statements can yield different architectures. Also, tools are part of the equation because different tools can also yield different results. Hence, when coding in HDL, two things are important:

- To design the desired underlying hardware architecture for the particular timing, area and power constraints of the module.
- To know which HDL constructs will synthesize that architecture.

This makes the discussion of how to code in HDL intrinsically related to the discussion of how to design the underlying digital circuit to achieve the require timing, area and power performance. It also makes HDL coding patterns, that are known to synthesize particular architectures, very important.

Moreover, the ability of HDL to accurately describe the desired hardware architecture can make HDL code very hard to understand, debug and maintain. To describe the concurrent nature of hardware, HDL code usually includes many low-level concurrent statements that become very difficult to understand when the number of concurrent statements surpasses some small amount. This makes bugs very hard to find. Hence, coding guidelines and patterns are important not only to assure the desired hardware is synthesized, but also to enable understandable code among a variety of designers, a vital point in moderate-to-big designs.

Complete books are written on HDL coding guidelines and patterns, so a detailed description is clearly out of scope of this thesis. However, it is important to point out some ideas to get a picture of the problem the work on this thesis aims to tackle. That is, that *the growing complexity of digital circuits make it very difficult for designers to model the functional intent of the system and implement it using HDL languages and traditional FPGA design flows.*

In following sections, the different types of circuits and very small amount of coding guidelines are presented. This design methodology together with coding guidelines are taken from Pong P. Chu's book "FPGA Prototyping by VHDL Examples" [3][2]. General considerations are taken from [2] and [1].

General Considerations

Inference and Instantiation

As already stated, each RTL construct in HDL code synthesize to a particular hardware structure. In order to design good and optimized circuits, it is very important to know what hardware is *inferred* when writing a particular code. Another way to determine which particular hardware will be put in the resulting digital circuit is to *instantiate* a particular IP core that wraps a physically existing resource in the FPGA (for example, a DSP block or BRAM). Inference is preferred when the objective is design flexibility or design portability; instantiation when seeking to take the greatest advantage of architecture specific structures and architectural features.

Procedural and Structural Coding

Procedural and structural are two general coding styles:

- *Procedural Coding Style*: the "behavior" of a circuit is described in sequential, top-to-bottom code, similar to the code implemented in procedural languages like C. Much procedural-style code is supported for RTL synthesis, and does not require a behavioral synthesis tool, and for that reason this style is usually referred to as behavioral code. In order for tools to understand top-to-bottom statements, they need to be wrapped in VHDL *processes* or Verilog *always* blocks.
- *Structural*: the structure of the circuit is explicitly written, including instantiating components and specifying which signals (nets, wires) are

[2]Chu also has this book in Verilog flavor, see [69]

connected to each pin of the component. It is similar to specifying schematic connections explicitly in text.

It is possible for users to mix structural and procedural coding styles in the same VHDL or Verilog source file. Procedural style code is easier to read, however, structural style code is typically used for instantiating technology-specific library cells (such as I/O pads), memory cells, third-party IP blocks, and lower-levels of the hierarchy from the HDL code. Structural code is also used to explicitly show parallel sections.

Types of circuits

Combinatorial circuits

A combinatorial circuit is a circuit in which the output is a function of the input only. Combinatorial circuits are important because they are the building blocks of higher-level architectures. They include logical operators, arithmetic operators such as adders or multipliers, and relational operators such as comparators. Also, routing circuits made with multiplexers, such as priority or parallel routing. Guidelines for coding this combinatorial circuits can be found in chapter 3 of [3].

Sequential circuits

A sequential circuit is a circuit with *memory*, which forms the internal state of the circuit. Unlike a combinatorial circuit, in which the output is a function of input only, the output of a sequential circuit is a function of the input and the internal state.

Synchronous Design is the most commonly used practice in designing a sequential circuit. In this method, all storage elements are controlled (i.e., synchronized) by a global clock signal and the data is sampled and stored at the rising or falling edge of the clock signal. It allows designers to separate the storage components from the circuit and greatly simplifies the development process. It also simplifies design reuse, and timing simulation, static timing analysis and constraints.

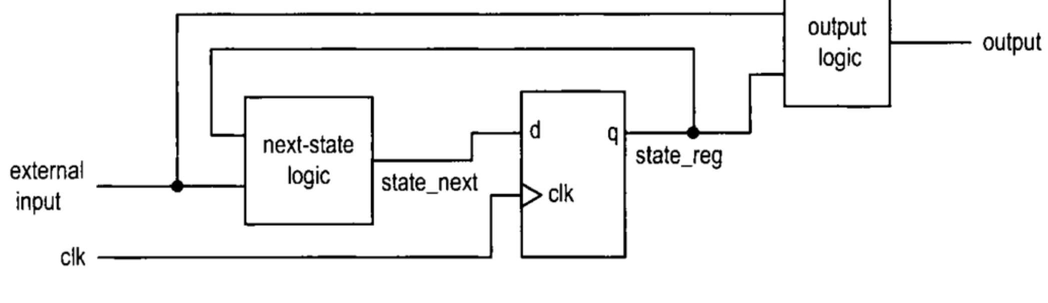


Figure 3.13: Block diagram of a synchronous system (figure taken from [3])

Of course, different clocks may be used. In that case, it may be necessary to synchronize at the clock domain interfaces, which is usually a place for headaches. The interfacing function depends on the application. For data exchange, FIFOs between the clock domains may be used. For signals, a common method is to use two or more successive flip-flops clocked with the frequency of the clock domain the signals are transitioning into.

The block diagram of a synchronous system is shown in Figure 3.13 . It consists of:

- *State register*: a collection of flip-flops controlled by the same clock signal.
- *Next-state logic*: combinatorial logic that uses the external input and internal state (i.e., the output of register) to determine the new value of the state register.
- *Output logic*: combinatorial logic that generates the output signal.

The code development follows the basic block diagram in Figure 3.13. The key is to separate the memory component (i.e., the register) from the system. Once the register is isolated, the remaining portion is a pure combinatorial circuit, and the coding and analysis schemes used for combinatorial circuits can be applied. While this approach may make the code a bit more cumbersome at times, it helps to better visualize the circuit architecture and avoid unintended memory and subtle mistakes.

Based on the characteristics of the next-state logic, sequential circuits can be divided into three categories:

- *Regular sequential circuit.* The state transitions in the circuit exhibit a “regular” pattern, as in a counter or shift register. The next-state logic is constructed primarily by a predesigned, “regular” component, such as an incrementor or shifter.
- *Finite State Machine - FSM.* The system transits among a finite number of internal states. The transitions depend on the current state and external input. Unlike a regular sequential circuit, the state transitions of an FSM do not exhibit a simple, repetitive pattern. Its next-state logic is usually constructed from scratch and is sometimes known as “random” logic.
- *FSMD.* The circuit consists of a regular sequential circuit and an FSM. The two parts are known as a data-path and a control-path, and the complete circuit is known as an FSMD (FSM with data path). This type of circuit is used to implement an algorithm represented by register-transfer (RT) method, which describes system operation by a sequence of data transfers and manipulations among registers.

Sequential Circuits coding guidelines

In all the cases, the *state register* needs to be coded as a process. The process has the clock and reset signals in the activation list, both signals necessary to synthesize memory.

The rest of the coding guidelines depend on the type of sequential circuit.

- Regular sequential circuits: the next-state and output logic can be coded as combinatorial circuits.
- FSMs: the next-state logic is coded as a process that has as activation signals the previous state and the input signals. Inside the process, a case statement defines which is the next state for each current state and inputs- i.e, it codes the FSM. The output logic is also coded as process that defines the output for each state. The activation signals of this output logic process depend on the type of FSM. In Moore type machines, the output depends only on the state and hence the activation signal only includes the state. In the case of Mealy machines, it depends both on the state and the input signals; and hence both are in the process' activation list.
- FSMDs: this can be coded following the block diagram in 3.14. One process codes the state and data path registers, having clock and reset as activation signals. Another processes codes the control path (next state logic + control signals), having as activation signals the state register and input signals. A third process codes the output signals. Finally, the data path can be coded as a combinatorial circuit.

Of course, other coding patterns can be applied to code this circuits. Details con be found in chapter 4,5 and 6 of [3].

Optimizations

To close the HDL Capture section, a few words on optimization techniques. When talking of optimizations in FPGA design, there are several conflicting goals:

- Performance: The logic must run at a required rate. Performance can be measured in several ways, such as throughput and latency (usually conflicting among them). Clock rate is usually used as a measure of performance.
- Power/energy: The chip must often run within an energy or power budget. Energy consumption is clearly critical in battery-powered systems.
- Area: Area occupancy is usually related to the cost of the chip that will host the design.

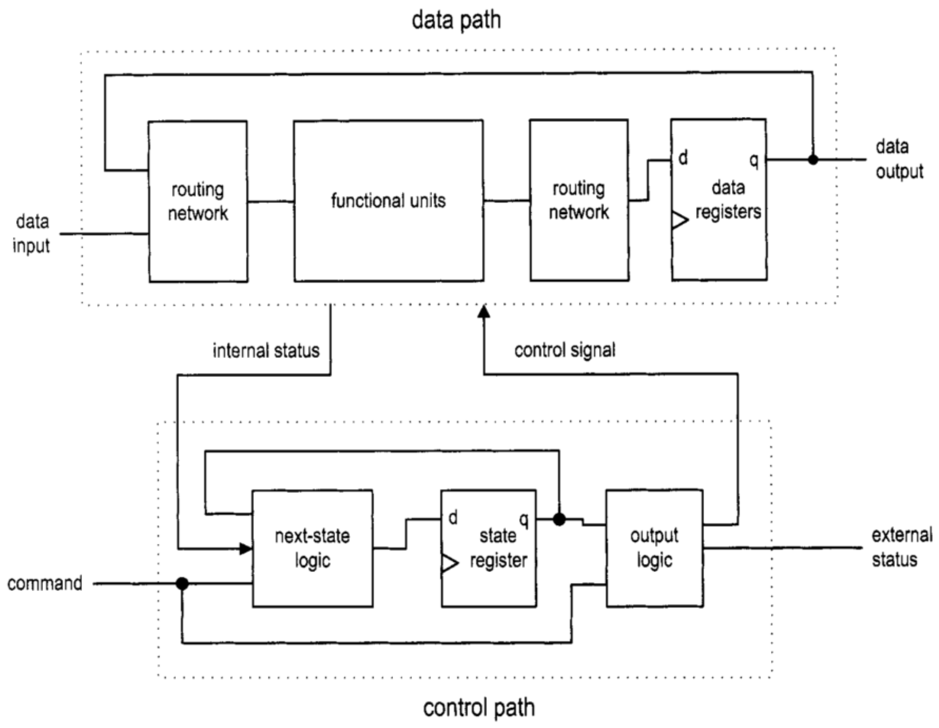


Figure 3.14: Block Diagram of a Data Path - Controller Architecture (figure taken from [3])

Although constraints for synthesis tools are useful to optimize designs in these vectors, in this type of HDL coding, optimizations are mostly given by the general architecture of the system and the designed hardware architecture of each module. There are many techniques to optimize designs in each of the mentioned topics. For example, pipelining is a very usual approach to increase throughput. Resource sharing is very common for reducing area- but it usually decreases throughout. On the contrary, replicating hardware modules can be used to increase throughput and decrease latency, but of course, increasing area. Although almost all FPGA design books include some section for optimizations, an excellent book on detailed optimizations techniques is Steve Kilts' "Advanced FPGA Design. Architecture, Implementation and Optimization" [5].

The above are usual performance goals taking into account only the design itself. However, at least two other goals must be taken into account:

- Design time: It cannot take forever to design and optimize the system. This means that actions need to be taken to reduce design and verification time.
- Design cost: Of course, design time affects costs. Another important cost is tools: in FPGA design there are many very interesting tools that can

help with reducing design and verification time. However, many times this tools are prohibitively expensive[3].

Logic synthesis

Logical synthesis is the process of translating an HDL language design description into a pure RTL design description. The synthesis process occurs as a sequence of stages. The first stage is the parsing of the HDL code for syntax errors. When the code is verified to be syntactically correct, the synthesis tool begins the process of translating the design into an RTL description (registers, Boolean equations, clocks and interconnecting signals). The output of the synthesis process is a netlist file, which is used as an input to the place-and-route tools discussed in following subsection.

Physical implementation

The Physical process consists of three smaller processes: translate, map, and place and route. The translate process merges multiple design files to a single netlist. The map process, which is generally known as technology mapping, maps the generic gates in the netlist to FPGAs logic cells and IOBs. The place and route process, which is generally known as placement and routing, derives the physical layout inside the FPGA chip. It places the cells in physical locations and determines the routes to connect various signals. Static timing analysis, which determines various timing parameters, such as maximal propagation delay and maximal clock frequency, is performed at the end of the implementation process.

Although the process of translating code to gates is a fairly mature technology; choosing the gates, their placement and interconnect routing in order to meet the specified design timing requirements and area goals remains a significant challenge. Hence, physical implementation is an iterative process: different placement and interconnections are tried until the timing constraints are met, while the I/O and area constraints are followed. Of course, this may not happen, and design changes may be needed to achieve the desired timing constraints.

Generate and download the programming file

In this process, a configuration file is generated according to the final netlist. This file is downloaded to an FPGA device serially to configure the logic cells and switches. The physical circuit can be verified accordingly.

[3]This is much worse in ASIC design.

Simulation and Functional Verification

Two primary methods are used for FPGA design validation: simulation and board-level testing. Board-level testing is implemented after the design has been placed and routed and is performed on the target hardware platform. Although board-level testing is an effective design test and debug approach, validating a design in the lab at the board level all at once without significant block-level testing is only practical for small to medium designs with limited complexity. Simulation plays a critical role in the FPGA design verification process, especially for rapid system development efforts.

The primary benefit simulation provides is the ability to begin validation of design functionality at the earliest phases of the project, independent of the availability of a hardware target platform. Simulation can begin before the synthesis process and can continue throughout all the implementation phases of the FPGA design flow until a hardware target platform becomes available.

There are three main stages of simulation. Each of these stages is related to the phases of implementation relative to the synthesis process. The typical terms associated with each simulation stage are behavioral, functional and timing. A short description of each stage is:

- *Behavioral*: Used to validate the behavior of the HDL code. Performed before the synthesis stage. May not be synthesizable to hardware.
- *Functional*: Used to validate that the functionality of the design blocks meet functional design block requirements. Performed after synthesis stage. Timing analysis is based on assumed gate and routing delays since the design has not yet been placed or routed
- *Timing*: Used to validate the functionality, timing and performance of the design. Performed after design place and route. Based on actual back-annotated timing delays and thus more accurate than functional simulation.

For rapid system prototyping applications, the use of behavioral simulation coupled with timing and board level validation is typically sufficient. Although functional simulation is not commonly used, there are cases where it can be beneficial to implement. The main idea in simulation is to have a testbench that stimulates the design, captures the output, and compares it to the expected outputs. HDL languages supports testbench creation to some extent. However, special languages have appeared for full-featured testbench environments, such as SystemVerilog [4], and also C++ environments have been proposed [70].

Fig. 3.15 shows what a full-feature functional verification environment diagram might look like.

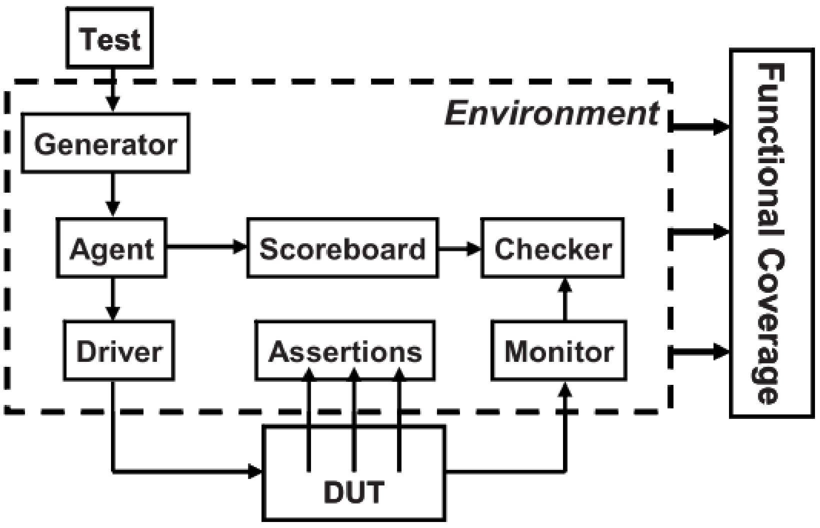


Figure 3.15: Block diagram of full testbench- DUT stands for Design Under Test (figure taken from [4]).

There are many books covering functional verification, including topics such as assertion driven simulation, functional coverage, random stimuli generation or formal techniques. This is a vast area of FPGA-based design. A more detailed discussion of this topics exceeds the aim of this thesis, that is focused in the design methodology more than the functional verification of designs. However, this is an area of much recent development, and a path for future research: the inclusion of functional verification methodologies and frameworks with the proposed co-design methodology. Some important books cover SystemVerilog for Verification [4], Open Verification Methodology [71] or C++ based verification [70].

Constraints

Constraints are used to influence the FPGA design implementation tools including the synthesizer, and place-and-route tools. They allow the design team to specify the design performance requirements and guide the tools toward meeting those requirements. The implementation tools prioritize their actions based on the optimization levels of synthesis, specified timing, assignment of pins, and grouping of logic provided to the tools by the design team. The four primary types of constraints include synthesis, I/O, timing and area/location constraints.

Synthesis constraints influence the details of how the synthesis of HDL code to RTL occurs. There are a range of synthesis constraints and their context, format and use typically vary between different tools.

I/O constraints (also commonly referred to as pin assignment), are used to assign a signal to a specific I/O (pin) or I/O bank. These constraints may also be used to specify the user-configurable I/O characteristics for individual IOB.

Timing constraints are used to specify the timing characteristics of the design. Timing constraints may affect all internal timing interconnections, delays through logic and LUTs and between flip-flops or registers. Timing constraints can be either global or path-specific.

Area constraints are used to map specific circuitry to a range of resources within the FPGA. Location constraints specify the location either relative to another design element or to a specific fixed resource within the FPGA.

3.3.3 FPGA flows comparison

The concepts and ideas of the previous section where taken from several books. These books present design flows that are subsets of the presented design flow with different names for the same phases. In this section, we shortly show the design flow presented in each book and compare it to the one presented in this thesis.

For this, we will show the diagram for the design flow of each book, and mark in red the swimlines (constraints, implementation, simulation) and in blue the stages of the flow we presented in the preceding section.

The Design flow presented in Chu's book is very similar (see Fig. 3.16). It includes constraints, implementation and simulation. It describes in detail only the implementation stage, which is reasonable since the book is focused in coding guidelines and patterns for good design in VHDL. It is also interesting to note that the HDL Capture stage is called RTL code. This is because the design patterns presented in the book are mainly at the RTL level of HDL. Although it uses processes and "high-level" VHDL constructs, in the presented pattern designs, the timing is completely defined: it is an RTL design.

The Design flow presented in Maxfield's book is a simplified HDL based flow (see Fig. 3.17). Since this book is mostly an overview on many aspects of FPGA based design, there are no detailed design flows. Moreover, it is interesting to note that here again, the HDL capture phase is called RTL.

The design flow presented in Kilts' book includes all the stages presented in this chapter (see Fig. 3.18). This book is focused on detailed design, coding and implementation for optimized performance. The chapters on the book are organized following this design flow. Moreover, although the constraints are not explicitly described in the diagram, they are an important topic on the chapters describing Synthesis Optimization and Place and Route optimization.

This design flow presented in Cofer's book is focused in the implementation

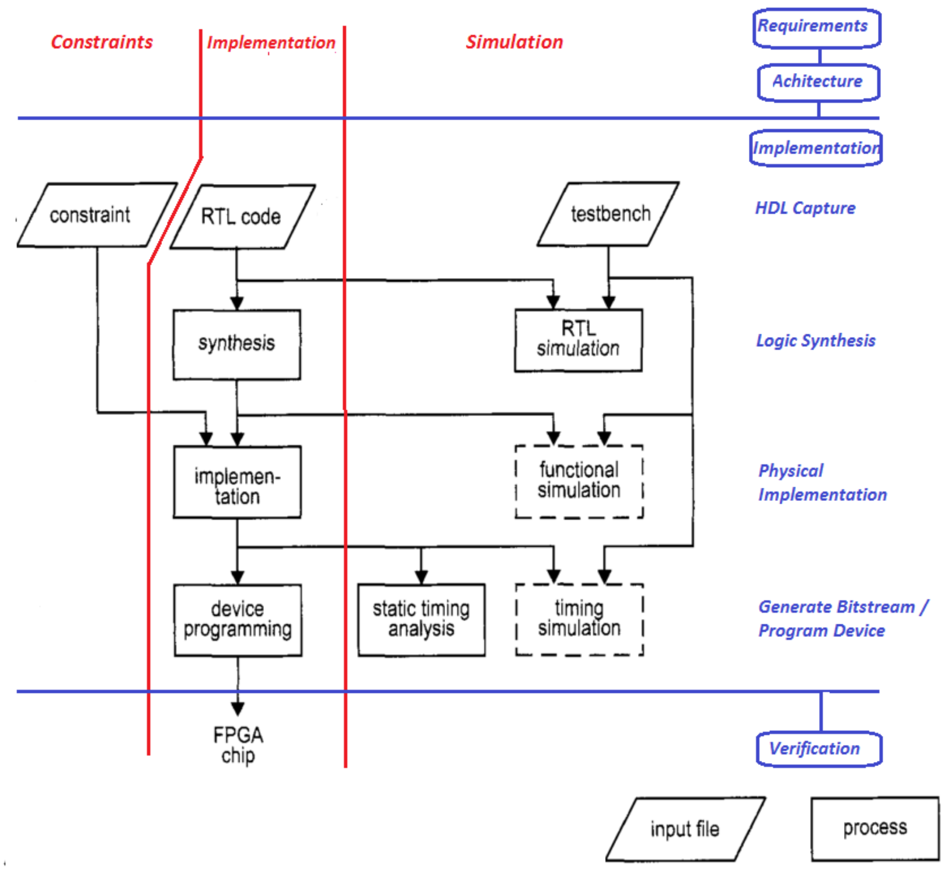


Figure 3.16: Design flow presented in Chu's book [3] (see page 16 of the book)

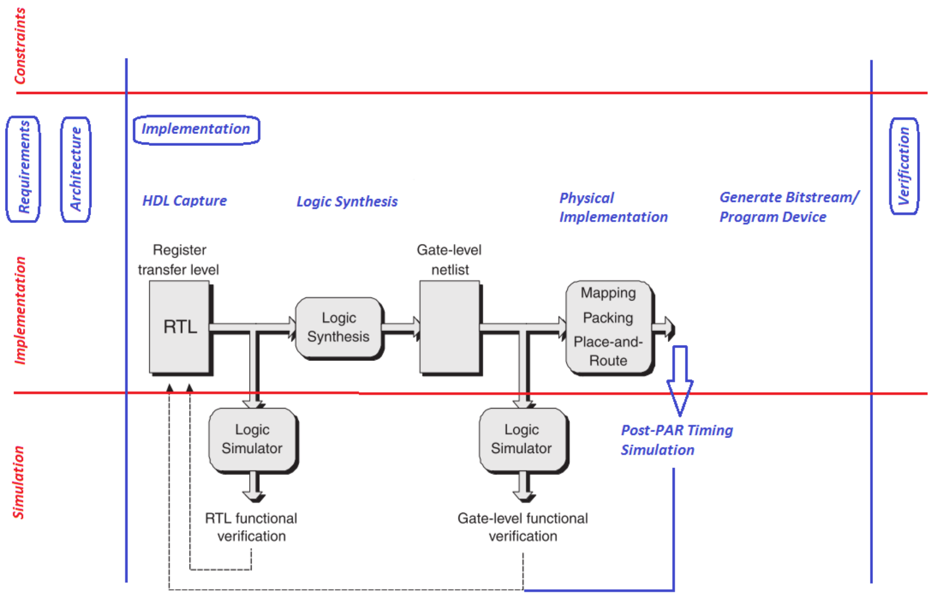


Figure 3.17: Design flow presented in Maxfield's book [1] (see page 159 of the book)

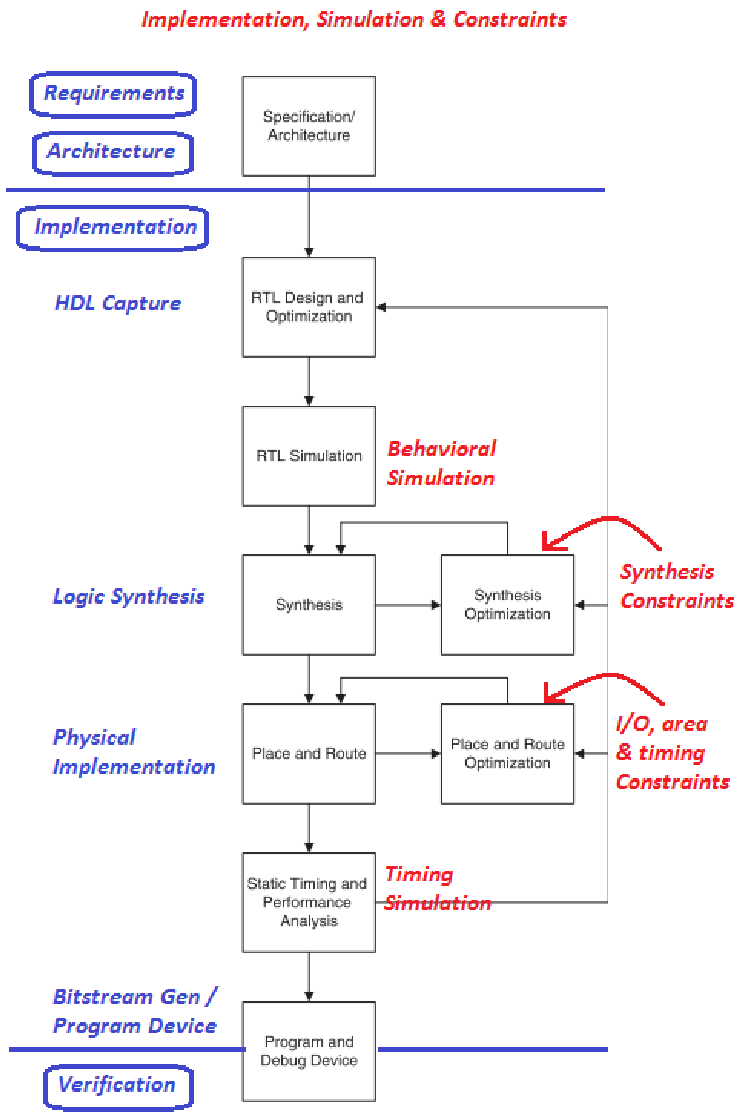


Figure 3.18: Design flow presented in Kilts's book [5] (see page 9 of the book)

phase (see Fig.3.19), although a description of each phase is provided in the book. Moreover, although it is not represented in the diagram, the book includes a complete chapter on Optimizations and Constraints, and also on Simulation.

Figure 3.20 shows the design flow found in the book by Wayne Wolf. This is a very basic flow that does not really reflect the amount of information in the book. Chapter 4 of the book focuses on Combinatorial circuit design, and Chapter 5 on Sequential circuits including FSM. In Chapter 6 the Data-Path-Controller architecture is presented.

As can be seen, each book names things and partition stages a little different. Moreover, the focus of each book makes the flows more or less detailed

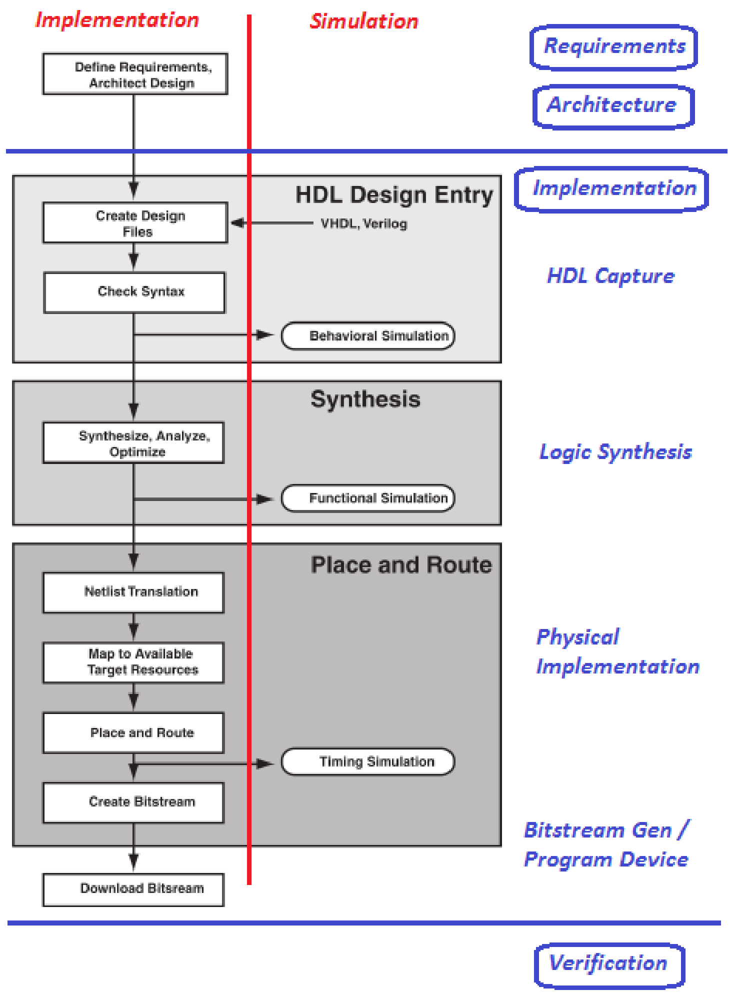


Figure 3.19: Design flow for the Implementation stage in Cofer's book [2] (see page 121 of the book)

in different sections of the flow. However, this comparison comes to show that the described design flow in this section extracts the important phases in FPGA-based Design.

3.4 Case Study: Real time hot spot detection using FPGA

In this section, we apply the aforementioned FPGA design flow to a remote sensing case study that requires on board, real time processing with low power consumption. As already stated, for many embedded digital signal processing applications, today's general purpose microprocessors are not longer able to handle them [72]. FPGAs offer a highly parallel hardware architecture with

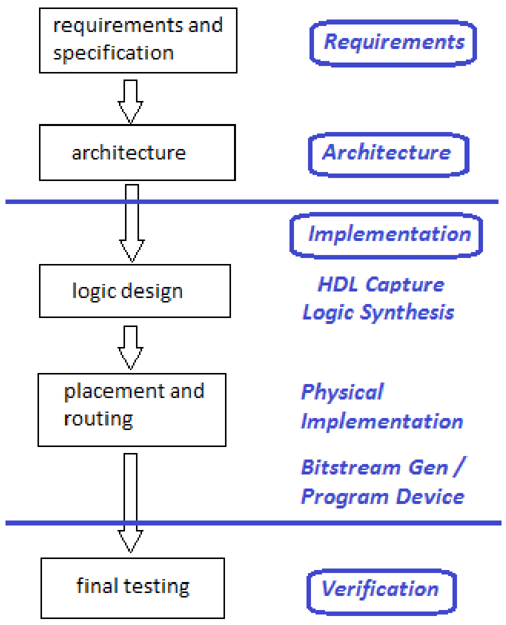


Figure 3.20: Design flow in Wolf's book [6] (see page 414 of the book)

low power consumption that is an alternative for such digital signal processing implementations.

3.4.1 Requirements and Specification

The problem consists on processing video captured by an IR camera on an Unmanned Aerial Vehicle (UAV) in real-time. The aim of the algorithm is to identify fire embers (that is, hotspots) in the captured images. The location and characteristics of the detected hotspots are then transmitted by the UAV to a firemen team fighting a forest fire. This solution was developed for an UAV System of the Department of Computer Architecture, Escola Politècnica Superior de Castelldefels, Universitat Politècnica de Catalunya [73].

The UAV has a network centric architecture, in which all sensors and different processing units are connected to an Ethernet network. It is required that the FPGA solution is inserted between the IR camera and the network. It should take the analogical output of the IR camera, process the frames in real time and return the location and spatial configuration of the found hot spots (if any) in UDP packets.

From each hotspot, the needed information is the bounding box that contains the complete hotspot and the hotspot's centroid. In geometry, the centroid or geometric center of a two-dimensional shape X is the intersection of all straight lines that divide X into two parts of equal moment about the line.

Informally, it is the "average" (arithmetic mean) of all points of X. For a finite set of k points $\{\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_k\}$ where $\mathbf{p}_i = (x_i, y_i)$, the centroid is given by equation 3.1.

$$\mathbf{C} = \frac{\sum_{i=1}^{k} \mathbf{p}_i}{k} = \begin{pmatrix} \frac{\sum_{i=1}^{k} x_i}{k} \\ \frac{\sum_{i=1}^{k} y_i}{k} \end{pmatrix} \tag{3.1}$$

To calculate the enclosing square, it is sufficient to obtain the maximum and minimum X and Y of the pixels that are in the hotspot. Similarly, to calculate the centroid, it is sufficient to calculate the amount of pixels k that are in the hotspot, and sum of their positions in X and in Y. The division to calculate the average can be calculated in the off-chip processor. Hence, the needed information from each hotspot is {`maxX`, `minX`, `maxY`, `minY`, `sumX`, `sumY`, `#pixels`}.

The most important constraint for the solution is that the image needs to be processed in real time, with the minimum possible delay between the acquisition of the last pixel and the transmission of the results. It is also desirable that the whole application works at the slowest possible clock frequency, to minimize the FPGA's power consumption.

Segmentation algorithm

The algorithm needs to segment the image in hot and cold regions, storing the location and spatial configuration of the found hot regions (i.e, hotspots). There are several possible algorithms to do this. However, we are looking for a solution that groups the pixels in hotspots and updates the stored hotspot's data as the image is being captured (if possible). Not every algorithm can be implemented in such a manner. In order to process the image as it is being captured, the algorithm must use only the data of the pixels previous to the one that is being processed.

The IR camera's output video is first digitalized, the temperature pixel is extracted and then classified as a hot o cold pixel (i.e, if the pixel belongs to a hotspot or not). This is done using a parametrizable threshold T. If the current pixel- say pixel(m,n)- is hot, the segmentation algorithm checks the adjacent pixels. These are the neighbors of the current pixel that have already been acquired, i.e, the pixel in the same column but previous row- pixel(m-1,n)- and the one in the same row and previous column- pixel(m,n-1). For this, the algorithm uses a list L that stores to which hotspot (if any) the previous line of pixels belong to (see Fig. 3.21).

If none of the two adjacent pixels belong to hotspots, then the current pixel

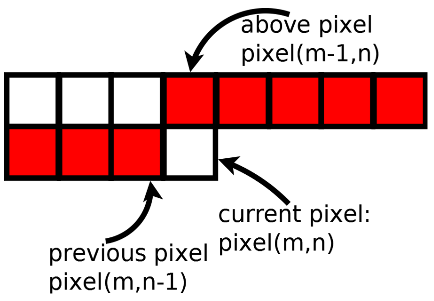


Figure 3.21: Line of previous pixels stored in L.

is the beginning of a new hotspot. If only one belongs to a hotspot, or both belong to the same hotspot, then the current pixel belongs to that hotspot. Finally, if both adjacent pixels are hot but belong to different hotspots, then the current pixel unifies those two previously discovered hotspots. In each step, the information of the current pixel- pixel(m,n)- is added to line L, and the information of the pixel in the same column but previous row- pixel(m-1,n)- is discarded, since it will not be used again. The algorithm also updates the stored hotspot's data according to the situation. The algorithm's pseudo code is shown in Listing 1

With this algorithm, there is no need for extra memory to store parts or the complete image, and the total processing delay is independent on the image size. Moreover, it allows an implementation that processes the image as it is being captured.

3.4.2 Architecture

In this section, architecture design for the solution is addressed. As explained in section 3.3.1, the main concept is hierarchical design. Figure 3.22 shows the hardware platform for the design. The board is connected to the IR camera and the Router. Besides the FPGA, it includes a SAA7113 video digitalizer [74] and an Ethernet physical driver.

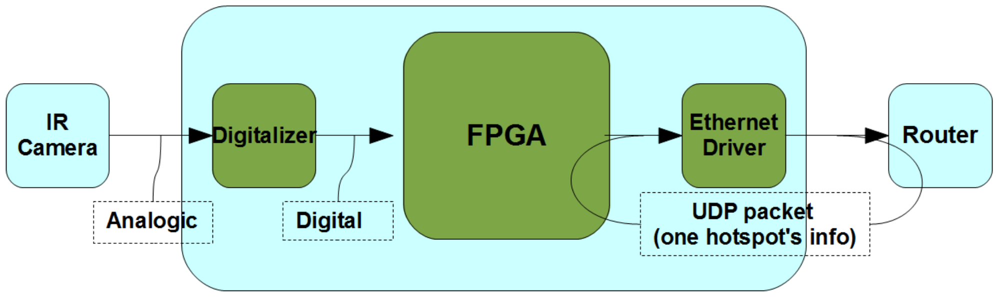


Figure 3.22: Hardware platform including the video digitalizer, the FPGA and Ethernet physical driver.

Algorithm 1 Segmentation Algorithm.

```
function SEGMENT IMAGE(pixel(m,n), line L, threshold T)
    if pixel(m,n) $<$ T then
        mark in the line L that pixel(m,n) does not belong to a hotspot.
    end if
    if pixel(m,n) $\geq$ T then
        if pixel(m-1,n) and pixel(m,n-1) do not belong to hotspots then
            create a new hotspot for pixel(m,n)
            mark pixel(m,n) in the line L as belonging to the new hotspot
        end if
        if (pixel(m-1,n) or pixel(m,n-1) $\in$ hot_spot_x) and neither are in other hotspot then
            add pixel(m,n) to hot_spot_x in the memory
            mark pixel(m,n) in line L as belonging to hot_spot_x
        end if
        if (pixel(m-1,n) $\in$ hot_spot_x & pixel(m, n-1) $\in$ hot_spot_y & id(hot_spot_x) $<$ id(hot_spot_y)) then
            add hot_spot_y data to hot_spot_x in the memory
            add pixel(m,n) to hot_spot_x in the memory
            mark hot_spot_y as invalid in the memory
            mark pixel(m,n) in line L as belonging to hot_spot_x
            for each pixel in line L do
                if pixel $\in$ hot_spot_y then
                    mark pixel as belonging to hot_spot_x
                end if
            end for
        end if
    end if
end function
```

From the description of the algorithm, three top hierarchy modules are clear. One to configure the digitalizer of the IR camera and to create the raw data that the segmentation algorithm needs from the camera's digitalized output. The second one to perform the segmentation algorithm described in Listing 1. And the third one to transmit the resulting hotspots in each image through UDP. Fig. 3.23 shows this global view of the architecture.

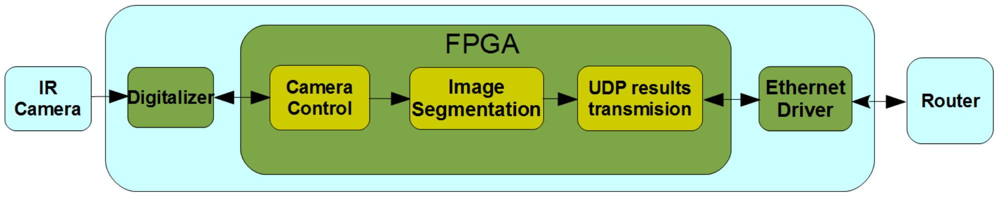


Figure 3.23: First view of the hierarchichal design

At this stage, it is important to define which are the interfaces for these modules, and some general constraints and definitions that will affect the next stages. The amount of design details vary, and the final design is, as already stated, an iterative work between architecture, implementation, testing and timing analysis. However, this iterative process cannot be completely pictured in a few pages, hence it is important to keep in mind that some of the detail explained in the Architecture phase was actually achieved after some iterations.

Camera module

As already stated, the camera module needs to perform two functions: to configure de SAA7113 digitalizer, and to create the raw data for the segmentation algorithm from the digitalized video.

Digitalizer Configuration

The SAA7113 digitalizer is configured using a I^2C bus. Through this bus, a set of internal registers of the digitalizer need to be set. Hence, this configuration module needs to drive the I^2C bus pins- including the 200 Khz clock-, the digitalizer's chip enable and also a pin to inform that the digitalizer has been configured and it is ready for use.

The chip can be configured to transmit in several video codings. In this application, we use it to transmit video coded in ITU-R BT 656 YUV 4:2:2 format at the standard's clock of 27 Mhz. In this configuration, the digitalizer sends the Y component coded in 8 bits in one clock, and the UV component in another clock. Hence, it takes two clocks to transmit the information of one pixel.

Raw Generator

The first thing to think when architecting this submodule is: what information needs to be extracted from the YUV digitalized image data for the segmentation algorithm to work? For one, the IR level information from the pixel to classify it as hot or cold. This information is in the Y component of the video stream. Also, the (X,Y) position of the current pixel is needed, since the algorithm needs to calculate their maximums and minimums, and their sum. The (X,Y) position needs to be generated from the horizontal and vertical synchronization signals that the SAA7113 exports. Finally, a frame signal is needed to know when the frame is over and the information of a new frame starts.

A requirement of the application is that it runs at the slowest possible clock in order to consume less power. A possible solution is that the application runs at the digitalizer's clock frequency, i.e. 27 Mhz. This imposes the need for the segmentation algorithm to process each pixel as it is being acquired, that is, with a throughput of one pixel every two clocks. This restriction will impact the design of the segmentation module further along the way. Using the digitalizer's clock also simplifies the integration between the camera and the proposed solution. Hence, the raw generator module also generates the clock signal for the rest of the application, propagating the clock signal from the SAA7113. It also generates a *sync_y* signal that identifies the rising edge of the clock at which the pixel data is changed.

It is important to note that the digitalizer and camera used is abstracted from the rest of the application by the Raw Generator. They can be completely changed and the rest of the application will continue to work, provided the Raw Generator module continues to produce the same control signals; i.e, that it still produces one (X,Y) and pixel data every two clocks and correctly generate the frame and sync_y signals.

Figure 3.24 shows the architecture of the Camera Module.

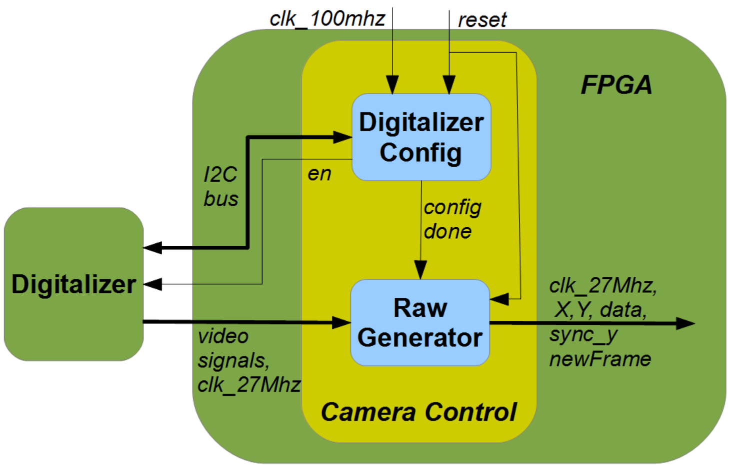


Figure 3.24: Camera control module architecture

Table 3.1: Raw processing output signals according to the situation.

the pixel...	**id1**	**id2**	**unifHS**	**newHS**
..is not hot	0	0	0	0
..starts new hotspot	newId	0	0	1
..is in hotspot with idA	idA	0	0	0
..unifies hotspots with idA & idB; and idA<idB	idA	idB	1	0

Image Segmentation module

This module implements the segmentation algorithm. In order to partition this module, it is important to keep in mind that it needs to process the pixel as it is being acquired.

A way to start is to think about the memory needed: the list L of previous pixels needs to be stored, and also the information of the previously found hotspots. L is relatively small, and can be stored in flip-flops, but the hotspots will need a BlockRAM. Since the list L needs to be read and written, and so does the hotspot memory, it is not possible to perform the whole processing at pixel rate (i.e, two clocks). However, the solution can be pipelined: a first stage can process the pixel using list L to determine to which hotspot it belongs to; and a second stage can make the necessary changes to the hotspot information in memory. At least another module is needed: the memory module for the hotspots. A first approximation to the Image Segmentation module design includes three sub-modules: Raw Processing, HotSpot Reconstructor and HotSpot Memory.

Raw processing

The Raw Processing module needs to determine if the current pixel is hot or not. If the pixel is hot, the module decides if it is the beginning of a new hotspot, if it belongs to a previously discovered hotspot or if it unifies two previously discovered hotspots. For this, the only stored information it needs is the list L that stores to which hotspot the previous line of pixels belong to.

In Fig. 3.25, the needed inputs and the produced outputs of this module can be seen. Important things to note is that the IR data from the pixel is not propagated to the next module, since it is not needed any more. The outputs include two control signals: new hotspot and unify hotspot (*newHS* and *unifyHS* in the figure). It also includes two hotspot ids (*id1* and *id2*). In Table 3.1, the values of these output signals according to the situation are shown.

The module needs to update the list L according to the processing results and generate the corresponding output signals at pixel rate (i.e, in 2 clocks). This means the output signals from the current pixel are stable for two clocks. For this, it uses the *sync_y* signal.

Hotspot Reconstructor and Hotspot Memory

As discussed in section 3.4.1, the information stored for each hotspot is {`maxX`, `minX`, `maxY`, `minY`, `sumX`, `sumY`, `#pixels`}. The Hotspot Reconstructor module is in charge of updating the appropriate hotspot with the information from the current pixel, as shown in the algorithm in Listing 1. For example, adding one to the amount of pixels, checking if the X component is bigger than the stored `maxX`, etc. For this, it uses the X,Y position of the pixel and the control signals produced by the Raw Processing module. It also has to access the Hotspot Memory, retrieve the corresponding hotspot(s), recalculate the data and write the results back.

Adding the information of a new pixel to a particular hotspot or creating a new hotspot is rather straightforward. Unifying two hotspot is a little trickier. To start, two hotspots need to be accessed. Hence, the both ports of the Hotspot Memory are used: one is accessed with *id1* and the other with *id2*. The information of both hotspots and the current pixel needs to be merged. Since the stored information are maximums, minimums and sums, this is easily performed. For example, the equations for the X coordinate are:

$$\begin{aligned} maxX_{new} &= max(maxX_{id1}, maxX_{id2}, X) \\ minX_{new} &= min(minX_{id1}, minX_{id2}, X) \\ sumX_{new} &= sumX_{id1} + sumX_{id2} + X \end{aligned}$$

The unified hotspot data is always written to the *id1* hotspot. However, the hotspot with *id2* is also overwritten to state that this hotspot is no longer valid. In this manner, it will not get transmitted as a valid hotspot when the frame is completely processed.

Finally, it is important to remember that all this processing needs to be performed at pixel rate. Luckily, pixel rate means two clock cycles. Hence, during the first clock the hotspots can be read from memory and the data can be merged, and the results can be written during the second clock. Of course, this requires that the logic to merge the data is fast enough to be done in one clock, but since the clock is rather slow (27Mhz) and the logic is relatively simple, this restriction is achieved.

Fig. 3.25 shows this initial architecture for image segmentation.

Final modifications

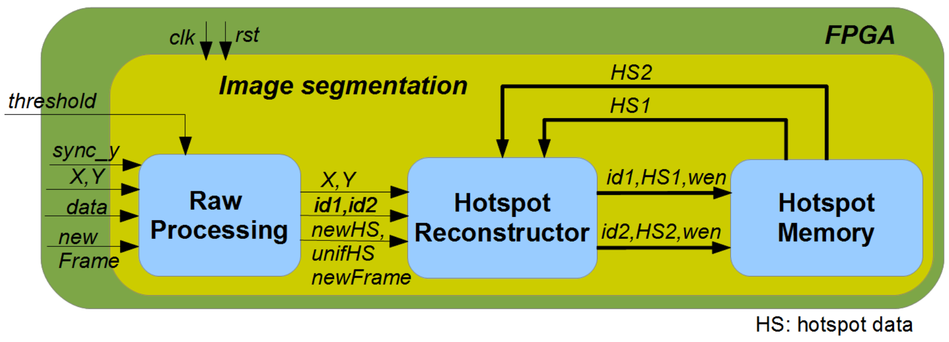


Figure 3.25: Initial Architecture for the Image Segmentation module. The clock is the 27 Mhz digitalizer clock. Both the clock and reset signals are distributed to all the submodules, although this is not shown in the figure for simplicity.

To process one frame and not transmit the results, this architecture would be sufficient. However, the application is required to process a video stream. Hence, the results of the previous frame need to be transmitted as the current frame is processed. As both ports of the Hotspot Memory are being used, the transmission module cannot access that memory to retrieve the data from the previous frame. A solution is to use a double buffer: to have two hotspot memories, one for the current frame and another for the previous frame. For this to work, the hotspot reconstructor needs to switch the memory it fills in each frame. Moreover, the logic that will take out the hotspots' information for transmission also needs to switch memories in each frame. Hence, we have three new modules: an extra Hotspot Memory, the Switching Box and a Hotspot Retriever for transmission. The Switching Box connects the memories to the appropriate modules whenever the *newFrame* signal becomes active.

There is another problem: the clock at which the image segmentation is done (27 Mhz) is quite different from the UDP transmission clock (100 Mhz). Hence, there is a clock domain transition at this spot. For this, a FIFO can be used.

The Hotspot Retriever generates all the possible Hotspots ids, accessing the memory in an ordered manner. If the hotspot is valid, it adds the frame id (i.e, to which frame this hotspot belongs to), and puts the information in the FIFO at the 27Mhz clock rate. It also overwrites the read hotspot in memory with hotspot initialization values (i.e, all zeros). When the *newFrame* signal is asserted, the module increments the frame id and resets the hotspot id generator. The data FIFO that is filled by this module at 27 Mhz will be read by the UDP transmission module at it's clock rate, completing the clock domain transition.

Fig. 3.26 shows the final architecture of the image segmentation module.

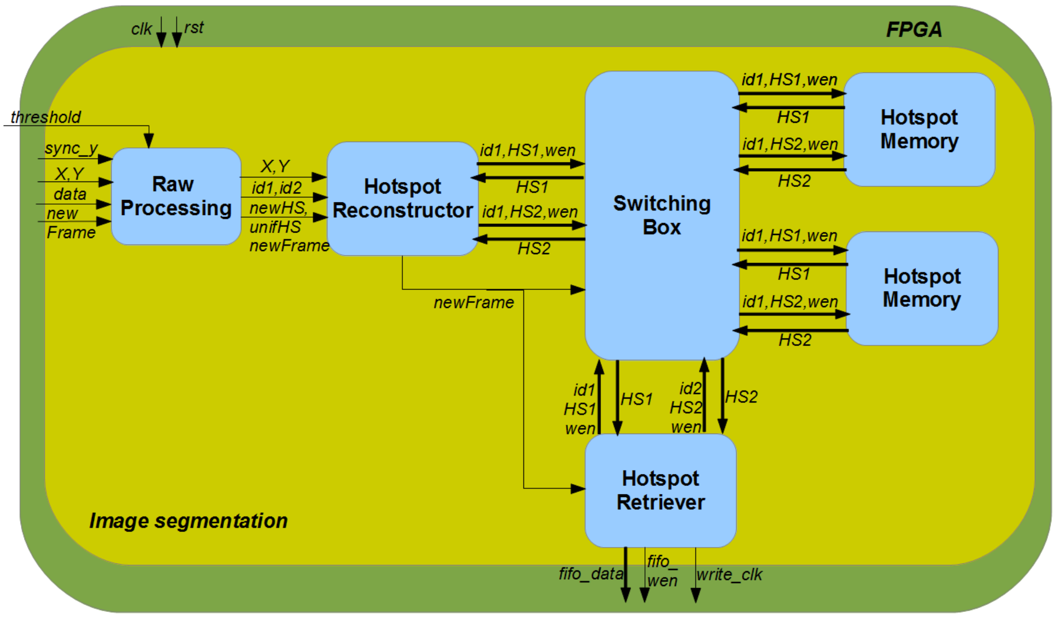


Figure 3.26: Final Architecture for the Image Segmentation module.

Net Control module

The Net Control module has two submodules: the FIFO from where it takes the hotspot data, and a net transmission module that communicates with the physical Ethernet driver, and generates Ethernet, IP and UDP packets with the hotspot information and control for a host application in a remote computer. Hence, this module together with the Physical Ethernet Driver implement a simplified but complete OSI model: from the Application layer to the Physical one.

Fig. 3.27 shows the architecture of the Net Control module.

3.4.3 Implementation

In this subsection we describe the FPGA implementation, simulation and constraints. The implementation description is focused in the Image Segmentation module, particularly the Raw Processing and the Hotspot Reconstructor modules that are the core of the solution. For the rest of the submodules, only a short description is provided, commenting which of the HDL capture patterns described in section 3.3.2 were used.

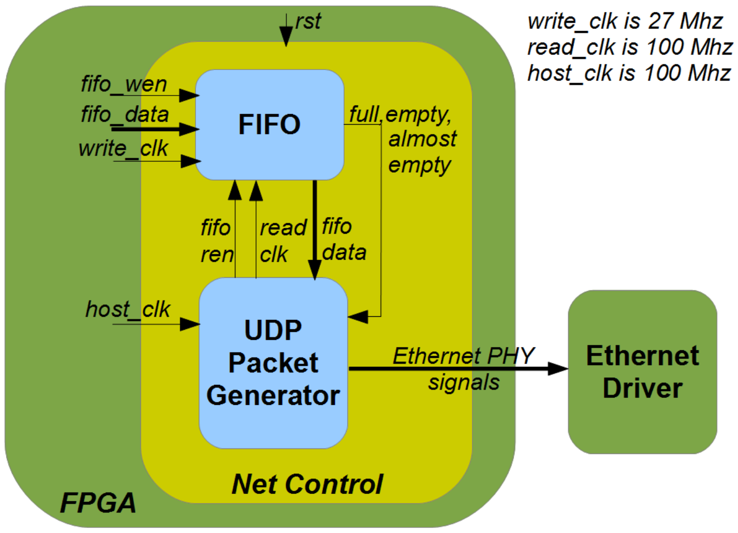


Figure 3.27: Architecture of the Net Control module.

Camera Module

In this module, the Digitalizer Configuration is implemented using two FSM sequential circuits. One is an “outer loop” that transits through the several steps of the configuration (idle, reset, configure, etc). The other FSM performs the I^{2}C bus protocol, and using this protocol sends the SAA7113 registers following the chip’s data-sheet to obtain the desired video coding.

The Raw Generator module is implemented with several processes that generate each output signal from the digitalizer’s output. This transformation are rather straightforward.

Image Segmentation Module

Raw Processing module

The core of the Raw Processing module is the implementation of L such as to calculate which hotspot the current pixel belongs to and update all the list at the pixel rate. For this purpose, L is implemented as a stack: the top of the stack stores the id of the hotspot that pixel(m,n-1) belongs to, and the bottom of the stack stores the id of the hotspot pixel(m-1,n) belongs to. This two special records can be accessed to obtain the hotspot ids needed in the algorithm. The remaining records in L hold the information of the line of pixels between pixel(m-1,n) and pixel(m,n-1), i.e, the previous line of pixels. At pixel rate, the hotspot id corresponding to the new pixel is pushed onto the stack, all the middle records are updated if necessary and moved to the next stack position, and the bottom record (i.e, the hotspot id of pixel(m-1,n)) is discarded. Fig. 3.28 shows this stack implementation of L.

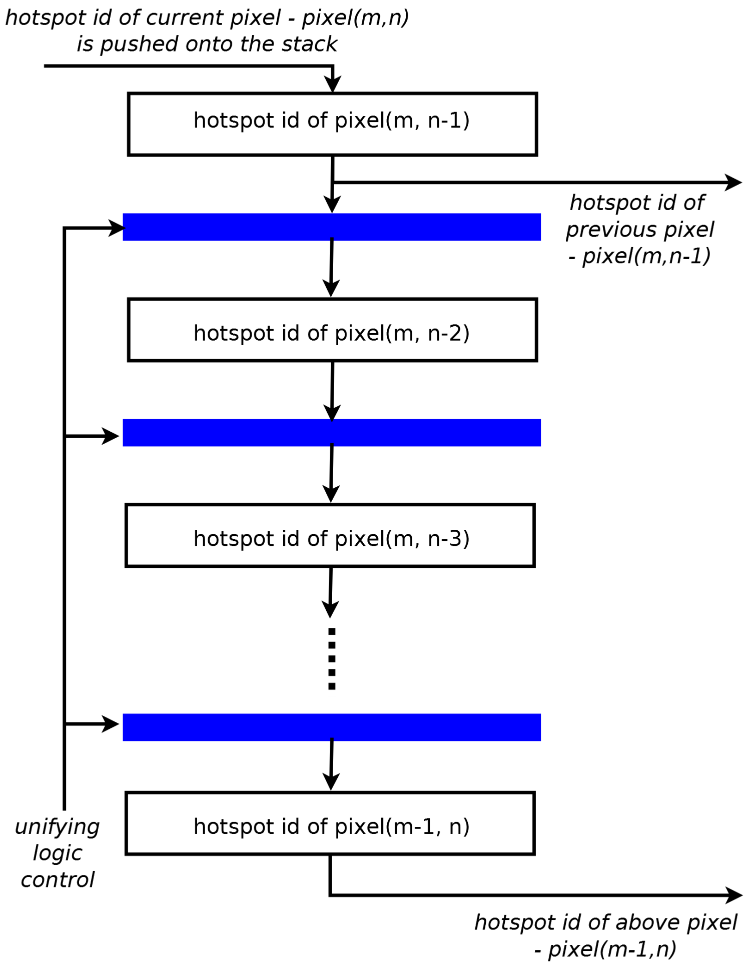

hotspot id of current pixel - pixel(m,n)
is pushed onto the stack
hotspot id of pixel(m, n-1)
hotspot id of previous pixel - pixel(m,n-1)
hotspot id of pixel(m, n-2)
hotspot id of pixel(m, n-3)
unifying logic control
hotspot id of pixel(m-1, n)
hotspot id of above pixel - pixel(m-1,n)

Figure 3.28: Line L of previous pixels implemented as a stack.

When a pixel unifies two previously discovered hotspots, say hot_spot_y and hot_spot_x, hot_spot_y is marked as invalid and all the pixels belonging to that hotspot are added to hot_spot_x. In that case, all records in the stack have to be accessed, compared with the id of hot_spot_y and changed to the id of hot_spot_x (if needed) in one pixel cycle. To accomplish this, each record of the stack has a comparator and a multiplexer. The result of comparing the record's id, say idA, with the id of hot_spot_y enables the multiplexer that either propagates idA or the id of hot_spot_x to the next record as needed. The implementation of each record in list L is shown in Fig. 3.29.

L is implemented as a regular sequential circuit: at pixel rate, the data is transmitted from one record to the next as previously indicated. The pixel rate is generated using the clock and the *sync_y* signal as an enable. The logic to create the output signals (*unifyHS*, *newHS*, *id1*, *id2*), that are also the signals used in list L's multiplexing logic, is completely combinatorial. At pixel rate, the ids of the top and bottom stack records together with the input signals are

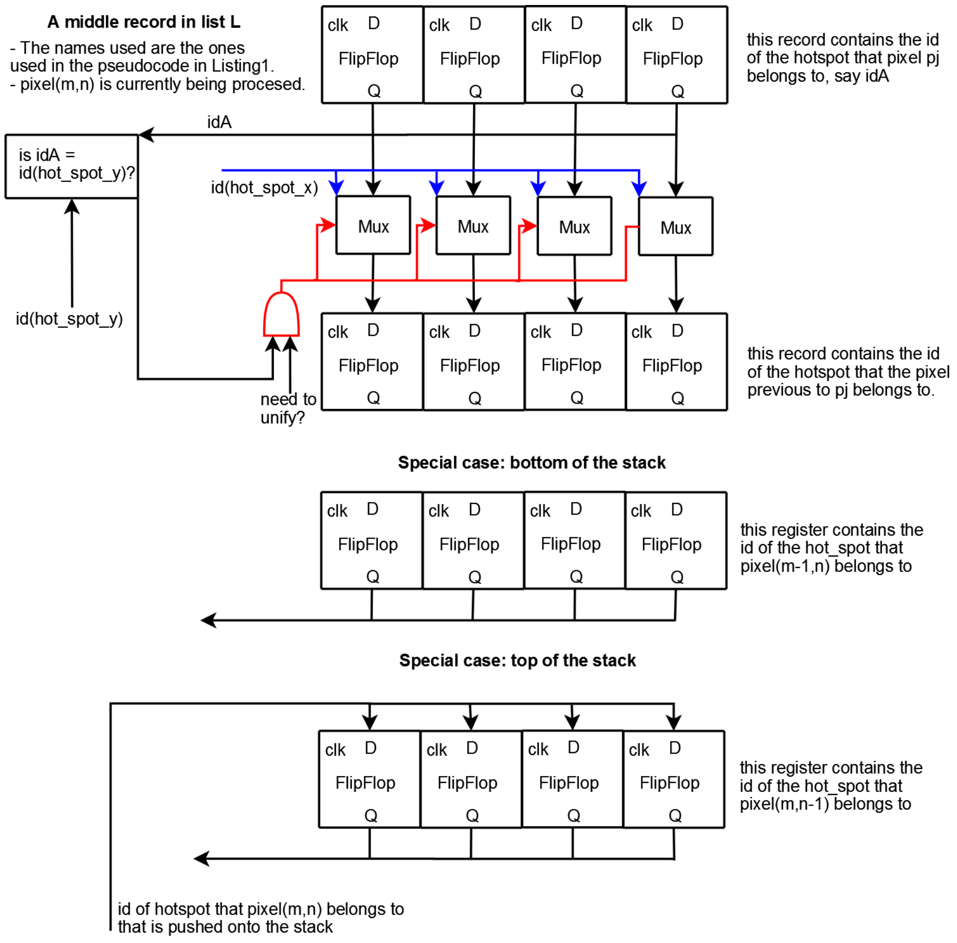

A middle record in list L
- The names used are the ones used in the pseudocode in Listing1.
- pixel(m,n) is currently being procesed.
clk D FlipFlop Q
this record contains the id of the hotspot that pixel pj belongs to, say idA
idA
is idA = id(hot_spot_y)?
id(hot_spot_x)
Mux
id(hot_spot_y)
need to unify?
this record contains the id of the hotspot that the pixel previous to pj belongs to.
Special case: bottom of the stack
this register contains the id of the hot_spot that pixel(m-1,n) belongs to
Special case: top of the stack
this register contains the id of the hot_spot that pixel(m,n-1) belongs to
id of hotspot that pixel(m,n) belongs to that is pushed onto the stack

Figure 3.29: Detailed implementation of the stack L in the Raw Processing module, including the general middle records and the special top and bottom records of the stack.

read, the combinatorial logic creates the control signals appropriate for this pixel, and when the next pixel arrives, the records in list L are updated and the output signals are already stable. Finally, to generate a *newId* when a pixel starts a new hotspot, a simple counter is used. Both the list L and the id generator counter are reset to zero when a new frame starts (i.e, when the *newFrame* input signal is driven high).

Hotspot Reconstructor module

As already described in the architecture section, this submodule runs at clock frequency. It uses the control signals generated by the previous module, which are stable for two clock cycles. It reads the information of the hotspots from memory and merges it with the pixel info during the first clock cycle, and writes the results back to the memory during the second clock cycle. The implementation of this module is shown in Fig. 3.30 and Fig. 3.31.

Switching Box

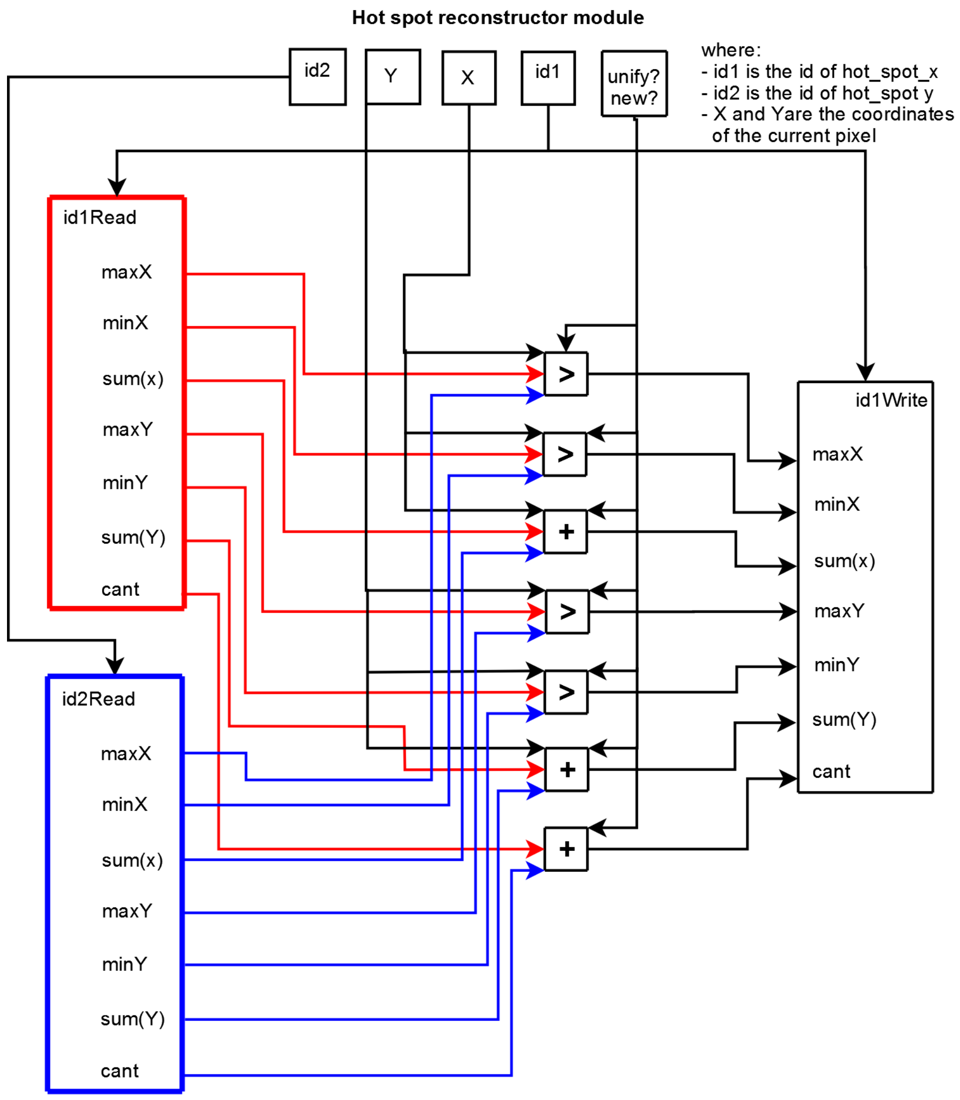


Figure 3.30: HotSpot Reconstructor module.

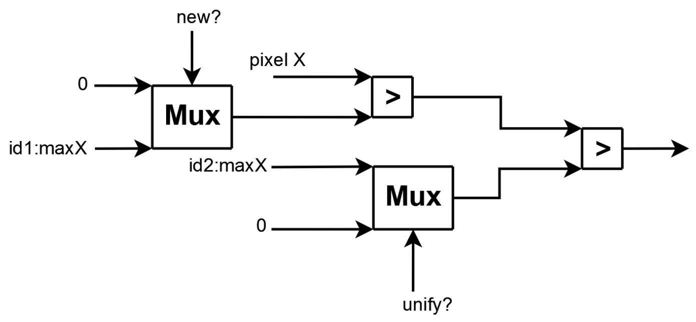


Figure 3.31: Detail of the logic for the maximum calculation in the hotSpot Reconstructor Module

This module is implemented as a combinatorial circuit using several multiplexers. The control signal of the multiplexer is a frame signal that is toggled every time that the *newFrame* signal is asserted. This successfully switches the memories, reconstructor and retriever output and input signals.

Hotspot Memory

This module is implemented using a BRAM ipcore instantiation, and creating a combinatorial wrapper that maps the module's inputs and outputs to the particular BRAM ipcore interface. The instantiated ipcore is Xilinx's LogicCore Dual-Port Block Memory Core v6.3 [75].

Hotspot Retriever

This module is implemented with an FSM. It needs to follow several steps to retrieve the data from the Hotspot Memory and write it in the FIFO (control implemented with a FSM). It also needs to transform the hotspot data, formatting it in a particular way and adding frame information.

Net Control Module

FIFO

The FIFO module is implemented by instantiating a FIFO ipcore, and creating a combinatorial wrapper that maps the module's inputs and outputs to the particular ipcore interface. The instantiated ipcore is Xilinx's LogicCore FIFO Generator v4.3 [76]. This core is actually a particular wrapper for a BRAM.

UDP packet Generator

This is implemented using several FSMs. It needs to follow several steps to retrieve the data from the FIFO, encapsulate it in the several packet layers, and drive the Ethernet physical signals so that the driver can actually send the data to the Router.

To configure and access the Ethernet Physical Driver, a wrapper ipcore provided by Xilinx is also used. This wrapper file instantiates the full Virtex-4 FX Ethernet MAC (EMAC) primitive.

Constraints, Logic Synthesis and Physical Implementation

The tool used for logic synthesis and physical implementation is the Xilinx ISE Webpack 10.1, with default settings for all the involved processes. As can be seen from previous sections, the core of the design is very much tailored to a particular hardware implementation. Moreover, the desired clock frequency is rather slow (27 Mhz). This reduces the importance of using special area or timing constraints to aid synthesis tools, and hence the default constraints were used. After synthesis and physical implementation, since the timing and area requirements were met, there was no need to review this decision. Particular I/O constraints were added to map the FPGA pins for the SAA7113 digitalizer, the Ethernet driver and clock signals. This constraints are the ones provided by Avnet, the manufacturer of the particular development board used.

Simulation

The basic verification tool used in this project was Behavioral Simulation. Each of the submodules described has its own HDL testbench. Some are simple, and some are quite complex. The submodules were gradually integrated in the modules, and each module has its own testbench to check for correct integration of the submodules.

In order to test the Image Segmentation module, a fake camera module was created that emulates the acquisition of several frames and generates the appropriate Raw Generator signals. This module uses a memory that stores the image information. This fake module was key to simulating the complete Image Segmentation module.

3.4.4 Solution sizing

Area and Memory equations

In order to implement the high parallelism needed to achieve the proposed real time processing of the image, much space and hardware resources of the FPGA are used. The area needed for this implementation depends only on the size of the image and the maximum amount of hotspots that can be found in each image. In order to make the application suitable for different IR cameras, those parameters can be easily configured.

There are two modules in the implementation that are resource consuming: the Raw Processing module and the Hotspot Memory module. Since the design of these modules is tailored, equations to estimate area and Block RAM consumption can be derived.

The Raw Processing module implements the list L that stores the hotspot id for each pixel in the previous line, as explained in subsection 3.4.3. In terms of FPGA area, the list has *im_width* records. Each record is wide enough to store a hotspot id, that is *log(max_hotspot_amount)* bits, and has extra logic needed for the hotspot unifying process. The area equations is as follows:

$$\begin{aligned}area &= im\_width \times [\\ &\quad log(max\_hotspot\_amount) \times (1FF + 1Mux) +\\ &\quad 1comp\_of\_log(max\_hotspot\_amount)\_bits]\end{aligned}$$

The Hotspot Memory module stores the information of each hotspot found in the image, that is, the memory has *max_hotspot_amount* records. As already explained, each record stores the following information: maxX, minX, sumX, maxY, minY, sumY, #pixels. Moreover, there are two memory buffers. Hence, the needed Block RAM equation is as follows:

$$\begin{aligned}BRAM &= 2 \times max\_hotspot\_amount \times [\\ &\quad log(im\_width) + log(im\_width) + (2 \times log(im\_width) - 1) +\\ &\quad log(im\_height) + log(im\_height) + (2 \times log(im\_height) - 1) +\\ &\quad log(im\_width \times im\_height) - 1]\end{aligned}$$

From these equations, it can be seen that the amount of logic cells needed depend linearly on the image width, while the amount of memory (Block RAMs) depends linearly on the maximum amount of hotspots to be detected per frame. The image height is of little importance for area calculations. With these equations, it is straightforward to calculate the size of the needed FPGA given the size of the image and a maximum for the amount of hotspots expected in each frame.

Code Sizing

To have an idea of the size of the project, we present in table 3.2 a listing of the amount of VHDL module files and lines of code used for the implementation of each module.

3.4.5 Experiments and Results

The testing environment consists of a PAL-N composed video camera, and an Avnet development kit with a Xilinx Virtex 4 FX12-10C-ES FPGA, a SAA7113 video digitalizer and an Ethernet physical driver. The UDP packets with the resulting hotspots are routed to a PC in order to check the processing results. A program in the host PC shows the results. In this experimental

Table 3.2: Code sizing

	Module	lines of code	VHDL files
Implementation	Top	297	1
	Camera	1298	6
	Image Segmentation	1573	17
	Net Control	1605	6
	Total	*4773*	*30*
Testing	Testbenches	748	7
	Special Modules	344	3
	Total	*1092*	*10*
	Total	**5865**	**40**

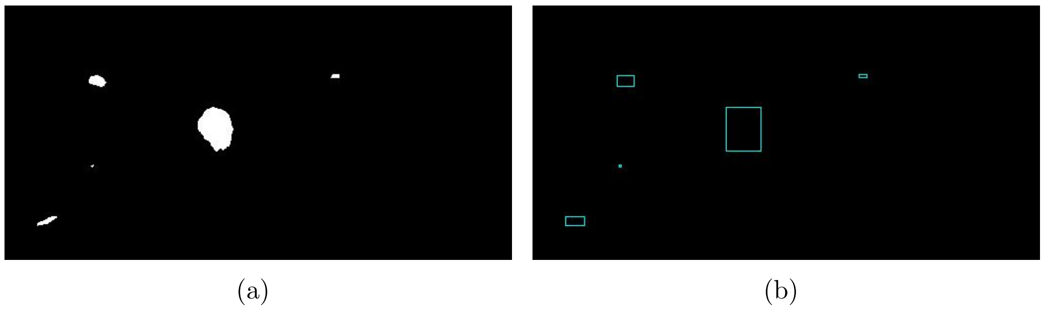


Figure 3.32: *a* hotspot image after classification in hot or cold pixels. *b* visual representation of the results showing the location of the detected hotspot (the centroid is not shown).

setup, the threshold is set using the board's Dip Switches. For the Ethernet transmission, the MAC and IP address are fixed, and to debug purposes, a switch is implemented to transmit either the original image or the processed and thresholded image.

From the video camera we process and analyze 50 frames per second, corresponding to half video images even and odd, with 512 pixels by 256 lines per frame. The solution was configured for a 256 maximum amount of hotspots per frame. In Fig. 3.32 some results are shown.

Area

In table 3.3, area results for the complete solution and the most area consuming modules are shown.

As all the memories were mapped into block rams, including the UDP FIFO and the configuration memory for the SAA7113, and the hardware MAC

Table 3.3: Area occupied by the complete solution

	Complete Sol.		Raw Processing		Hotspot Memory—	
	amount	%	amount	%	amount	%
Slices	**4972**	**90**	**4492**	**82**	0	0
Slice FF	4510	41	4109	37	0	0
LUTs	8576	78	7478	68	0	0
BRAMs	**9**	**25**	0	0	**6**	**17**
EMAC	1	100	0	0	0	0

Ethernet included in the Virtex 4 was used, the total area of the application was 90% of the FPGA slices and 25% of the Block RAMs (9 out of 36).

In particular, the occupied area of the most consuming module- Raw Processing- was 82% of the FPGA slices. The most memory consuming module (Hotspot Memory) consumed 17% of the Block Rams (6 out of 36).

This area corresponds to the size equations presented in subsection 4, and it means that the entire application fits in the smallest Virtex 4 FPGA available in the market.

Area equation comparison

In this section, we briefly compare the proposed solution sizing equations with the synthesis and physical implementation area results.

To start, it is important to note that the Raw Processing module is in fact the most area consuming module of the application. It can be seen in table 3.3 that this module alone corresponds to 4109/4510 = 91% of the used Flip Flops and 7478/8576 = 87% of the used LUTs. Hence, the presented area equation is a good measure of the area that will be needed to implement the solution for a particular camera and hotspot amount.

Following area equation 3.2, and taking into account that in this case study *im_width* is 512 and *max_hotspot_amount* is 256, the estimated area for the Raw Processing module is:

$$\begin{aligned} area &= 512 \times [log(256) \times (1FF + 1Mux) + 1comp\_of\_log(256)\_bits] \\ &= 512 \times [8 \times (1FF + 1Mux) + 1comp\_of\_8\_bits] \\ &= 4096 \quad FF + 4096 \quad Mux + 512 \quad comp\_of\_8\_bits \end{aligned}$$

In table 3.4, the Macro Statistics for module Raw Processing are shown.

It can be easily seen that the estimated area results are compatible with

Table 3.4: Advanced HDL Synthesis Report Macro Statistics for Raw Processing module

# Adders/Subtractors	1
8-bit adder	1
# Registers	4104
Flip-Flops	4104
# Comparators	514
8-bit comparator equal	512
8-bit comparator greater	1
8-bit comparator less	1

both the Macro Statistics (table 3.4) and the Area consumption results (table 3.3):

- *Flip Flops*: the equation yields that 4096 FF should be used, the Macro Statistics assigns 4104 and the area consumption after placement uses 4109.
- *Comparators*: the equation yields 512 8-bit comparators. The Macro Statistics actually assigns 512 8-bit equal comparators, that are the ones taken into account in the equation, i.e, the ones between each register for the id propagation logic. The other 2 comparators are used in generating the control signals (e.g, when a pixel unifies two hotspots, the module selects the hotspot with smaller id, using a less comparator).
- *Multiplexers*: there are no particular mentions to multiplexers in the tables.
- *Adder*: the adder that is in the Macro Statistics corresponds to the *newId* generator (in case the pixel starts a new hotspot).

BRAM equation comparison

Looking at the memory requirements, equation 3.2 refers to the BRAM bits needed in the implementation of both Hotspot Memory modules. Taking into account that in this case study *im_width* is 512, *im_height* is 256 and *max_hotspot_amount* is 256, the estimated memory bits needed in both Hotspot Memory modules is:

$$\begin{aligned}
BRAM &= 2 \times 256 \times [log(512) + log(512) + (2 \times log(512) - 1) + \\
&\quad log(256) + log(256) + (2 \times log(256) - 1) + log(512 \times 256) - 1] \\
&= 2 \times 256 \times [9 + 9 + (2 \times 9 - 1) + 8 + 8 + (2 \times 8 - 1) + 17 - 1] \\
&= 2 \times 256 \times [9 + 9 + (2 \times 9 - 1) + 8 + 8 + (2 \times 8 - 1) + 17 - 1] \\
&= 2 \times 256 \times 82 = 41984 bits
\end{aligned}$$

This yields a need for 41984 bits, arranged in two modules, each having 256 records of 82 bits each. So, how many BRAMs are needed to arrange a module with 256 records of 82 bits? The ports in the Virtex4 BRAM can be configured in any of three #entries×bits_per_entry "aspect ratio": 16K×1, 8K×2, to 512×36. For this application, the best one is 512×36, and since 82 bits per entry are needed and each BRAM accommodates at most 36 bits per entry, 3 BRAMS will be needed per module. That is exactly the informed result in the area consumption report, see table 3.3.

Both Hotspot memories use 6 BRAMs, and the complete application uses 9 BRAMs. The remaining 3 BRAMs are occupied as follows: one is used for the FIFO between the Image Segmentation Module and the Net Control module to achieve the clock domain interface, and the other two are used to store the configuration for the SAA7113 and the hardware EMAC.

Power Consumption

Estimate measures of power consumption were done using Xilinx's XPower tool. The total estimated power is 260.92 mW, taking into account both the logic and driving the I/O of the FPGA to communicate both with the digitalizer and physical Ethernet driver. Taking into account that the processing delay of an image is equal to the acquisition time of one pixel, the Energy (i.e, W×s or Power×Time) can be calculated for any particular image size. In the case of an image of 512 × 256 pixels that is digitalized at a pixel clock of one pixel every two 27 Mhz clocks, this yields:

$$acq\_time = (512 \times 256) \times \frac{2}{27M} s = 0.0097s = 9.7ms \tag{3.2}$$

Hence, the estimated Energy for the processing of one frame is:

$$E(J) = P \times time = 0.26092W \times 0.0097s = 0.0025J = 2.5mJ \tag{3.3}$$

Timing

The maximum operation clock obtained was of little over 100 Mhz, which is enough to work with the 27 Mhz clock output of the SAA7113. In the UAV application, the mounted IR camera is the FLIR A320, that delivers 320 pixels by 240 lines images at a rate of 9 fps. Therefore, the results of the tests in the laboratory experiments indicate that the solution is well suited for the UAV application. The proposed method successfully segments the image with a total processing delay equal to the acquisition time of one pixel (that is, at the video rate). This processing delay time is independent of the image size.

3.5 Conclusions

In this chapter, FPGAs were introduced and a commonly used FPGA design flow was discussed. The presented design flow has four broad stages: Requirements, Architecture, Implementation and Verification. The Implementation is done capturing the design with Hardware Description Languages and using synthesis and physical implementation tools to translate HDL to the final placed and routed design in a particular FPGA chip. Although HDLs support a Behavioral coding level, most of the design patterns found in books require that the designer think at RTL level, picturing the intended synthesized hardware while architecting the solution and capturing it in HDL. This also means that designers keep in mind to what hardware each HDL constructs is synthesized to. Constraints and synthesis tools aid designers in this task, while powerful simulators and verification languages aid him or her in the verification phase.

This type of design allows to create extremely tailored, top performance designs that meet tight real-time constraints with low power consumption and small occupied area. However, it requires a lot of engineer hours from knowledgeable designers. It is also highly prone to error, and hard to test and verify even using top notch simulation tools and complex verification environments.

This design flow was applied to a case study to achieve real-time processing of an IR image for hot spot detection. The algorithm was thought for parallel implementation, and the design was carefully tailored to achieve the real-time constraints. In this manner, the proposed method successfully segments the image with a total processing delay equal to the acquisition time of one pixel (that is, at the video rate). This processing delay time is independent of the image size. There is also no need for extra memory to store parts or the complete image. Since the design was clearly mapped to the implementation, FPGA area equations could be presented in order to calculate the needed FPGA size for a particular application. The synthesis area reports confirm this equations. Finally, as the configuration and communication with the camera is encapsulated in one module, the proposed solution is not tied to one specific IR camera, and may be used with several IR camera with minor adjustments.

The tailored design in this case study was necessary to achieve the tight real-time constraint. However, it also shows the level of detail and care needed to achieve a working solution, including the development of many testbenches and particular modules for testing. It shows the difficult, low level and time-consuming process of the traditional HDL-based design flow.

Clearly, measures to decrease design time by raising the abstraction level of design and implementation are needed to cope with the ever increasing complexity of applications. This is addressed in the following chapter.

Chapter 4

A new co-design methodology for processor-centric embedded systems in FPGAs

Vision-based multiple robot localization

For many applications designing the entire system in FPGAs or hardware is not the most practical solution. As discussed in the previous chapter, although the traditional HDL-based design flow is usefull to generate top performance tailored designs, it comes at the cost of time-consuming, complex and error-prone development. At the same time, even the most data intensive processing methods frequently contain sequential sections that are easier implemented in processors. These hardware/software co-designed solutions try to combine the best of both software and hardware worlds, making use of the ease of programming a processor while designing tailored hardware accelerator modules for the most time-consuming sections of the application. This not only accelerates the resulting system, as compared with the processor solution, but also allows savings in energy.

The inclusion of processor cores embedded in programmable logic has made FPGAs an excellent platform for these approaches. During 2011, the two major FPGA vendors (Xilinx and Altera) announced new chip families that combine powerful ARM processor cores with low-power programmable logic [14, 15]. While FPGA vendors have previously produced devices with on-board processors, the new families are unique in that the ARM processor system, rather than the programmable logic, is the center of the chip [16]. This strengthens the growing trend towards co-designed processor-centric solutions in FPGA-based chips. According to the 2012 Embedded Market Survey, 37% of the engineers that do not use FPGAs in their current designs confirmed that this trend would make them reconsider the matter [13].

The novelty of this approach together with its potential in the embedded system world makes academic research in hardware/software co-design in FPGA-based chips an important field. The main problem to tackle is time-consuming development. The rising complexity of these applications makes it difficult for designers to model the functional intent of the system in languages that are used for implementation, such as C or HDLs (Hardware Description Languages). Moreover, the traditional HDL-based FPGA design flow is time consuming and error-prone, as discussed in the previous chapter. Engineers regard the difficulty in programming FPGAs in HDL as an important reason for not using FPGAs [13]. This creates a great need for methodologies, languages and tools that reduce development time and complexity by raising the abstraction level of design and implementation [18] [19]. Advances have been made in high-level modeling using specific Unified Modeling Language (UML) profiles to simplify design. Besides, much work is being done in high-level synthesis tools, which translate constructs in C/C++ to HDL to simplify hardware implementation. However, there is still a great need for research in co-design methodologies, languages and tools, so that the recent combination of powerful processors with programmable logic can reach its full potential.

In this chapter, we present a co-design methodology for processor-centric embedded systems with hardware acceleration using FPGAs; and apply it to a global vision algorithm for the localization of multiple robots [77] [78] [24]. The main contributions in this chapter are:

- The proposal of a co-design methodology for the growing field of processor-centric embedded systems with hardware acceleration in FPGA-based chips. The goal is to achieve real-time embedded solutions, using hardware acceleration, but achieving development time similar to that of software projects. To reduce the development time, well established methodologies, techniques and languages from the software domain are applied, such as Object-Oriented Paradigm design, Unified Modelling Language and multithreaded programming. Moreover, to reduce hardware coding effort, semiautomatic C-to-HDL translation tools and methods are used and compared.

- The proposal of a simple and robust algorithm for multiple robot localization in global vision systems. This algorithm integrates an e-learning robotic laboratory for distance education that allows students from all over the world to perform experiments with real robots in an enclosed arena. Hence, the algorithm was specifically developed to work reliably 24/7 and to detect the robot's positions and headings even in the presence of partial occlusions and varying lighting conditions expectable in a normal classroom.

- The co-designed implementation of this algorithm following the proposed methodology. This solution processes 1600×1200 pixel images at a rate of 32 fps with an estimated energy consumption of 17mJ per frame. It achieves a 16× acceleration and 92% energy saving comparing to the most optimized embedded software solution. This solution presents -to the best of our knowledge- the first implementation of such an algorithm in FPGA-based hardware. It also shows the usefulness of the proposed methodology for embedded real-time image processing applications.

This chapter is organized as follows. Section 4.1 presents related work on the methodology proposal and global vision localization for multiple robots. Section 4.2 provides details of the proposed methodology. Section 4.3 describes the developed algorithm for multiple robot localization, while section 4.4 describes how the methodology was applied to this application to obtain an accelerated embedded solution. Section 4.5 presents and discusses acceleration, area and energy consumption results; finally Section 4.6 provides conclusions.

4.1 Related Work

This section discusses related work on the image processing algorithm, FPGA-based image processing solutions and the proposed methodology. Multiple robot localization in global vision systems is discussed in Section 4.1.1; and FPGA co-designed implementations of image processing applications related to robot localization are dealt with in Section 4.1.2. Key parts of the proposed co-design methodology -high level modeling and high-level synthesis- are discussed in Section 4.1.3.

4.1.1 Vision-based multiple robot localization

The most popular localization approaches in mobile robotics are based on GPS. However, its precision is often insufficient and its signal is not available indoors. One way to tackle the problem of GPS unavailability is to calculate a mobile robot position from its sensory measurements and an environment map [79]. Another solution involves installing localization infrastructure based on radio waves [36], multicamera systems [37] or similar technology. These alternative global positioning systems are more useful in multirobotic systems, which require complete information of the environment for efficient coordination of robot actions. Moreover, they allow the use of small robots unable to carry heavy sensory and computational equipment and are often used to provide “ground truth” data in robotic experiments.

A typical application that requires a small-scale positioning system is the FIRA and RoboCup contests, where teams of small mobile robots compete in a soccer-like game. The robotic soccer rules require the robots to carry identification marks with a specific team color. Thus, the robot dress design usually comprises a team color and a combination of individual colors for distinguishing each robot [80, 81, 82]. Since a robotic soccer round takes five minutes, the playing field illumination is strong, the teams can recalibrate their systems every time a goal is scored, and the robots move very fast, the visual localization approaches of the robotic soccer domain focus on real-time response and precision rather than robustness to variable lighting conditions. For example, [83] proposes to diminish the processing time and increase accuracy by localizing a color patch surrounded by a white border directly in the raw Bayer format image. The localization methods used in robotic soccer have influenced design of localization systems used in path and motion planning for robots in industrial settings [84], in teaching activities [85], and also in experimental setups for multi-agent collaborative tasks, such as platoon formations [86].

Although the setup of the localization system presented in this work is similar to the ones used in robotic soccer, its main requirement is a reliable and continuous (24/ 7) operation in lighting conditions of a normal classroom. It has to deal not only with uneven and variable illumination of the operation area (see Fig. 4.1) but also with dynamic objects in its field of view. Therefore, the robot dresses might be partially obstructed or visually connected to some other object which would cause color segmentation approaches to fail. In addition, the system does not require such high framerates as in robot soccer, because the robots move ten times more slowly than a typical soccer robot of a MIROSOT league. Hence, the system presented in this work is based on convolution rather than segmentation, because although convolution-based approaches are more computationally demanding, they are also more robust to realistic lighting conditions and offer good position estimation precision. To achieve real-time operation, this convolution-based algorithm needs to process the image at rates higher than the camera framerate. In this work, the real-time goal was to process 1600×1200 pixel images at a rate of 30 fps.

4.1.2 Co-designed FPGA solutions for image processing algorithms related to robotic localization

FPGAs are evolving as complex hardware/software platforms providing powerful embedded microprocessor cores. This enables several acceleration approaches to run image and video algorithms in real-time. Modern devices for image acquisition and visualization can benefit from these acceleration techniques. For this reason, much work can be found in the literature about hardware/software implementations on FPGA for computer vision applications. In particular, some vision algorithms that can be applied to robot localization, such as object tracking [87] or background subtraction [88], have their co-designed FPGA solutions. Furthermore, FPGA is particularly suitable for other robotic applications too [89].

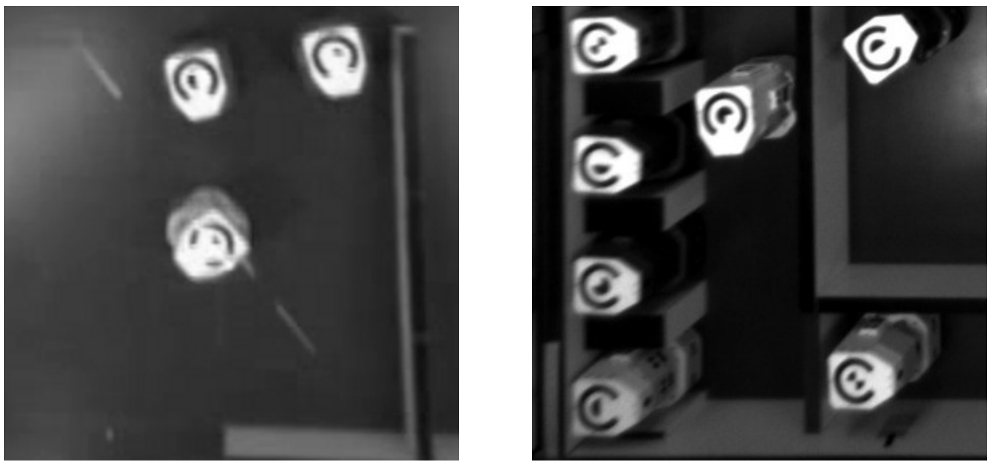

(a) Light stripes (b) Uneven illumination

Figure 4.1: Lighting conditions on the SyRoTek Arena.

In [87] a video object tracking application is presented. The application is based on a Sequential Monte Carlo method that uses color segmentation and is targeted to CPU/FPGA hybrid systems. Based on a multi-threaded programming model, the authors have developed a framework that allows design space exploration with respect to the hardware/software partitioning. Additionally, the application can adaptively switch between several partitioning states during run-time by means of partial reconfiguration in order to react to changing input data and performance requirements.

A hardware computing engine to perform background subtraction in video streams is presented in [88]. The embedded system detects people on low-cost FPGAs and is able to segment objects in sequences with resolution 768×576 at 50 fps with an estimated power consumption of 5 W.

In [89] authors propose an optical flow-based algorithm that estimates and compensates ego-motion to allow for object detection from a continuously moving robot. The system is implemented using a traditional HW/SW co-design approach on a Virtex-5 FPGA from Xilinx with an embedded PowerPC Processor. This implementation can process 31 fps at a resolution of 640×480 pixels.

Although the results in these three papers are competitive in flexibility, performance, and power consumption, as compared with state-of-the-art articles, in all of them the designer must design the solution and the hardware coprocessors in a traditional way. In our work, we propose a co-design methodology aimed at reducing development time, and show its applicability to the acceleration of embedded image processing algorithms. Moreover, to the best of our knowledge, our work presents the first co-designed FPGA-based solution to the problem of multiple robot localization in global vision systems. This solution points to the development of “intelligent cameras”: an embedded system including the image acquisition and processing on board so that there is no need to transfer the whole image to other computers.

4.1.3 High level modeling and high level synthesis

High level modeling and high level synthesis are two related areas that propose to raise the abstraction level of design and implementation in an effort to reduce the long development times associated with increasingly complex designs.

UML is a widely used language for system modeling in the software domain. It is a standard language of the Object Management Group (OMG), which has given rise to OMG standard UML profiles for SoCs, embedded systems and real time systems. The most closely related are UML for SoC, SysML and MARTE. The UML for SoC profile [90] was first introduced in 2009 and mainly defines structure diagrams through specific SoC stereotypes. SysML[91] is a general purpose modeling profile for systems engineering applications. It supports the specification, analysis, design, verification and validation of a broad range of systems. Although it is useful for embedded system design, it also includes support for things not specifically designed for this field, such as personnel or facilities. The UML MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [92] is the most adequate profile, adding capabilities to UML for model-driven development of Real Time and Embedded Systems (RTES). Since its standardization in November 2009, it has become the most recent industry standard in this field.

Based on these OMG standards, several profile extensions have been proposed to enable automatic code generation and/or extend modeling capabilities to more detailed aspects of embedded system design. For example, extensions of the MARTE profile for particular sub-domains include proposals for partial run-time FPGA reconfigurability [93] and for dynamic power management [94]. Authors of [95] propose a UML-ESL profile for cache usage analysis. The key to these approaches is not automatic code generation, but the extension of the modeling language to include the particular sub-domains.

To enable automatic code generation, the most common approach involves extending the UML profiles to model SystemC or VHDL structures. In [96] a UML2.0 profile for SystemC was first proposed, and this trend continued in several works like [97], [98] and [99]. In [100], the SysML profile is extended for SystemC constructs, enabling automatic SystemC code generation for simulation, and automatic VHDL code generation for synthesis. In all these approaches, UML designs are just representations of SystemC models, so low-level details such as ports, modules, or SystemC data types need to be modeled in the UML diagrams. In fact, this is what enables the automatic

SystemC code generation. In [101], authors propose to use a subset of the MARTE profile and generate rules to translate that subset to VHDL code. A key feature is that they use a subset of C++ as an Action Language to describe the behavior of states in state machines. Then, that C++ code is translated to VHDL using a high level synthesis tool.

In order to perform automatic HDL code generation, the aforementioned approaches restrict the modeling possibilities, for example with "one class-to-one module" synthesis, or the restriction that only complete objects -and not particular methods- can be mapped to hardware. Moreover, many SystemC or VHDL implementation details need to be modeled in the UML diagrams for the automatic translation to work. Our approach is to model the whole system using standard UML2.0 in an Object Oriented manner *prior* to hardware/software partitioning. This is particularly suitable for the processor-centric approach, in which most of the final application will be running in the processor. By modeling the whole application before hardware/software partition, the implementation details related to both hardware and software are abstracted away. This is crucial at this stage: it is what allows engineers to perform a good, modularized OOP design. This type of design will later on enable the engineer to use profiling tools and find precisely which OOP methods need to be accelerated by hardware. In this way, useless translations to hardware can be prevented. Moreover, by using standard UML2.0, many tools from the software domain that have been developed for years can be used, such as Sparx Enterprise Architect [102]. These tools support not only explicit modeling of parallel algorithms, but also automatic C++ headers and code generation from the UML diagrams. Many tailored UML profiles, although better suited for modeling certain aspects of particular domains, are still behind in tool support.

Complementing the high-level modeling approaches, another area of much development is High Level Synthesis tools. These include automatic or semiautomatic tools to translate constructs using a subset of languages like C/C+ to HDL code. Examples of open-source semiautomatic tools include ROCCC [103], SPARK [104] and DWARV[105]. There are also many proprietary tools such as DIME-C [106], CatapultC [107] or AutoESL [108]. All of them require rewriting effort on the original methods to particular coding styles and C/C++ language subset, and hardware knowledge in order to generate optimized HDL. Since they have specific interface definitions, hand-coded interface modules are required to integrate the automatic-generated modules into the complete system. A comparison of open-source tools can be found in [109], and an excellent study of the proprietary AutoESL tool can be found in [110]. Another approach is to hand-code the modules, but using a clear coding method. In [111] the two-process method is proposed: a Behavioral VHDL design applied on several designs made for the European Space Agency, including the LEON3 processor [112]. In [113], authors show that the method decreased man-years, code lines and bugs in many important projects, and that it is well suited for implementing OOP methods, thereby creating short and readable code. Although the two-process method is not a (semi)automatic tool, the coding guidelines create a schema that could be the starting point for such a tool. In this work, we compare three different ways to translate the C++ class methods that need to be accelerated by hardware to HDL: open-source tool ROCCC, proprietary tool AutoESL, and the two-process VHDL coding method.

A key feature of the proposed methodology is the union of a good modularized OOP design captured in UML and implemented in C++ that makes it possible to find precisely which methods need to be accelerated by hardware; with semi-automatic tools or guidelines to translate these C++ methods to HDL. This union reduces hardware coding effort, traditionally the most time-consuming and error-prone stage of a hardware-accelerated application.

4.2 Methodology

The proposed methodology has four broad stages: A) OOP Design; B) C++ Implementation and Testing in a general purpose processor; C) Software migration, optimization and hardware/software partition; and D) Translation, testing and integration of each hardware module in the final embedded platform (see Fig. 4.2). In this Section, the steps of each stage are described, and useful tools are presented.

4.2.1 OOP Design

In this stage, an OOP approach and the use of UML language for modeling is proposed. As already stated, these are widely used in the software community, and many modified UML approaches for hardware specific applications exist. Hence, many software and hardware engineers are familiar with these techniques and at least some of the associated tools. The use of well established techniques and tools is important to help reduce the design effort by raising the abstraction level, while not imposing the need for engineers to learn new languages, methods and tools.

The OOP design includes concepts such as encapsulation, inheritance, messaging, modularity, polymorphism, and data abstraction. In the proposed approach, abstraction, encapsulation and modularity are particularly important. A modular design -in which the responsibilities of each class are clearly identified, and concise methods are created- is a key part of the methodology. This kind of design assists the engineer in finding the exact methods that need to be accelerated by hardware using profiling tools. This helps to reduce hardware

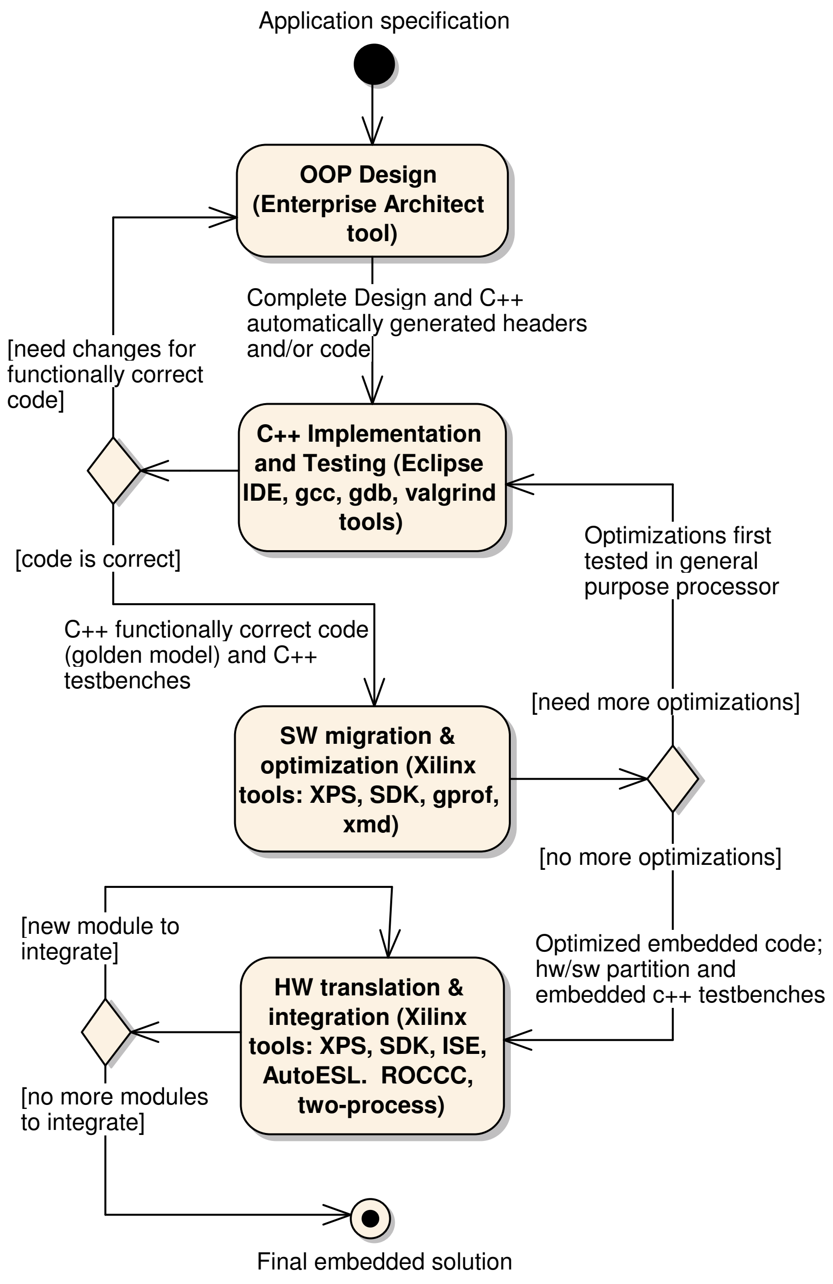

Application specification
OOP Design (Enterprise Architect tool)
Complete Design and C++ automatically generated headers and/or code
[need changes for functionally correct code]
C++ Implementation and Testing (Eclipse IDE, gcc, gdb, valgrind tools)
[code is correct]
Optimizations first tested in general purpose processor
C++ functionally correct code (golden model) and C++ testbenches
[need more optimizations]
SW migration & optimization (Xilinx tools: XPS, SDK, gprof, xmd)
[no more optimizations]
[new module to integrate]
Optimized embedded code; hw/sw partition and embedded c++ testbenches
HW translation & integration (Xilinx tools: XPS, SDK, ISE, AutoESL. ROCCC, two-process)
[no more modules to integrate]
Final embedded solution

Figure 4.2: Methodology overview

coding effort because no useless translations to hardware are done.

At this stage, the possible parallelizable sections of the algorithms can be identified and modeled. Standard UML offers several ways to model parallel thread-like behavior. One way is to model threads in the structural design as Active classes. Active objects (instances of an active class) model the concurrent behavior of real world objects, and can own an execution thread and initiate control activities. Threads can also be modeled in behavioral diagrams. Activity diagrams include specific fork/join bars to illustrate thread behavior. These diagrams also include swimlines to show which objects perform the activities modeled. Sequence diagrams also include `par` sections to indicate that the interactions in those sections may be executed in parallel or in any order. The most appropriate diagrams to capture the intended design vary in different applications. Several profiles have been proposed to model designs for cluster, parallel and heterogeneous computer architectures, which are beyond the scope of this work.

The complete design is done using Spark's Enterprise Architect [102], a comprehensive UML analysis and design tool. This tool allows for structural design - i.e, classes- and behavioral design -the description of the system behavior by interaction sequences, state machines, activity diagrams, etc. It automatically generates documentation and the class skeleton in several languages (C++, Java, Python, etc.). It can also generate automatic code in these languages, provided there is enough detail in the UML diagrams. The tool also allows reverse engineering, making it possible to reflect in the UML design changes made during implementation.

The output of this stage is the Structural and Behavioral design of the solution, and also the class skeleton with members, methods and all comments in C++ files.

4.2.2 C++ Implementation and Testing

The second stage is coding and testing the designed solution in a general purpose processor. Although any programming language may be used, we prefer C++ for several reasons. It is a widely used language with full OOP support, allowing to port the UML design perfectly. It also has ample support for running in different embedded processors. Thus, the software solution developed in this stage serves not only as a reference model but also as part of the final software that will run in the embedded processor. Finally, C++, along with C and SystemC, is one of the strong contenders for the best input language for High-level synthesis tools [19]. This means that much work is being done on tools to automatically translate C++ constructs to HDL.

The first step is to code all the methods of the class skeleton exported from the Enterprise Architect, using the behavioral design from the previous stage as a guideline. The code for some or all of the methods might be automatically generated. At this stage, the POSIX Thread API is used for multithreaded programming. This API is the most common interface for thread programming, with support in virtually all OS, including embedded OS such as Linux or Xilinx's lightweight embedded kernel xilkernel.

Next, the testing step includes both functional and correctness tests. Correctness tests assure that the available resources are correctly used -e.g, that the solution does not have any memory leaks. This testing stage uses the tools and resources available in a general purpose processor, making it much easier to obtain a functionally correct complete solution. Any changes that may be needed at this point to get to a functionally correct solution need to be reflected in the UML OOP Design in the previous stage. To do this, a very useful capability of the Enterprise Architect and many other UML capture tools is reverse engineering, so the UML design can be captured from C++ code. Tests developed at this stage can also be used for testing in the embedded platform later on. Functional verification against the specification can also be done.

The tool used is the Eclipse IDE, with the GCC compiler and GDB debugger. The GNU valgrind is used to assess correctness in the memory usage. All of them are widely used open source tools.

The output of this stage is a correct, executable solution of the problem running in a general purpose processor. This software solution may be used both as a reference model and as part of the final software that will run in the embedded processor, given that it uses the right language for codification, as well as widely ported libraries for OS services, such as threads or memory management. This reduces the software coding effort.

4.2.3 Software Migration, optimization and HW/SW partition

In this stage, the whole software solution must be migrated to the final embedded processor in a testing environment, in order to fully characterize the resources needed in the embedded platform, perform all possible software optimizations, and decide which parts of the system need to be accelerated by hardware. See Fig. 4.3 for an Activity Diagram of the steps in this stage.

The first step is to characterize the resources needed to execute the complete software solution in the embedded platform. This includes an analysis of the amount of memory needed, the different peripherals and their interfaces, and a general measure of the processor usage. It must also be determined whether an embedded OS will be necessary or not.

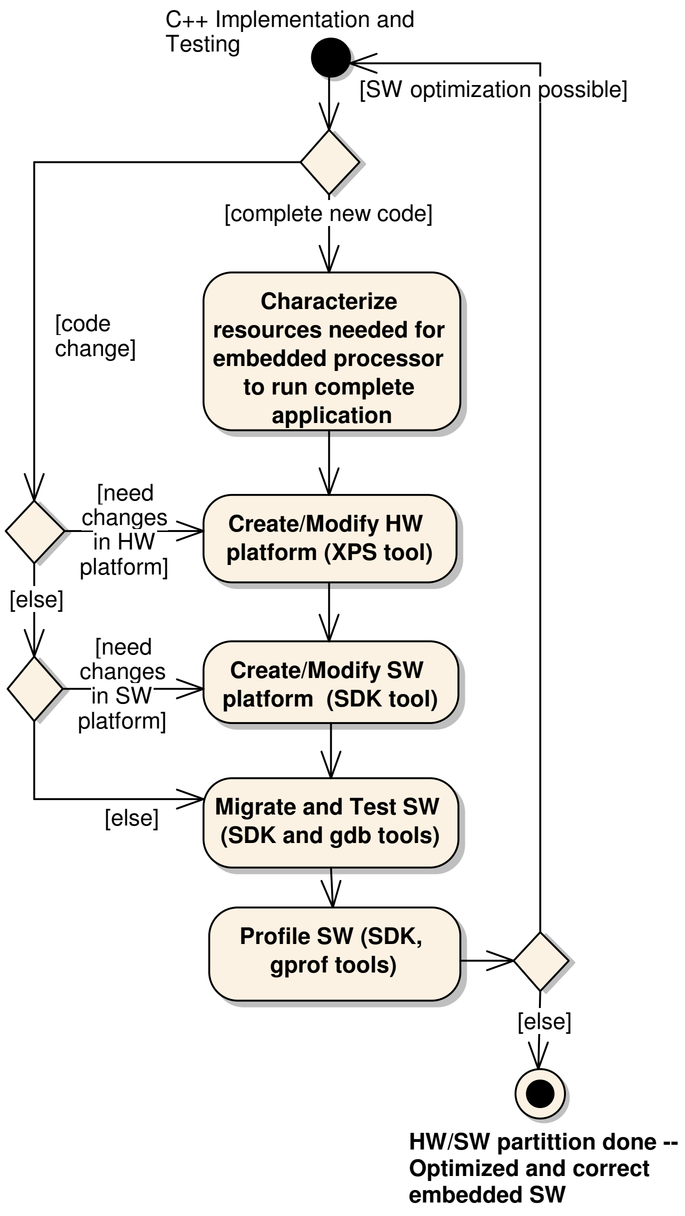

C++ Implementation and Testing
[SW optimization possible]
[complete new code]
[code change]
Characterize resources needed for embedded processor to run complete application
[need changes in HW platform]
Create/Modify HW platform (XPS tool)
[else]
[need changes in SW platform]
Create/Modify SW platform (SDK tool)
[else]
Migrate and Test SW (SDK and gdb tools)
Profile SW (SDK, gprof tools)
[else]
HW/SW partittion done -- Optimized and correct embedded SW

Figure 4.3: Activity Diagram for the software migration, optimization and HW/SW partition stage

The second step is to configure and generate the hardware platform to execute the whole software solution in the embedded processor. The configuration of the embedded processor includes processor frequency and bus frequency, debugging and profiling configurations, coprocessors such as floating point units, internal memories and caches, interrupts, buses for peripherals, etc. The output of this step is all the necessary hardware so that the embedded processor may run, including inputs and outputs for the application, for debugging and for profiling.

The third step is to configure the software platform in the embedded processor. This includes the choice and configuration of operating system (if any), input/output libraries, debugging and profiling libraries and memory map. The output of this step is a software platform configured to run the whole software solution in the embedded platform.

The fourth step is to migrate the software solution to the embedded platform. The changes in the code needed for this migration depend on the embedded processor, needed peripherals and software platform. For a C++ solution with standard libraries such as `malloc` or `pthread`, most embedded operating systems and processors need only slight changes in the library interfaces to communicate with peripherals and memory. The output of this step is a complete executable software solution in the embedded platform.

The final step for this stage is to profile the software solution running in the target embedded processor. The profiler is useful to point out the methods that are time-consuming, although it is important to note that it may be impossible to profile some libraries that have not been compiled with the appropriate flags and also that profiling does not work with multi-threaded environment. For more reliable time measures in these settings, the internal processor's clock or some specially included external timer may be needed. Using this profiling information, the most time consuming methods are pointed out.

If possible, software optimizations for the particular architecture of the embedded processor can be done (e.g. the use of fixed point arithmetics in a non-FPU processor). Method optimizations that change the precision of the algorithms but save operations can also be studied. Taking into account that the tools for debugging and functional tests in a general purpose processor are much better than in the embedded processor, the changes are first performed in the C++ code in the general purpose processor. This affects the previous stage of the methodology (C++ Implementation and Testing) and probably even the OOP Design stage, as shown in the alternative branch in Fig. 4.3. As can be seen in that figure, the changes in code may or may not call for modifications in the previously generated hardware and software platforms. For example it may require to add Block RAM memory (i.e., hardware platform change) or a new software library (i.e., software platform change). This is an iterative stage: the procedure might be repeated as long as the real-time performance is not reached, and more software optimizations may be done.

Once no more software optimizations are possible, the final profiling points to the methods that need to be accelerated by hardware to achieve the required performance. Hence, the output of this step is the final hardware/software partition.

The tools for this stage depend on the embedded processor and FPGA, development boards and associated tools to be used. In this case study, the Avnet Virtex4-FX12 evaluation board was used. This board includes a Xilinx Virtex4 FPGA with an embedded PowerPC405, so Xilinx's Embedded Development Kit [114] is used. This kit includes the Xilinx Platform Studio (XPS) tool for hardware platform configuration and generation, and the Software Design Kit (SDK) used for software platform configuration and embedded software migration. SDK is an Eclipse-based tool that comes with a special GDB debugger and a special integrated GNU gprof profiler.

The output of this stage is the hardware/software partition and an optimized, functionally correct software solution running in the embedded processor.

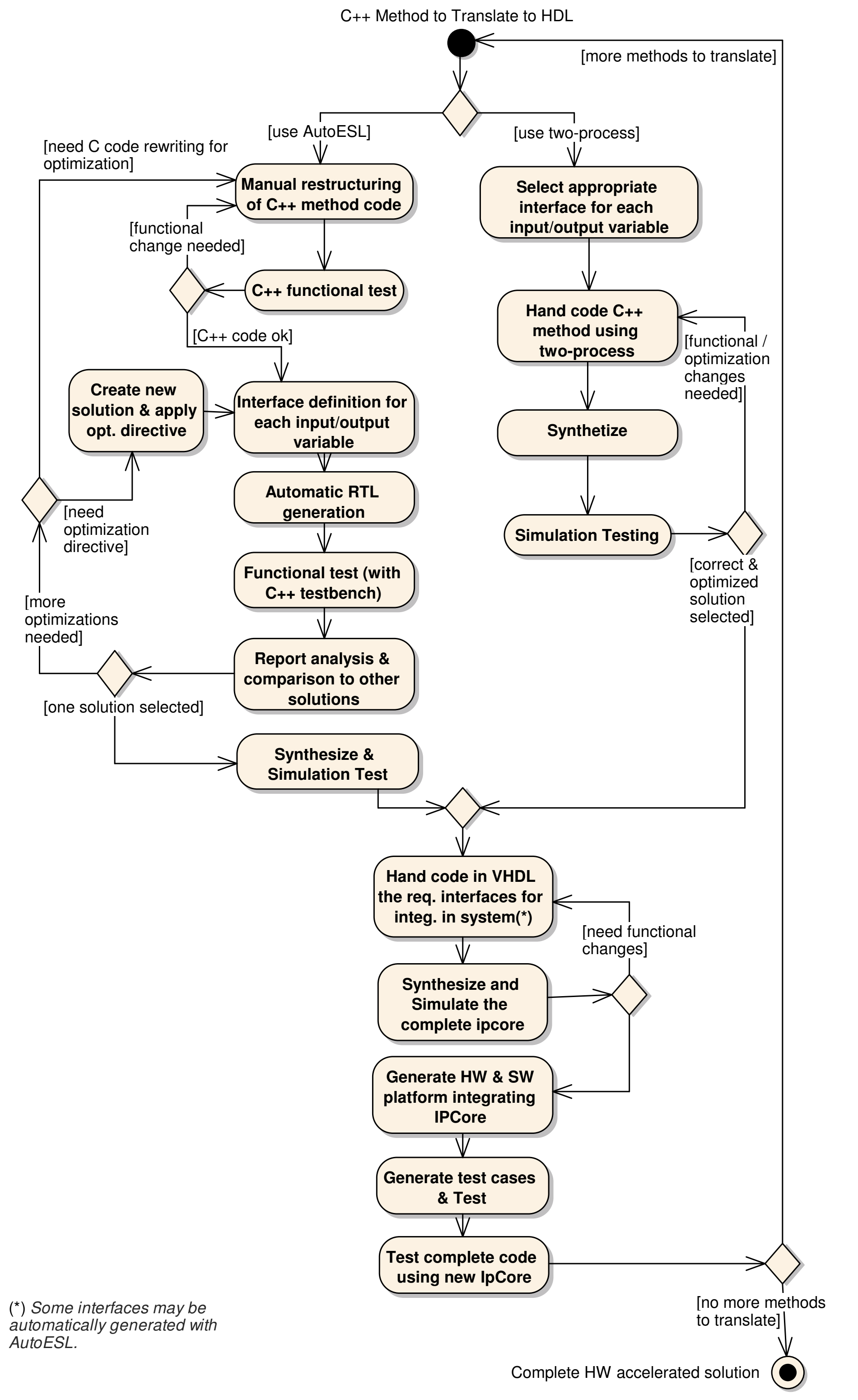
4.2.4 Hardware translation, testing and integration

The fourth stage is to translate the C++ methods that need to be accelerated by hardware to HDL, to test and integrate them into the system, by implementing the needed interfaces in hardware and software. In Fig. 4.4 the Activity diagram of this stage can be seen.

In this work, three ways to translate the selected C++ methods to HDL are evaluated: the ROCCC tool, the AutoESL tool and the two-process coding method. These are shortly described in the following subsections. The entity and its interfaces are tested using simulation and testbenches, which change depending on the tool used. The output of this step is the hardware IP core that passes all the simulated tests.

The following step involves performing unit tests for the IP core implemented in the embedded platform. Hardware and software platforms needed to run these tests in the embedded processor must be generated, following the steps mentioned in Section 4.2.3. Then, unit test cases developed in Section 4.2.2 must be migrated. New test cases that take into account the particular details of the hardware implementation and its interaction with the embedded processor must also be done. The output of this step is the tested hardware IP core.

Finally, integration tests must be done to see if the whole system with the added hardware module is still functionally correct. For this purpose, the



C++ Method to Translate to HDL
[more methods to translate]
[use AutoESL]
[use two-process]
[need C code rewriting for optimization]
Manual restructuring of C++ method code
[functional change needed]
C++ functional test
[C++ code ok]
Create new solution & apply opt. directive
Interface definition for each input/output variable
Automatic RTL generation
[need optimization directive]
Functional test (with C++ testbench)
[more optimizations needed]
Report analysis & comparison to other solutions
[one solution selected]
Synthesize & Simulation Test
Select appropriate interface for each input/output variable
Hand code C++ method using two-process
[functional / optimization changes needed]
Synthetize
Simulation Testing
[correct & optimized solution selected]
Hand code in VHDL the req. interfaces for integ. in system(*)
[need functional changes]
Synthesize and Simulate the complete ipcore
Generate HW & SW platform integrating IPCore
Generate test cases & Test
Test complete code using new IpCore
[no more methods to translate]
Complete HW accelerated solution
(*) Some interfaces may be automatically generated with AutoESL.

Figure 4.4: Activity Diagram for the hardware translation, testing and integration stage, including the choices to use the AutoESL tool or the hand-coded two-process

system tests developed in Section 4.2.3 can be migrated. The output of this step is the tested system, including the added hardware module.

Since only part of the system is accelerated by hardware, it is important to keep in mind the theoretical maximum expected improvement to the overall system. That theoretical maximum S'_{max} can be calculated by Amdahl's law. This law is concerned with the speedup achievable from an improvement to a computation that affects a proportion P of that computation, where the improvement has a speedup of S. For example, if 30% of the computation may be the subject of a speedup, P will be 0.3; if the improvement makes the portion affected twice as fast, S will be 2. Amdahl's law states that the overall speedup S' of applying the improvement will be:

$$S' = \frac{1}{(1-P) + P/S} \tag{4.1}$$

The theoretical maximum S'_{max} can be calculated by assuming that the portion P will have a infinity speedup ($S = \infty$). This maximum helps to indicate when translating a part to hardware or increasing parallelism is not worth the effort, since the impact on the complete application will not be significant enough. This equation is also useful to check if the measured complete accelerations are consistent with the measured partial accelerations.

The tools for this stage also depend on the embedded processor and FPGA. In this case study, Xilinx's Embedded Development Kit is used for hardware and software configuration and development. Xilinx's ISE is used for hardware modules implementation, and Xilinx's ISim simulator is used for unit simulation and testing. The AutoESL and ROCCC semi automatic tools are used to generate HDL code for the C++ methods.

The output of this stage is the tested system, including the complete software and all the integrated hardware modules.

The ROCCC tool

The ROCCC compiler tool by Jacquard Computing is designed to create hardware accelerators from a subset of C. The hardware generated is not intended to replace entire software applications, but instead provide an application speedup by replacing critical regions in software with a dedicated hardware component. Users code hardware module in C, and then use these modules in larger programs. ROCCC generates platform independent VHDL code from C descriptions.

In order to run the VHDL code on a particular platform, users must create glue code that attaches the automatically generated code to the system. There are specific guidelines as to how to present the data to the automatically generated hardware module, so the glue code must follow those guidelines. The tool implements several optimizations -loop unrolling, systolic array generation, pipeline, etc- which must be configured by the user in order to achieve an optimized code.

The ROCCC GUI is a plugin designed for the Eclipse IDE that works on both Linux and Mac systems. After VHDL code has been generated, the modules need to be imported to Xilinx ISE or similar in order to synthesize the design for the target FPGA.

The ROCCC is an open-source development, so download, documentation and examples are available for free from Jacquard Computing.

The AutoESL tool

The AutoESL is a High Level Synthesis tool developed since 2006 by the AutoESL company, which was acquired by Xilinx in 2011. Xilinx has released this tool and also used it as a base for the new Vivado Design Suite. AutoESL takes as its input a C, C++ or SystemC description of functionality at a high level of abstraction. It then generates a device-specific Verilog or VHDL description of a hardware implementation.

AutoESL has several optimization directives that can be applied to a given design. From a given C/C++ code, different HDL solutions can be achieved using different directives. AutoESL provides reports that compare each solution in terms of timing, area and power consumption of the generated design. Although these measures are estimates, in a few minutes they enable the designer to see the achieved solution after applying an optimization directive, and also to compare it with other possible solutions.

The AutoESL GUI is an eclipse-based complete tool. It can be used to code and test the C/C++ original code. Moreover, the C/C++ testbench can also be used to test the generated design. In order to create some optimizations (like the use of bit-accurate variables) special C/C++ types exist and the C/C++ code may need changes. The Xilinx ISE tool is integrated into AutoESL, so the design can be synthesized from within the tool. Moreover, for some interface types, AutoESL can create automatic master or slave ports for Xilinx EDK's standard buses. This greatly simplifies the integration of the generated modules. However, for other types of interfaces -such as particular memory access patterns or other buses- the interface modules need to be hand coded.

AutoESL is a proprietary tool, with a substantial price per license and many years of development from both previous the company (called AutoESL) and Xilinx.

The two-process design method

Two-process is a structured Behavioral VHDL design method that is particularly well suited for implementing OOP methods. The main goals for this design method are to provide a uniform algorithm encoding, increase abstraction level and improve readability and bug finding. These goals are reached by simple means: using record types in all ports and signal declarations, using only two processes per entity, and high-level sequential statements to code the algorithm.

The biggest difference between a program in VHDL and standard programming language, such as C, is that VHDL allows concurrent statements and processes that are scheduled for execution by events rather than in the order they are written. This reflects indeed the data-flow behavior of real hardware, but becomes difficult to understand and analyze when the number of concurrent statements passes some threshold. On the contrary, analyzing the behavior of programs written in sequential programming languages does not become a problem even if the program tends to grow. These programs are easier to understand because there is only one thread of control, and execution is done sequentially from top to bottom.

The two-process method only uses two processes per VHDL entity: one process that contains all combinational (asynchronous) logic, and one process that contains all sequential logic (registers). Using this structure, the complete algorithm can be coded in sequential (non-concurrent) statements in the combinational process while the sequential process only contains registers, i.e., the state.

In this way, methods of classes that need to be migrated to hardware can be translated as one VHDL entity, coding the whole algorithm using very similar sequential statements to the ones used in C++. For this stage, a correct, modular OOP design is vital, so that every method to be translated is short and concise. The key is not having to rethink the algorithm to accommodate the concurrent nature of hardware, but translating the algorithm from C/C++ syntax to VHDL syntax in a sequential manner. This is a Behavioral level HDL coding method, i.e. the precise RTL architecture is not defined in the code.

The two-process method has shown to greatly decrease man-years, code lines and bugs. In [113] some comparisons for ESA projects can be found, including a comparison between the ERC32 memory controller MEC (designed with ad-hoc methods) and the whole LEON3 processor designed with the two-process method. Even though the LEON3 processor is a 100k gates design and the MEC is only 30k gates; the LEON3 took 2 man-years, 15000 code lines and had no bugs in the first silicon, while the MEC took 10 man-years, 25000 code lines and had to go through 3 silicon iterations.

All these reasons make the two-process method a good choice for translat-

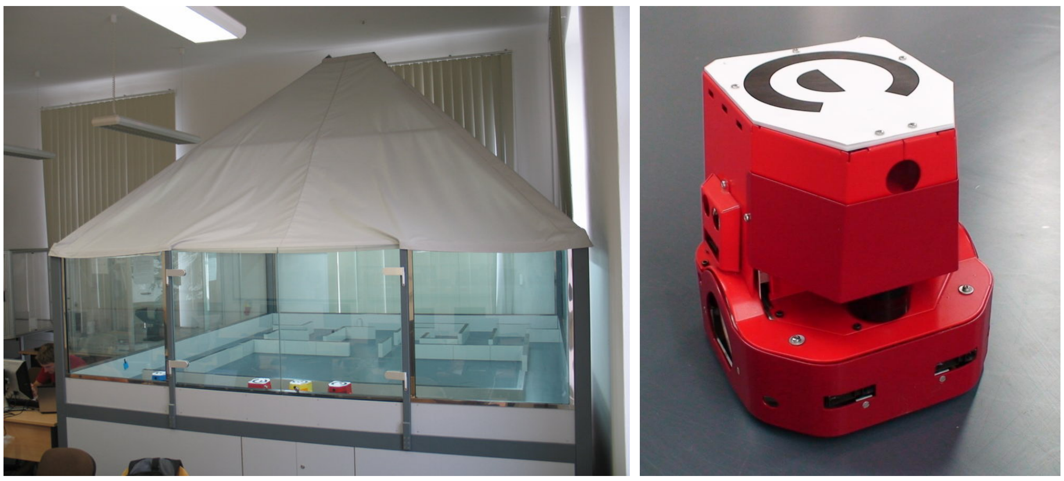


Figure 4.5: SyRoTek arena and robot with dress arc.

ing the C++ object methods to VHDL.

4.3 Multiple Robot Localization

The System for Robotic Teleeducation (SyRoTek) [115] is an e-learning platform for distance education of artificial intelligence, control engineering, motion planning and other fields related to mobile robotics. It has been successfully used in education and research by institutions across Europe and the Americas. The platform consists of fourteen autonomous mobile robots operating on a 24/7 basis in an enclosed area with dynamically reconfigurable obstacles (see Fig. 4.5). Users anywhere around the world can upload their algorithms to the robots, gather their sensorymotor data and analyze their behavior.

An important component of the platform is a visual localization system, which determines position and heading of each robot in the arena. It consists of a dedicated PC, an overhead camera and unique identification patterns placed on the individual robots. Due to the system 24/7 operation, it is desirable to implement the localization system on an embedded device with low power consumption. The real-time constraint for the system is that it should be able to process 1600×1200 pixel images at a rate of 30 fps.

4.3.1 Method overview

The $1600{\times}1200$ gray scale image that is provided by the localization system camera is processed in four consecutive steps. In the first step, the image is transformed to make the arena appear as a rectangle aligned with the image

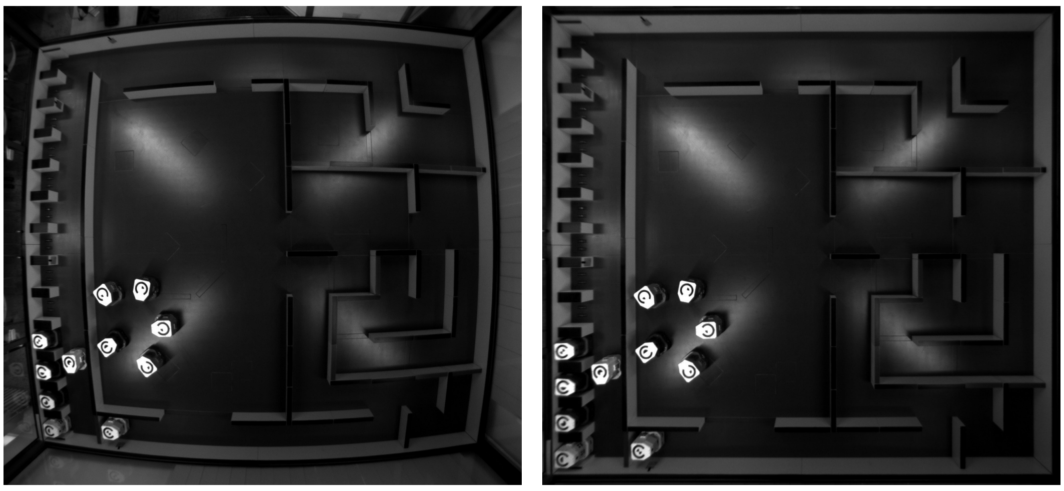


Figure 4.6: Original and rectified arena image.

edges. The rectified image is then convolved with a 40×40 annulus pattern, and local maxima of the convolution are found. After that, endpoints of the robot dress arcs are found to determine the robot heading. Finally, binary identification tags at the robot dress centers are decoded.

4.3.2 Image rectification

The purpose of this step is to remove radial and perspective distortion of the arena image, so that it appears as a rectangle aligned with the image edges (see Fig. 4.6). The radial distortion, which is caused by camera lens imperfection, has been modeled by the method described in [116], and its parameters have been established by using the MATLAB calibration toolbox [117]. The perspective transformation, which results from the camera misalignment, was modeled by a 3×3 projective transformation matrix. This matrix was calculated from the positions of the arena corners in the undistorted and rectified image by means of solving a set of linear equations.

Using the established parameters of both transformations, a look-up table mapping pixel coordinates of the rectified and captured image was generated. The look-up table allows to perform both transformations in a single step, thus reducing the number of floating point operations per pixel of the generated image. Since the mapping of the pixels is not one-to-one, the brightness of each rectified image pixel was calculated from four pixels of the captured image by bilinear interpolation.

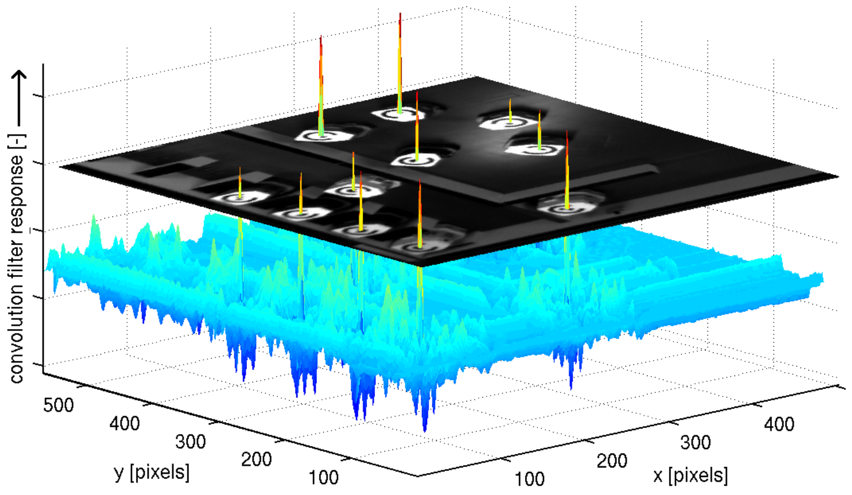


Figure 4.7: Rectified image part and convolution filter response

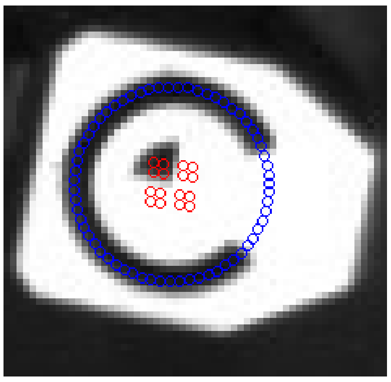
4.3.3 Position estimation

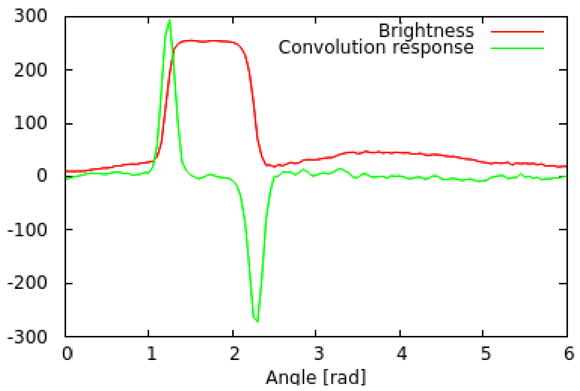
The circular shape of the robot dress outer arc allows to decompose the robot localization to 2D position estimation followed by orientation calculation. To determine the robot position, the rectified image is convolved with an 40×40 pixel annulus pattern with outer and inner diameter equal to the sizes of the dress arc (see Fig. 4.5). The response of the convolution filter is then searched for local maxima, which indicate robot positions in the arena (see Fig. 4.7).

Given the limited robot speed, camera resolution and fps, the convolution of the entire image is not necessary. During standard system operation, the convolution is performed only in a neighborhood of each robot's position in the previous frame. This also means that the image rectification can be performed only in the areas where convolution is about to be calculated. Convolution of the entire image is performed only when the system starts, resumes from idle state, or if the robots are removed or added to the arena. The system administrator can also force to search robots in the entire image in the case the tracking algorithm fails.

4.3.4 Orientation calculation

As soon as positions of the robots are known, the robot orientation is established from the dress arc. First, positions of several sampling points along the dress ring are calculated. The brightness of each point is estimated from its neighbouring pixels by bilinear transformation, constructing a vector that contains the brightness of the pixels on the dress ring. The vector is normalized and convolved with a kernel corresponding to the expected brightness gradient


(a) Sampled pixels



(b) Convolution response

Figure 4.8: Orientation and identification process

at the endpoints of the dress arc (see Fig. 4.8).

Minimum and maximum of the convolution correspond to the positions of these endpoints, and therefore, orientation of the robot is computed as the average of the argmin and argmax of the resulting vector. To verify the angle estimation, the distance of the found minimum and maximum is compared to the dress arc angle.

4.3.5 Robot identification

The last step establishes the robot number by decoding a binary tag in the robot dress center. Once the position and orientation of the robot is known, brightness of sixteen pixels around its center is measured (see Fig. 4.8). Average brightness of each pixel group is then calculated and a threshold separating white and black values is established (each identification tag has at least one white and one black segment). The calculated brightness are thresholded and the sequence of the four results encodes the robot number. Since the robots are tracked, the identification is performed only once. After that, the identification is run only to verify correctness of the localization algorithm.

4.4 Hardware/Software co-designed solution

In this section, the steps of the proposed methodology were applied to the multiple robot localization problem in order to achieve the accelerated embedded solution.

4.4.1 OOP Design

In this stage, an OOP Design was made and expressed in UML diagrams. Also, parallelizable sections were identified. The overall structural design of the solution is shown in Fig. 4.9.

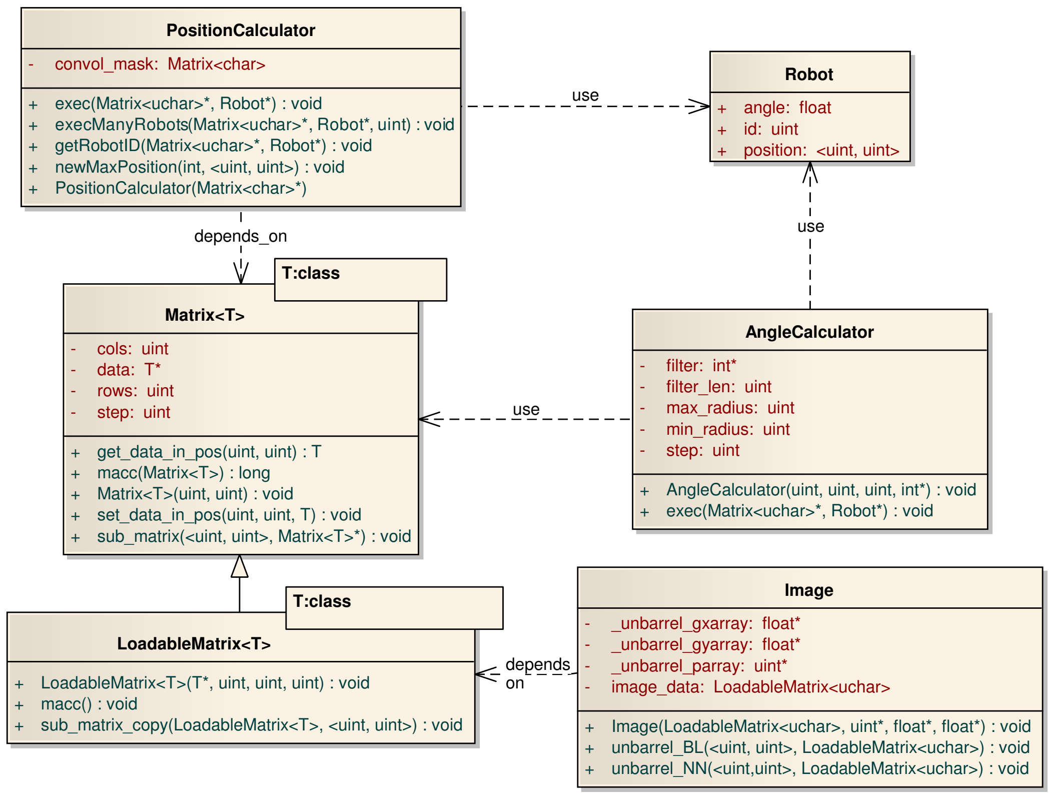
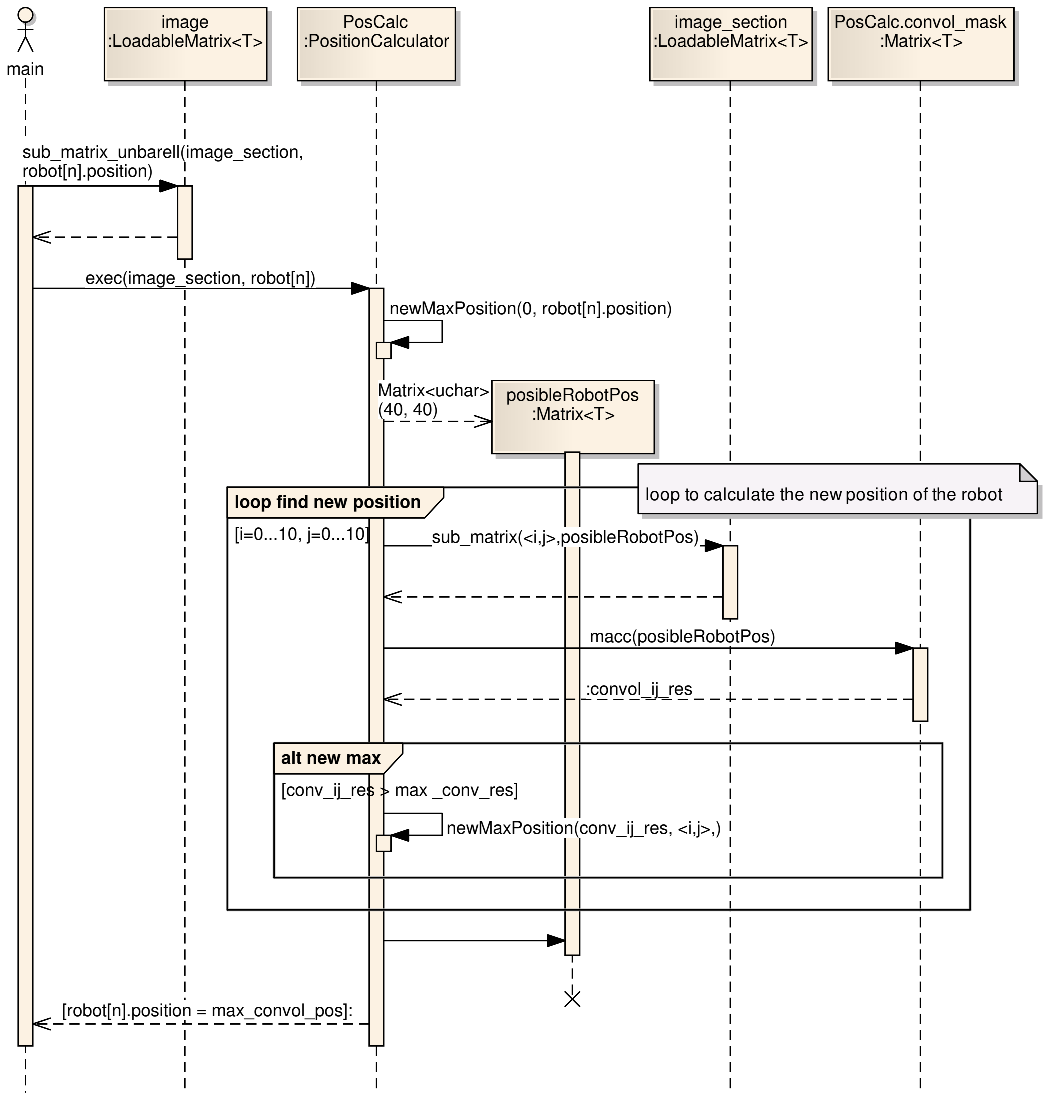

PositionCalculator
- convol_mask: Matrix<char>
+ exec(Matrix<uchar>*, Robot*) : void
+ execManyRobots(Matrix<uchar>*, Robot*, uint) : void
+ getRobotID(Matrix<uchar>*, Robot*) : void
+ newMaxPosition(int, <uint, uint>) : void
+ PositionCalculator(Matrix<char>*)
use
Robot
+ angle: float
+ id: uint
+ position: <uint, uint>
use
depends_on
T:class
Matrix<T>
- cols: uint
- data: T*
- rows: uint
- step: uint
+ get_data_in_pos(uint, uint) : T
+ macc(Matrix<T>) : long
+ Matrix<T>(uint, uint) : void
+ set_data_in_pos(uint, uint, T) : void
+ sub_matrix(<uint, uint>, Matrix<T>*) : void
use
AngleCalculator
- filter: int*
- filter_len: uint
- max_radius: uint
- min_radius: uint
- step: uint
+ AngleCalculator(uint, uint, uint, int*) : void
+ exec(Matrix<uchar>*, Robot*) : void
T:class
LoadableMatrix<T>
+ LoadableMatrix<T>(T*, uint, uint, uint) : void
+ macc() : void
+ sub_matrix_copy(LoadableMatrix<T>, <uint, uint>) : void
depends on
Image
- _unbarrel_gxarray: float*
- _unbarrel_gyarray: float*
- _unbarrel_parray: uint*
- image_data: LoadableMatrix<uchar>
+ Image(LoadableMatrix<uchar>, uint*, float*, float*) : void
+ unbarrel_BL(<uint, uint>, LoadableMatrix<uchar>) : void
+ unbarrel_NN(<uint,uint>, LoadableMatrix<uchar>) : void

Figure 4.9: Structural Design

The `Robot` class contains the information of each robot, i.e., position, heading and id. The class `PositionCalculator` calculates the new position of a robot. For this task, the `exec` method takes as parameters a Robot and a neighborhood of 50×50 pixels of the image around its position in the previous frame. By convolving the 40×40 `conv_mask` in that image section, the new position of the robot is found. The sizes for the convolution masks and neighborhood are configurable. In Fig. 4.10, a sequence diagram for the calculation of the new position of one robot can be seen. The class `AngleCalculator` calculates the new heading of a robot, by sampling the arc points on the 40×40 unbarreled section of the image where the robot actually is. The identification of the Robot is done with the method `getRobotID` of the class `PositionCalculator`.

Matrix operations are performed by a `Matrix` class. Since most matrices are sub-matrices of bigger ones (e.g., an image section is a sub-matrix of image), memory is only dealt with in very specific moments. The `Loadable`

-`Matrix` class inherits from `Matrix` and performs actual memory movements. Finally, the `Image` class depends on `LoadableMatrix`, as it has an instance of that class to contain the image data. The `Image` class knows about image undistortion operation, implementing both bilinear and nearest neighbor interpolation for comparison purposes.



main
image :LoadableMatrix<T>
PosCalc :PositionCalculator
image_section :LoadableMatrix<T>
PosCalc.convol_mask :Matrix<T>
sub_matrix_unbarell(image_section, robot[n].position)
exec(image_section, robot[n])
newMaxPosition(0, robot[n].position)
Matrix<uchar> (40, 40)
posibleRobotPos :Matrix<T>
loop find new position
loop to calculate the new position of the robot
[i=0...10, j=0...10]
sub_matrix(<i,j>,posibleRobotPos)
macc(posibleRobotPos)
:convol_ij_res
alt new max
[conv_ij_res > max _conv_res]
newMaxPosition(conv_ij_res, <i,j>,)
[robot[n].position = max_convol_pos]:

Figure 4.10: Sequence Diagram for the calculation of the new position of one robot

It is important to note that there are a variety of possible structural designs for the solution. In the software community, much work has been done in design pattern, starting in 1994 from the foundational book “Design Patterns: Elements of Reusable Object-Oriented Software” [118]. Evaluating which patterns are applicable and useful in embedded software, and which are not, is a promising and vast research area.

At this stage the possible parallel sections need to be identified. Analyzing the algorithms allows one to see that the different stages in finding a robot’s new position and orientation are not parallelizable. For the position calculation, the image section the robot was in needs to be already undistorted, and then for angle calculation the robot's new position is needed. However, the complete process for each robot is independent of any other robot, so the complete process can be done in parallel for each of the fourteen robots, for example in fourteen different threads. These are shown in the activity diagram of Fig. 4.11, where the bars show the fork and joins, and the horizontal swimlanes show which class is responsible for each activity.

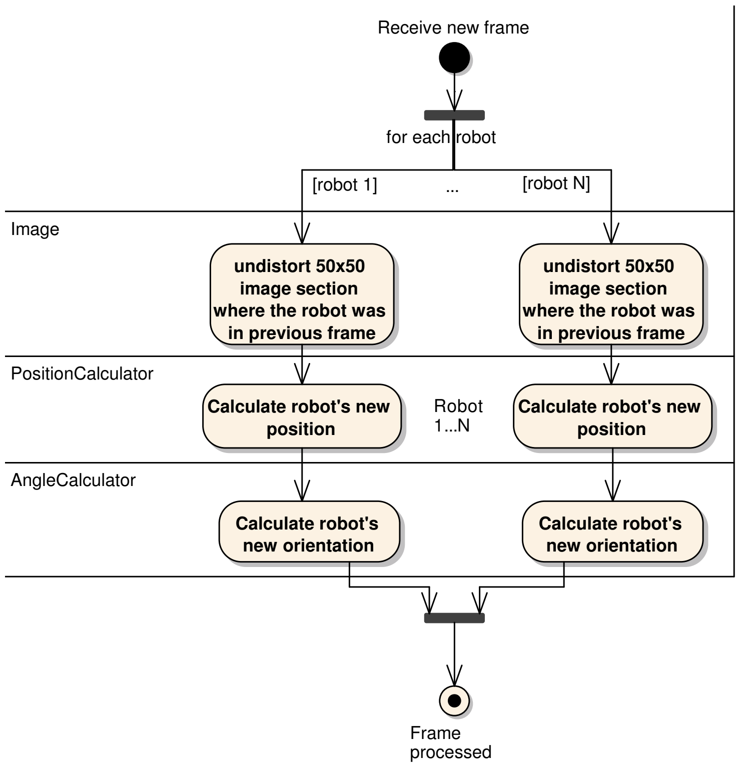

Receive new frame
for each robot
[robot 1]
...
[robot N]
Image
undistort 50x50 image section where the robot was in previous frame
undistort 50x50 image section where the robot was in previous frame
PositionCalculator
Calculate robot's new position
Robot 1...N
Calculate robot's new position
AngleCalculator
Calculate robot's new orientation
Calculate robot's new orientation
Frame processed

Figure 4.11: Activity Diagram showing the parallel nature of the new position and orientation calculation for each robot (two robots in this diagram).

4.4.2 C++ Implementation and Testing

The class skeletons were automatically generated with the Enterprise Architect, along with partial code for some methods. The whole system was coded using C++ and the Eclipse IDE. For multi-threaded programming, the Posix threads C library was used, implementing the calculation of each robot's position and orientation in a separate thread.

Unit tests were developed for the Matrix classes and also system tests with a test suite of images from the arena and the known positions and angles of the robots. The OpenCV library was used for image handling.

4.4.3 Software Migration, optimization and HW/SW partition

An Avnet development kit including a V4-FX12 FPGA with a PowerPC405 embedded processor was used. The development tools used were Xilinx's Design Suite 11.2 for hardware and embedded software development, and GNU valgrind and gprof for preliminary resource characterization.

First, the peripherals, memory and resources needed to run the software solution in the embedded processor were characterized. The application needs extensive memory, since the image is a 1600×1200 grayscale image and there are three precalculated undistortion arrays, each storing one floating point per pixel. That is close to 24 Mbytes of required memory. Hence, the on-chip fast BlockRAMs included in the FPGA were not enough, and off-chip memory, such as Flash and SDRAM, was needed. For initial tests, the images from the camera could be loaded into this external memory, so the peripherals related to image capture could be -for the time being- left apart. To have an idea of the necessary processor resources, profiling was done in a general-purpose processor. The execution time to process an image with 14 robots in a Core i5 M480 (2 cores@2.67GHz) is 30.74 ms, including undistortion, localization and angle estimation for each robot.

Using this resource analysis, the hardware platform needed to run the software solution in the embedded processor was generated. The memories included were Flash, SDRAM and Block RAMs, and they were connected through an IBM PLB (Processor Local Bus) bus to the processor. The Block RAMs were included so that small image sections that are used many times (such as the 50×50 pixel neighborhood of the robot) could be stored in these on-chip memories. The PowerPC405 data and instruction internal caches were also configured. For the initial testing phases, special programmable logic modules for debugging and profiling were also added. The PowerPC405 and the PLB bus were set at their maximum frequency (300 MHz for the PPC, 100MHz for the PLB).

Next, the software platform needed to be generated. Since the solution uses threads, an embedded operating system with thread support is needed, preferably following the POSIX API. Xilinx offers support for two possibilities: xilkernel and Linux. Xilkernel is an open source kernel shipped with EDK, which supports the core features required in a lightweight embedded kernel, with a POSIX API for thread and mutexes. The Linux distribution does not come with EDK but it is available for compilation for the PPC405 core. Both are suitable solutions for the application, but xilkernel offers a simpler solution that is enough for the requirements of this application.

However, since the target platform has only one processor with one core, the execution of all threads is sequential, and hence there is no need for multithreaded programming for the initial software optimizations. In this platform, the use of threads can only come in handy for parallel processing if more than one hardware accelerator is included, as explained in the following subsection. Using a standalone (not OS) platform in this stage simplifies measuring execution times (and debugging), and allows the use of Xilinx's profiling tool to guide the software optimizations (since this tool only works in the standalone platform). Hence, for this stage, a standalone software platform was generated for the processor.

A version of the C++ code without threads was built and tested in the general purpose processor, since that is the base for software migration in this stage. The migration of the complete software solution to the embedded processor required only two minor changes. In the embedded solution, images were loaded from the Flash memory instead of using OpenCV, and dynamic memory for image sections was replaced by BlockRAMs. These interface changes were encapsulated in a single configuration file, so the rest of the code was unchanged.

Finally, the complete software solution was profiled in the embedded processor. Since Xilinx's profiler does not measure the time completely (e.g., the time for interrupts or some of Xilinx's libraries), it was only used to estimate the percentages of time spent in each method. The real overall time that the application takes was measured using the internal timer of the PPC, saving the timestamp when execution starts and then when it ends. These time measurements were corroborated with oscilloscope measures.

Since the PowerPC405 has no FPU, all floating point operations were emulated by Xilinx's library. Hence, software optimizations were developed for the PowerPC's particular architecture. Profiling results for each code version are shown in Table 4.1, and the corresponding bar graphic can be seen in Fig. 4.12.

The first column corresponds to the original code. The complete software solution takes 1.6 seconds. Most of the time is spent in angle calculation, so it was the first thing to tackle. The first optimization consisted in using pre-calculated cosine and sine masks to find the arc-points that needed to be sampled for angle calculation (see second column). Also, all floating point operations in the angle calculation were changed to fixed point arithmetics (see third column). These changes took the total time down to 0.9 seconds, and the percentage of time spent in angle calculation down to 1.71%.

At this point image undistortion and matrix convolution took almost half of the time each. Image undistortion could be simplified by taking the nearest neighbor to calculate the pixel in the undistorted image, instead of the bilinear interpolation with the four closest pixels. Of course, the nearest neigbour interpolation would decrease precision of robot position and angle estimation.

Table 4.1: Profiling results for software optimizations. Times for complete solution in milliseconds.

	PPC405@300 MHz					Core i5
	orig.code	cos_mask	fixed pt.	unbarrel_NN	angle_NI	all opt.
Matrix::macc	28.56%	44.13%	51.00%	93.95%	95.46%	93.33%
angleCalc::exec	44.96%	14.96%	1.71%	3.16%	1.60%	2.73%
Image::unbarrel	26.48%	40.91%	47.29%	2.89%	2.94%	2.34%
complete code	*1630*	*1078*	*926*	*501*	*495*	*30.74*

The impact of nearest neighbor estimation on the algorithm precision can be estimated as follows. The nearest neighbor estimation can be modeled as a quantization noise, which adds errors up to half a pixel. It can be assumed that this maximal noise- i.e half a pixel- does not appear more than twice per one robot dress since the distortion is not so high. Therefore, in comparison to the bilinear transformation, the robot position estimation precision might be decreased by one pixel, which corresponds to an extra 3 mm (or 0.1% relative to the arena dimensions) localization error. If both the arc endpoints and the robot position estimation are affected by the noise, the angle estimation algorithm errs by two pixels. Since the arc circumference is approximately 120 pixels, the maximal introduced error is 0.1 radians (or 1.6%). To verify the aforementioned estimations, one thousand images of the arena were taken with robots at known positions, and the localization errors with the bilinear and nearest neighbor undistortion methods were calculated. The largest error introduced by the nearest neighbor interpolation was 1.59% for angle estimation and 3 mm for position estimation, which is in good accordance with the aforementioned calculation.

Switching to nearest neighbor interpolation caused the image undistortion times to fall dramatically (see fourth column). The same change was introduced for the brightness calculation in the angle estimation (see fifth column). The aforementioned precision loss was far outweighed by the increase in the algorithm's efficiency.

All these changes were first implemented and tested in the general purpose Corei5 processor, using its debugging and testing resources, and keeping the golden reference model up to date. Migration to the PowerPC did not require code changes. The test suite was images with fourteen robots in the arena loaded in the Flash memory. Results for profiling in the Corei5 processor are also shown in this table. The fastest code was used for this test (including all optimizations and floating point arithmetics).

The final column for the PowerPC in the profiling table shows that 95.46% of the time is spent in the `Matrix::macc` method. Although so far all methods could be optimized by software taking into account the PowerPC architecture, the `Matrix::macc` that does the convolution could not be accelerated by soft-

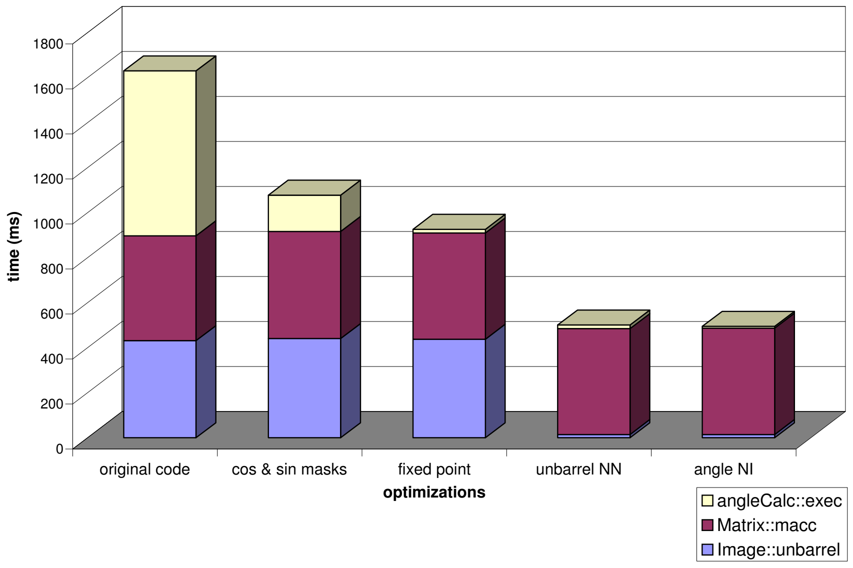


Figure 4.12: Profiling results for software optimizations in the PPC.

ware. Moreover, although slightly over 3× acceleration was achieved with software optimizations, the total time was still very high, and only enabled a 2 fps throughput. The only solution for decreasing this time and get closer to the 30 fps goal was to accelerate the `Matrix::macc` in hardware.

This method is called 100 times in `PositionCalculator::exec`, which searches for the new position of a robot, representing almost all the time spent in position calculation. It is important to note that the modularity of the OOP design and the encapsulation of the matrix operations in a separate class allowed the profiling to accurately point where the most time-consuming operation was, thus preventing useless translations to hardware.

An output of this stage was the complete, correct and optimized software version running in the embedded PowerPC405 processor. The definite hardware-software partition led to the translation of the `Matrix::macc` method to hardware.

4.4.4 Hardware translation, testing and integration

Next, the hardware module for the `Matrix::macc` was implemented, including its interface with the memory and embedded processor. Hardware and software changes were introduced to integrate this hardware module in the solution. In this section, we show the translation using two-process. In section 4.4.4, we show a comparison with AutoESL and ROCCC solutions.

The first step was to decide which interfaces were best suited for the module. The `macc` method of a `Matrix` class object takes as a parameter a matrix of the same size to convolve with itself. Hence, a good solution was to connect the hardware module to two Block RAMs, one per matrix. The PowerPC is connected to the other port of each Block RAM so it can load the matrices data. The convolution is performed between two 40x40 matrices, but it is known that one of those matrices is part of a bigger one (the 50x50 image section). Hence, the PowerPC needs to tell the `macc` module in which address of each Block RAM the matrices to be multiplied start. The size and step of the matrixes are configurable parameters. When the convolution is done, the hardware module can send the result back to the processor. This means that for each convolution, there are three data exchanges: two addresses and one result. The default bus used by EDK 11 to connect peripherals is the PLB bus. This is a complex bus that is prepared for many different types of slaves. A much simpler bus is the Device-Control Register(DCR), a bus that can connect many slave modules in a daisy chain manner. This bus takes up less logic and is the simplest solution that achieves the desired communication pattern.

When Block RAM memories are added in EDK, they are wrapped in an interface and connected automatically through a 64-bit PLB bus. This poses the restriction that the other Block RAM port (that is, the port that is connected to the `macc` hardware module) needs to be also 64 bits wide. Hence, an interface between the Block RAM wrapper and the `macc` module needs to be coded to extract the desired bytes from the 64-bit wide memory data.

Each module (the macc module, the DCR interface module, and the memory interface modules) was tested separately and then integrated into a single ipcore. This ipcore was included in the hardware platform in XPS. Software was changed to use this hardware accelerator instead of calculating the macc in software. For this purpose, the only change was to replace the `Matrix::macc` method code by sending the two addresses through the DCR bus, sleeping the processor while waiting for the hardware module to work, and asking through the DCR bus for the result. The amount of time needed for the `macc` hardware module to complete the convolution was $17\mu s$.

The best possible complete-system performance was achieved since each part (hardware and software) ran at its maximum frequency. For this, a Digital Clock Manager (DCM) available in the FPGA was used and the connection between the embedded processor and hardware was done in an asynchronous way, i.e, using memories and the DCR bus. Table 4.2 shows the execution times using one hardware macc accelerator. The measures were all taken using the PowerPC internal timer.

Table 4.2: Profiling results for hardware accelerated solution.

	ms	%
posCalc::exec	27.35	54.82
angleCalc::exec	7.96	15.96
Image::unbarrel	14.58	29.22
complete solution	*49.89*	100

As can be seen from Table 4.2, with one core the reached acceleration was 9.92×: the last software accelerated version took 495 ms and the hardware accelerated version took 49.89 ms. This factor multiplied the already achieved 3× improvement through software accelerations, achieving so far almost 30× acceleration. The achieved throughput so far was 20 fps, a very good throughput but still short from the 30 fps goal. The throughput could be further improved by making use of the inherent parallel sections in the algorithm, multithreaded programming and the ability to replicate the hardware `macc` module.

Multithreaded programming

An analysis conducted during OOP Design indicated that the complete process of finding the new position and orientation of a robot is independent from other robots -and hence parallelizable. When having only one processor and one hardware accelerator, this property of the algorithm cannot be exploited. However, if more hardware accelerators are added, there is a chance for parallelization.

As already discussed, a xilkernel platform with thread support was set up in the PowerPC; and the complete process (undistortion, position calculation and angle calculation) for one robot was placed in separate threads. Since each position calculation calls the `Matrix::macc` method 100 times, and that method sleeps waiting for the hardware to end, the processor is idle to execute another thread during that time. The most usual way for multithreading scheduling is preemptive multitasking, in which the OS decides when to switch the thread context, using a scheduling policy. However, the time slot assigned to each thread in xilkernel scheduler is 10 ms, too large compared with the 17μs each thread is sleeping while waiting for the hardware to end. Hence, cooperative multitasking needs to be used. In this approach, each thread relinquishes control when it reaches a stopping point, using the `yield()` function that makes the next thread to continue execution. While one thread is waiting for one hardware convolution to end, the other thread can send a new pair of addresses to the other hardware convolution accelerator. In this way, the

Table 4.3: Profiling results with one, two, four and six hardware accelerators

	one ipcore		two ipcore		four ipcore		six ipcore	
	ms	%	ms	%	ms	%	ms	%
posCalc::exec	27.35	54.82	14.66	39.55	8.13	26.44	7.62	25.18
angleCalc::exec	7.96	15.96	7.83	21.12	8.04	26.15	8.06	26.64
Image::unbarrel	14.58	29.22	14.58	39.33	14.58	47.41	14.58	48.18
complete solution	*49.89*		*37.07*		*30.75*		*30.26*	

processor and two hardware modules are working in parallel. This is achieved with very small software code changes: just by creating the threads, using the `yield` function in the macc method, and joining the threads. The xilkernel has to be set up and the hardware accelerator replicated.

Table 4.3 and Figure 4.13 show the results for profiling with one, two, four and six hardware accelerators.

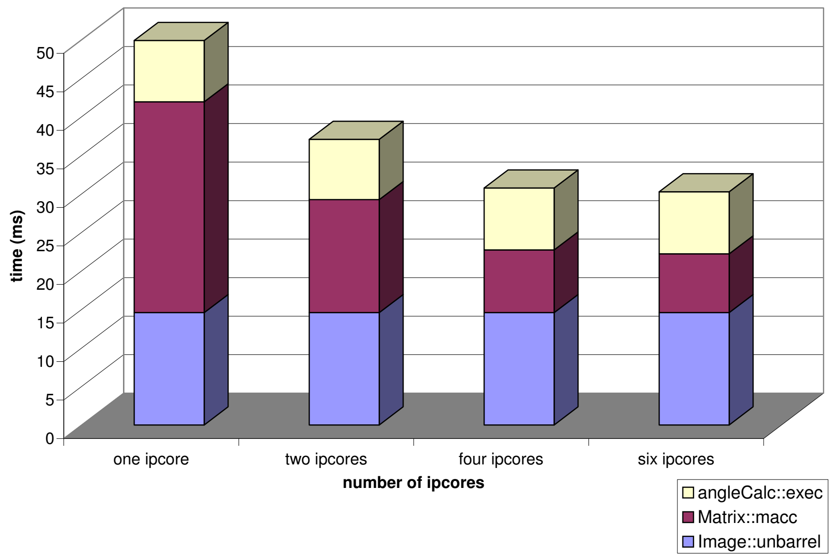


Figure 4.13: Profiling results for hardware acceleration

From these results, it can be seen that with four cores the solution processes 32, 5 fps, achieving the goal of 30 fps. To do so, software optimizations, hardware acceleration and parallelization using multithreaded programming and many ipcores were needed.

AutoESL, ROCCC and two-process comparison

For comparison purposes, the AutoESL and ROCCC high level synthesis tools were also used to synthesize the methods that needed hardware acceleration. With the ROCCC tool only the most time consuming part of the `Matrix::macc` method was implemented: the multiplication and accumulation of two vectors, without taking into account that they were matrices. This was done in this manner since the ROCCC version used did not have support for multidimensional arrays -this has been included in the latest version. Table 4.4 presents a comparison of area, frequency and VHDL code lines between the ROCCC generated code and the two-process code.

This table shows that the area requirements for the ROCCC generated code are around 11 times higher than the two-process implementation. The maximum frequency obtained with the ROCCC tool is 57% of the one obtained with the two-process. The C code had to be rewritten in order for the ROCCC generator to work. Moreover, there are specific guidelines as to how to present the data for the hardware module to use, so all the modules to feed data need to be hand-coded in VHDL/Verilog, as the modules to connect with the PPC.

With AutoESL, the complete `Matrix::macc` was implemented. In order to get a synthesizable C code, many changes had to be made. `Matrix` is a template class, since both unsigned and signed char matrices are used in the application (the image is unsigned char, but the convolution mask is signed). The template had to be taken away, making the translation for a particular type. Moreover, the method had to be translated from a class method to a standalone function. Hence, the attributes of the class object (rows, cols, step, data) had to be translated to either function parameters or variables. Also, for the tool to synthesize a BRAM memory port for the matrices, the matrices could not be passed as memory addresses, but had to be passed as fixed sized arrays (40×40 for the convolution mask and 50×50 for the image section). Hence, an extra parameter had to be included to tell the function at which offset of the big 50×50 matrix, the 40×40 matrix to convolve starts. After these changes were made, a first synthesizable solution was achieved, and HDL was generated.

From this solution several optimization directives can be applied, such as bit accurate data types, correct interfaces, loop unrolling or pipelining. Data types could not be more optimized, since the 8 bit char representation is the smallest possible for this problem. From the interfaces, the code had already been changed to take both matrices as single-port BRAMs. For the extra offset parameter, an `ap_none` interface was selected, so that AutoESL would not generate any particular protocol for the variable, and later on this new parameter could be included in the DCR bus. The complete module has AutoESL's `ap_hs` handshake protocol, which would later on need a hand-coded module to integrate to the DCR bus interface. Taking as a guide the already

	ROCCC	Two-Process
Slices	652	59
Slice FF	779	107
LUTs	1099	112
BRAMs	2	0
GCLKs	4	1
DSP48s	1	1
Latency	3200	1600
Freq (Mhz)	125.98	216.29
lines of code	1467	170

Table 4.4: ROCCC and Two-Process comparison for vector MACC

	AutoESL		Two-process
	First code	Opt. code	
Slices	71	95	144
Slice FF	73	89	128
LUTs	104	125	214
BRAMs	0	0	0
DSP48s	1	1	1
Latency	4882	1606	1606
Freq (Mhz)	167	144	166

Table 4.5: AutoESL and Two-Process comparison for Matrix::macc

hand-coded design, a pipeline optimization to the inner loop was applied. With these optimizations, the AutoESL design achieved the two-process design throughput.

Table 4.5 provides a comparison between the first AutoESL synthetizable solution, the solution after optimization and two-process solution. Although comparative reports previous to actual synthesis are provided, the results in this table are the ones provided after Xilinx's ISE implementation from the automatically generated VHDL source.

It is very interesting to see that the optimized version of the automatic code achieves a smaller solution with the same throughput- and similar maximum operating frequency- as the two-process hand-coded version. It is true that the two-process is focused on code clarity and size more than area optimization, and that the optimization directives used in AutoESL were inspired in the hand-coded design. However, these results are very good for an automated tool like AutoESL. They are also consistent with the reported results in a BDTI Benchmark that implemented a wireless communications DQPSK receiver with the AutoESL tool and compared it with hand-coded design [110].

To include this solution into the complete system, all the interface modules need to be hand-coded. The DCR bus interface is not one of the supported buses in this AutoESL version, so a hand-coded interface module between the DCR bus and the `ap_hs` handshake interface is needed. Since in XPS the BRAMs are automatically included in a 64-bit wrapper and the module takes in 8 bit data, a memory interface also needs to be handcoded.

4.5 Acceleration, area and power consumption results and analysis

In this section, an analysis of the acceleration, power and area results is presented. All the results in this section use the hardware cores generated with the two-process method.

4.5.1 Acceleration

Overall Acceleration

This section analyzes the overall acceleration based on software optimization, hardware acceleration and parallelization with many ipcores and multithreading programming. Figure 4.14 shows the execution times of each different solution, and Fig. 4.15 shows the overall acceleration of each solution over the original code and the most optimized software version.

From these figures, it can be seen that software optimizations account for approximately 3×, hardware acceleration for almost 10×, and parallelization using multithreading and hardware replication for another 1.6×. It is interesting to note that, while hardware acceleration yields the most speedup, it is traditionally the most time-consuming step. Software optimization, as well as multithreaded programming, are achieved with smaller software changes. This also shows the importance of research in high level synthesis area that seeks to reduce the time spent in a step that has the potential of providing vast acceleration.

Theoretical maximum

The maximum theoretical acceleration is an important analysis, bound by Amdahl's law, as described in Section 4.2.4. It demonstrates how close each solution is to the most optimized possible, and helps the designer decide when it is not worth doing any more work , because the maximum possible acceleration achievable from further optimizations is too low, as compared with the extra work required. Since Amdahl's law applies to parallelization, it makes

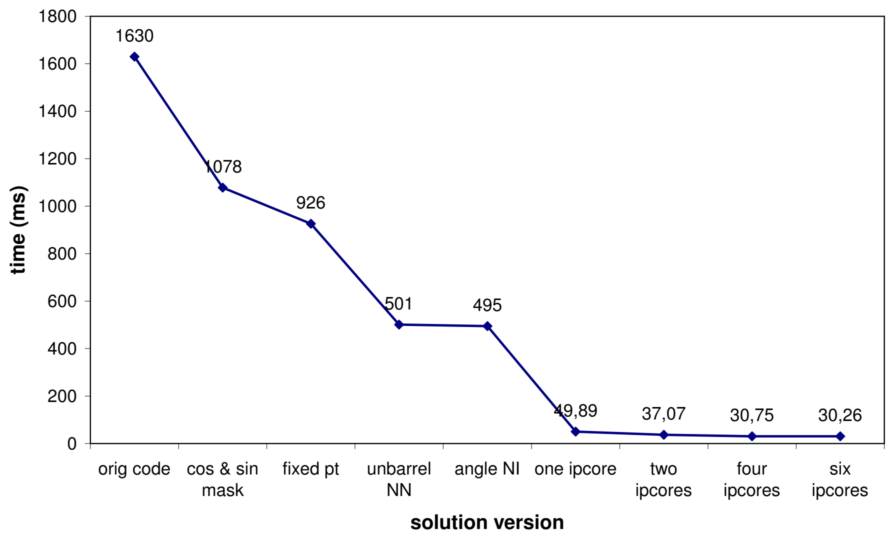


Figure 4.14: Execution times of all solutions

sense to apply it only from the most optimized software version, and analize the effect of hardware acceleration and parallelization with many ipcores and multithreading programming.

As can be seen from the profiling information of the most optimized code, the portion P that can be accelerated, i.e., the `Matrix::macc`, is 0.9546. Assuming an infinite speedup of that portion, Amdahl's law yields:

$$\begin{aligned} S'_{max} &= \lim_{S\to+\infty} \frac{1}{(1-P)+P/S} \\ &= \frac{1}{(1-0.9546)+0} = 22 \end{aligned}$$

With one core, the acceleration reached 9.92× (Fig. 4.15). That is 45% of the theoretical maximum, so it seems reasonable to try to improve it. An interesting point is that in that solution, acceleration is not really obtained by parallelization in the sense of Amdahl's law, i.e., multicore parallelization, but by changing the implementation platform from software to hardware. When adding parallelization in the multicore sense, with four cores and multithreaded programming, the acceleration goes up to 16.1×, which is 73% of the theoretical maximum. Figure 4.16 shows the percentage of the theoretical maximum

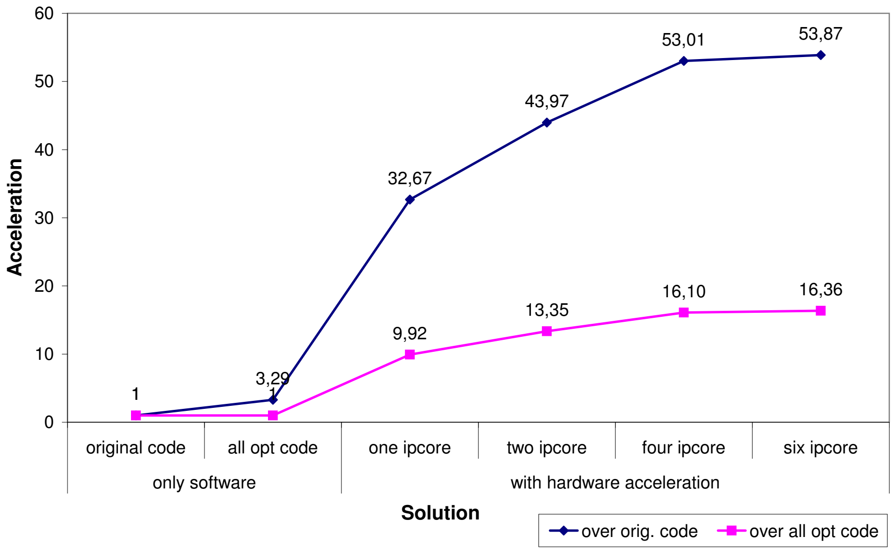


Figure 4.15: Acceleration of each hardware accelerated solution compared with software solutions

achieved by adding each hardware ipcore, and compares it with the most optimized software solution.

Theoretical maximum for acceleration using only parallelization with many ipcores and multithreading programming

Figure 4.17 shows the acceleration obtained with each new hardware ipcore, and compares with the solution with only one hardware ipcore. This Figure takes into account only Position Calculation, which is the section of the algorithm where parallelization was really done, and also the acceleration of the complete solution.

As already discussed, the addition of one core amounts to 10× acceleration. The addition of four cores and multithreading programming amounts to an extra 1.6×. Although this extra 1.6× is what allows the solution to get to the 30 fps performance goal, it seems small as compared with the 10× original gain obtained by adding hardware acceleration. An interesting analysis involves using Amdahl's law to check which theoretical maximum is achievable by the parallelization step, that is, by comparing each ipcore added to the solution that already has software optimizations and hardware acceleration with one ipcore.

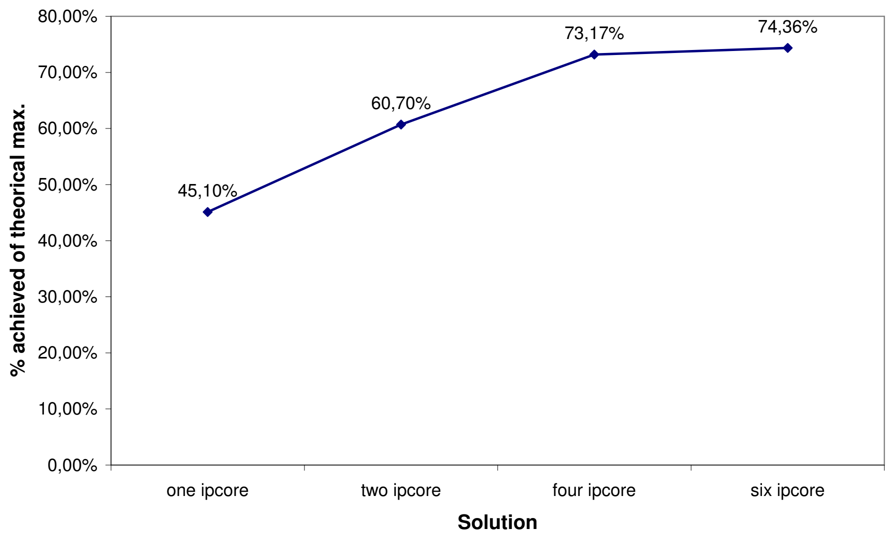


Figure 4.16: Acceleration achieved with respect to Amdahl's theoretical maximum

The portion P that can be accelerated by parallelization, i.e., the `Matrix::macc`, is 0.5482 (see Table 4.2). Of course, it is lower than in the most optimized software solution, since it already includes the acceleration provided by hardware implementation. Assuming an infinite speedup of that portion, Amdahl's law yields:

$$\begin{aligned} S'_{max} &= \lim_{S \to +\infty} \frac{1}{(1-P) + P/S} \\ &= \frac{1}{(1-0.5482)+0} = 2.21 \end{aligned}$$

Therefore, even though $1.6\times$ seems small, it amounts to $(1.6/2.21) * 100 = 72,4\%$ of the theoretical maximum acceleration that Amdahl's law yields.

The analysis with Amdahl's law can help decide when the possible achievable acceleration is not worth the effort of adding a new ipcore. However, in this problem there is a way of obtaining another interesting measure: what is the theoretical maximum amount of hardware maccs that can be added and still provide some acceleration? The answer to this question is related to the software overhead to manage an extra ipcore. For example, if the time spent in the `yield` function was exactly half the 17 μs that the hardware module takes, then it would not be worth adding more than one hardware accelera-

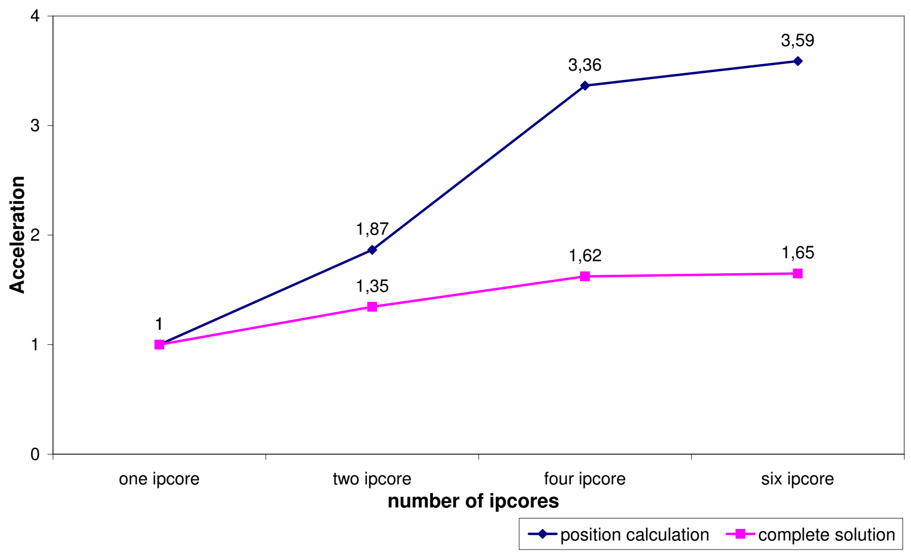


Figure 4.17: Acceleration of each additional hardware ipcore, as compared with the solution with only one hardware ipcore.

tor. With two accelerators, the processor would not be able to feed data into both of them on time: one would always be idle. Hence, the answer lies in the relationship between the time taken by the `yield` function and the really parallel activity, in this case, the hardware `macc`. Although it is not possible to measure the exact time the `yield` takes by itself, an approximation was measured in a system that has only two threads and only yields among them, resulting in $1.9\mu s$. This means that approximate 8.9 yields can be theoretically executed while a hardware macc is working, so that at most 7 or 8 hardware maccs can be included. This is a rough upper limit, but it still means, for example, that the position of the fourteen robots in the arena would not be able to be processed completely in parallel under this setting. The processor, which is the one executing the whole control of the application, will be the bottleneck.

After four cores, no acceleration is obtained by adding a new core (see Fig.4.17). That means that five threads are running: one for the main application and one for each ipcore. This is less than the calculated theoretical maximum of 7 cores, but still reasonable since this system has more threads and each thread does more than just yield, and, therefore, uses more processor time -which is now the bottleneck of the system.

Another important thing to note in Figure 4.17 is that when adding a

Table 4.6: Area occupied by each solution. Hardware implemented using two-process.

	only PPC(sw)		1 ipcore		2 ipcore		4 ipcore		6 ipcore	
	amount	%	amount	%	amount	%	amount	%	amount	%
Slices	**3530**	**64**	**3882**	**70**	**4245**	**77**	**5037**	**92**	**5443**	**99**
Slice FF	4136	37	4379	40	5054	46	6408	58	7763	70
LUTs	3690	33	4236	38	5083	46	6871	62	8665	79
BRAMs	13	36	13	36	17	47	25	70	33	91
DSP48	0	0	1	3	2	6	4	12	6	18
PowerPC	1	100	1	100	1	100	1	100	1	100

second core, the obtained acceleration for the Position Calculation is not 2 but 1.87. This is expected, since there is a software overhead for adding a core: the overhead involved in creating an extra thread and all the yielding logic between the threads, and in joining the threads after processing In the case of four cores, the difference between the theoretical speedup of 4 and the obtained one of 3.36 is bigger. From four cores on, there is no acceleration gain. From these results it is likely that the exact point at which the processor becomes 100% busy is around four cores.

4.5.2 Area

The area occupied by each solution can be seen in Table 4.6. It should be noted that there are extensive area requirements just to get the PowerPC embedded processor and the necessary memories to run the software solution. Figure 4.18 shows the percentage of extra slices required for each solution.

An interesting fact to note is the impressive routing effort of Xilinx's tools, which achieve a working design that occupies 99% of the slices in the FPGA. It is also important to notice not only the occupied slices, but also each category, to get a clearer idea of the occupancy of the FPGA. The routing tool at the begining leaves more slices with less occupation, and it then not only occupies more slices, but also more things in each slice.

4.5.3 Power and Energy consumption

Table 4.7 presents an estimation of power and energy consumption obtained by using Xilinx's Xpower. This analysis only includes the consumption of the FPGA core: the external load and the I/O consumption are not considered. Since these are estimations, perhaps the most significant number is the saving

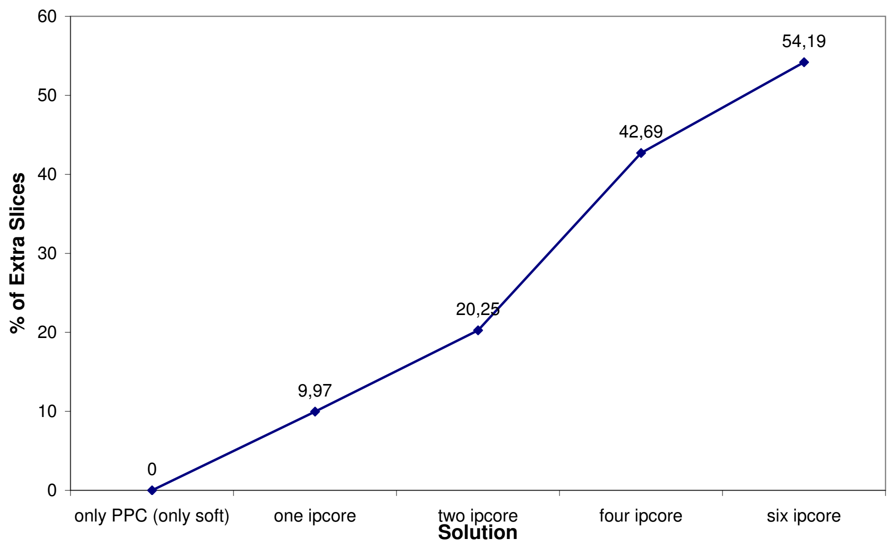


Figure 4.18: Percentage of extra slices occupied as compared with the only software solution

of energy shown in the last column.

Figure 4.19 shows the estimated energy consumption per frame, and Figure 4.20 presents the estimated energy saving. The energy saving is expressed in the same way as the acceleration is portrayed in the previous figures,i.e., in terms of how many times less energy each solution consumes. Of course, this shows the obvious correlation between acceleration and energy consumption: although adding new ipcores makes the power consumption higher, since the acceleration obtained is very important, the energy consumption is significantly reduced. However, when the time acceleration is smaller (e.g., from 2 to 4 ipcores) or almost zero (e.g., from 4 to 6 cores), the energy saving is not so big, or is even smaller than in previous solutions. This is because the acceleration gain is not enough to counteract the increase in power consumption. Hence, taking into account only energy consumption, adding more than 2 ipcores does not make much sense.

Finally, 92% energy saving with four ipcores, as compared with the software solution, or in other words, the possibility to process one frame consuming 13× less energy, entails a very important result that advocates for hardware acceleration and parallelization in FPGA based chips.

Table 4.7: Power and Energy consumption. Hardware implemented using two-process.

Design	I(mA)	P (mW)	T (ms)	E (mJ)	Save
opt. code	377	453	495	224	0%
1 ipcore	394	473	49,89	24	89%
2 ipcore	422	506	37,07	19	92%
4 ipcore	460	552	30,75	17	92%
6 ipcore	506	607	30,26	18	92%

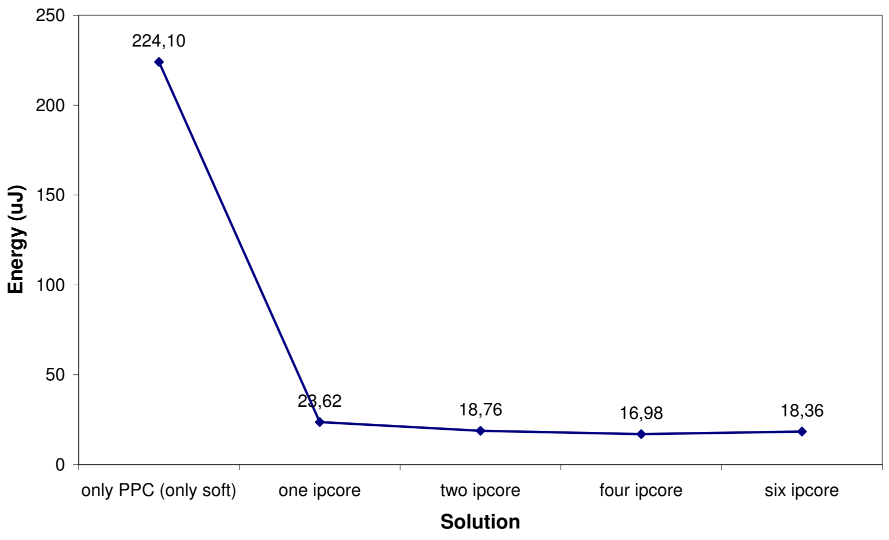


Figure 4.19: Estimated energy consumption per frame for each solution

4.5.4 Overall analysis

A relevant question about acceleration, area and energy consumption results would be: which is the solution that best balances all these measures? Data indicates that the solution with 2 ipcores seems more appropiate. This solution achieves almost 27 fps, a $13,15\times$ acceleration from the most optimized code solution, 92% energy saving, and with only 20% area increase. The one-core solution only achieves 20 fps with 89% energy saving and 10% area increase. The four-core solution achieves 32 fps, but with almost no extra energy saving and doubling the area increase to 40% as compared with the two-core solution. The six-core solution, on the other hand, offers almost no acceleration with

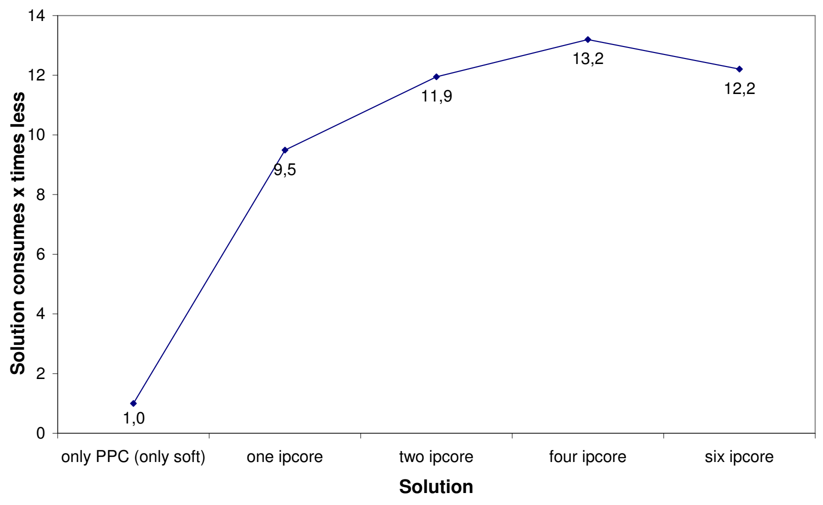


Figure 4.20: Estimated energy saving

an extra energy and area penalty. Hence, it seems reasonable to point the two-core solution as a good balance between acceleration, energy consumption and area.

However, the design target was to process 30 fps, and the two-core solution falls 3 fps -i.e. 10%- short of this performance goal. Hence, the four-core solution is the most suitable. Using four cores, the final hardware accelerated solution processes over 32 fps of 1600×1200 pixel images, thereby achieving a real-time embedded solution to the problem. The acceleration from the original solution to the final software optimized and hardware accelerated solution is 53×, while the acceleration from the optimized software solution is 16×. The XC4VFX12 FPGA -which is the smallest Virtex4 FPGA- is 92% occupied, as compared with the original 64% for only software solution. On the other hand, the use of four cores represents an estimated 92% energy saving from the software solution, that is, 13× less energy consumption to process each frame. This embedded solution takes 30.7 ms to process an image, while the most optimized software solution in a Corei5 (2 cores@2.67GHz) takes 30.4 ms. This means that the embedded solution achieved by following the proposed methodology runs with a comparable speed as to the method implementation on an up-to-date general purpose processor, but is smaller, cheaper, and demands less power and energy.

4.6 Conclusions

In this chapter, we proposed a methodology to achieve real-time embedded solutions using hardware acceleration, but with development times similar to software projects. This methodology applies to the growing field of processor-centric embedded systems with hardware acceleration in FPGA-based chips. The methodology is applied to a novel algorithm for multiple robot localization in global vision systems, demonstrating its usefulness for embedded real-time image processing applications.

The methodology helps to reduce design effort by raising the abstraction level while not imposing the need for engineers to learn new languages and tools. Taking advantage of the processor centric approach, the whole system is designed using well established high level modeling techniques, languages and tools from the software domain. In other words, it is an OOP design approach expressed in UML and implemented in C++ using multithreaded programming. The methodology also helps to reduce software coding effort since the C++ implementation provides not only a golden reference model, but may also be used as part of the final embedded software. Hardware coding, traditionally the most time-consuming and error-prone stage of hardware-accelerated applications, is simplified. The key to reducing hardware coding effort is to join a good OOP design implemented in C++, which allows engineers to precisely find the methods that need to be accelerated by hardware, with automatic tools or guidelines to translate the selected C++ methods to HDL.

A simple and robust algorithm for multiple robot localization in global vision systems is also presented. The algorithm was specifically developed to work reliably 24/7 and to detect the robot's positions and headings even in the presence of partial occlusions and varying lighting conditions. To achieve a real-time embedded solution able to process over 30 fps, we applied the methodology, and performed software optimizations, used hardware acceleration, and extracted parallelism by including multiple ipcores in a multithreaded programming environment. The final embedded solution processes 1600×1200 pixel images at 32 fps, uses four hardware acceleration cores, occupies 92% of the XC4VFX12 FPGA and consumes approximately 17mJ of energy per frame. This represents a $16\times$ acceleration with respect to the most optimized software solution, with a 43% increase in area but a 92% energy saving.

Chapter 5

Conclusions

The overall goal of this thesis was to contribute to the field of hardware/software co-design of embedded systems. We proposed a new co-design methodology that reduces design and implementation effort in an important field of embedded systems design- processor-centric embedded systems in FPGA-based chips-, at a time when the growing complexities of these designs make the need for new methodologies, languages and tools vital. Processor-centric embedded systems in FPGA-based chips is a growing and novel field of embedded systems: during 2011, both Xilinx and Altera launched new chip families that combine powerful ARM processor cores with low-power programmable logic.

To achieve the overall goal, we proposed three particular goals:

1. the study of traditional co-design flows using processors and off-the-shelve ICs, and their application to the co-design of an embedded system with real-time, power consumption and size requirements.

2. the study of traditional design flows using FPGAs and their application to the design of an embedded system that requires massive data processing with real-time constraints

3. the proposal of a new co-design methodology for a significant class of embedded systems: processor-centric embedded systems with hardware acceleration in FPGA-based chips. The new methodology should be focused in reducing design and implementation effort, integrating methodologies, languages and tools from both the software and hardware domain.

The main contributions in each of these goals are discussed in the following three sections.

5.1 Embedded systems using processors and ICs

We devoted chapter 2 to this goal. We described the co-design of a control embedded system applying the traditional flow in which processors and ICs are combined: the development of the mini-robot ExaBot. This system has stringent real-time, power consumption and size requirements, providing a challenging case study.

The main contributions regarding this goal are:

- The adaptation of traditional co-design flows in which processors and off-the-shelve ICs are combined, to the autonomous robotics field. The particular co-design flow is explained and the development of the robot following its different stages is shown.
- The design, construction and testing of the ExaBot robots. The main goal for pursuing this task was to obtain a low-cost robot that could be used not only for research, but also for outreach activities and education. In this sense, neither the commercially available research robots nor the commercially available educational robots were a suitable solution. Six ExaBot robots are currently in use in the Laboratorio de Robótica y Sistemas Embebidos of the FCEN-UBA. They have been used for educational robotics activities for high school students, research experiments in mobile robotics, and education in graduate and undergraduate university courses.

5.2 Embedded systems using FPGAs

Chapter 3 is devoted to the study of traditional design flows using FPGAs. Some embedded systems require massive data processing with real-time constraints that cannot be met with the standard microprocessor and IC approach. In these cases, solutions based on FPGAs are common. These approaches take advantage of the inherent parallelism of many data processing algorithms and allow to create massive parallel solutions. Nowadays, around 35% of the embedded engineers use FPGAs in their designs, which makes this an important field in embedded system design.

The main contributions regarding this goal are:

- The introduction of a traditional FPGA design flow, derived from the analysis of design flows presented in several foundational FPGA books. A short comparison among those design flows is also presented.

- The application of this HDL-based design flow to achieve real-time processing of an IR image for hot spot detection. The novel image segmentation algorithm was thought for parallel implementation, and its design was carefully tailored to achieve the real-time constraints. In this manner, the achieved embedded solution successfully segments the image with a total processing delay equal to the acquisition time of one pixel (i.e., at video rate). This processing delay time is independent of the image size. There is also no need for extra memory to store parts or the complete image. Sizing equations are presented, and timing, area and power consumption parameters are discussed.

From bibliographical research and our own experience- shown in that chapter -, we discussed pros and cons of this HDL-based design flow. For one, this type of flow allows to create extremely tailored, top performance designs that meet tight real-time constraints with low power consumption and small occupied area. However, it requires a lot of man power from knowledgeable designers. It is also highly prone to error, and hard to test and verify even using top notch simulation tools and complex verification environments. Clearly, measures to decrease design time by raising the abstraction level of design and implementation are needed to cope with the ever increasing complexity of applications. This is addressed in the next goal.

5.3 A new co-design methodology for processor-centric embedded systems in FPGA-based chips

Chapter 4 is devoted to the proposal of a new co-design methodology in FPGA-based chips. Hardware/software co-designed solutions try to combine the best of both software and hardware worlds, making use of the ease of programming a processor while designing tailored hardware accelerator modules for the most time-consuming sections of the application. The inclusion of processor cores embedded in programmable logic has made FPGAs an excellent platform for these approaches. During 2011, both Xilinx and Altera (the two major FPGA vendors) launched new chip families that combine powerful ARM processor cores with low-power programmable logic. According to the 2012 Embedded Market Survey, 37 % of the engineers that do not use FPGAs in their current designs confirmed that this trend will change their minds.

The novelty of this approach together with its potential in the embedded system world makes academic research in hardware/software co-design in FPGA-based chips an important field. The main problem to tackle in these designs is time-consuming and complex development. The rising complexity of applications makes it difficult for designers to model the functional intent of the system in languages that are used for implementation, such as C or HDLs. Moreover, although the traditional HDL-based design flow is useful to generate top performance tailored designs, it comes at the cost of time-consuming, complex and error-prone development. Although advances have been made in high-level modeling and high-level synthesis, there is still a great need for co-design methodologies, languages and tools, so that the recent combination of powerful processors with programmable logic can reach its full potential.

The main contributions regarding this goal are:

- The proposal of a co-design methodology for the growing field of processor-centric embedded systems with hardware acceleration in FPGA-based chips. The goal is to achieve real-time embedded solutions, using hardware acceleration, but achieving development time similar to that of software projects. The methodology's main advantages are:
  - It helps to reduce design effort by raising the abstraction level while not imposing the need for engineers to learn new languages and tools. Taking advantage of the processor centric approach, the whole system is designed using well established high level modeling techniques, languages and tools from the software domain. In other words, it is an OOP design approach expressed in UML and implemented in C++ using multithreaded programming.
  - The methodology also helps to reduce software coding effort since the C++ implementation provides not only a golden reference model, but may also be used as part of the final embedded software.
  - Hardware coding, traditionally the most time-consuming and error-prone stage of hardware-accelerated applications, is simplified. The key to reducing hardware coding effort is to join a good OOP design implemented in C++, which allows engineers to precisely find the methods that need to be accelerated by hardware, with semi-automatic tools or guidelines to translate the selected C++ methods to HDL.
- The proposal of a simple and robust algorithm for multiple robot localization in global vision systems. The algorithm was specifically developed to work reliably 24/7 and to detect the robot's positions and headings even in the presence of partial occlusions and varying lighting conditions.
- The co-designed implementation of this algorithm to achieve a real-time embedded solution able to process over 30 fps. For this, we applied the methodology, and performed software optimizations, used hardware acceleration, and extracted parallelism by including multiple ipcores in a multithreaded programming environment. The final embedded solution

processes 1600 × 1200 pixel images at 32 fps, uses four hardware acceleration cores, occupies 92% of the XC4VFX12 FPGA and consumes approximately $17\mu J$ of energy per frame. This represents a 16× acceleration with respect to the most optimized software solution, with a 43% increase in area but a 92% energy saving. This case study shows the usefulness of the proposed methodology for embedded real-time image processing applications.

5.4 Publications

During the course of this work, we have published 11 conference papers and 3 journal articles.

Journal Articles:

- 2013 *Accelerating embedded image processing for real time: a case study*, Sol Pedre, Tomáš Krajník, Elías Todorovich and Patricia Borensztejn. Journal of Real Time Image Processing, Springer-Verlag Berlin Heidelberg, ISSN 0018-9162. *in press.* JCR 2011 IF 1.020

- 2013 *A Behavior-Based approach for educational robotics activities*, Pablo de Cristóforis, Sol Pedre, Matias Nitsche, Thomas Fischer, Facundo Pessacg, Carlos Di Pietro, IEEE Transactions on Education, vol 56, no 1, pp 61-66, ISSN 0018-9359. JCR 2011 IF 1.021

- 2010 *A mobile mini robot architecture for research, education and popularization of science*, Sol Pedre, Pablo de Cristóforis, Javier Caccavelli and Andrés Stoliar, Journal of Applied Computer Science Methods, vol 2, no 1, pp 41-59, ISSN 1689-9636.

Conference Full Papers:

- 2012 *Hardware/Software co-design for real-time embedded image processing: a case study*, Sol Pedre, Tomáš Krajník, Elías Todorovich and Patricia Borensztejn. Progress in Pattern Recognition, Image Analysis and Applications, 17th Iberoamerican Congress on Pattern Recognition, CIARP 2012, Buenos Aires, Argentina, Septiembre 2012. Lecture Notes in Computer Science (LNCS), Springer-Verlag Berlin Heidelberg, vol 7441, pp 621-628, ISSN 0302-9743.

- 2012 *A co-design methodology for processor-centric embedded systems with hardware acceleration using FPGA*, Sol Pedre, Tomáš Krajník, Elías Todorovich and Patricia Borensztejn. VIII IEEE Southern Programmable Logic Conference – SPL 2012, Bento Goncalvez, Brazil,20 al 23 de Marzo 2012,pp 7-14, ISBN 978-1-4673-0185-5, Published in IEEEXplore.

- 2012 *A simple visual navigation system for an UAV*,Tomáš Krajník, Matias Nistche, Sol Pedre, Libor Přeucil and Marta Mejail, 9th IEEE International Multi-Conference on Systems, Signals and Devices – SSD12, Chemnitz, Germnay, March 20-23 , 2012, pp 1-6. ISBN: 978-3-9814766-1-3. Published in IEEEXplore.

- 2011 *A new programming interface for Educational Robotics*, Javier Caccavelli, Sol Pedre, Pablo de Cristóforis, Andrea Katz and Diego Bendersky, 4th International Conference on Research and Education in Robotics, EUROBOT 2011. Prague, Czech Republic, June 2011. Communications in Computer and Information Science (CCIS), Springer-Verlag Berlin Heidelberg, vol 161, pp 68-77, ISSN 1865-0929.

- 2009 *Real Time Hot Spot Detection using FPGA*, Sol Pedre, Andrés Stoliar and Patricia Borensztejn. Progress in Pattern Recognition, Image Analysis and Applications, 14th Iberoamerican Congress on Pattern Recognition, CIARP 2009, Guadalajara, Mexico, November 2009. Lecture Notes in Computer Science (LNCS) 5856, Springer-Verlag Berlin Heidelberg, pp 595-602, ISSN 0302-9743.

- 2009 *Decision Support System for Hot Spot Detection*, Esther Salami, Sol Pedre, Patricia Borensztejn, Cristina Barrado, Andrés Stoliar and Enric Pastor, Intelligent Environments 2009, Proceedings of the 5th International Conference on Intelligent Environments, Barcelona, Spain, 2009. ISBN: 978-1-60750-034-6. IOS Press, pp 277-284.

Conference Short Papers:

- 2011 *Layered Testbench for Assertion-Based Verification*, José Mosquera, Sol Pedre and Patricia Borensztejn. VII IEEE Southern Programmable Logic Conference – SPL 2011, Designer Forum, Córdoba, Argentina, 13-15 April 2011.

- 2010 *Derivation of PBKFD2 keys using FPGA*, Sol Pedre, Andrés Stoliar and Patricia Borensztejn. VI IEEE Southern Programmable Logic Conference – SPL 2010, Designer Forum, Ipojuca, Porto Gallinas Beach, March 24-26 2010.

- 2010 *Audio sobre ethernet: implementación utilizando FPGA*, José Mosquera, Andrés Stoliar, Sol Pedre, Maximiliano Sacco and Patricia Borensztejn. VI IEEE Southern Programmable Logic Conference – SPL 2010, Designer Forum, Ipojuca, Porto Gallinas Beach, March 24-26 2010.

- 2009 *ExaBot: a mini robot for research, education and popularization of science*, Pablo De Cristóforis, Sol Pedre, Javier Caccavelli and Andrés Stoliar. VI Latin American Summer School in Computational Intelligence and Robotics - EVIC2009, Santiago, Chile, December 2009.

- 2008 *Exabot: un robot para divulgación, docencia e investigación*, Pablo De Cristóforis, Sol Pedre and Juan Santos. V Jornadas Argentinas de Robótica – JAR08, Bahía Blanca, Argentina, November 2008.

5.5 Future work

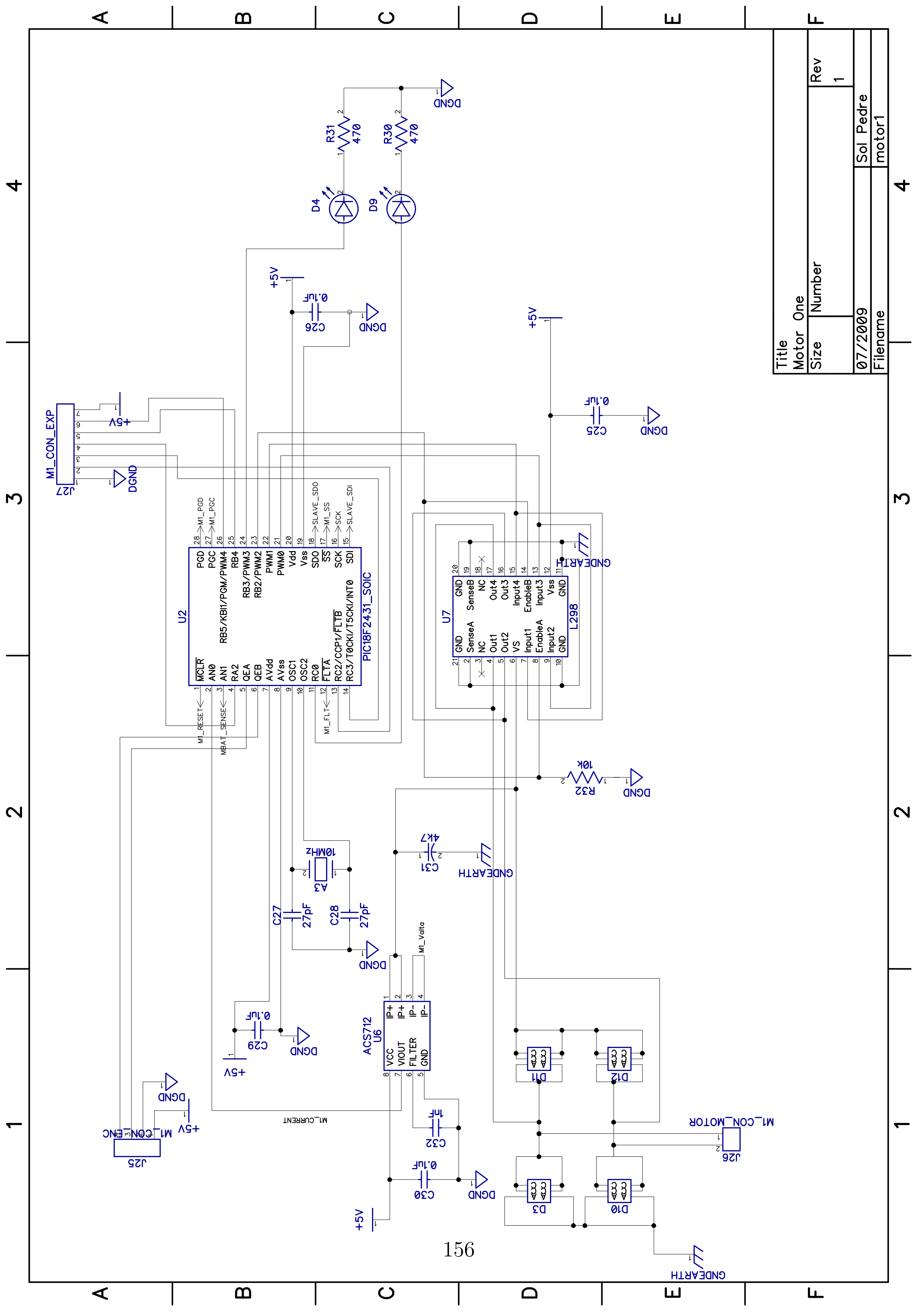
There are several paths to follow the work of this thesis. One important aspect to develop is the union of functional verification methodologies and frameworks with the proposed co-design methodology, both from the software domain as from the hardware domain. As discussed in section 3.3.2, there are several works in the field of functional verification of FPGA-based designs, including topics such as assertion driven simulation, functional coverage, random stimuli generation or formal techniques. Some important books cover SystemVerilog for Verification [4], Open Verification Methodology [71] or C++ based verification [70]. This is a vast area of FPGA-based design, that has seen much development in recent years. Studying these methodologies, and merging them with software functional verifications techniques to achieve a comprehensive co-design functional verification framework to include in the proposed co-design methodology, is a challenging and interesting path to continue.

Another path to continue is to apply the methodology to further case studies. Here, it would be particularly interesting to choose case studies that cover several fields of embedded systems, to show that the methodology is useful for a wide range of applications. In this sense, an interesting study is "The Landscape of Parallel Computing Research: A View from Berkeley" [119], a study conducted during two years by eleven specialists of several areas of computer science led by David Patterson. In this study, the authors propose a list of 13 "dwarfs", patterns of computation and communication that are the core of any application. They propose that new computer architectures are tested with sample algorithms of these dwarfs, instead of using traditional benchmarks that are biased to already established architectures. From these 13 dwarfs, 8 have applications in the embedded system domain. For example, the proposed localization algorithm is an example of the Dense Linear Algebra dwarf. These dwarfs provide a good guide to choose further case studies, and be able to state that the proposed methodology is useful for the core of most embedded system applications. Moreover, these case studies would provide the chance to further test the AutoESL high level synthesis tool, or it's new version, the Vivado Design Suite, that are clearly very promising tools. Finally, these implementations could be done using Xilinx's new Zynq-7000 platform, which combines a powerful ARM with low power programmable logic, to have a taste of what these new platforms can achieve.

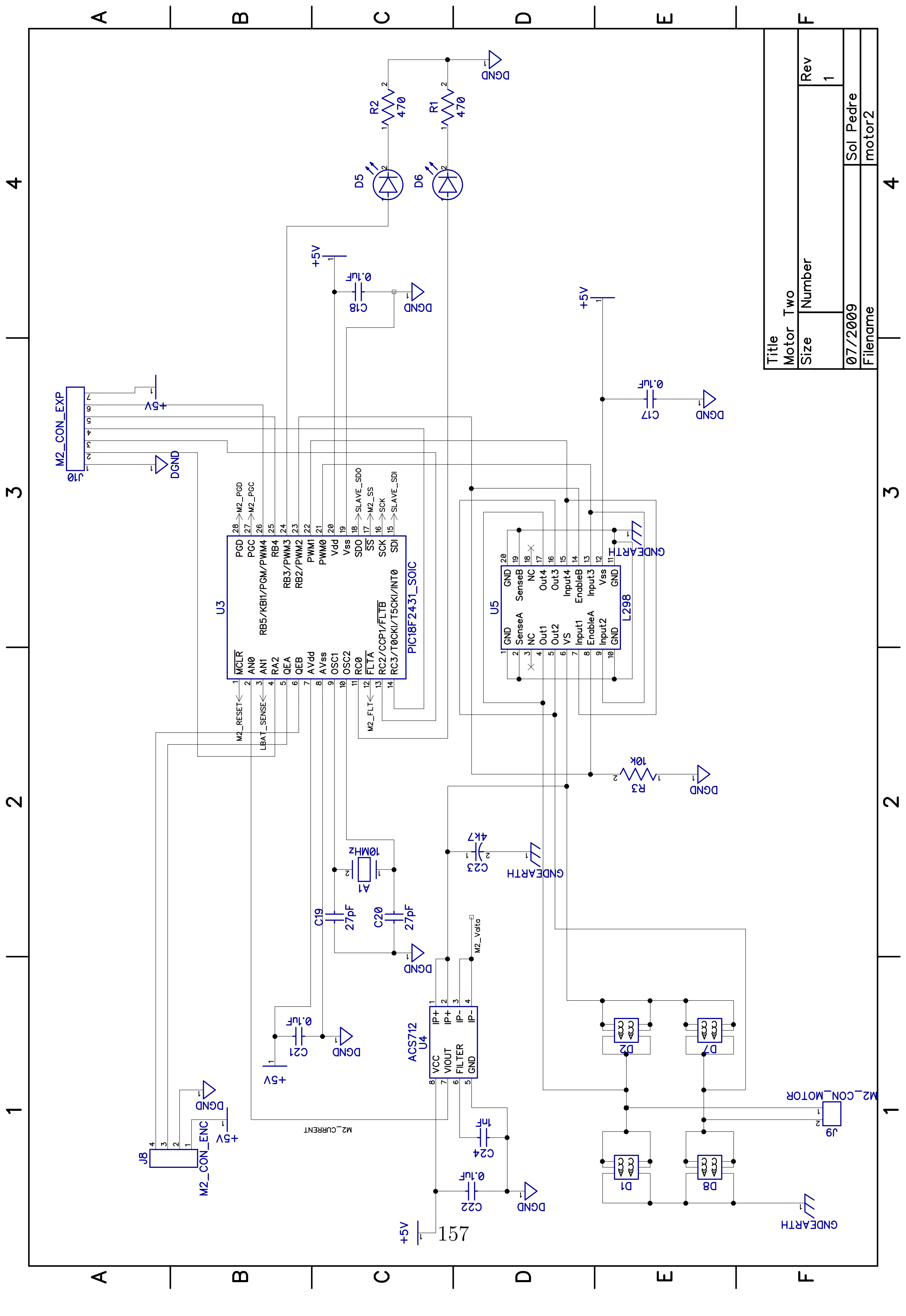
Appendices

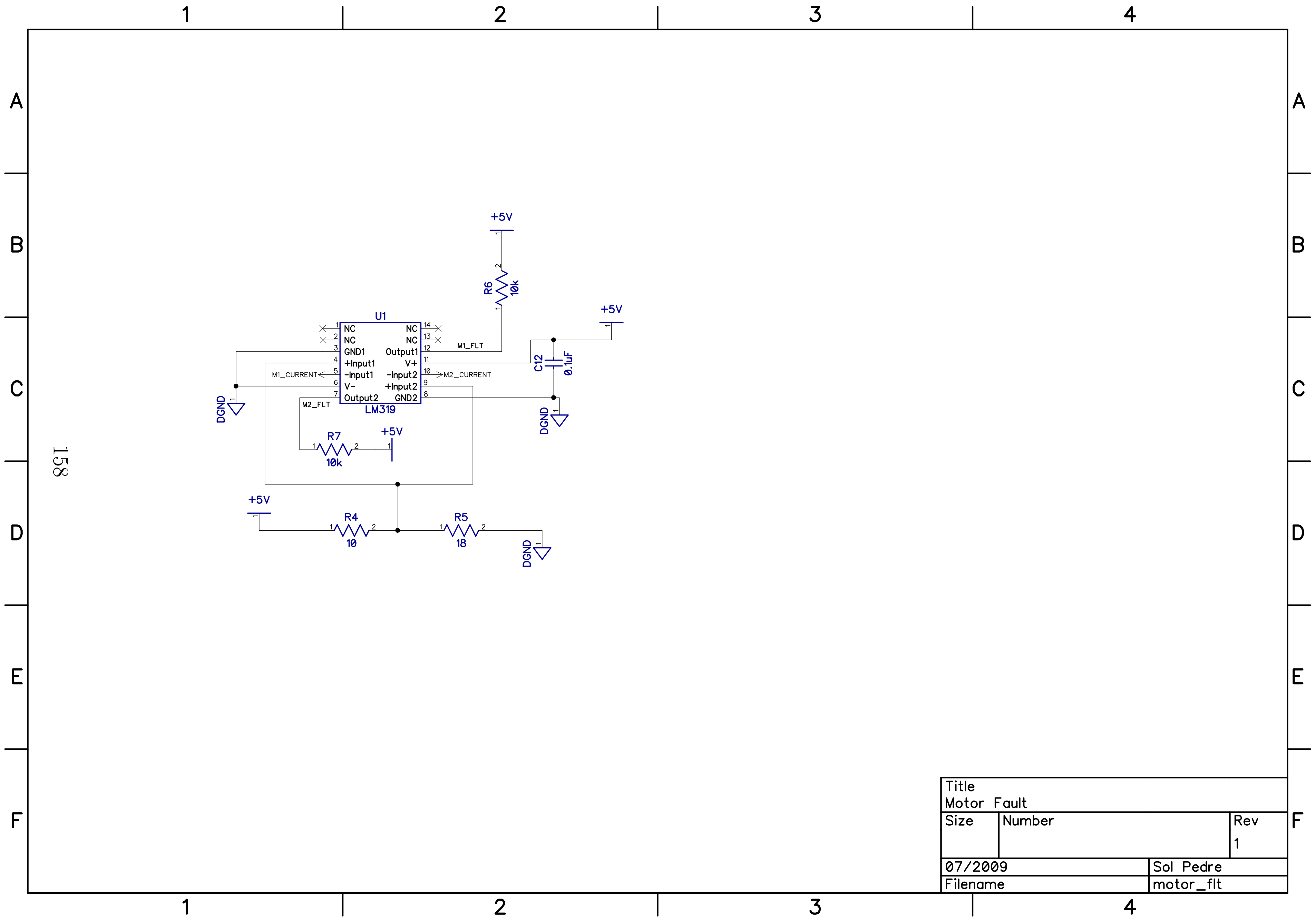
Appendix A

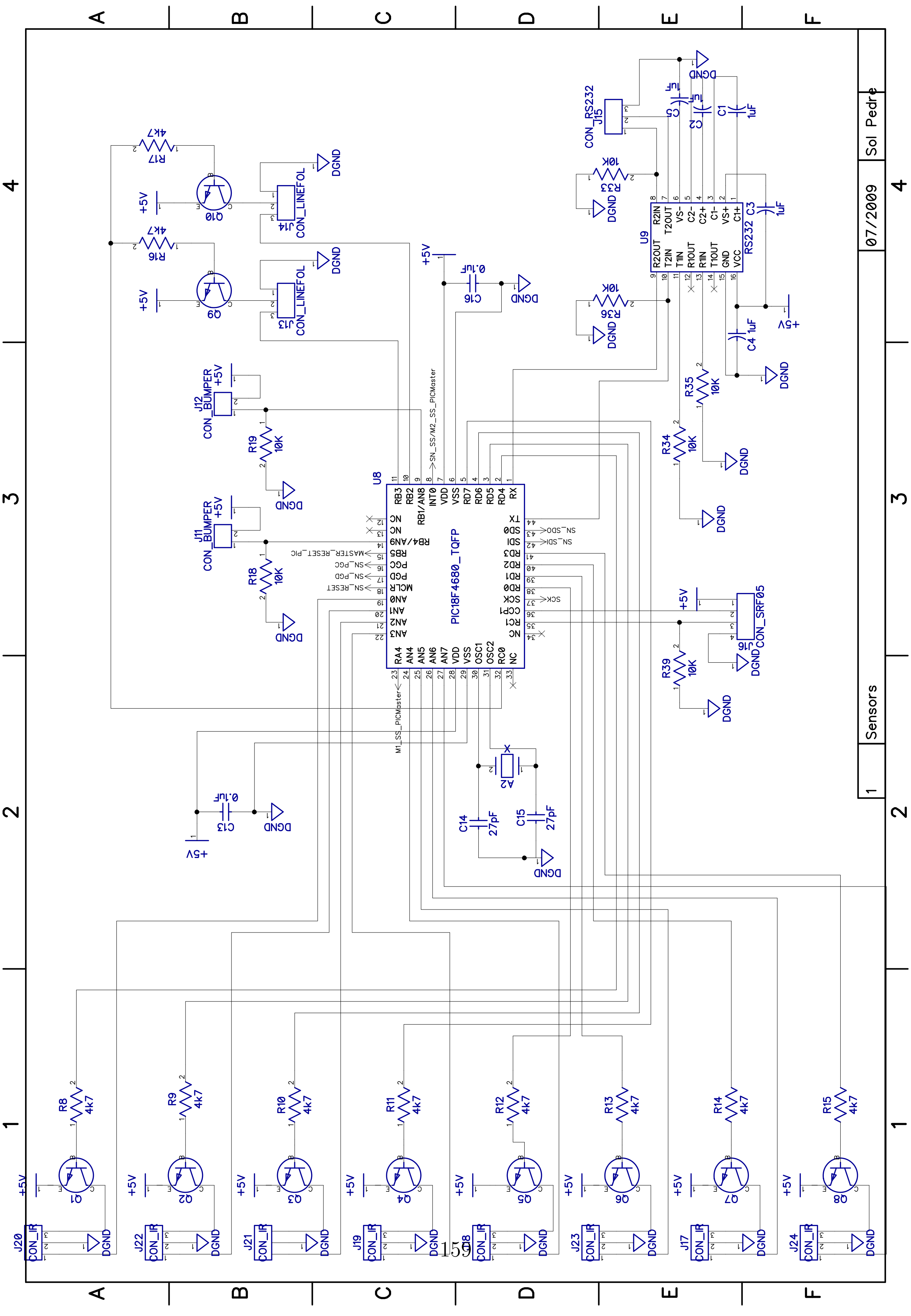
Exabot Schematics and PCB



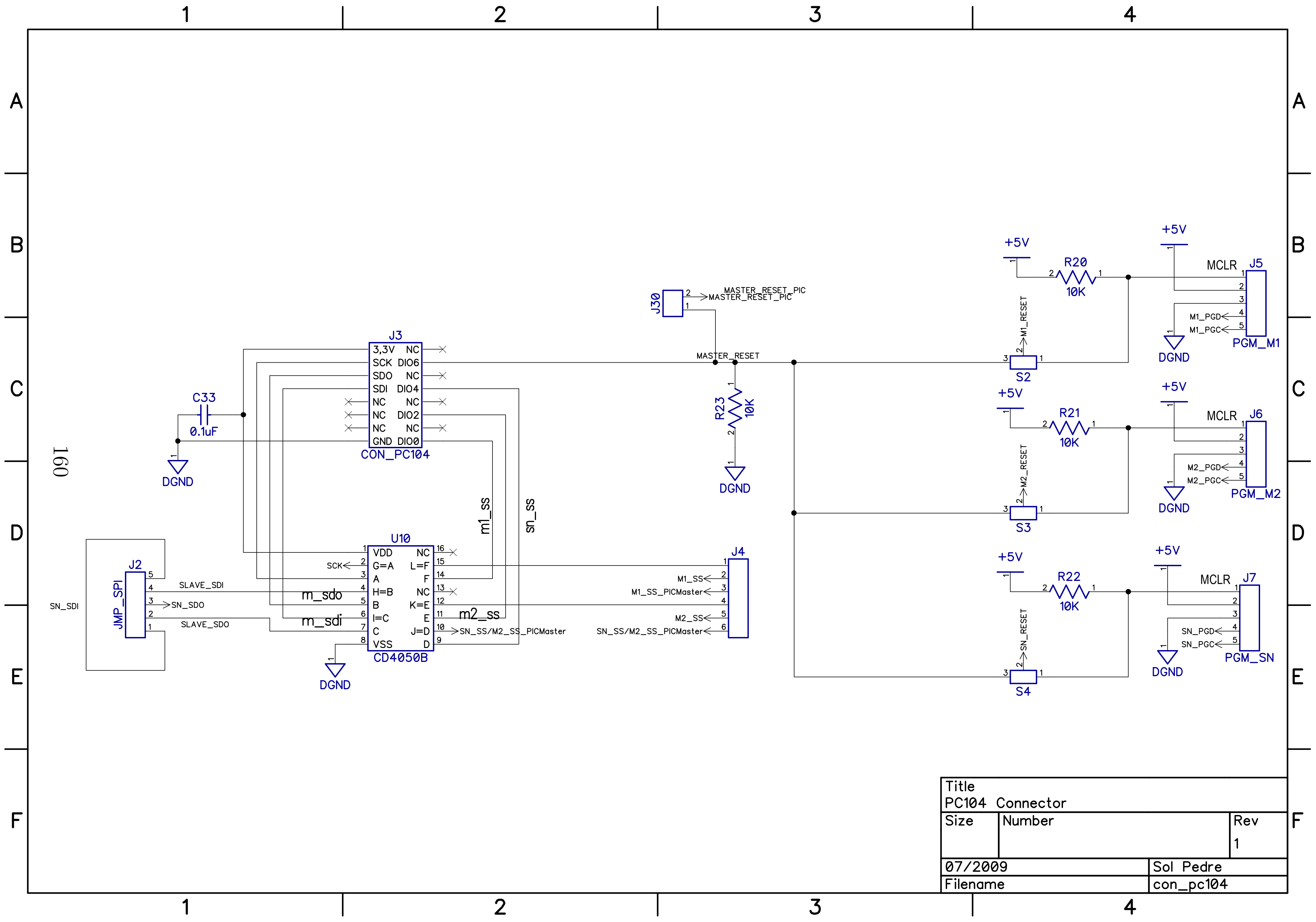
Title
Motor One
Size
Number
Rev
1
07/2009
Sol Pedre
Filename
motor1
U2
PIC18F2431_SOIC
MCLR
AN0
AN1
RA2
QEA
QEB
AVdd
AVss
OSC1
OSC2
RC0
FLTA
RC2/CCP1/FLTB
RC3/T0CKI/T5CKI/INT0
PGD
PGC
RB5/KBI1/PGM/PWM4
RB4
RB3/PWM3
RB2/PWM2
PWM1
PWM0
Vdd
Vss
SDO
SS
SCK
SDI
M1_PGD
M1_PGC
SLAVE_SDO
M1_SS
SCK
SLAVE_SDI
M1_RESET
MBAT_SENSE
M1_FLT
U7
L298
GND
SenseA
NC
Out1
Out2
VS
Input1
EnableA
Input2
SenseB
Out4
Out3
Input4
EnableB
Input3
Vss
GNDEARTH
J27
M1_CON_EXP
+5V
DGND
D4
D9
R31
R30
470
C26
0.1uF
C25
R32
10K
C31
4K7
A3
10MHz
C27
C28
27pF
C29
U6
ACS712
VCC
VIOUT
FILTER
IP+
IP-
M1_Volta
M1_CURRENT
C32
1nF
C30
J25
M1_CON_ENC
J26
M1_CON_MOTOR
D11
D12
D3
D10

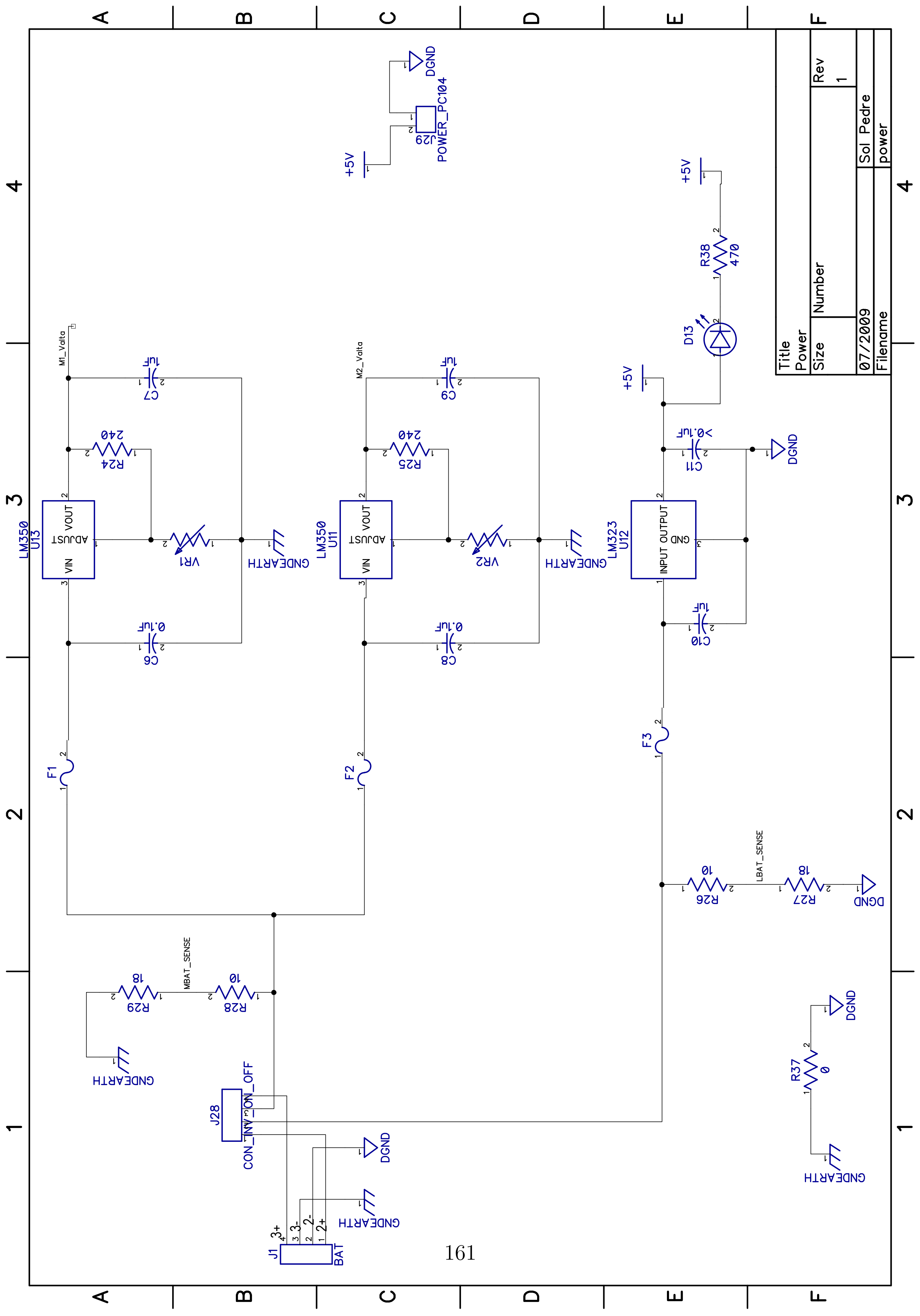

Title
Motor Two
Size
Number
Rev
1
07/2009
Sol Pedre
Filename
motor2
U3
PIC18F2431_SOIC
MCLR
AN0
AN1
RA2
QEA
QEB
AVdd
AVss
OSC1
OSC2
RC0
FLTA
RC2/CCP1/FLTB
RC3/T0CKI/T5CKI/INT0
PGD
PGC
RB5/KBI1/PGM/PWM4
RB4
RB3/PWM3
RB2/PWM2
PWM1
PWM0
Vdd
Vss
SDO
SS
SCK
SDI
M2_PGD
M2_PGC
SLAVE_SDO
M2_SS
SCK
SLAVE_SDI
M2_RESET
LBAT_SENSE
M2_FLT
U5
L298
GND
SenseA
NC
Out1
Out2
VS
Input1
EnableA
Input2
SenseB
Out4
Out3
Input4
EnableB
Input3
Vss
U4
ACS712
VCC
VIOUT
FILTER
IP+
IP-
M2_Vdto
M2_CURRENT
J10
M2_CON_EXP
J8
M2_CON_ENC
J9
M2_CON_MOTOR
D1
D2
D5
D6
D7
D8
R1
470
R2
470
R3
10k
A1
10MHz
C17
0.1uF
C18
0.1uF
C19
27pF
C20
27pF
C21
0.1uF
C22
0.1uF
C23
4k7
C24
1nF
+5V
DGND
GNDEARTH
A
B
C
D
E
F
1
2
3
4


+5V
R6
10k
U1
NC
NC
GND1
+Input1
-Input1
V-
Output2
NC
NC
Output1
V+
-Input2
+Input2
GND2
LM319
M1_FLT
M1_CURRENT
M2_CURRENT
M2_FLT
C12
0.1uF
DGND
R7
10k
R4
10
R5
18
Title
Motor Fault
Size
Number
Rev
1
07/2009
Sol Pedre
Filename
motor_flt

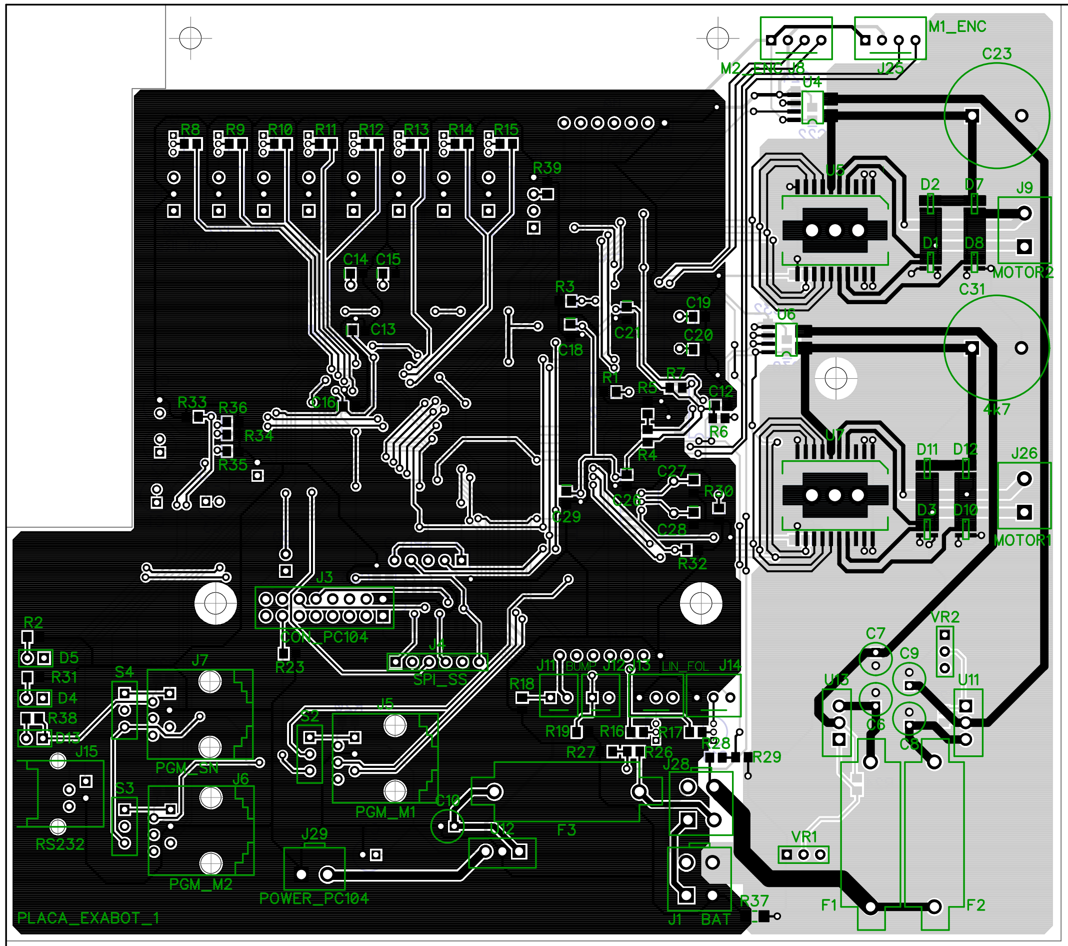


Sensors
Sol Pedre
07/2009
1
U8
PIC18F4680_TQFP
U9
RS232
J15 CON_RS232
J16 CON_SRF05
J11 CON_BUMPER
J12 CON_BUMPER
J13 CON_LINEFOL
J14 CON_LINEFOL
J17 J19 J20 J21 J22 J23 J24 CON_IR
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10
R8 R9 R10 R11 R12 R13 R14 R15 R16 R17 4k7
R18 R19 R33 R34 R35 R36 R39 10K
C1 C2 C3 C4 C5 1uF
C13 C16 0.1uF
C14 C15 27pF
A2
+5V
DGND
MASTER_RESET_PIC
SN_PGC
SN_PGD
SN_RESET
SN_SDO
SN_SDI
SCK
SN_SS/M2_SS_PICMaster
M1_SS_PICMaster

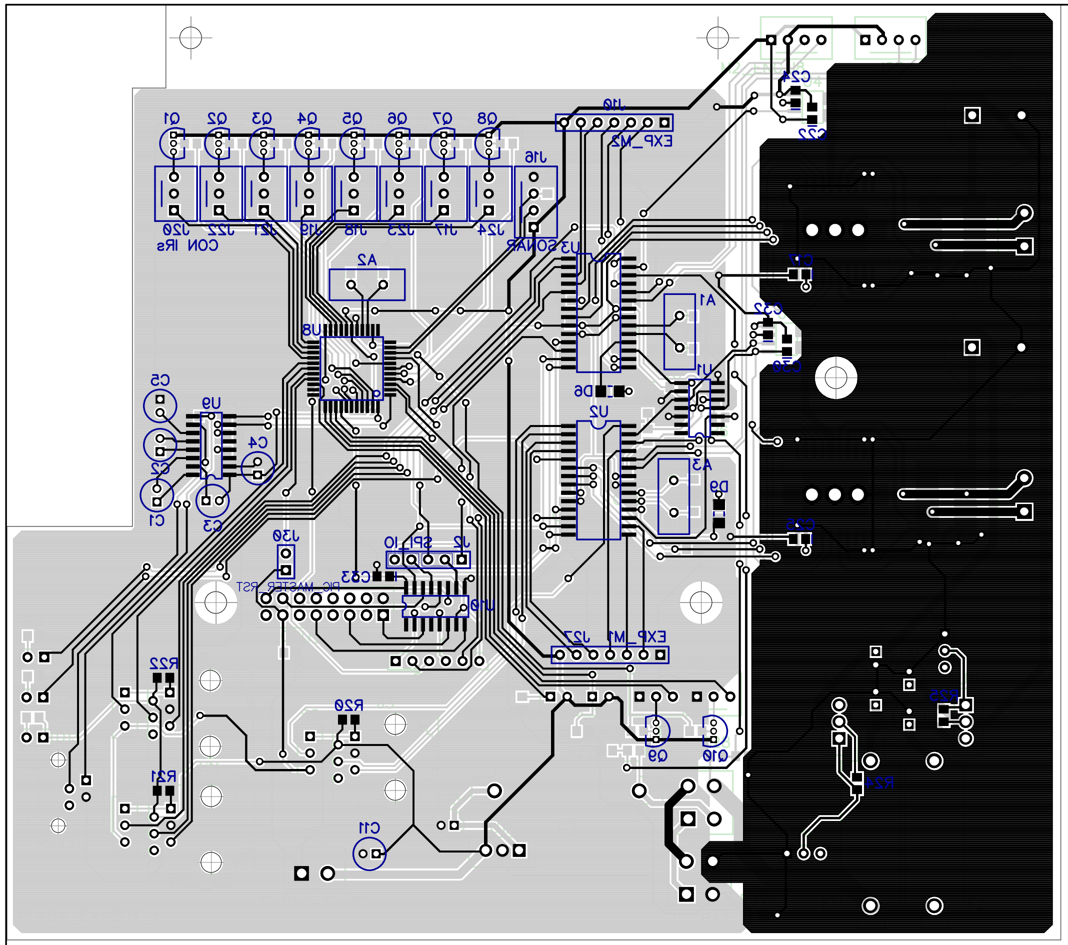

J3
3,3V NC
SCK DIO6
SDO NC
SDI DIO4
NC NC
NC DIO2
NC NC
GND DIO0
CON_PC104
C33
0.1uF
DGND
U10
VDD NC
G=A L=F
A F
H=B NC
B K=E
I=C E
C J=D
VSS D
CD4050B
SCK
m_sdo
m_sdi
m1_ss
sn_ss
m2_ss
SN_SS/M2_SS_PICMaster
J2
JMP_SPI
SLAVE_SDI
SN_SDO
SLAVE_SDO
SN_SDI
J30
MASTER_RESET_PIC
MASTER_RESET
R23
10K
J4
M1_SS
M1_SS_PICMaster
M2_SS
SN_SS/M2_SS_PICMaster
+5V
R20
10K
MCLR
J5
M1_PGD
M1_PGC
PGM_M1
M1_RESET
S2
R21
J6
M2_PGD
M2_PGC
PGM_M2
M2_RESET
S3
R22
J7
SN_PGD
SN_PGC
PGM_SN
SN_RESET
S4
A
B
C
D
E
F
1
2
3
4
Title
PC104 Connector
Size
Number
Rev
1
07/2009
Sol Pedre
Filename
con_pc104



Title
Power
Size
Number
Rev
1
07/2009
Sol Pedre
Filename
power
LM350
U13
LM350
U11
LM323
U12
VIN
VOUT
ADJUST
INPUT OUTPUT
GND
VR1
VR2
R24
240
R25
240
C6
0.1uF
C7
1uF
C8
0.1uF
C9
1uF
C10
1uF
C11
>0.1uF
F1
F2
F3
M1_Volta
M2_Volta
GNDEARTH
DGND
+5V
J29
POWER_PC104
D13
R38
470
R26
10
R27
18
LBAT_SENSE
R28
10
R29
18
MBAT_SENSE
J28
CON_INV_ON_OFF
J1
BAT
3+
3-
2-
2+
R37
0
A
B
C
D
E
F
1
2
3
4



Title		
ExaBot Top Side		
Size	Number	Rev
		1
07/2009	Sol Pedre	
Filename	Sheet	



Title ExaBot Bottom Side		
Size	Number	Rev 1
07/2009		Sol Pedre
Filename		Sheet

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs.* Elsevier, 2004.

[2] R. Cofer and B. Harding, *Rapid System Prototyping with FPGAs.* Elsevier, 2006.

[3] P. P. Chu, *FPGA prototyping by VHDL Examples.* Wiley, 2008.

[4] C. Spears, *SystemVerilog for Verification*, 2nd ed. Springer, 2008.

[5] S. Kilts, *Advanced FPGA Design. Architecture, Implementation and Optimization.* John Wiley & Sons, 2007.

[6] W. Wolf, *FPGA-Based System Design.* Prentice Hall, 2004.

[7] R. Siegwart and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*, 1st ed. MIT Press, 2004.

[8] T. Noergaard, *Embedded System Architecture - A Comprehensive Guide for Engineers and Programmers.* Elsevier, 2005.

[9] S. Heath, *Embedded Systems Design.* Newnes Publishers, 2003.

[10] P. Marwedel, *Embedded Systems Design.* Springer, 2006.

[11] A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *Computer*, vol. 38, no. 2, pp. 63–69, Feb. 2005.

[12] J. Ganssle and M. Barr. (2011, Oct.) Embedded Systems Glossary. [Online]. Available: http://www.netrino.com/Embedded-Systems/Glossary

[13] D. U. E. Blanza and C. U. E. Holland, "Embedded Market Survey," *EETimes and Embedded.com*, p. 84, 2012. [Online]. Available: http://seminar2.techonline.com/~additionalresources/esd_apr2012/ubme_embeddedmarket2012_full.pdf

[14] Altera. (2011, Oct.) Altera Introduces SoC FPGAs: Integrating ARM Processor System and FPGA into 28-nm Single-Chip Solution. [Online]. Available: http://www.altera.com/corporate/news_room/releases/2011/products/nr-soc-fpga.html

[15] Xilinx. (2011) Xilinx Introduces Zynq-7000 Family, Industry's First Extensible Processing Platform. [Online]. Available: http://press.xilinx.com/index.php?s=34135&item=18

[16] M. Santarini, "Zynq-7000 EPP sets stage for new era of innovations," *Xcell Journal*, vol. 75, pp. 8–13, May 2011.

[17] Reinaldo Bergamaschi et al., "The State of ESL Design [Roundtable]," *IEEE Design & Test of Computers*, vol. 25, no. 6, pp. 510–519, 2008.

[18] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A prescription for Electronic System-Level Methodology.* Morgan Kaufmann, 2007.

[19] B. Bailey and G. Martin, *ESL Models and their Application Electronic System Level Design and Verification in Practice.* Springer, 2010.

[20] S. Pedre, P. de Cristóforis, J. Caccavelli, and A. Stoliar, "A mobile mini robot architecture for research, education and popularization of science," *Applied Computer Science Methods*, vol. 2, no. 1, pp. 41–59, Feb. 2010.

[21] S. Pedre, A. Stoliar, and P. Borensztejn, "Real Time Hot Spot Detection Using FPGA," in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, ser. Lecture Notes in Computer Science, E. Bayro-Corrochano and J.-O. Eklundh, Eds. Springer Berlin Heidelberg, 2009, vol. 5856, pp. 595–602.

[22] S. Pedre, T. Krajnik, E. Todorovich, and P. Borensztejn, "A co-design methodology for processor-centric embedded systems with hardware acceleration using FPGA," in *2012 VIII IEEE Southern Conference on Programmable Logic (SPL)*, L. Agostini, E. Boemo, C. Zeferino, M. Glesner, and L. Rosa, Eds. IEEE, march 2012, pp. 1–8.

[23] S. Pedre, T. Krajník, E. Todorovich, and P. Borensztejn, "Hardware/Software Co-design for Real Time Embedded Image Processing: A Case Study," in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, ser. Lecture Notes in Computer Science, L. Alvarez, M. Mejail, L. Gomez, and J. Jacobo, Eds. Springer Berlin Heidelberg, 2012, vol. 7441, pp. 599–606.

[24] S. Pedre, T. Krajník, E. Todorovich, and P. Borensztejn, "Accelerating embedded image processing for real time: a case study," *Journal of Real Time Image Processing*, 2013, to appear.

[25] IFR. (2012, Aug.) International Federation of Robotics - Industrial Robots Statistics 2011. [Online]. Available: http://www.ifr.org/industrial-robots/statistics/

[26] R. Siegwart and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots - Chapter 1: Introduction*, 1st ed. MIT Press, 2004.

[27] IFR. (2012, Aug.) International Federation of Robotics - Service Robots Statistics 2011. [Online]. Available: http://www.ifr.org/service-robots/statistics/

[28] F. B. V. Benitti, "Exploring the educational potential of robotics in schools: A systematic review," *Computers & Education*, vol. 58, no. 3, pp. 978–988, Apr. 2012. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0360131511002508

[29] (2012, Aug.) RobotShop. [Online]. Available: http://www.robotshop.com/

[30] K-team corporation. (2013) Khepera, Khepera II and Khepera III. [Online]. Available: www.k-team.com

[31] Adept Mobile Robotics. (2013) Pioneer 2-DX and 3-DX. [Online]. Available: robots.mobilerobots.com

[32] C.-T. Chen, *Analog and Digital Control System Design - Transfer Function, State Space, and Algebraic Methods.* Sounders College Publishing, 2006.

[33] P. Horowitz and W. Hill, *The art of electronics*, 2nd ed. Cambridge University Press, 1989.

[34] T. Bräunl, *Embedded Robotics - Mobile Robot Design and Applications with Embedded Systems*, 2nd ed. Springer-Verlag, 2006.

[35] RoboticsConnections. (2012, Aug.) Traxster Kit. [Online]. Available: http://www.roboticsconnection.com/p-15-traxster-robot-kit.aspx

[36] P. Steggles and S. Gschwind, "The ubisense smart space platform," in *Third International Conference on Pervasive Computing*, 2005.

[37] N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar, " The GRASP multiple micro UAV testbed ," *IEEE Robotics and Automation Magazine*, 2010.

[38] "General Purpose Type Distance Measuring Sensors - GP2D120 Data sheet," Sharp, p. 10, may 2006. [Online]. Available: www.sharpsma.com/webfm_send/1205

[39] Devanatech. (2012, Aug.) SRF05 Ultrasonic Ranger documentation. [Online]. Available: http://www.robot-electronics.co.uk/htm/srf05tech.htm

[40] "ACS712 Fully Integrated, Hall-Effect-Based Linear Current Sensor IC with 2.1 kVRMS Voltage Isolation and a Low-Resistance Current Conductor Data Sheet," Allegro Microsystems Inc., p. 15, aug 2012, revised bla. [Online]. Available: http://www.allegromicro.com/Products/Current-Sensor-ICs/Zero-To-Fifty-Amp-Integrated-Conductor-Sensor-ICs/ACS712.aspx

[41] "TS-7250 Data Sheet," Embedded ARM, p. x, revised xxxx. [Online]. Available: www.embeddedarm.com/epc/ts7250-spec-h.html

[42] T. Wilmshurst, *Designing Embedded Systems with PIC Microcontrollers. Principles and Applications.* Elsevier, 2007.

[43] "L298 Dual Full Bridge Driver Data Sheet," STMicroelectronics, p. 13, revised 2000. [Online]. Available: www.st.com/internet/analog/product/63147.jsp

[44] "PIC18F2331/2431/4331/4431 Data Sheet - 28/40/44-Pin Enhanced Flash Microcontrollers with nanoWatt Technology, High-Performance PWM and A/D," Microchip, p. 392, sep 2010. [Online]. Available: www.microchip.com/wwwproducts/Devices.aspx?dDocName=en010271

[45] "LM319 Dual Comparator Data Sheet," Fairchild Semiconductor, p. 8, revised 2012. [Online]. Available: http://www.fairchildsemi.com/ds/LM/LM319.pdf

[46] K. Ogata, *Modern Control Engineering*, 4th ed. Pearson Prentice Hall, 2003.

[47] Cadsoft. (2012, Aug.) Eagle. [Online]. Available: www.cadsoftusa.com

[48] Novarm. (2012, Aug.) DipTrace. [Online]. Available: www.diptrace.com

[49] "L293 Datasheet - Push-Pull four channel driver with diodes," ST Microelectronics, p. 7, july 2003. [Online]. Available: www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00000059.pdf

[50] "DC Micromotors - Series 2224...SR Data Sheet," Faulhaber, p. 2, revised 2012. [Online]. Available: www.faulhaber.com/uploadpk/EN_2224_SR_DFF.pdf

[51] "LM78XX/LM78XXA Data sheet - 3-Terminal 1A Positive Voltage Regulator," Fairchild semiconductors, p. 28, August 2012. [Online]. Available: www.fairchildsemi.com/ds/LM/LM7805.pdf

[52] Microchip. (2012, Aug.) MPLAB Integrated Development Environment. [Online]. Available: www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en019469&part=SW007002

[53] "PIC18F2585/2680/4585/4680 Data Sheet - 28/40/44-Pin Enhanced Flash Microcontrollers with ECAN Technology, 10-Bit A/D and nanoWatt Technology," Microchip, p. 482, oct 2009. [Online]. Available: http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en010305

[54] "CD4049UB, CD4050B Data Sheet - CMOS Hex Buffer/Converters," Texas Instruments, p. 38, August 1998, revised May 2004. [Online]. Available: www.ti.com/lit/ds/symlink/cd4050b.pdf

[55] "MAX232, MAX232I Data Sheet - Dual EIA-232 Drivers/Receivers," Texas Instruments, p. 18, February 1989, revised March 2004. [Online]. Available: www.ti.com/lit/ds/symlink/max232.pdf

[56] "LM350 3.0 A, Adjustable Output, Positive Voltage Regulator Data Sheet," ON Semiconductor, p. 10, August 2006, revision 4. [Online]. Available: http://www.onsemi.com/pub_link/Collateral/LM350-D.PDF

[57] "LM323 Three-terminal 3 A adjustable voltage regulators Data Sheet," STMicroelectronics, p. 15, February 2008, revision 4. [Online]. Available: www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00000466.pdf

[58] M. A. Nitsche and P. Cristóforis, "Real-Time On-Board Image Processing Using an Embedded GPU for Monocular Vision-Based Navigation," in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, ser. Lecture Notes in Computer Science, L. Alvarez, M. Mejail, L. Gomez, and J. Jacobo, Eds. Springer Berlin Heidelberg, 2012, vol. 7441, pp. 591–598.

[59] P. De Cristóforis, M. Nitsche, T. Krajník, and M. Mejail, "Real-time monocular image based path detection," *Journal of Real Time Image Processing*, 2013, to appear.

[60] P. De Cristóforis, "Vision-based mobile robot system for monocular navigation in indoor/outdoor environments," Ph.D. dissertation, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2013.

[61] T. Piré, "Evasión de obstáculos en tiempo real usando visión estéreo," in *VII Jornadas Argentinas de Robótica*, Nov. 2012.

[62] T. Pire, P. de Cristóforis, M. Nitsche, and J. J. Berlles, "Stereo vision obstacle avoidance using disparity and elevation maps," in *IEEE RAS Summer School on Robot Vision and Applications*, 2012.

[63] J. Caccavelli, S. Pedre, P. de Cristóforis, A. Katz, and D. Bendersky, "A New Programming Interface for Educational Robotics," in *Research*

and Education in Robotics - EUROBOT 2011, ser. Communications in Computer and Information Science, D. Obdržálek and A. Gottscheber, Eds. Springer Berlin Heidelberg, 2011, vol. 161, pp. 68–77.

[64] P. De Cristoforis, S. Pedre, M. Nitsche, T. Fischer, F. Pessacg, and C. Di Pietro, "A Behavior-Based Approach for Educational Robotics Activities," *IEEE Transactions on Education*, vol. 56, no. 1, pp. 61–66, 2013.

[65] INVAP. (2012) INVAP. [Online]. Available: www.invap.com.ar

[66] M. Heimlicher and M. Oberholzer. (2010) FPGA Technology and Industry Experience. [Online]. Available: http://www.enclustra.com/assets/files/download/ETH_FPGA_Technology_and_Industry_Experience_100526.pdf

[67] P. Dillien. (2010) An Overview of FPGA Market Dynamics. [Online]. Available: http://www.soccentral.com/results.asp?CatID=488&EntryID=30730

[68] Xilinx. (2012) Field Programmable Gate Array. [Online]. Available: http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm

[69] P. P. Chu, *FPGA Prototyping by Verilog Examples.* John Wiley & Sons, 2008.

[70] M. Mintz and R. Ekendahl, *Hardware Verification with C++: A Practitioner's Handbook.* Springer, 2006.

[71] C. D. Systems, *Open Verification Methodology User Guide.* Cadence Design Systems, 2010.

[72] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient Embedded Computing," *Computer*, vol. 41, no. 7, pp. 27–32, Jul. 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4563875

[73] E. Salami, S. Pedre, P. Borensztejn, C. Barrado, A. Stoliar, and E. Pastor, "Decision Support System for Hot Spot Detection," in *Proceedings of the 5th International Conference on Intelligent Environments*, ser. IE'09. IOS Press, 2009, pp. 277–284.

[74] "SAA7113H - 9-bit video input processor Data sheet," Philips, p. 81, jul 1999. [Online]. Available: http://www.datasheetcatalog.org/datasheet/philips/SAA7113H_1.pdf

[75] "Dual-Port Block Memory Core v6.3," Xilinx, p. 20, Aug 2005. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/dp_block_mem.pdf

[76] "FIFO Generator v4.3," Xilinx, p. 19, March 2008. [Online]. Available: http://www.xilinx.com/products/intellectual-property/FIFO_Generator.htm

[77] S. Pedre, T. Krajník, E. Todorovich, and P. Borensztejn, "A co-design methodology for processor-centric embedded systems with hardware acceleration using FPGA," in *IEEE 8th Southern Programmable Logic Conference*. Brazil: IEEE, 2012, pp. 7–14.

[78] S. Pedre, T. Krajník, E. Todorovich, and P. Borensztejn, "Hardware/Software Co-design for Real Time Embedded Image Processing: A Case Study," in *CIARP*, 2012, pp. 599–606.

[79] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, September 2005. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&amp;path=ASIN/0262201623

[80] J. Bruce and M. Veloso, "Fast and accurate vision-based pattern detection and identification," in *IEEE International Conference on Robotics and Automation - ICRA*. IEEE, sept. 2003, pp. 1277 – 1282.

[81] G. Klančar, M. Kristan, S. Kovačič, and O. Orqueda, "Robust and efficient vision system for group of cooperating mobile robots with application to soccer robots," *ISA Transactions*, vol. 43, no. 3, pp. 329 – 342, 2004.

[82] N. Gunay and E. Dadios, "A robust and accurate color-based global vision recognition of highly dynamic objects in real time," in *8th Asian Control Conference (ASCC)*. IEEE, may 2011, pp. 90 –95.

[83] M. Brezak, I. Petrović, and E. Ivanjko, "Robust and accurate global vision system for real time tracking of multiple mobile robots," *Robotics and Autonomous Systems*, vol. 56, no. 3, pp. 213–230, Mar. 2008.

[84] R. Rao, C. Taylor, and V. Kumar, "Experiments in Robot Control from Uncalibrated Overhead Imagery," in *Experimental Robotics IX*, ser. Springer Tracts in Advanced Robotics, J. Ang, MarceloH. and O. Khatib, Eds. Springer Berlin Heidelberg, 2006, vol. 21, pp. 491–500.

[85] O. Keskin and E. Uyar, "A framework for multi robot guidance control," in *Holonic and Multi-Agent Systems for Manufacturing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5696, pp. 315–323.

[86] G. Klančar, D. Matko, and S. Blažič, "Wheeled mobile robots control in a linear platoon," *Journal of Intelligent and Robotic Systems*, vol. 54, pp. 709–731, 2009.

[87] M. Happe, E. Lubbers, and M. Platzner, "A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking," *Journal of Real-Time Image Processing*, pp. 1–16, 2011.

[88] R. Rodriguez-Gomez, E. Fernandez-Sanchez, J. Diaz, and E. Ros, "Codebook hardware implementation on FPGA for background subtraction," *Journal of Real-Time Image Processing*, pp. 1–15, 2012.

[89] J. Paul, A. Laika, C. Claus, W. Stechele, A. El Sayed Auf, and E. Maehle, "Real-time motion detection based on SW/HW-codesign for walking rescue robots," *Journal of Real-Time Image Processing*, pp. 1–16, 2012.

[90] OMG. (2006) UML Profile for System on a Chip (SoC) Version 1.0.1 - formal/06-08-01. [Online]. Available: www.omg.org/spec/SoCP/1.0.1/

[91] ——. (2010) Systems Modeling Language (SysML) Version 1.2 - formal/2010-06-01. [Online]. Available: www.omg.org/spec/SysML/1.2/

[92] ——. (2011, June) UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems Version 1.1 - formal/2011-06-02. [Online]. Available: www.omg.org/spec/MARTE/1.1/

[93] I. R. Quadri, A. Gamatie, P. Boulet, S. Meftali, and J.-L. Dekeyser, "Expressing embedded systems configurations at high abstraction levels with UML MARTE profile: Advantages, limitations and alternatives," *Journal of Systems Architecture*, vol. 58, no. 5, pp. 178 – 194, 2012.

[94] T. Arpinen, E. Salminen, T. D. Hämäläinen, and M. Hännikäinen, "MARTE profile extension for modeling dynamic power management of embedded systems," *Journal of Systems Architecture*, vol. 58, no. 5, pp. 209 – 219, 2012.

[95] A.G. Silva-Filho et al., "An ESL Approach for Energy Consumption Analysis of Cache Memories in SoC Platforms," *International Journal of Reconfigurable Computing*, vol. 2011, pp. 1–12, 2011.

[96] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, "A SoC Design Methodology Involving a UML 2.0 Profile for SystemC," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, ser. DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 704–709. [Online]. Available: http://dx.doi.org/10.1109/DATE.2005.37

[97] ——, "A model-driven design environment for embedded systems," in *Proceedings of the 43rd annual Design Automation Conference.* New York, NY, USA: ACM, 2006, pp. 915–918.

[98] W. Mueller, A. Rosti, S. Bocchio, E. Riccobene, P. Scandurra, W. Dehaene, Y. Vanderperren, and L. Ku, "UML for ESL Design - Basic Principles, Tools, and Applications," in *IEEE/ACM Int. Conf. on Computer Aided Design*, Nov. 2006, pp. 73–80.

[99] E. Riccobene and P. Scandurra, "Weaving executability into UML class models at PIM level," in *First European Workshop on Behaviour Modelling in Model Driven Architecture (BM-MDA).* Enschede, The Netherlands: CTIT Workshop Proceedings Series, 2009, pp. 10–28.

[100] F. Mischkalla, D. He, and W. Mueller, "Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, march 2010, pp. 1201 –1206.

[101] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet, "A co-design approach for embedded system modeling and code generation with UML and MARTE," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 226–231. [Online]. Available: http://dl.acm.org/citation.cfm?id=1874620.1874674

[102] Sparx Systems, "Visual Modelling Platform," Oct. 2011. [Online]. Available: http://www.sparxsystems.com/products/ea/

[103] Jacquard, "ROCCC 2.0," Oct. 2011. [Online]. Available: www.jacquardcomputing.com/roccc/

[104] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations," in *16th Intl. Conf. on VLSI Design.* IEEE, 2003, pp. 461–467.

[105] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV : Delftworkbench automated reconfigurable VHDL generator," in *Intl Conf on Field Programmable Logic and Applications.* IEEE, 2007, pp. 697–701.

[106] Nallatech, "DIME-C," Oct. 2011. [Online]. Available: www.nallatech.com/Development-Tools/dime-c.html

[107] Mentor-Graphics, "CatapultC," Oct. 2011. [Online]. Available: www.mentor.com/esl/catapult/overview

[108] Xilinx. (2012, Nov.) Vivado Desing Suite. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/index.htm

[109] Virginia et al., "An empirical comparison of ANSI-C to VHDL compilers : SPARK, RORCC and DWARV," in *Anual Workshop on Circuits Systems and Signal Processing ProRISC*, 2007, pp. 388–394.

[110] B. D. Technology, "The AutoESL AutoPilot High-Level Synthesis Tool," Tech. Rep., 2010.

[111] J. Gaisler, "A structured VHDL design method," in *Fault-tolerant Microprocessors for Space Applications*. Gaisler Research, 2004, pp. 41–50. [Online]. Available: http://www.gaisler.com/doc/vhdl2proc.pdf

[112] ESA, "European Space Agency VHDL - www.esa.int," Oct. 2011. [Online]. Available: http://www.esa.int/TEC/Microelectronics/SEMS7EV681F_0.html

[113] J. Gaisler. (2011, Oct.) A structured VHDL Design Method. [Online]. Available: http://www.gaisler.com/doc/structdes.pdf

[114] Xilinx. (2011, Oct.) Platform Studio and the Embedded Development Kit (EDK). [Online]. Available: http://www.xilinx.com/tools/platform.htm

[115] M. Kulich, J. Chudoba, K. Kosnar, T. Krajnik, J. Faigl, and L. Preucil, "SyRoTek-Distance Teaching of Mobile Robotics," *IEEE Transactions on Education*, vol. 56, no. 1, pp. 18 –23, feb. 2013.

[116] Z. Zhang, "Flexible camera calibration by viewing a plane from unknown orientations," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 1, 1999, pp. 666 –673 vol.1.

[117] J. Y. Bouguet, "Camera calibration toolbox for Matlab," 2008. [Online]. Available: http://www.vision.caltech.edu/bouguetj/calib_doc/.

[118] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[119] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. Webb Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research : A View from Berkeley," University of California at Berkeley, Tech. Rep., 2006. [Online]. Available: www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html