Automatic Platform Synthesis and Application Mapping for Multiprocessor Systems On-Chip

MASTER'S THESIS

by

Kai Huang kkhuang@liacs.nl

Ji Gu jgu@liacs.nl

Leiden Embedded Research Center
LIACS - Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Supervisors: Dr.ir. Todor Stefanov (LIACS - Leiden University)
Prof.dr.ir. Ed F. Deprettere (LIACS - Leiden University)

The work in this thesis was carried out in the context of the Artemisia project supported by PROGRESS/STW.

Printed in the Netherlands

Contents

Acknowledgments

This thesis is the result of a research work performed at the Leiden Embedded Research Center of the Leiden Institute of Advanced Computer Science (LIACS) - Leiden University. We would like to thank all the people who guided and supported us during our research.

First of all, we would like to thank Prof. Ed Deprettere for giving us the opportunity to do our Master's research at the Leiden Embedded Research Center.

Special thanks to our supervisor Todor Stefanov, who provided us with the most support and guidance. He was always willing to spend time helping us to solve difficult problems related to our research work presented in this thesis. Without him we could not have overcome all the difficulties we found along our research path.

Kai Huang & Ji Gu
Leiden, The Netherlands
August 30, 2005

Chapter 1

Introduction

For modern embedded systems, the complexity of embedded applications has reached a point where the performance requirements of these applications can no longer be supported by embedded system architectures based on a single processor. Meanwhile, the Moore's law predicts *exponential growth* over time of the number of transistors that can be integrated in an IC. It predicts that chips in 2010 will count over 4 billion transistors, operating in the multi-GHz range [1]. Therefore, the emerging embedded System-on-Chip platforms are increasingly becoming multiprocessor architectures. Multiprocessors can provide enhanced processing efficiency by exploiting parallelism between loops, functions, or even coarse-grained tasks. As in the case of a uniprocessor architecture, multiprocessor architectures are much more efficient when each of the processors can be customized to a specific task it performs. Moreover, there is a tendency that the multiprocessor Systems-on-Chip (MPSoCs) need to support ever-increasing functionality and complexity of applications while being subject to stringent performance and power consumption requirements. Also, MPSoCs need to be flexible enough that the design can be re-used between different product variants or versions, and easily modified in response to bugs, market shifts, or user requirements, during the design cycle and even after production.

All these lead to the fact that existing design methodologies and tools can no longer keep up with the trends because they cannot deal with such complex and highly flexible systems. Without a disciplined design methodology, system designers will have to resort to *ad hoc* techniques to implement concurrent applications on complex multiprocessor platforms, which is a doubtful proposition. We believe that new design methodologies should be introduced with regard to the following two concepts and approaches:

- First, the sequential languages widely used to specify applications will no longer match the increasing amount of parallelism that will be enforced by multiprocessor platforms. Thus, there is a need for a parallel language or more realistic, a translator to convert sequential specifications into parallel specifications.

- Second, methods have to be designed to map application models onto platform models. This includes techniques and tools to automatically and systematically map applications onto hardware platforms, meeting severe system performance and cost requirements, in a relatively short amount of time.

In the system design community, a few challenges are agreed upon in order to master the ever growing complexity of Embedded Systems-on-Chip. The first challenge is the application specification. Applications have to be specified in some parallel language and modeled at a high level of abstraction. Currently, applications are specified using sequential programming languages like C or Matlab. The lack of appropriate methodology and tool support for extracting and modeling of concurrency in its various forms is an essential limiting factor in commonly used programming languages to express design complexity and to exploit parallelism available in applications. The second challenge is the platform specification. Platforms have to be specified in a parameterized form and modeled at a high level of abstraction. Today, designers are familiar with working at levels of abstraction that are too close to implementation. Therefore, sharing design components and verifying designs before prototypes are built is nearly impossible. For most designers the highest level of abstraction of their design (platform) is the register transfer level (RTL). The RTL level is clearly too low for complex platform design. The third challenge is the mapping. Methods have to be provided to systematically and automatically map the application models onto platform models in terms of system performance and cost in a relatively short amount of time.

The three challenges presented above are closely related and equally important. Each challenge has its own specific problems that have to be solved. The problems further discussed in this thesis and the proposed solutions are related to the second and third challenge. This thesis focuses on methods, techniques, and tools for systematic and automated mapping of a parallel application model onto multiprocessor platforms.

This chapter is further organized as follows. In Section 1.1, we state the actual problem that we want to solve. A description of the approaches and techniques we propose to solve this problem is given in Section 1.2. Section 1.3 gives a brief overview of the related work and Section 1.4 summarizes the main contributions of this thesis. Finally, Section 1.5 describes the organization of this thesis.

1.1 Problem Description

Applications in the realm of high throughput multimedia, imaging, and digital signal processing usually consist of a variety of complex algorithms, such as FFT, DCT, image/video codecs, and modems. They perform highly repetitive arithmetic tasks and demand extremely high processing performance on the platforms. To achieve this performance, the emerging embedded systems on a chip for applications in this realm, have to be multiprocessor platforms, thereby allowing task-level parallelism available in applications to be exploited efficiently.

Fortunately, the state-of-the-art technologies allow us to build very complex multiprocessor platforms. Three examples are the *Picochip* from PicoChip [2], the *VirtexII-Pro* from Xilinx [3], and the *SpaceCAKE* architecture [4] from Philips. The PicoChip combines 308 simple RISC processors on a single die. Xilinx combines FPGA technology with four embedded PowerPC processors on their VirtexII-Pro chips. The SpaceCAKE architecture is a homogeneous network of tiles where each tile consists of a heterogeneous mix of memories, CPUs like MIPS or ARM, DSPs, and hardware IP cores. An abstract model of such platforms is shown in the bottom part of Figure 1.1. This model is composed of fully programmable components (CPUs),

reconfigurable components (RPUs), and dedicated hardware blocks (IP cores). These components are linked via some kinds of communication structure, e.g., high speed on-chip bus or multiple buses. This type of multiprocessor platforms implies that task-level parallelism can be exploited to satisfy the performance needs of applications.

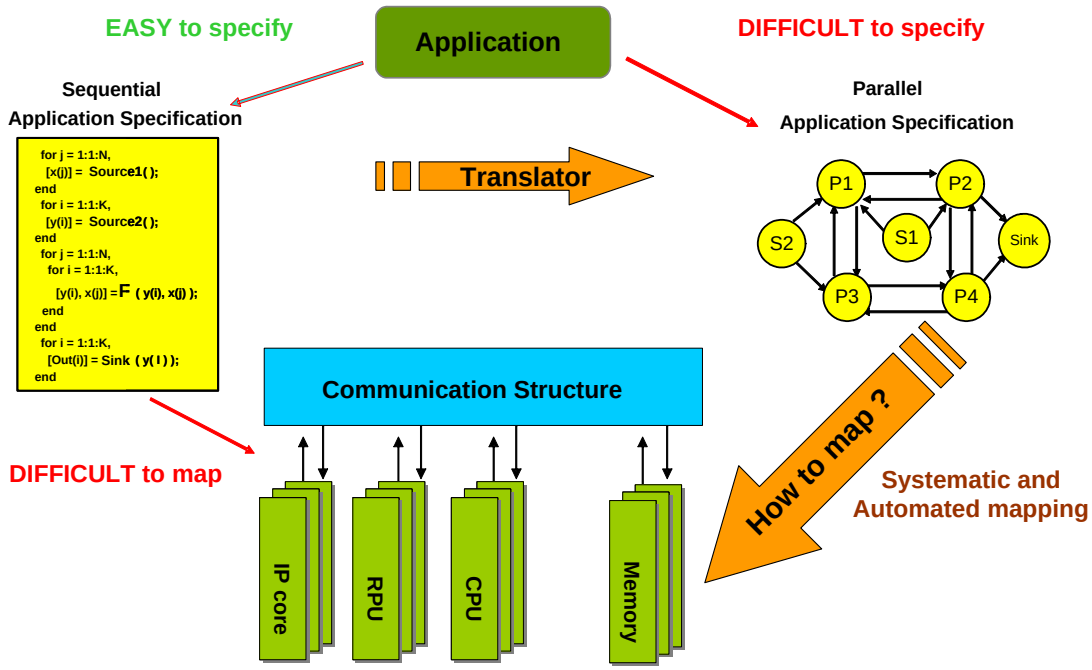

Figure 1.1: The mapping problem.

However, achieving high performance when mapping applications onto multiprocessor platforms currently depends very much on the expertise of the hardware designer, who has to possess an accurate knowledge of the underlying hardware platforms and applications. Moreover, the mapping of applications onto this type of platforms is in most cases done manually, which leads to a slow, difficult, and error prone design process. Embedded system designers are faced with expanding array of challenges in both application and platform design. One challenge is the task of modeling the concurrency in an application. Another is the mapping of the concurrent model to multiprocessor platforms. Therefore, methodologies need to be developed that allow efficient and effective mapping of a class of multimedia and signal processing applications onto multiprocessor platforms in an automated and systematic way.

Mapping applications onto multiprocessor platform is difficult because the way an application is specified does not match the way a multiprocessor platform operates. This mapping problem is shown in the left part of Figure 1.1. The multiprocessor platform has components that can run concurrently, since the control is distributed over the components and the memory is distributed as well. To satisfy the high performance needs of applications, such platform must be programmed in a way that all components that comprise the multiprocessor platform run as concurrently as possible. This implies that the parallelism available in an application must be revealed and exploited efficiently. However, most of the applications are typically specified as sequential programs using a high-level programming language such as C/C++ or Matlab. Such specifications do not reveal parallelism due to their inherent sequential nature. The sequential model of computation makes it easy to reason about a program, as only a single memory

and a single thread of control need to be considered. But the single memory and single thread of control are contradictory to the need for distributed control and distributed memory for the platform. So, an abstract concurrent model is needed to reveal the implicit concurrency of the application. A translator, such as COMPAAN [5], can be used to convert the sequential application to an abstract concurrent model, which is show in the right part of Figure 1.1. This model consists of several concurrent tasks making the task-level parallelism available in an application explicit.

Now the challenge is how to map the concurrent model onto a multiprocessor platform. The Moore's law has described the exponential growth over time of the number of transistors that can be integrated in a single chip. The intrinsic computational power of a chip must not only be used efficiently and effectively, but also the time and effort to design a system containing both hardware and software must also remain acceptable. Unfortunately, current platform design methodologies are still based on Register Transfer Level (RTL) platform descriptions created in Verilog/VHDL by hand. Such methodologies were effective in the past. Applications and platforms used in many of today's new system designs are so complex that traditional design practices are now inadequate, because creating RTL descriptions of complex multiprocessor platforms becomes error-prone and time-consuming. Moreover, the complexity of high-end, computationally intensive applications in the realm of high throughput multimedia, imaging, and digital signal processing enlarges the difficulties associated with the traditional hand-coded RTL design. At the same time, using traditional logic simulation to verify a large design represented in RTL is computationally expensive and extremely slow.

From all these reasons, we can conclude that the bottleneck is the use of a RTL system specification as a starting point of multiprocessor system design methodologies. Although the RTL system specification has the advantage that the state of the art synthesis tools can use it as an input to automatically implement a system, the system design community believes that a system should be specified at a higher level of abstraction called System-Level. This is the only way to solve the problems caused by the low level RTL specification discussed above. Moving up from the detailed RTL specification to a more abstract System-Level specification opens a gap which we call *Implementation Gap*. Indeed, on the one hand the RTL system specification is very detailed and close to an implementation, thereby allowing automated system synthesis path from RTL system specification to implementation. This is obvious if we consider the current commercial synthesis tools where the RTL-to-netlist synthesis is very well developed and efficient. On the other hand, the complexity of the today's systems forces us to move to higher levels of abstraction when designing a system, but currently we do not have mature methodologies, techniques, and tools to go back from the high-level system specification to an implementation. Therefore, the *Implementation Gap* has to be closed by finding a systematic and automated way to effectively and efficiently convert a System-Level specification to a RTL-Level specification.

1.2 Solution Approach

In this section we give an overview of the solution approach and the techniques we have developed to close the *Implementation Gap*, described in Section 1.1, in a particular way. Figure 1.2 shows our approach integrated in a system design flow.

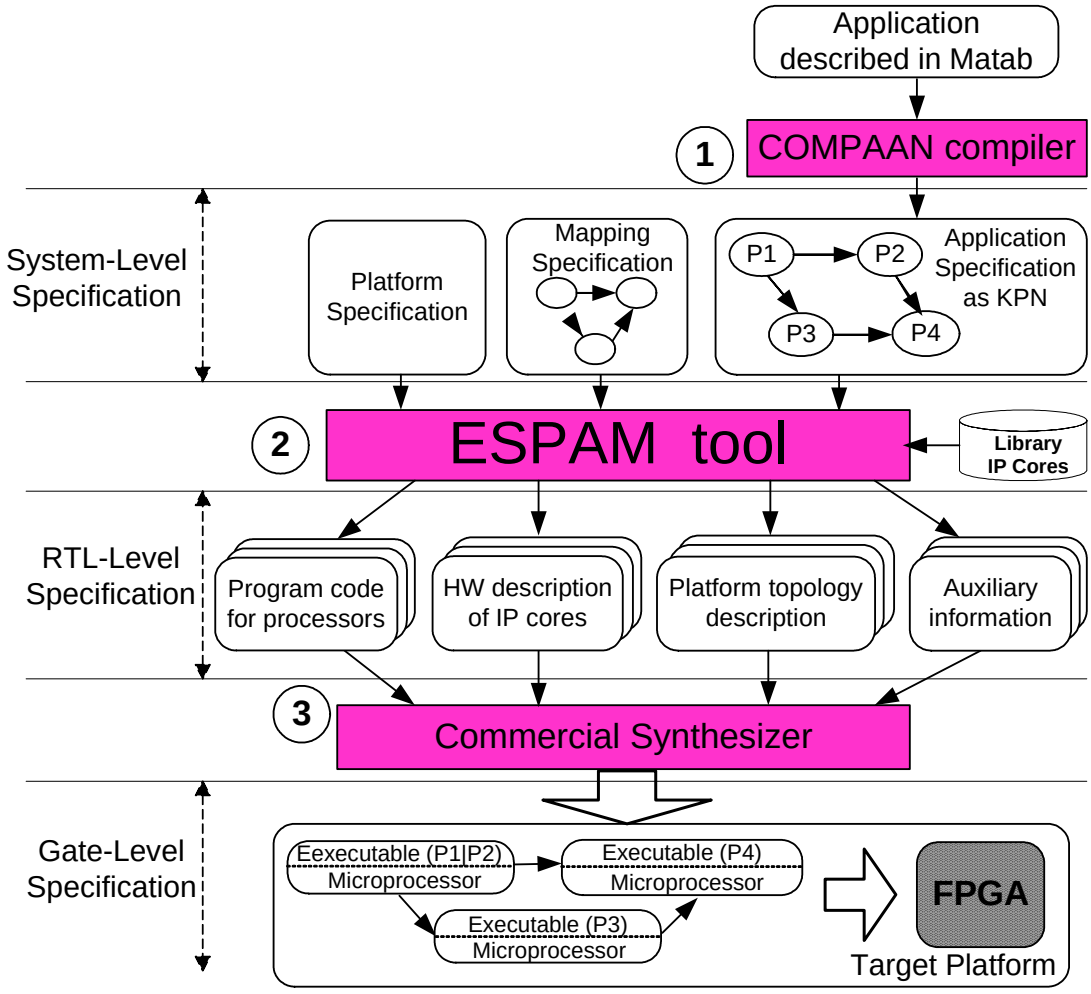

Figure 1.2: System design flow.

There are three levels of specifications in our system design flow. They are *System-Level* specification, *RTL-Level* specification, and *Gate-Level* specification.

On the top of Figure 1.2 is the *System-Level* specification, which consists of three parts:

- *Platform Specification*: It specifies the topology of a platform using generic parameterized system components. There are two types of components, namely processing components and communication components. The processing components are processor component, IP component, memory component, and controller component. The communication components are FIFO component, crossbar component, and bus component.

- *Application Specification*: It specifies an application as a Kahn Process Network (KPN) where a number of concurrent processes are connected in a network and they communicate data via FIFO channels. The KPN specification reveals the task-level parallelism available in the application.

- *Mapping Specification*: It specifies the relation between all processes and FIFO channels of *Application Specification* and all components of *Platform Specification*, i.e., for each

processing component in *Platform Specification* which processes from *Application Specification* are mapped onto it as well as for each communication component in *Platform Specification* which FIFO channels are mapped onto it.

In this thesis, we consider only one-to-one mapping, which means the following: First, the number of processor components in *Platform Specification* is equal to the number of processes in *Application Specification*, a process is mapped onto only one processor, and each processor has only one process mapped onto it. Second, a channel in *Application Specification* is mapped onto a FIFO component in *Platform Specification* and each FIFO component has only one channel mapped onto it, so that all the connections are point-to-point connections. Since we consider the one-to-one mapping described above, *Platform Specification* and *Mapping Specification* are straightforward and ESPAM derives them automatically according to the topology of the KPN described in *Application Specification*. Therefore, currently the input of ESPAM is only *Application Specification*.

To reveal the implicit parallelism inside an application, a concurrent model is needed to explicitly describe the task-level parallelism in *Application Specification*. Also, the data dependencies and the communications among tasks are needed to be explicit via distributed memory buffers in the model. There are many existing parallel models of computation [6] [7]. In this thesis we have chosen the Kahn Process Network (KPN) model of computation [8] because its operational semantics are simple, yet general enough, to specify conveniently stream-oriented data processing that fits nicely with the application domain of high throughput multimedia and signal processing applications described in Section 1.1. We describe the KPN model in detail in Section 2.1. Although, the KPN parallel model of computation is very suitable for multiprocessor platforms mapping, specifying an application manually using such a model is a difficult, error prone and time consuming process. To facilitate the migration from a sequential specification of an application to an equivalent KPN specification, we use the COMPAAN compiler, i.e., Step 1 in Figure 1.2. This compiler, introduced in [5] and further developed in [9] [10], fully automates the transformation of Matlab code into KPN specification. The applications that COMPAAN can handle have to be specified as parameterized static affine nested loop programs, which is a subset of the Matlab language. This conversion is fast and correct by construction.

Central to the design flow is our ESPAM tool, which is shown in Step 2 in Figure 1.2. Our ESPAM tool is designed to bridge the gap between the *System-Level* specification of a system and the *RTL-Level* specification of a system. The state-of-art RTL-based design approaches are error-prone and time-consuming procedures which are lagged behind the design complexity because of the continuing exponential growth of the on-chip transistor count. By using our ESPAM tool, a correct-by-construction mapping process is achieved by systematically and automatically converting a *System-Level* specification to a *RTL-Level* specification. This allows the design process to be moved up to a higher abstraction level and later this provides a systematic and automated path to an implementation. ESPAM first converts the given KPN specification into an equivalent network of processors in which no information on the target physical platform is taken into account. The processors network is composed of generic parameterized system components. Then ESPAM instantiates the generic system components by setting parameters based on the target physical platform, by which an elaborate platform specification is obtained for an implementation on the target platform. Finally, ESPAM generates program code files for each processor in the processors network. Detailed description of ESPAM is given in

Chapter 2 and Chapter 3.

The output of ESPAM, namely a *RTL-level* specification of a system is a model that can adequately abstract and export the key features of the target programmable platform at the register transfer level. It consists of four parts:

- *Platform topology description* defines in great detail the processors network (multiprocessor platform) in which all generic parameterized system components are used as well as the connections between these components are specified. It has the same topology as *Platform Specification* in the *System-Level* specification. The difference is that it describes the network in a lower level with more details.

- *Hardware descriptions of IP cores* contains all predefined IP cores as well as reconfigurable IP cores used in *Platform topology description*. ESPAM selects the predefined IP cores from *Library IP Cores*, see Step 2 in Figure 1.2, and generates the reconfigurable IP cores according to the KPN specification of an application.

- *Program codes for processors* are program source code files for each processor component in the processors network. To execute an application on the synthesized multiprocessor platform, the platform has to be programmed, which means writing program source code files for each processor in the platform using high level programming languages like C/C++. ESPAM generates source code files in C for each processor component according to the behavior of the corresponding process in the KPN.

- *Auxiliary information* contains supply files which give tight control of the overall specifications, such as defining precise timing requirements and prioritizing signal constraints.

With the above descriptions, a commercial synthesizer can convert a *RTL-Level* specification to a *Gate-Level* specification, thereby generating the target platform gate-level netlist, see the bottom part of Figure 1.2. This *Gate-Level* specification obtained in Step 3 of our system design flow is actually the system implementation.

The ESPAM tool together with the COMPAAN compiler, allows a fully automated system design flow that maps sequential applications written in Matlab onto multiprocessor platforms. This automation significantly reduces the design time of a system as well as possible errors in the mapping process are eliminated. Using our system design flow, the design focus can be moved up to the high *System-Level* or even to more abstract level, namely the sequential application, without sacrificing the possibility for automatic and systematic design implementation. Thus, our design flow closes in a particular way the *Implementation Gap* described in Section 1.1.

1.3 Related Work

Systematic and automated application-to-architecture mapping has been widely studied in the research community.

The closest work to our work is the LAURA tool [11] which has been developed at the Leiden Embedded Research Center (LERC). LAURA accepts a KPN specification and converts

this KPN specification together with predefined non-programmable IP cores into synthesizable VHDL code. The KPN is generated by COMPAAN from an application described as a Matlab program. The IP cores are needed preemptively as they implement the functionality of the functions used in the initial Matlab program. On the contrary, our ESPAM tool map KPN specifications to multiprocessor platforms. The functions used in the initial Matlab program can be mapped to programmable processor cores and run on top of them as software, which gives much more flexibility in the system implementation.

An automatic logic synthesis method has been presented in [12]. This automated synthesis is focused on the mapping of the functionality of an asynchronous logic directly to the FPGA. A hardware description language called CHP is used to describe the sequential program. At the same time, their mapping is limited to a pipelined architecture. On the contrary, in our design flow, more abstract programming languages are supported, e.g., C and Matlab. Besides the pipelined architecture, more flexible parallel system architectures can be mapped to the target platform.

In Philips Research Laboratory, a top-down design methodology called C-HEAP [13] is introduced which starts with a functional description and proceeds to a silicon implementation in an incremental way. Seven abstraction levels that are traversed throughout the design process have been identified. It also proposes an architecture template based on distributed shared memory that allows for the use of a variety of processing devices. Our design flow is similar to this. We introduce four levels, e.g., application level, system level, RTL level and Gate level, and we traverse them from the application level to system level using COMPAAN, from system level to RTL level using ESPAM then to Gate level by a commercial synthesis tool. Another major difference is that our platform model uses distributed memory instead of a shared memory.

Another similar work has been presented in [14]. This work is focused on synthesis of application specific multiprocessor System-on-Chip architectures for process networks of streaming applications. Our work is close to this, but the key difference is that they map the channels of the KPN model onto shared memories. Therefore, possible data communication conflicts need to be estimated and taken into account in the mapping process. In contrast, in our methodology, the communication is distributed over hardware FIFO buffers. There is no notion of a shared memory that has to be accessed by multiple processors. Therefore, resource contention does not occur.

A few projects deal with high level analysis of systems on chip, such as ZIPPY [15] and AEthereal [16]. The ZIPPY project follows a systematic design methodology to investigate architectures for dynamically reconfigurable processors for the domain of handhelds and wearables. It intends to design a domain-specific, hybrid, dynamically reconfigurable processor. It is a processor co-processor architecture optimized on instruction level in which the most used instruction set is executed by a co-processor. The AEthereal Network-On-Chip project tries to offer guaranteed services to obtain a QoS-based design by using a mix of time-division-multiplexed circuit switching and packet switching. A network protocol stack is adopted to model the communication among different services. All these projects present novel ways for exploring embedded system on chip at system level. However, there is still no runnable prototyping system, let alone systematic and automated implementation path, therefore all candidate designs can only be verified by simulation.

There are a number of exploration environments, such as SPADE [17], Sesame [18], MMM [19]

and Polis [20], that facilitate flexible system-level design space exploration by providing support for mapping a behavioral application specification to an architecture specification. SPADE is a method and tool for architecture exploration of heterogeneous signal processing systems. SPADE supports the construction of abstract executable models for evaluation of alternative architectures by modeling applications and architectures as well as capturing the mapping of application models onto architecture models on the system level by using a library of generic building blocks. The Sesame project, which builds up on the ground-laying work of Spade, uses Y-Chart Modeling Language (YML) to describe the application model and uses the Pearl discrete-event simulation language [21] to describe the architecture model. A mapping layer is developed to map the application model onto the resources in the architecture model. In the MMM project, a three-stage process network refinement-based approach has been proposed for heterogeneous multiprocessor mapping of process networks. All these projects offer trace-driven co-simulation of application and architecture models, yet there is still no runnable prototype. Another problem in these three approaches is that modeling the application as a process network and mapping have to be done manually. The Polis environment provides a totally automated flow from high-level specifications such as ESTEREL, down to performance optimized machine code for a reconfigurable target architecture. It uses an intermediate model of computation called Extended Finite State Machines. This model is well suited for control dominated applications, but less for stream oriented applications which is the application domain targeted by our system design approach.

A lot of multiprocessor platform manufacturers offer RTL-level design tool chains. Three examples are the picoTools from PicoChip [2], the EDK and Platform Studio from Xilinx [3], and the SpaceCAKE [4] environment from Philips. PicoChip's newly picoArray processor, namely PC102 has 308 processors. It divides tasks across multiple, independent, processors in the array. In picoTools, the relationships and block diagram structure among different tasks is specified in structural VHDL. Programmers can use ANSI C to program the individual elements. For time critical blocks, the programmer needs to write an assembly code. The Xilinx Embedded Development Kit (EDK) and the Platform Studio form a comprehensive suite which allows designers to configure a HW/SW platform including automatic generation of device drivers, application code, and Board Support Packets (BSPs) for their VirtexII-Pro chips which combine FPGA technology with four embedded PowerPCs. The SpaceCAKE environments uses YAPI [22] as modeling language for its homogeneous multiprocessing platform. The above tools move up the design process from the gate level to the RTL level, which reduces the design time and complexity. However, the RTL level is still too low for future complex system design. Therefore, the above tools can be used as a back-end to ESPAM, see Step 3 in Figure 1.2.

In the embedded design industry, there have been a few attempts to find a path that takes high-level programming language specification of applications and automatically transforms them into efficient hardware designs. The CriticalBlue [23] has developed a tool called Cascade that automatically generates processor-coprocessor system at RTL level and a testbench from compiled executable software code. This synthesis tool extracts parallelism directly from the compiled code exploiting the full power of standard software languages such as C/C++. Tensilica's XPRES Compiler [24] is another synthesis tool that creates tailored processor descriptions for the Xtensa processor from native C/C++ code. Designers can use the XPRES compiler to synthesize highly optimized processor RTL codes directly from C/C++ reference code or algorithmic specifications. The Celoxica [25] has a compiler, i.e., Agility which provides be-

havioral design and synthesis with SystemC. It can automatically generate IEEE compliant RTL (VHDL and Verilog HDL) from SystemC source code. Mentor Graphics has also developed a synthesis tool, i.e., Catapult C [26]. The Catapult C uses industry-standard C++ source code augmented with SystemC data types, which allows specific bit-widths to be associated with variables and constants, to employ models in a non implementation-specific representation. AccelChip's product [27], namely AccelChip DSP Synthesis provides an automated implementation and verification flow for DSP algorithms, developed using Matlab and targeting ASIC or FPGA devices. The final output is a synthesizable RTL VHDL or Verilog model optimized for the specified device and automatically verified against the initial Matlab source. The above mentioned tools focus on generating optimized hardware for system that are described using hight-level programming languages. However, the generated hardware is limited to either a single processor system or a processor-coprocessor system, hence these tools cannot fully exploit the task-level parallelism existing in an application.

1.4 Research Contributions

The work presented in this thesis focuses on systematic synthesis of application-specific multiprocessor platforms and automated mapping of applications onto these platforms. We make the following research contributions:

- We bridge the gap between the system-level design and the RTL-level design in a particular way. We present a novel way of mapping an application onto a multiprocessor platform. Using our approach, an application which is described in a system-level specification as KPN can be systematically and automatically converted to a RTL-level specification for a multiprocessor platform. The automated and systematic approach is correct-by-construction and the automation reduces significantly the design time of a system and possible errors in the mapping process are eliminated. Therefore, the design focus can be move up to the system level and even to the sequential application. The proposed mapping approach has been implemented as part of a software tool called ESPAM.
- We have implemented the proposed system design method in the context of a commercial configurable soft processor design flow (using MicroBlaze embedded soft processor of Xilinx Inc.) and prototyped the multiprocessor platform on an FPGA-based board. The benefit of using a soft processor core is the ability for designers to build systems with multiple processors using the same FPGA. The number of processors a designer can incorporate within any given FPGA is only limited by the size of the FPGA itself.

1.5 Thesis Organization

The remaining part of this thesis is organized as follows. Chapter 2 presents our approach for systematic synthesis of multiprocessor platforms and automatic mapping of applications. This chapter describes in great detail the methods and techniques we have developed and used. First, we describe the way to model an application specified as a sequential program. Second, we

present our approach for synthesizing multiprocessor platforms for KPN specifications. The synthesis process consists of several steps, which are selection of reusable components, synthesis of communication structure, mapping of processes of a KPN onto processor components, and mapping channels onto FIFO components. Finally, we explain how to program the multiprocessor platforms.

Chapter 3 explains how we prototype our multiprocessor platforms onto a Field Programmable Gate Array (FPGA) chip. To prototype a multiprocessor platform onto an FPGA, we describe how to implement each component of the platform on the FPGA chip. Specifically, as we use a single Xilinx Virtex-II Pro FPGA for prototyping, we use configurable MicroBlaze embedded soft processor core for each processor in the platform. For the hardware FIFO buffers, we instantiate predefined generic FIFO IP cores. Besides, the connection between the MicroBlaze soft processor cores and the hardware FIFO buffers, the control mechanism of the FIFOs will also be elaborated.

In Chapter 4 we present two case studies that we conducted in order to validate and evaluate our approach presented in Chapter 2 on real-life applications. We analyze the results obtained from the experiments performed in these case studies.

In Chapter 5 we give a tutorial showing how to build a multiprocessor embedded system using our COMPAAN/ESPAM tool chain and the commercial synthesis tool Xilinx Platform Studio. In the tutorial, we use a complex application, namely an M-JPEG encoder as our example to explain each step of our design flow in detail. In the final chapter we give some conclusions and recommendations for future work.

Chapter 2

Multiprocessor Platform Synthesis and Application Mapping

In Chapter 1 we discussed that in order to map efficiently sequential applications onto multiprocessor platforms, we first need to specify them as Kahn Process Networks (KPN). This is because on the one hand KPN specifies an application as a composition of concurrent process where the computation, control, and memory are distributed. On the other hand, the multiprocessor platforms have components that run concurrently, i.e, the computation and control are distributed over the components. Thus the KPN parallel processing model matches the multiprocessor platforms very well, thereby mapping of KPN specifications onto multiprocessor platforms can be done in a systematic and automated way. We discussed that we can use the COMPAAN compiler [5] [9] [10] to facilitate the migration from a sequential application specification to an equivalent KPN specification. Also, we proposed a systematic approach implemented as part of the ESPAM tool to automatically synthesize multiprocessor platforms and map applications onto the platforms.

In this chapter we present our approach and system design flow depicted in Figure 1.2. We elaborate in more details on the techniques and tools integrated in our flow in order to show how for an application written in Matlab, a Kahn Process Network specification can automatically be derived and systematically mapped onto a multiprocessor platform. To make the design flow specific, we demonstrate our design flow in the context of implementing a matrix multiplication application onto a multiprocessor platform. In Section 2.1, we first describe the KPN model of computation for specifying applications. We introduce the COMPAAN compiler that we use as a tool to automatically convert sequential programs written in Matlab into KPN specification. Then in Section 2.2, we present our systematic approach for automatic synthesis of application-specific multiprocessor platforms for applications expressed as KPNs. The synthesis process involves selection of computation modules and communication components from the platform component library defined in ESPAM. We construct the multiprocessor platform by instantiating a processor for each process and a hardware FIFO buffer for each channel in the KPN model. This leads to multiprocessor platform that has the same topology as the input KPN model of the original application. In Section 2.3, we discuss the programming of the multiprocessor platform. Each process in the KPN is specified as a sequential program that is modeled as a syntax tree by ESPAM. A software engineering technique called $visitor$, implemented in

ESPAM, is able to traverse such a tree and generate a program code for each processor in the multiprocessor platform.

2.1 Application Modeling

As we map an application onto a multiprocessor platform, we have to expose task-level parallelism available in the application and make communications explicit. This means that the sequential application has to be specified as a parallel model of computation (MOC) in order to be mapped onto the multiprocessor platform in a systematic and efficient way. In our system design approach, we use the Kahn Process Networks [8] (KPNs) model of computation for application specification.

2.1.1 Kahn Process Networks

The KPN model of computation assumes a network of concurrent autonomous processes that communicate in a point-to-point fashion over unbounded FIFO channels, using a blocking-read synchronization primitive. A simple example of the KPN model is shown in Figure 2.1. The three concurrent processes A, B and C are connected via FIFO channels.

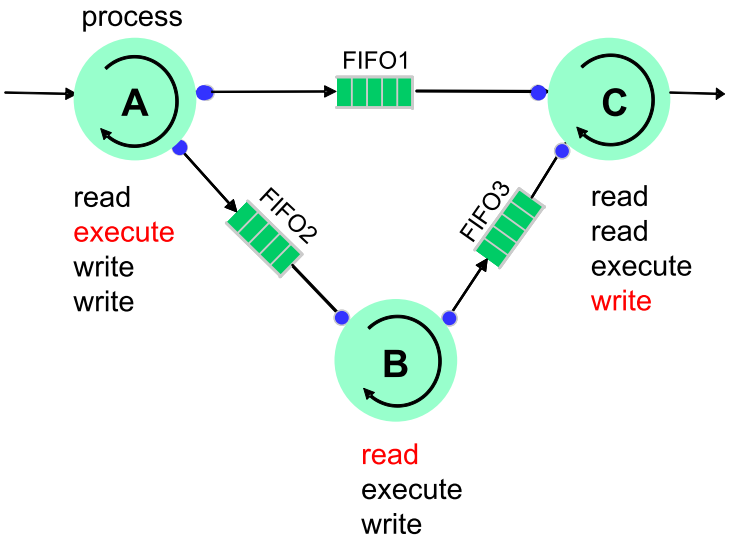

Figure 2.1: A simple KPN model.

Each process in a Kahn Process Network is specified as a sequential program that executes concurrently with other processes. There are three primitives that are executed by each process upon execution of the KPN model:

- *Read* primitive for communication events. This primitive is used to read data from a FIFO channel via a process input port.
- *Write* primitive for communication events. This primitive is used to write data to a FIFO channel via a process output port.

- *Execute* primitive for computation events. This primitive performs the actual data processing to implement the program.

These primitives can be demonstrated by the KPN example in Figure 2.1. Process A reads data from its input port, executes the program and then writes the result data to FIFO1 and FIFO2 connected to process C and process B, respectively. Process B has to read data from FIFO2 before it executes its program, and then writes the result to FIFO3 connected to process C. For the process C, it first reads data from FIFO1 and FIFO3, then executes the program, and finally writes the result to its output port.

A KPN has the following characteristics:

- The control is completely distributed to the individual processes, there is no global scheduler present. As a consequence, partitioning a KPN over a number of processors is a simple task.
- The exchange of data is distributed over the FIFO channels. There is no notion of a global memory that has to be accessed by multiple processes. Therefore, resource contention does not occur.
- Processes run autonomously and synchronize via a blocking read mechanism. When a KPN specification is mapped onto a multiprocessor platform, the blocking write may occur since the FIFO buffers in the platform are bounded. The blocking *Read/Write* primitives are easy to be implemented in hardware and software. They are used as a synchronization mechanism in the multiprocessor platform.

The KPN model fits nicely with signal processing applications as it conveniently models stream processing and as it guarantees that no data is lost. Further, the execution of a KPN is deterministic, meaning that for a given input always the same output is produced and the same workload is generated, irrespective of the execution schedule. The key characteristic of the KPN model is that it specifies an application in terms of distributed control and distributed memory, which allows us to map the application onto the multiprocessor platform in a systematic and efficient way.

2.1.2 The COMPAAN compiler

Although the KPN model of computation is very suitable for multiprocessor platform mapping, specifying an application manually using such a model is a difficult, error prone and time consuming process. To facilitate the migration from a sequential specification of an application to an equivalent KPN specification, we use the COMPAAN compiler.

The COMPAAN compiler [5] [9] [10] is a methodology and tool developed at the Leiden Embedded Research Center, The Netherlands. It automates the transformation of DSP applications written in Matlab into Kahn Process Networks. The COMPAAN compiler framwork consists of three tools. The first tool transforms the initial Matlab code into single assignment code (SAC), which resembles the dependence graph (DG) of the initial nested loop program. The second tool converts the SAC into a Polyhedral Reduced Dependence Graph (PRDG) data structure,

which is a compact mathematical representation of the DG in terms of polyhedra. The third tool converts the PRDG into a process network by generating a process for each node and a FIFO channel for each edge in the PRDG. The parallel processes communicate with each other according to the data-dependency given in the DG.

The COMPAAN compiler fully automates the transformation of an application written in Matlab into KPN specifications. The applications that this tool can handle have to be specified as parameterized static affine nested loop programs, which is a subset of the Matlab language. This conversion is fast and correct by construction.

2.1.3 Mapping Example

Here, we illustrate the approach described above by an example to show how a sequential application specification is automatically transformed into an equivalent KPN specification by the COMPAAN compiler. The application we take is a two dimensional (2D) matrix multiplication. We choose this application because it is an embedded system benchmark application. It is a fundamental, yet real-life algorithm that can often be found in many applications in the realm of digital signal processing. It is not complicated, but has enough features to illustrate the use and usefulness of our approach for application specification.

We start with a standard sequential Matlab code of a matrix multiplication. This code is modified and structured by hand to meet the subset of Matlab that our design flow accepts and to match the features of the matrix multiplication. The reason we use Matlab is only because the COMPAAN compiler uses a simple Matlab parser. The code is shown in Figure 2.2.

We structure the matrix multiplication algorithm as a set of routines (functions) that are called by the Matlab code. It can be debugged easily and the functional correctness of the application can be easily verified. The code lines 1-3 specify that the number of elements of the two dimensions of a matrix can be any integer value between 2 and 100. The code lines 5-9 initialize the first input matrix *X*. The code lines 11-15 initialize the second input matrix *Y*. The matrix Z is used to store the final result of the matrix multiplication, with its initial value set to zero - see code lines 17-21. The *MultProp* function and the *Sum* function perform the multiplication of matrix X and Y - see code lines 23-30. Finally, in code lines 32-36, the *Write_z* function outputs the result of the multiplication stored in matrix Z.

The Matlab program in Figure 2.2 is a convenient way to describe the matrix multiplication application. Nonetheless, this program does not reveal the inherent task-level parallelism available in the application due to the sequential nature of the program. Therefore, the first step in our system design flow is to convert this sequential program into an executable parallel specification, in our case Kahn Process Network (KPN). We rely on the COMPAAN compiler to convert fully automatically the matrix multiplication Matlab program into the KPN specification shown in Figure 2.3.

To obtain a KPN specification from a sequential application specification, the general partitioning strategy employed in the COMPAAN compiler is to create a process for every function call in the program. Therefore, the Kahn Process Network shown in Figure 2.3 consists of six processes. The *Read_x*, *Read_y*, *Zero_z*, *MultProp*, *Sum* and *Write_z* processes form the central data-flow processing of the matrix multiplication algorithm. These six concurrent processes

```
1    %parameter N 2 100;
2    %parameter M 2 100;
3    %parameter T 2 100;
4
5    for i = 1:1:N,
6      for j = 1:1:M,
7        [x(i,j)] = Read_x();
8      end
9    end
10
11   for j = 1:1:T,
12     for i = 1:1:M,
13       [y(i,j)] = Read_y();
14     end
15   end
16
17   for i = 1:1:N,
18     for j = 1:1:T,
19       [z(i,j)] = Zero_z();
20     end
21   end
22
23   for i = 1:1:N,
24     for j = 1:1:T,
25       for k = 1:1:M,
26         [t, x(i,k) , y(k,j)] = MultProp( x(i,k), y(k,j) );
27         [z(i,j)] = Sum ( z(i,j), t );
28       end
29     end
30   end
31
32   for i = 1:1:N,
33     for j = 1:1:T,
34       [Sink(i,j)] = Write_z( z(i,j) );
35     end
36   end
```

Figure 2.2: Task-level specification of the matrix multiplication in Matlab.

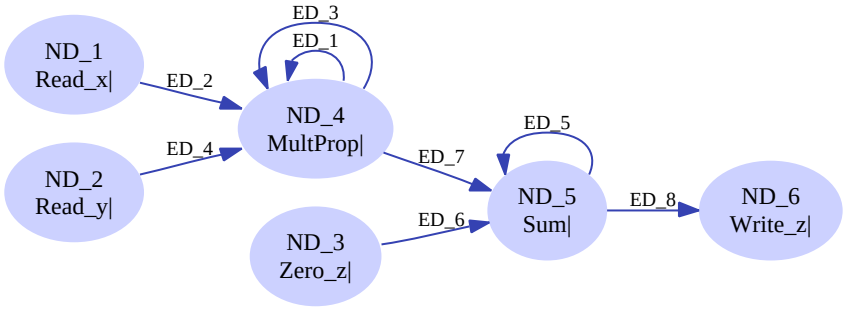

Figure 2.3: The KPN specification of the matrix multiplication.

make the task-level parallelism available in the application explicit. Also, the data dependencies and the communication between the processes is explicit via distributed FIFO channels.

In the KPN specification of the matrix multiplication shown in Figure 2.3, process *Read_x* and *Read_y* get every element from matrix X and Y, repectively and send them to process *MultProp*. *MultProp* calculates the product of every two elements from matrix X and Y, and then sends the product to process *Sum* to calculate the sum of products. Process *Zero_z* generates matrix Z to store the final result of the matrix multiplication, with all its initial elements set to value zero. *Zero_z* also sends its initial elements of value zero to process *Sum* to be the initial value

for the sum of products. Finally, process *Write_z* outputs the result of the multiplication stored in matrix Z.

2.2 Synthesis of Multiprocessor Platform for KPN

After the KPN specification of the application has been derived, the next step is to map it onto multiprocessor platforms. To map a KPN specification onto a platform, we have implemented in the ESPAM tool a strategy that performs the mapping in two steps:

- In the first step, the ESPAM tool converts the given KPN specification into an equivalent network of processors. This is an abstract model of a multiprocessor platform onto which we map the KPN specification. This step is implementation independent as no information on the target physical platform is taken into account. The model defines the key components of the platform and their attributes. Such an abstract model is constructed using parameterized building blocks from a component library.
- In the second step, information is added to the abstract model that is specific for the target physical platform. At this stage, we need to select the proper processors that can implement the functions of the original application. Also, we set parameters of the components like bit-width and size of the hardware FIFO buffers. This step leads to an elaborate platform specification ready for an implementation on the target platform.

2.2.1 Platform Modeling

To facilitate the construction of platform models, the ESPAM tool provides a component library that consists of parameterized building blocks from which platform models can be composed. These components are designed in a parameterized way so that they can be used to model a broad class of programmable or dedicated components that are specific for a target physical platform.

Currently the component library defined in ESPAM consists of six components. They are *Processor*, *Memory*, *Memory Controller*, *FIFO*, *FIFO Controller*, and *Bus*. Table 2.1 lists these components and their parameter descriptions. These components are abstract modules that represent a large number of concrete component specifications. We explain the components and their parameters in detail as below:

Processor *P_FREQ* denotes the frequency of the processor. It is one of the most important parameters of the processor and describes the processing speed of the processor. *P_PROG* denotes whether the processor is a programmable unit or not. For example, a processor could be a non programmable unit like ASIC or a programmable unit such as a RISC or DSP processor. *P_HARVARD* denotes whether the processor has a Harvard memory architecture. If yes, then the processor has two separated buses connected to its local memory for data access and instruction access, respectively. Otherwise the processor has only one bus connected to its local memory. *P_LM_SIZE* describes the local memory size

Table 2.1: Component modules and parameters description.

Component	Parameter Description	Parameter Name	Allowable Values
Processor	Frequency	P_FREQ	1MHz - 1GHz
	Programmable	P_PROG	Yes/No
	Harvard Architecture	P_HARVARD	Yes/No
	Local Memory Size	P_LM_SIZE	1KB - 1GB
	Number of I/O Ports	P_IO_PORTS	1 - 32
	Instruction Bus Width	P_ILMB_WIDTH	16/32/64/128
	Data Bus Width	P_DLMB_WIDTH	16/32/64/128
	Address Bus Width	P_ADDR_WIDTH	16/32/64/128
Memory	Memory Size	M_SIZE	1KB - 1GB
	Data Bus Width	M_DWIDTH	16/32/64/128
	Address Bus Width	M_AWIDTH	16/32/64/128
	Number of I/O Ports	M_IO_PORTS	1/2
Memory Controller	Data Bus Width	MC_DWIDTH	16/32/64/128
	Address Bus Width	MC_AWIDTH	16/32/64/128
	Number of I/O Ports	MC_IO_PORTS	1 - 32
FIFO	FIFO Size	FIFO_SIZE	1 - 16384
	FIFO Data Width	FIFO_DWIDTH	16/32/64/128
FIFO Controller	Data Bus Width	FC_DWIDTH	16/32/64/128
	Address Bus Width	FC_AWIDTH	16/32/64/128
	Number of FIFOs to Read	FC_FIFO_READ	0 to 128
	Number of FIFOs to Write	FC_FIFO_WRITE	0 to 128
Bus	Clock Rate	BS_CLKRATE	16 - 500MHz
	Data Rate	BS_DRATE	33 - 533MB/s
	Number of Masters on the bus	BS_MASTER	1 to 16
	Number of Slaves on the bus	BS_SLAVE	1 to 16
	Data Width	BS_DWIDTH	16/32/64/128
	Address Width	BS_AWIDTH	16/32/64/128

of the processor. *P_IO_PORTS* describes the number of I/O ports of the processor. These ports can be used to connect the processor to other components of the multiprocessor platform. *P_ILMB_WIDTH*, *P_DLMB_WIDTH*, and *P_ADDR_WIDTH* denote the bitwidth of the instruction bus, data bus, and address bus of the processor, respectively. The values of the bitwidth can be 16, 32, 64, or 128 bits.

Memory A memory component could be a local memory along with a processor, or a shared global memory in a network for data exchange. *M_SIZE* denotes the size of the memory component. *M_DWIDTH* and *M_AWIDTH* denote the bitwidth of the data bus and address bus of the memory, respectively. The allowable values of the bitwidth can be 16, 32, 64 or 128 bits. *M_IO_PORTS* describes the number of I/O ports of the memory, i.e., single or dual port memory.

Memory Controller The memory controller is used as an interface between a processor and a memory component, translating the processor data bus protocol into a memory component specific protocol. *MC_DWIDTH* and *MC_AWIDTH* denote the bitwidth of the data bus and address bus of the memory controller, respectively. The allowable values of the bitwidth can be 16, 32, 64 or 128 bits. *MC_IO_PORTS* describes the number of I/O ports of the memory controller. It determines the number of memory components that can be connected to this controller.

FIFO The FIFO component is a memory buffer for point-to-point connection between two processors. The processors communicate with each other and exchange data via the FIFO components. *FIFO_SIZE* denotes the size of the FIFO components. It determines the quantity of data that can be stored in the FIFO. *FIFO_DWIDTH* denotes the bitwidth of the data that can be stored in the FIFO.

FIFO Controller The FIFO controller is used as an interface between a processor and a FIFO component. *FC_DWIDTH* and *FC_AWIDTH* denote the bitwidth of the data bus and address bus of the FIFO controller, respectively. The allowable values of the bitwidth can be 16, 32, 64 or 128 bits. *FC_FIFO_READ* denotes the number of input FIFO components from which the processor can read data. *FC_FIFO_WRITE* denotes the number of output FIFO components to which the processor can write data. Currently, the FIFO controller in our component library can support up to 128 input FIFO components and 128 output FIFO components.

Bus The bus component is an arbiter controlling a fast local bus for connecting a processor and its local memory or a slow peripheral bus connecting a processor with slow peripheral components. *BS_CLKRATE* denotes the clock rate of the bus. *BS_DRATE* denotes the data rate of the bus. *BS_DWIDTH* and *BS_AWIDTH* denote the bitwidth of the data and address that the bus supports. *BS_MASTER* denotes the number of masters that can be on the same bus. *BS_SLAVE* denotes the number of slaves that can be on the same bus.

2.2.2 Synthesis Algorithm

As a good component library has been defined in the ESPAM tool, synthesizing multiprocessor platforms then becomes as easy as instantiating building blocks from the library and intercon-

necting them. The synthesis procedure can be stated as follows:

Given an application in the form of a Kahn Process Network and a platform component library, synthesis of a multiprocessor platform consists of allocation of processors and communication components, binding of processes to processors, and FIFO channels to communication components.

In our methodology, we define the synthesis algorithm as follows:

- A single processor has to be instantiated for each Kahn node (process).
- A FIFO buffer has to be instantiated for each channel in the KPN model.
- A FIFO controller has to be instantiated as an interface between a processor and all FIFO buffers connected to the processor.
- A memory module has to be instantiated as a local memory along with each processor.
- A memory controller has to be instantiated as an interface between each processor and its local memory.
- A bus has to be instantiated for a connection between any two components of processor, FIFO, FIFO controller, memory and memory controller.

Thus, the synthesis procedure involves a selection and reuse of all six component modules from the component library to construct the multiprocessor platform. To make clear the synthesis of a multiprocessor platform done in ESPAM, we use the generated KPN specification of the matrix multiplication given in Figure 2.3 as an example. After the synthesis procedure described above, we get the multiprocessor platform shown in Figure 2.4.

The KPN shown in the upper part of Figure 2.4 is mapped by the platform synthesis methodology in the ESPAM tool onto an abstract model of multiprocessor platform. This model is composed of the component modules defined in our component library. The lower part of Figure 2.4 represents the network of processors that has the same topology as the input KPN. This is because our synthesis methodology performs a one-to-one mapping. The six processes *Read_x*, *Read_y*, *Zero_z*, *MultProp*, *Sum*, and *Write_z* are mapped onto the processors P1, P2, P3, P4, P5, and P6, respectively. The KPN unbounded FIFO channels *ED_1* to *ED_8* are mapped onto the hardware FIFO buffers FIFO1 to FIFO8, respectively. A FIFO controller (FC) is instantiated as an interface between each processor and all FIFO buffers connected to it. Besides, a memory component is instantiated as a local memory (LM) for each processor, with a controller (MC) between them as an interface. A bus is instantiated for each connection between two components in the multiprocessor platform.

The synthesis of a multiprocessor platform for a KPN specification is performed in a systematic and automated way by the ESPAM tool. A multiprocessor platform model is specified by means of a textual description using an architecture description language. The textual description is automatically generated by our platform synthesis algorithm. We use a software engineering technique called Visitor [28] to traverse the KPN model topology derived from the original application by the COMPAAN compiler and to generate the platform description code.

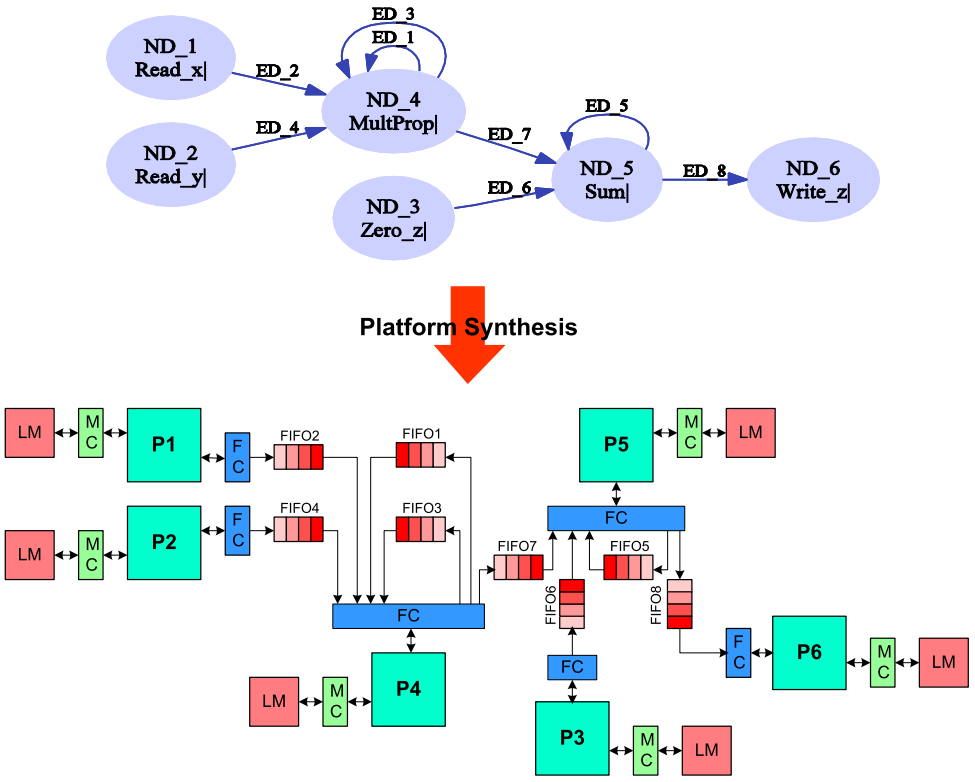

Figure 2.4: Synthesis of multiprocessor platform for KPN.

2.2.3 Target Platform Implementation

After the first step for synthesis of a multiprocessor platform for a KPN, which has been described in the last subsection, we obtain an abstract model of a platform onto which we map the KPN application. This model of a platform defines the key components of the platform and their attributes.

In the second step, we start to add information to the abstract model that is specific for the target physical platform. At this stage, we need to select the proper processors that can implement the functions of the original application. Also, we set parameters of the components like bit-width and size of the hardware FIFO buffers. This step leads to an elaborate platform specification ready for an implementation on the target physical platform.

In our case, as we implement the multiprocessor platform on a single Xilinx Virtex-II Pro FPGA for prototyping, the information to be added to this abstract model must be specific for this FPGA chip. For example, we select a configurable MicroBlaze embedded soft processor core for each processor in the platform. The MicroBlaze embedded soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx FPGAs. Since it is a soft core, we are able to instantiate and synthesize multiple MicroBlaze processors on the FPGA, thereby implementing the multiprocessor platform an a single FPGA. For the hardware FIFO buffers, we implement them by instantiating predefined generic FIFO IP cores found in the Xilinx library. The local memory of a processor is implemented using the Block RAMs on the FPGA. As the MicroBlaze processor core has a Harvard memory architecture, the memory controllers in the abstract model are now split into two controllers for instruction and data access, respectively.

Also, the generic bus between the memory and the memory controller needs to be replaced with the LMB bus that is specific for Xilinx FPGA. For the FIFO controller, as there are no corresponding components on the target FPGA, we need to design a custom FIFO controller using VHDL and implement it on the target FPGA. More details on our implementation of multiprocessor platforms on FPGAs is discussed in Chapter 3.

2.3 Programming Multiprocessor Platforms

In section 2.2, we discussed the synthesis of a multiprocessor platform for an application specified as a Kahn Process Network. Now having the platform, the next step is to generate the program code for each processor in the multiprocessor platform.

2.3.1 What is Programming

To execute an application on the synthesized multiprocessor platform, the platform has to be programmed. Programming the multiprocessor platform means writing software for each processor in the platform using high level programming languages like C/C++.

In our case, we model an application as a Kahn Process Network (KPN) and map each process of the KPN onto a processor of the multiprocessor platform. Therefore, each processor must be programmed according to the behavior of the corresponding process in the KPN.

2.3.2 Program Modeling

Each process in the Kahn Process Network is specified as a sequential program that executes concurrently with other processes. In the KPN specification, such sequential program is modeled as a syntax tree [29]. The benefit of a syntax tree representation is that a sequential program is modeled at an abstract level that is independent on a specific programming language. Thus, it is easy to convert a syntax tree representation into a program specified in any high level programming language. A syntax tree gives a valid execution order between function calls which have to be executed inside a process. It completely defines the internal behavior of the process. An example of the syntax tree of process *Sum* depicted in Figure 2.4 is shown in Figure 2.5.

The resulting syntax tree in Figure 2.5 consists of three types of node. They are ”For”-statement nodes, ”If”- statement nodes, and ”Code”- statement nodes. Every program statement in the sequential program has a corresponding node in the syntax tree. ”For”-statement nodes contain the information about the loops present in the sequential program. ”If”-statement nodes contain the information about the condition present in the sequential program. Similarly, ”Code”-statement nodes correspond to the code segments.

If we parse this tree top-down from left to right, we can obtain a valid sequential execution order among function calls *read()*, *Sum()* and *write()*. This sequential order describes the behavior of process *Sum*.

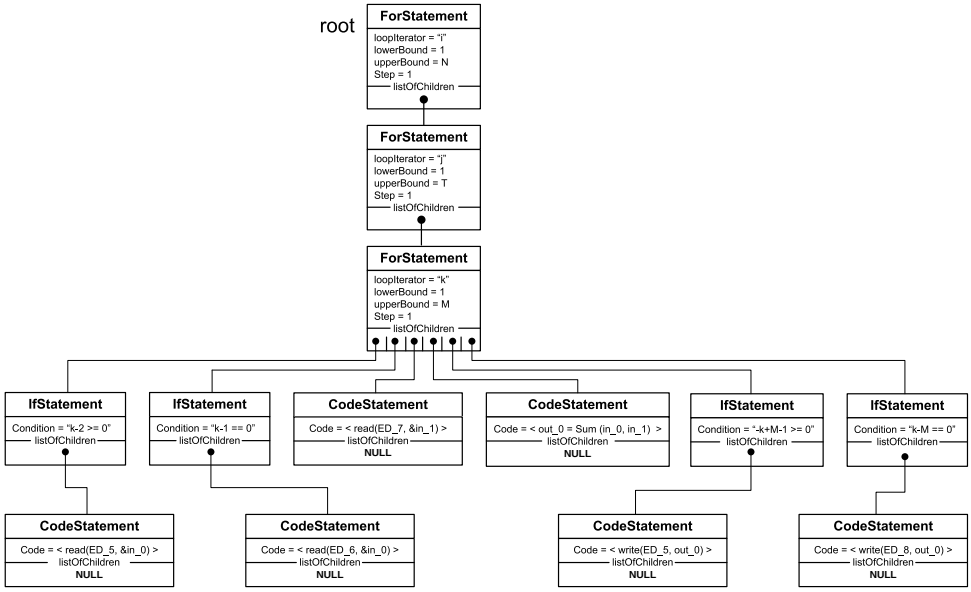

Figure 2.5: The syntax tree of process *Sum*.

2.3.3 Program Code Generation

In order to execute an application on the multiprocessor platform, our methodology implemented in the ESPAM tool is able to generate the program code for each processor. We use the software engineering technique called *Visitor* to traverse the syntax tree and to generate the program code. This program code can be expressed in any programming language for which a compiler support exists for the processors used in the platform.

An example of C programming code generated from the syntax tree of process *Sum* is given in Figure 2.6.

```
0   int main () {
      for ( i =1 ; i <=N ; i += 1 ) {
        for ( j =1 ; j <=T ; j += 1 ) {
          for ( k =1 ; k <=M ; k += 1 ) {
            if ( k-2 >= 0) {
5             read(ED_5, &in_0);
            }
            if ( k-1 == 0) {
              read(ED_6, &in_0);
            }
10          read(ED_7, &in_1);
            out_0 = Sum (in_0, in_1) ;
            if ( -k+M-1 >= 0) {
              write (ED_5, out_0);
            }
15          if ( k-M == 0) {
              write (ED_8, out_0);
            }
          } // for k
        } // for j
20    } // for i
    }
```

Figure 2.6: C program code generated from the syntax tree of process *Sum*.

If we look at the program code together with Figure 2.4, we can see that in each *for* loop the processor first reads data from FIFO *ED_5* or *ED_6* - code lines 6-11. If it is the first iteration of *k* (*k-1 == 0*), then it reads data from FIFO *ED_6* as this is the initial value of zero for the sum of products. Otherwise it reads data from FIFO *ED_5* which is the sum of products from the previous iteration. The second data is always read from FIFO *ED_7* - code line 12. This is the value that has to be added to the sum of products. In line 14, it calculates the sum of products. Then, it writes result to one of its output FIFOs. If it is the last iteration of loop *k* (*k-M == 0*), it writes the final result to FIFO *ED_8* as an element of the resulting matrix *Z*, otherwise the result is written to FIFO *ED_5* - code lines 16-21. Processor *Sum* executes the body of the loops repeatedly till the end of the *for* loops.

Chapter 3

Multiprocessor Platforms FPGA Prototyping

In Chapter 2 we have presented in detail the techniques and tools that we have developed and used to show how for an application written in Matlab, a Kahn Process Network specification can automatically be derived and systematically mapped onto a multiprocessor platform.

This chapter focuses on the second and third step shown in Figure 1.2 in which we implement the abstract model of a multiprocessor platform onto a Field Programmable Gate Array (FPGA) chip for prototyping. This generates an elaborate platform specification ready for an implementation on the target platform. To prototype a multiprocessor platform as described in Figure 2.4 onto an FPGA, we present in this chapter our methodology of how to implement each component of the multiprocessor platform on a target FPGA board. Specifically, as we use a single Xilinx Virtex-II Pro FPGA for prototyping, we use configurable MicroBlaze embedded soft processor core [30] [31] for each processor of the platform. For the hardware FIFO buffers, we implement them by instantiating predefined generic FIFO IP cores found in the Xilinx EDK library [32]. Besides, the connection between the MicroBlaze soft processor cores and the hardware FIFO buffers, the control mechanism of the FIFO will also be elaborated in details.

This chapter is organized as follows. In Section 3.1, we first introduce the target FPGA board on which we implement the multiprocessor platform. The multiprocessor platform FPGA prototyping is presented in Section 3.2. In this section we present in detail how the six components in the abstract model of the multiprocessor platform are implemented on the FPGA. In Section 3.3, we describe the program code generation for each processor and the software communication interface that also need to be automatically generated. Section 3.4 presents what is needed to generate a Xilinx Platform Studio project and the tree architecture of the project suite. This section also describes how to use the software engineering technique called $visitor$ to systematically and automatically generate the whole project suite to implement the multiprocessor platform on our target FPGA. Finally, Section 3.5 presents some discussion and conclusions.

3.1 Target FPGA Platform

We implement the multiprocessor platform on an FPGA prototyping board, namely the ADM-XPL board manufactured by Alpha Data Parallel Systems, Ltd [33]. The ADM-XPL board is a high performance PCI Card, designed for supporting development of applications using the Xilinx Virtex-II Pro series of FPGAs. The board is mainly based on a Virtex-II Pro 2VP20 FPGA. This FPGA chip contains two PowerPC hard processor cores, 88 distributed on-chip dual-port RAM blocks and 9,280 slices. The board has a PCI interface PLX 9656 which connects to the host processor (pentium) with PCI bus. It also has some off-chip memories on the board, 1 bank ZBT 512k/1024K x 64 bits and 1 bank DDR 64MB /128MB. These plenty of resources make it possible to implement a multiprocessor platform on this FPGA board.

Traditionally, most of the design tools for translating software programs into hardware implementations assume that only the most computational intensive portions of the program are mapped onto hardware. Those tools use the FPGA as a coprocessor to a standard CPU. The CPU implemented most of the program, handling much of the operations that are necessary to implement the program. The most computational intensive portions of the program code are mapped onto the FPGA. In that way the strengths of both FPGA and standard processors are combined into a single system. In our case, the standard CPU of the host is not involved. We implement the entire multiprocessor platform onto a single FPGA. Each processor runs concurrently, communicating via the hardware FIFO buffers. This leads to a complete self-contained system on programmable chip implementing the entire functionality of the application.

3.2 Multiprocessor Platform Implementation on FPGA

This section presents how we prototype the multiprocessor platform on our target FPGA board described above. Specifically, we present in detail how the six components in the abstract model of a multiprocessor platform are implemented on the FPGA.

3.2.1 MicroBlaze Soft Processor, Local Memory and Memory Controller

In most cases, the generated multiprocessor platform in our system design flow consists of more than two processors. Thus the two PowerPC hard processors [30] on the FPGA are not enough for the multiprocessor system prototyping if we use only one FPGA. Instead, we use a configurable MicroBlaze embedded soft processor core [30] [31] for the processors. As it is a soft processor core, the number of processors we can implement on a given FPGA is only limited by the size of the FPGA itself.

The MicroBlaze soft processor core provided by Xilinx is a 32-bit configurable processor core. A designer can create a system incorporating a MicroBlaze using the Xilinx Platform Studio in which a designer can quickly build a MicroBlaze processor system by instantiating and configuring cores from the provided libraries. Figure 3.1 presents a simple MicroBlaze system incorporating a MicroBlaze processor along with a local memory, memory controller, local memory bus and some other components to create a complete system.

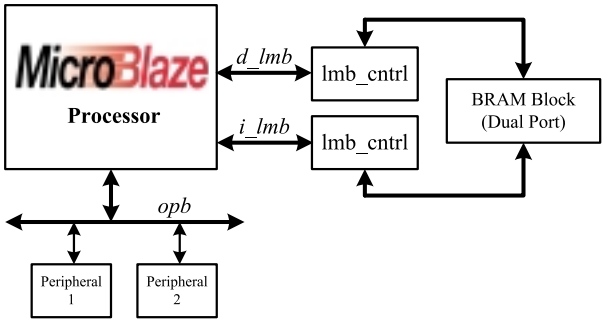

Figure 3.1: Simple MicroBlaze processor system.

The MicroBlaze processor core has a Harvard memory architecture and thus utilizes two Local Memory Busses (LMB), namely Data-side LMB (*d_lmb*) and Instruction-side LMB (*i_lmb*) for instruction and data memory, respectively. According to the attributes of the MicroBlaze processor core, the ESPAM tool now can specify the parameters of the processor component in the abstract platform shown in Figure 2.4. For example, *Frequency* is set to 100 MHz by default. *Programmable* is set to *YES*. *Harvard Architecture* is set to *YES*. The bitwidth of the Instruction Bus, Data Bus and Address Bus is set to 32 bits.

The local memory is implemented using the dual-port BRAM Blocks on our target FPGA. We can use from 4 to 64 dual-port BRAMs on the FPGA to provide memory sizes from 2KB to 128KB for the MicroBlaze processor. The parameters of the local memory can be set to be uniquely tailored for a system. In our case, the ESPAM tool sets the parameter *Memory Size* to 8KB by default, but this value can be changed according to different application requirements. The bitwidth of the Data Bus and Address Bus is set to 32 bits. The number of I/O ports is set to 2 (dual-port).

The two ports of the local memory must be connected to independent memory interface controllers. The system shown in Figure 3.1 includes two memory controllers and they are connected to the two ports of the local memory respectively, one for instruction access and the other one for data access. The parameters of these two memory controllers should be the same and the ESPAM tool specifies their values as follows: The bitwidth of the Data Bus and Address Bus is set to 32 bits. The number of I/O ports is set to 2. One port is connected to the local memory and the other one is connected to the MicroBlaze processor.

The system in Figure 3.1 also includes two peripherals connected via the On-Chip Peripheral Bus (OPB). After specifying the system architecture and configuring the MicroBlaze processor, the Xilinx Platform Studio tools synthesize the design and generate a bitstream file for the system as well as generate a set of software libraries that a design can use to interface with the various components in the system. Finally, a designer can compile an application and combine the application binary with the bitstream to produce the final system bitstream file.

3.2.2 Hardware FIFO Buffer

The communication structure in the multiprocessor platform is realized by hardware FIFO buffers. We implement them by instantiating predefined generic FIFO IP cores found in the

Xilinx EDK library [32].

FIFO IP cores utilize dual-port BRAM blocks as a data storage medium. The FIFO is designed using VHDL and the design incorporates special purpose counters and logic necessary to implement functional requirements of a channelized FIFO as shown in Figure 3.2. Counters are used to generate the read/write addresses. Additional control is required to check for empty and full states and to interface with the communication channels.

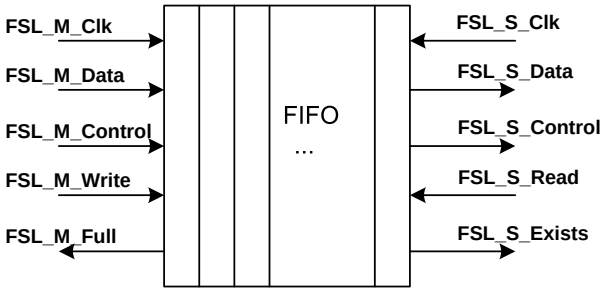

Figure 3.2: FIFO IP core design.

Figure 3.2 shows the FIFO interface signals. The signals on the left side are for a master processor to write data to the FIFO. *FSL_M_Clk* is the master clock signal to asynchronously control the master writes to the FIFO. *FSL_M_Data* is the data bus to write tokens to the FIFO. *FSL_M_Control* is a single bit control signal that is transmitted together with the data at each clock edge. *FSL_M_Write* is the input signal that controls the write enable signal of the FIFO. *FSL_M_Full* is the output signal from the FIFO indicating when the FIFO is full. The signals on the right side are for a slave processor to read data from the FIFO. *FSL_S_Clk* is the slave read clock to asynchronously read the FIFO. *FSL_S_Data* is the data bus to read tokens from the FIFO. *FSL_S_Control* is the output signal that indicates the control bit associated with the data at the read end of the FIFO. *FSL_S_Read* is the input signal that controls the read enable signal of the FIFO. *FSL_S_Exists* is the output signal indicating when the FIFO contains valid data.

For the parameters of the FIFO component in the abstract model of a multiprocessor platform, ESPAM sets the values that are specific for our target FPGA platform. The bitwidth of the Data to be stored in the FIFO is set to 32 bits. The allowable value for FIFO size ranges from 1 to 16384, ESPAM sets it to 512 by default for our target FPGA board, but it can also be changed according to different application requirements.

3.2.3 Bus connection

We have selected the MicroBlaze soft processor core for the processors in the abstract model of multiprocessor platform. Next, with alternative bus interfaces available in the MicroBlaze processor system, we need to choose a proper bus interface to interconnect the components in the multiprocessor platform.

Figure 3.3 shows that the MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data accesses and instruction accesses. Each bus interface unit is further split into a Local Memory Bus (LMB) [34] and IBMs On-chip Peripheral Bus (OPB) [35]. The LMB is a fast, local bus for connecting MicroBlaze instruction and data ports to high-speed

peripherals, primarily on-chip BRAMs. The OPB interface provides a slow connection to both on-and off-chip peripherals and memory. Besides these two bus interfaces, the MicroBlaze processor core provides 8 input and 8 output interfaces to Fast Simplex Link (FSL) bus [36]. The FSL buses are uni-directional non-arbitrated dedicated communication channels. They can be used as the fastest interconnection for Xilinx FPGA based embedded processor systems.

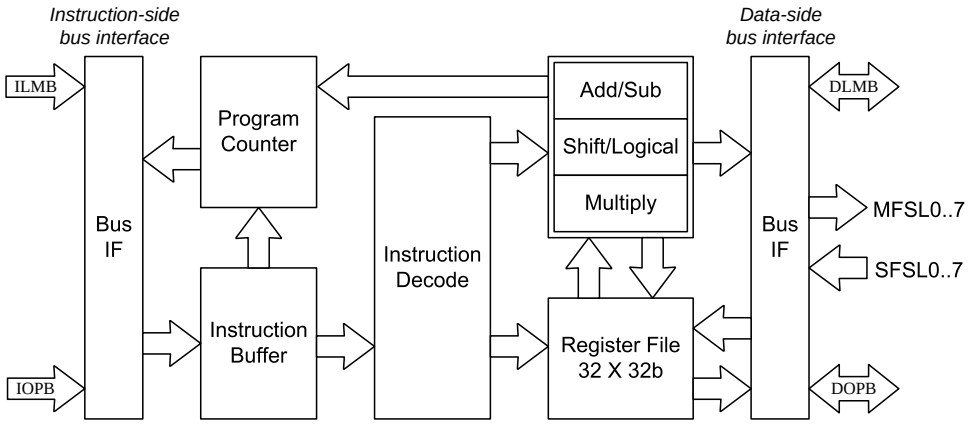

Figure 3.3: MicroBlaze core block diagram.

From the specification of these three bus interfaces, we find out that the FSL bus is ideal for our processor network. The FSL is implemented on the FPGA as a FIFO using the SRL16 primitive. The FSL bus provides a point-to-point communication channel between two components on the FPGA. Up to 8 master and slave FSL interfaces are available on the MicroBlaze soft core. The interfaces are used to transfer data in 2 clock cycles to and from a register file on the processor to hardware running on the FPGA. The FIFO depths can be as low as 1 and as high as 8K. It also supports both synchronous and asynchronous FIFO modes. This allows the master and slave side of the FSL to clock at different rates.

All these three buses discussed above are predefined generic IP cores that can be found in the Xilinx EDK library [32]. Some of the parameters defined in the abstract platform (See Table 2.1) of these three buses are listed in Table 3.1. The values of these parameters are specific to our target FPGA board. Thus the ESPAM tool can set the parameters for the bus component of the abstract platform according to this table 3.1. As the FSL bus is used for a point-to-point connection between two MicroBlaze processors, ESPAM sets *Clock Rate* to 150 MHz, *Data Rate* to 300 MB/s. The bitwidth of the Data Bus and Address Bus is set to 32 bits. Both the number of masters and slaves are set to 1.

Table 3.1: Bus parameters of LMB, OPB and FSL.

Parameter	LMB	OPB	FSL
Clock Rate	125 MHz	125 MHz	150 MHz
Data Rate	333 MB/s	167 MB/s	300MB/s
Number of Masters	1	2-8	1
Number of Slaves	4	2-8	1
Data Width	32	32	32
Address Width	32	32	32

Although the FSL bus has these advantages described before, the MicroBlaze processor can only support up to 8 input ports and 8 output ports for FSL connection. If a MicroBlaze pro-

cessor has more than 8 FIFO buffers connected to its input or output ports, we still have to use some other bus interfaces. From the table 3.1. we can see that LMB is faster than OPB. It has a data rate of 333 MB/s, twice as OPB. Therefore, we choose the data-side LMB bus [34]. The LMB bus is designed specially for the local on-chip memory. This brings another issue, which is, a custom FIFO controller between the LMB bus and the FSL FIFO is needed.

3.2.4 FIFO Controller

In the EDK, there is no standard controller for the LMB bus to connect to the FSL FIFO. Thus, a custom controller is developed as an interface for this connection. We use one controller to deal with all the input and output FIFOs connected to one processor. For the purpose of reducing the amount of gates, a parameterized controller is useful, so that we can reduce the size of the IP core by setting its parameters.

Figure 3.4 shows the FIFO controller interface and all its signals. The signals on the left side are standard signals of the LMB bus and the ports are connected according to the standard LMB bus. For example, *LMB_Rst* is the LMB reset signal. *LMB_WriteDBus* is the LMB write data bus. *SL_DBus* is the LMB read data bus. For the sake of brevity, we do not explain all the LMB bus signals here, more details of the LMB bus interface is described in [34]. On the right side, the signals can be connected to multiple FSL FIFOs. These signals are partitioned into 2n sets. Each set is an interface for connecting to a FIFO. For example, the five signals *FSL_M_Clk_1*, *FSL_M_Data_1*, *FSL_M_Control_1*, *FSL_M_Write_1*, and *FSL_M_Full_1* form the interface to the first writes FIFO. So we can see that all these signals can be connected to n writes FIFOs and n reads FIFOs. The value of the parameter n can be up to 128 in our design. Thus, for the FIFO controller in the abstract platform, the number of FIFOs to read/write can be set from 0 to 128. The bitwidth of the Data Bus and Address Bus is set to 32 bits.

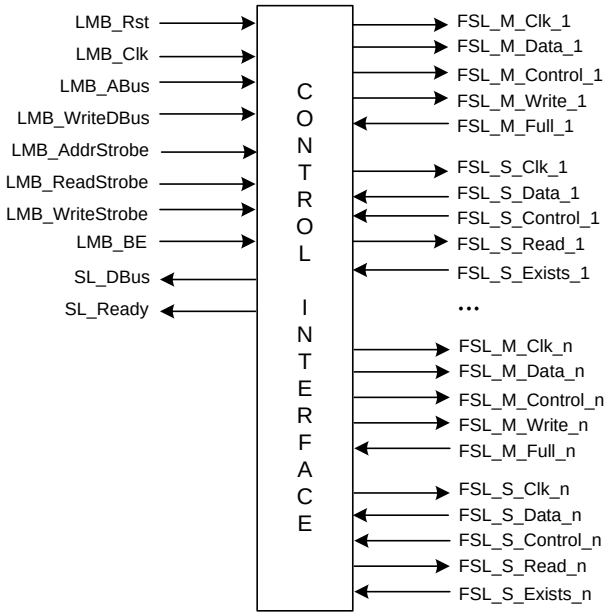

Figure 3.4: LMB FIFO controller interface.

We use the generic syntax to define the parameters for the controller. Figure 3.5 shows a fragment of the VHDL code to define the FIFO controller. There are two parameters, C_FIFO_WRITE and C_FIFO_READ, to define the maximum number of FIFOs to write or read, respectively. The values of these two parameters (see Figure 3.5 lines 7-8) can be changed easily for different FIFO controllers in the multiprocessor platform. Our automation tool implemented in ESPAM generates the VHDL code for the controllers dynamically, it traverses the topology of the multiprocessor platform and select the maximun number of writes/reads FIFOs connected to all the processor as the parameter value. Thus ESPAM first generates one core for all the processors. Further, it instantiates an instance of this controller core for each processor and set the parameters of the reads/writes FIFOs specific to that processor.

```
1    GENERIC (
2        C_HIGHADDR: STD_LOGIC_VECTOR(0 to 31):= X''ff00003f'';
3        C_BASEADDR: STD_LOGIC_VECTOR(0 to 31):= X''ff000000'';
4        C_ABi: INTEGER:= 26;
5        C_LMB_AWIDTH: INTEGER:= 32;
6        C_LMB_DWIDTH: INTEGER:= 32;
7        C_FIFO_WRITE: INTEGER:= 2;
8        C_FIFO_READ: INTEGER:= 2
9    );
```

Figure 3.5: Parameters of FIFO controller.

The FIFO controller is an interface for two different buses. It communicates with the MicroBlaze processor core using the LMB bus interface and communicates with the FIFO using the FSL bus interface. When a MicroBlaze processor writes/reads data to/from a FIFO buffer, the FIFO controller translates the LMB bus protocol into the FSL bus specific protocol. Thus, the MicroBlaze processors in the platform can communicate with each other via the FIFO buffers using the LMB bus for connection. Figure 3.6 shows the VHDL code of the FIFO controller that implements the mechanism of protocol translation. Code lines 1-7 translate the signals protocol for FIFO writing. We can see that when the signals on the LMB bus indicate a FIFO writing, the controller sets the write enable signal FSL_M_Write of the selected FIFO to "1" - code line 2-5. Then the data in the *LMB_WriteDBus* port are written to the data input port FSL_M_Data of the FIFO - line 6. Similarly, when the signals on the LMB bus indicate a FIFO reading, the controller sets the read enable signal FSL_S_Read of the selected FIFO to "1" - code line 10-12. Then the data on the data output port FSL_S_Data of the selected FIFO are written to the LMB read data bus *SL_DBus* - line 14.

```
1    G_1 : for I in 1 to C_FIFO_WRITE generate
2        FSL_M_Write(I) <= '1' when lmb_select = '1' and
3                                   LMB_ABus(24 to 29) =CONV_STD_LOGIC_VECTOR(2*I, 6) and
4                                   LMB_WriteStrobe = '1'
5                        else '0';
6        FSL_M_Data(I) <= LMB_WriteDBus;
7    end generate;
8
9    G_2 : for I in 1 to C_FIFO_READ  generate
10       FSL_S_Read(I) <= '1' when lmb_select='1' and
11                                   LMB_ABus(24 to 29)= ONV_STD_LOGIC_VECTOR(2*I, 6) and
12                                   LMB_ReadStrobe = '1'
13                           else '0';
14       SL_DBus <= FSL_S_Data(I);
15   end generate;
```

Figure 3.6: Signal instance of FIFO controller.

This simple controller mechanism reveals a flexible connectivity in our design methodology. Only by rewriting this controller, we can easily change the connection. For example, once the FSL FIFO is inapplicable, we only have to modify the bus interface on the right side of Figure 3.4 to adapt any other kind of FIFO. The other way around, we can adjust the bus interface on the left side of Figure 3.4 to connect the FSL FIFO to other kind of bus interface of the processor.

3.3 Programming Multiprocessor Platform and Code Generation

To execute an application on the synthesized multiprocessor platform, the platform has to be programmed. Programming the multiprocessor platform means writing program code for each processor in the platform using high level programming languages like C/C++. The MicroBlaze soft processor core supports GNU tools that support standard Executable and Linkable Format (ELF) [32]. The MicroBlaze GNU tools include mb-gcc compiler, mb-as assembler and mb-ld loader/linker, which can compile GNU compatible C/C++ source files to build an ELF executable file.

With the EDK GNU tools, we can build a separate ELF executable file for each MicroBlaze processor in the multiprocessor platform. To build this ELF executable file, our methodology implemented in the ESPAM tool is able to generate the program code for the MicroBlaze processors. We use the software engineering technique called *Visitor* [29] to generate C program code for each MicroBlaze processor. An example of C programming code generated for the MicroBlaze processor on which the process *Sum* is mapped can be seen in Figure 2.6. An explanation of this program code is also presented in Section 2.3, Chapter 2. For the sake of brevity, we do not explain it any more here.

When performing the application mapping, the major task is to construct the communication between processors in the platform. In our multiprocessor platform, a MicroBlaze processor gets data from other processors via hardware FIFO buffers using a read primitive. It sends data to other processors via the FIFO buffers using a write primitive. For example, the code line 6 - 12 in Figure 2.6 describes that the *Sum* processor reads data from its input FIFO buffers using a read primitive. It writes data to its output FIFO buffers using a write primitive - line 16 - 20. The hardware FIFO buffers in our platform are bounded, and thus the read/write operation is blocking. A blocking-read situation occurs when data is not available at a given input FIFO, i.e., the corresponding FIFO buffer is empty. A blocking-write situation occurs when data can not be written to a particular output FIFO, i.e., the corresponding FIFO buffer is full.

Below we explain the concrete implementation of the read and write primitives. As we use two different buses (FSL and/or LMB) to communicate with the FIFO buffers, we have to define two different sets of read/write primitives. The FSL primitives implement the blocking read/write mechanism in hardware, because we use the MicroBlaze specific assembly instructions, namely *put* and *get* [31] which are shown in Figure 3.7. To make a consistent user interface, we make a wrapper for these assembly instructions as shown in Figure 3.8.

In Figure 3.8, the variable *pos* denotes a port number for a FSL bus of the MicroBlaze processor.

```
1 #define microblaze_bread_datafsl(val, id) \
2             asm(''get %0, %1'' : ''=d'' (##val##) : ''m'' (rfsl##id##))
3
4 #define microblaze_bwrite_datafsl(val, id) \
5             asm(''put %0, %1'' : ''=d'' (##val##) : ''m'' (rfsl##id##))
```

Figure 3.7: MicroBlaze FSL bus read/write assembly code.

```
0  #define readFSL(pos, value, len) \
       do {\
           int i = 0;\
           for (i = 0; i < len; i++)\
               microblaze_bread_datafsl(((volatile int *) value)[i], pos);\
5      } while(0)

   #define writeFSL(pos, value, len) \
       do {\
           int i = 0;\
10         for (i = 0; i < len; i++)\
               microblaze_bwrite_datafsl(((volatile int *) value)[i], pos);\
       } while(0)
```

Figure 3.8: MicroBlaze FSL bus read/write primitive.

Its value ranges from 0 to 7. *value* is a variable used to store the data to be read/written. *len* denotes the length (measured in 32-bit words) of the data to be read/written. When performing the read operation, the MicroBlaze processor simply gets data from one of its FSL input ports and stores the data into the variable *value* - see code line 4. For the write operation, it puts the data stored in the variable *value* to one of its FSL output ports - see code line 11.

For the LMB bus, we can only implement the blocking mechanism in software, which means that we need a busy-wait mechanism in the program code. This can be seen in Figure 3.9. The variable *pos* in Figure 3.9 now denotes a physical address of a FIFO buffer in the LMB bus memory space connected to a MicroBlaze processor via the LMB bus. When performing the read operation, the MicroBlaze processor first gets the status of the FIFO buffer - see line 3. While the FIFO status is *Empty*, an empty ”while” loop is used to block the reading - see line 5. Otherwise, the processor gets the data from the FIFO buffer and stores the data into the variable *value* - see code line 6. For the write operation, see code line 13-16, the processor first gets the status of the FIFO buffer. While the FIFO status is *Full*, an empty ”while” loop is used to block the writing. Otherwise, the processor puts the data stored in the variable *value* to the FIFO buffer.

As shown above, the blocking read/write primitive is easily implemented as micros. It implements an efficient synchronization mechanism between the MicroBlaze processors in our multiprocessor platform. From the above discussion, we can see that the blocking read/write mechanism in the FSL primitives implementation is faster than the LMB primitives since the FIFO status checking is not necessary. Thus, the overhead of the communication between the processors is much less when we use the FSL ports to connect processors in our multiprocessor platform. However, sine the FSL primitives are implemented using the MicroBlaze specific assembly instructions, they depend much on a specific processor. For a comparison, the LMB primitives are more general. A processor usually has a local memory bus (LMB) and the LMB primitives can always be implemented for that processor.

```
0  #define readLMB(pos, value, len) \
       do { \
           int i;\
           Status = getFifoStatus (pos);\
           for (i = 0; i < len; i++) { \
5              while (Status == Empty) { };\  //FIFO is empty, reading is blocked.
               ((volatile int *) value)[i] = getFifoData (pos);\
           }\
       } while(0)

10 #define writeLMB(pos, value, len) \
       do { \
           int i;\
           Status = getFifoStatus (pos);\
           for (i = 0; i < len; i++) { \
15             while (Status == Full) { };\  //FIFO is full, writing is blocked.
               putFifoData (pos) = ((volatile int *) value)[i];\
           }\
   } while(0)
```

Figure 3.9: MicroBlaze LMB bus read/write primitive.

3.4 Project Generation for Xilinx Platform Studio

In this section, we explain the method inside our ESPAM tool that we apply to build an Xilinx Platform Studio (XPS) project using the components described in Section 3.2.

XPS is an Integrated Development Environment (IDE) used to develop Xilinx Embedded Development Kit (EDK)-based system designs. It allows designers to configure a HW/SW platform including automatic generation of device drivers and Board Support Packets (BSPs) for their VirtexII-Pro chips where an FPGA technology is combined with four embedded PowerPCs processors. However, directly using XPS to design an embedded system of processors network with dozens of processors and connections is extremely time-consuming and error-prone. At the same time, the parallelism implicit in an application can only be depicted manually. All these weak points restrict the building of a complex embedded system in XPS in a relatively short amount of time. To reduce the design time, the XPS tool can be used as a back-end tool of our ESPAM tool. ESPAM systematically synthesizes a platform and automatically generates all necessary files for an XPS project from an application specified as a KPN. Therefore, we can synthesize the system and build the bitstream file for a specific FPAG board efficiently and effectively.

3.4.1 Xilinx Platform Studio project Specification

By exploring XPS, we find out that all the information of a project is stored in four files: an Xilinx Microprocessor Project (XMP) file [30], a Microprocessor Hardware Specification (MHS) file [30], a Microprocessor Software Specification (MSS) file [30] and a User Constraint File (UCF) [37]. The XMP file stores the project options. The MHS file defines all hardware components used in a platform as well as the connections between these components. The MSS file contains directives for customizing libraries, drivers, and file systems. The UCF file contains pin information for the physical implementation in the selected FPGA device.

The XMP file points the location of the MHS file, the MSS file, and the C/C++ program codes

that need to be compiled into an executable file for a processor. It also includes the FPGA architecture family and the device type for which the XPS hardware tool flow needs to run. A sample XMP file is shown in Appendix A.3. Lines 3 - 4 specify the location of the MHS and MSS files. Lines 6 - 10 include the FPGA architecture family and the device type. Lines 25 - 39 define 5 processors. Lines 40 - 145 consists of all options of the program codes for the processors. For example, lines 40 - 61 define all options for the program codes of processor mb_P1. The detailed format of the XMP file is described in Appendix A.1 and A.2.

The MHS file defines all hardware components used in a platform as well as the connections between these components. Each component definition starts with *BEGIN* keyword and ends with *END* keyword. Between these keywords, three commands, namely *BUS_INTERFACE*, *PORT*, and *PARAMETER* are used to specify options. Each command has the following format: $name = value$, where *name* is the name of a bus interface, a port, or a parameter, and the *value* is a wire or a parameter number. A sample MHS file is shown in Appendix B. Figure 3.10 shows a small part of the sample MHS file, which defines an instantiation of a processor component. Six parameters are specified in lines 317 - 322 using the command *PARAMETER*. Five bus interface are specified in lines 323 - 227 using the command *BUS_INTERFACE*. One clock port is specified in line 328 using the command *PORT*.

```
    BEGIN microblaze
     PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
     PARAMETER HW_VER = 2.10.a
     PARAMETER C_NUMBER_OF_PC_BRK = 1
320  PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
     PARAMETER C_FSL_LINKS = 2
     PARAMETER INSTANCE = mb_ND_4
     BUS_INTERFACE DLMB = dlmb_ND_4
     BUS_INTERFACE DOPB = mb_opb
325  BUS_INTERFACE SFSL0 = sync_fifo_ED_1_ND_3_to_ND_4
     BUS_INTERFACE MFSL0 = sync_fifo_ED_2_ND_4_to_ND_5
     BUS_INTERFACE ILMB = ilmb_ND_4
     PORT CLK = sys_clk_s
    END
```

Figure 3.10: MicroBlaze processor component MHS definition.

The MSS file contains directives for customizing libraries, drivers, and file systems. It has the same format as the MHS file and the *PARAMETER* keyword is required before each assignment. In Figure 3.11, code lines 3 and 9 define the processor type and the operating system type for a processor.

```
  0   BEGIN PROCESSOR
       PARAMETER DRIVER_NAME = cpu
       PARAMETER DRIVER_VER = 1.00.a
       PARAMETER HW_INSTANCE = mb_ND_5
       PARAMETER COMPILER = mb-gcc
  5    PARAMETER ARCHIVER = mb-ar
      END

      BEGIN OS
       PARAMETER OS_NAME = standalone
 10    PARAMETER OS_VER = 1.00.a
       PARAMETER PROC_INSTANCE = mb_ND_5
      END
```

Figure 3.11: MicroBlaze processor component MSS definition.

The UCF file contains constraints such as timing, FPGA pin locations, FPGA resource specification, and I/O standards. For example, in Figure 3.12, the read port of a UART component is assigned to pin *F9* and the write port is assigned to pin *H11*.

```
NET RS232_Uart_1_RX  LOC = F9;
NET RS232_Uart_1_TX  LOC = H11;
```

Figure 3.12: Small part of a UCF file.

3.4.2 The Project Suite

The above sections have described all necessary files that comprise an XPS project. In order to use the Xilinx Platform Studio as a back-end synthesis tool of our ESPAM tool, ESPAM generates the project suite shown in Figure 3.13.

```
<PROJECT_ROOT>
|--- system.xmp
|--- system.mhs
|--- system.mss
|--- project.m: original Matlab code
|--- code/: Program codes
|-------- MemoryMap.h
|-------- aux_func.h
|-------- ND_5/
|------------- ND_5.cpp
|------------- default_link_script
|--- etc/: Optional files for implementation tools
|--- data/: UCF files
|--- pcores/: Customized IP cores for the EDK project
|---------- fifo_if_ctrl_v1_00_a/
|---------- myCLKRST_v1_00_a/
|---------- clock_cycle_counter_v1_00_a/
|---------- counter_input_ctrl_v1_00_a/
```

Figure 3.13: Project directory structure.

The *system.xmp*, *system.mhs*, and *system.mss* files are the corresponding XMP, MHS, and MSS file which have been explained in Section 3.4.1. The *project.m* stores the initial Matlab code. The directories *etc* and *data* contain some constant auxiliary files. Directory *etc* contains the files *bitgen.ut* and *fast_runtime.opt* which store options for the XPS tool. The directory *data* contains a file, namely *system.ucf* which specifies implementation constraints such as timing, FPGA pin locations, FPGA resource specification, and IO standards. Directory *code* stores the program codes for each processor in the platform. Each processor has a corresponding sub-directory, in which program source code files and a default linker script file are stored. We describe the program codes in Section 5.1.4 and the linker script file in Section 5.1.5. In the top level of the *code* directory, there are two files, namely *MemoryMap.h* and *aux_func.h*. They are common for program codes of all processors. The *aux_func.h* file declares read and write primitives and wrappers of all function calls in the initial Matlab code. The *MemoryMap.h* file specifies physical addresses of FIFOs connected to every processor. We explain this file in Section 5.1.6. The *pcores* directory contains predefined IP cores as well as IP cores generated by ESPAM. A concrete example of the whole project hierarchy is presented in Section 5.1.2

3.4.3 Visitor Hierarchy

In this section, we give an overview of the technique used in ESPAM to generate an XPS project. The classes hierarchy is shown in Figure 3.14. We use a software engineering technique called

Visitor [38] to traverse a KPN specification and to generate all necessary files for an XPS project.

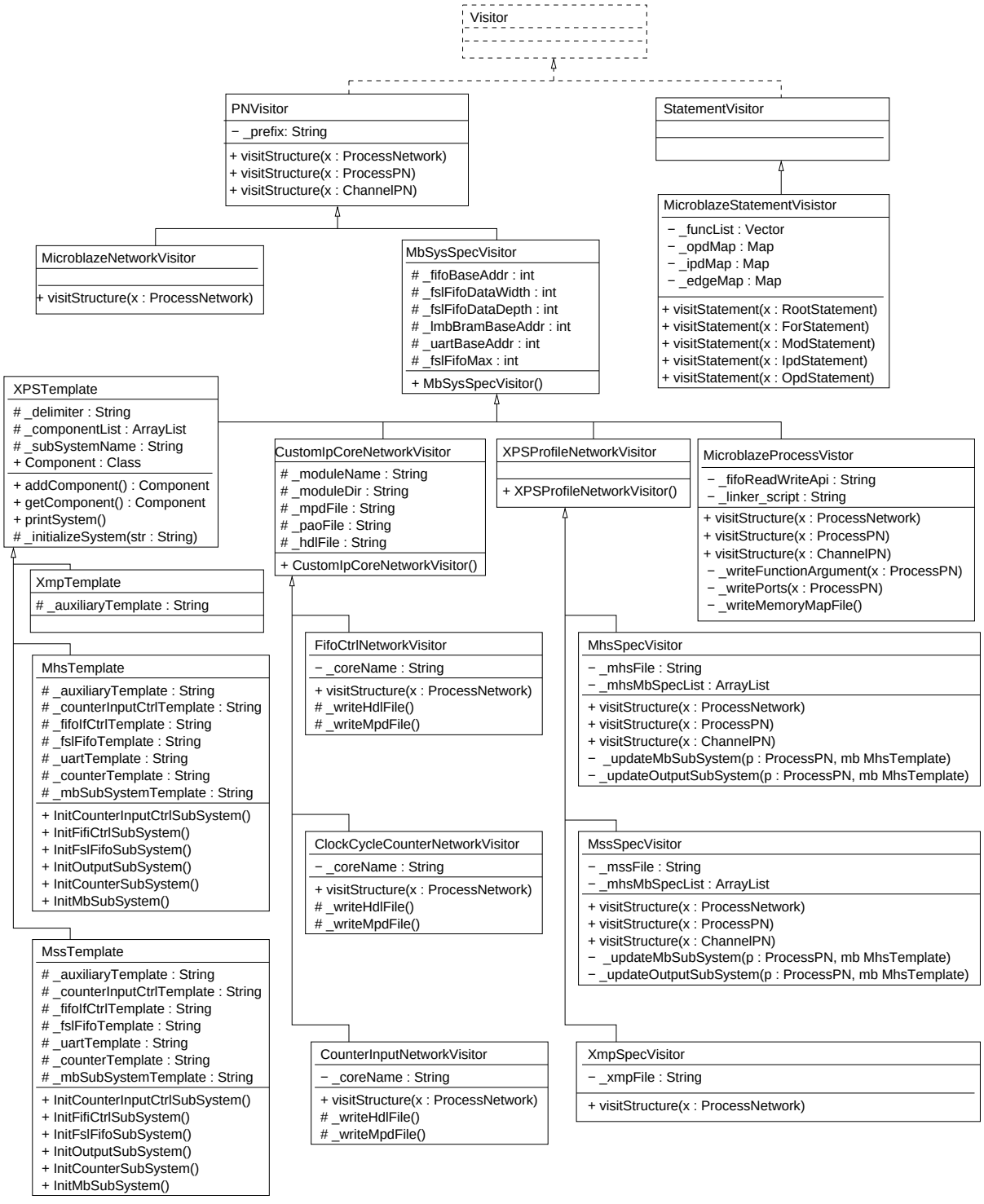

Figure 3.14: Visitor class hierarchy.

On the top level, an interface class called *Visitor* is defined to traverse a KPN specification. Two abstract classes *PNVisitor* and *StatementVisitor* implement the interface class.

The $PNVisitor$ class is an abstract class for a visitor to traverse a Process Network. The $StatementVisitor$ class is another abstract class for a visitor to traverse a processor's syntax tree which is explained in Section 2.3.2. The $MicroblazeStatementVisitor$ class extends $StatementVisitor$ to generate C codes for a MicroBlaze processor.

Concrete class $MicroblazeNetworkVisitor$ extends abstract class $PNVisitor$. It is an engine class called by ESPAM. It calls the method $visitStructure(x : ProcessNetwork)$ of all necessary concrete classes, i.e., $MhsSpecVisitor$, $MssSpecVisitor$, $XmpSpecVisitor$, $FifoCtrlNetworkVisitor$, $CounterInputCtrlNetworkVisitor$, $ClockCycleCounterNetworkVisitor$, and $MicroblazeProcessVisitor$ to generate XPS project files, respectively.

Abstract class $MbSysSpecVisitor$ also extends abstract class $PNVisitor$, in which all the common features of the project, such as the address space of each sub-system, is defined. Four classes, $XPSTemplate$, $CustomIpCoreNetworkVisitor$, $XPSProfileNetworkVisitor$, and $MicroblazeProcessVisitor$, are defined to extend the abstract class $MbSysSpecVisitor$.

Class $MicroblazeProcessVisitor$, which is called by class $MicroblazeNetworkVisitor$, generates global program code files $MemoryMap.h$ and $aux\_func.h$. At the same time, it traverses the syntax tree of each processor and calls method $visitStatement()$ of class $MicroblazeStatementVisitor$ to generate program codes for each processor.

Class $CustomIpCoreNetworkVisitor$ is an abstract class which defines all common features of the generation for custom IP cores. There are three concrete classes, namely $FifoCtrlNetworkVisitor$, $ClockCycleCounterNetworkVisitor$, and $CounterInputNetworkVisitor$, which extend $CustomIpCoreNetworkVisitor$ to generate three different IP cores, i.e., $fifo\_if\_ctrl\_v1\_00\_a$, *clock_cycle_counter*$\_v1\_00\_a$, and $counter\_input\_ctrl\_v1\_00\_a$ described in Section 3.2.4 and Section 5.4.2.

For the project related files, such as the MHS, MSS, and XMP files, we define a template operation. Abstract class $XPSTemplate$ is used to describe the common template descriptions. There are three concrete template classes extend $XPSTemplate$, namely $MhsTemplate$, $MssTemplate$, and $XmpTemplate$, which define the actual templates of the MHS, MSS, and XMP files, respectively.

The last abstract class is $XPSProfileNetworkVisitor$. It is a abstract class for the generation of the above three project files using their corresponding templates. Three concrete class, $MhsSpecVisitor$, $MssSpecVisitor$ and $XmpSpecVisitor$, extend this abstract class. They are used to generate the corresponding MHS, MSS, and XMP file. They are also called by class $MicroblazeNetworkVisitor$ when ESPAM is executed.

3.5 Discussion and Conclusion

In this chapter, we have described in detail how we implement the abstract model of a multiprocessor platform onto a Field Programmable Gate Array (FPGA) chip for prototyping. We use a commercial tool, namely Xilinx Embedded Development Kit (EDK) [32], as the final synthesis tool set to build the multiprocessor platform ready for an implementation on our target FPGA board.

We notice that Xilinx provides designers the EDK tool with a rich set of design tools and a wide selection of standard peripherals required to build embedded processor systems using MicroBlaze or PowerPC processor. But it does not provide a methodology how to design the systems in a systematic and automated way. Thus, the situation the designers are facing is that they have to do it all by hand. This is rather bearable if they only spend several hours to design a simple embedded processor system with a single processor. But for a multiprocessor system, this can be a nightmare. Such a design involves constructing the multiprocessor platform, partitioning an application and allocating them to multiple processors, and debugging the multiprocessor system. This design process is rather time consuming, error prone, difficult and depends very much on the expertise of the designer.

Therefore, we have developed a methodology and tool called ESPAM that allows fast and efficient mapping of an application onto multiprocessor platforms. One novelty in our ESPAM tool is that it automatically generates the multiprocessor platform and the program code for each processor in the platform for a given Matlab application as a project that can be accepted by EDK. The automatic and systematic approach is a correct-by-construction mapping of a KPN parallel specification onto multiprocessor platform. The automation will reduce significantly the design time of a system and possible errors in the mapping process will be eliminated.

Another novelty is the communication mechanism in our multiprocessor platform. We generate a FIFO controller between the processors and the hardware FIFO buffers. The FIFO controller translates the MicroBlaze LMB bus protocol into the FSL bus FIFO specific protocol. The MicroBlaze processor has to communicate with the FIFO buffers via this controller if the LMB bus is used for connection between a processor and a FIFO buffer. This offers a flexibility while implementing the multiprocessor platform on other target platform. If the bus protocol on either side of the controller changes, we only have to make some corrections of the protocol translation mechanism of the controller. The structure or topology of the multiprocessor platform need not to be changed.

Chapter 4

Case Studies

In this chapter we present two case studies that we have conducted in order to validate and evaluate our system design approach presented in Chapter 2 and prototyped in Chapter 3. We analyze the results obtained from the experiments performed in these case studies.

4.1 System Design Flow Using COMPAAN/ESPAM Tool Chain: a Matrix Multiplication Case Study

In this case study, we present some of the results we have obtained by mapping the matrix multiplication application introduced in Chapter 2 onto an FPGA using our system design flow that has been discussed in Chapter 2 and Chapter 3.

The application we consider is a two dimensional (2D) matrix multiplication. The reason for choosing this application can be explained from two aspects. On the one hand, what we need, first of all, is an application to validate the correctness of the multiprocessor platform generated by the proposed COMPAAN/ESPAM tool chain. The matrix multiplication is a fundamental benchmark application that is not complicated but has enough features to illustrate the correctness and usefulness of our system design method. On the other hand, matrix multiplication is an excellent application for parallel processing. It is commonly used in almost all areas of scientific research and it has significant appliance in the areas of graph theory, numerical algorithms, digital signal processing, and digital control.

The input to our system design flow is the application described in a subset of Matlab shown in Figure 2.2. We start with publicly available sequential C code of a matrix multiplication. This code is modified and structured by hand to meet the subset of Matlab that our design flow accepts and to match the features of the matrix multiplication. The only reason we use Matlab is that the COMPAAN tool uses a simple Matlab parser. The writing of the Matlab code together with the functional testing and debugging takes 2 days. After this preparation work, which is an one-time effort only, we start with the mapping of the matrix multiplication using our system design flow.

Our first experiment is to measure how much time it takes to map the 2D matrix multiplication

onto the FPGA using our system design flow. The two matrices we use for multiplication in this experiment are 20×20 integer matrices. Table 4.1 shows the processing time for every step in the flow. The last column shows the total time needed for every step.

Table 4.1: Processing time (hh:mm:ss).

	COMPAAN	ESPAM	Other tools	Manually	Total
STEP 1	00:00:18	–	–	–	00:00:18
STEP 2	–	00:00:24	–	–	00:00:24
STEP 3	–	–	00:45:10	00:10:00	00:55:10
Overall	00:00:18	00:00:24	00:45:10	00:10:00	00:55:52

The overall time of the whole design flow for the matrix multiplication experiment is around 56 minutes. The column "Other tools" now contains a commercial synthesis tool, namely Xilinx Platform Studio (XPS). The time for running this tool is about 45 minutes, which takes 80% of the time for the whole design flow. The column "Manually" indicates that we have to do some manual manipulations. For example, at the beginning of Step 3 in Figure 1.2, we have to manually import the generated project suite into the XPS environment and to initialize the values of the two matrices for the multiplication. All these manual procedures take a total of approximately 10 minutes.

The results show that the mapping of the matrix multiplication onto the FPGA is done in a short amount of time - about one hour. The main reason is the great time performance of the COMPAAN tool and our ESPAM tool that map fully automatically the KPN specification of the application onto a multiprocessor platform description in a few seconds. In XPS this platform description is further converted into synthesizable VHDL code in a few minutes. For comparison, a hand-made design of converting the KPN specification directly into VHDL will take several days or even months. Note that the time consumed by the COMPAAN/ESPAM tool chain in the system design flow is weakly dependent on the complexity of the application. For a very complex application, the time consumed by the first two steps is still at the level of a few seconds. Therefore, we can conclude that the automation realized by the COMPAAN/ESPAM tool chain reduces significantly the design time of a system.

Table 4.2 shows the FPGA resource utilization for the mapping of the matrix multiplication application. The KPN specification derived from COMPAAN in this experiment can be seen in Figure 2.3. The multiprocessor platform derived from this KPN specification is shown in Figure 2.4. It consists of six MicroBlaze processors, eight FIFO buffers, six FIFO control cores, one UART, and some memory blocks and FSL buses associated to each processor. The numbers in Table 4.2 show that on average 43% of the FPGA resources are used. This is not efficient in terms of resource usage for such a simple application. A manual design for sequential implementation on non-multiprocessor platform gives much better resource utilization. However, as has said before, the goal of this case study is to show that we can build very fast a complex multiprocessor platform for an application. A more complex and realistic example is the M-JPEG application which is discussed in Section 4.2.

In the second experiment, we map a set of different KPN specifications of this 2D matrix multiplication onto the FPGA and compare their performance. A sequential application can be expressed as a KPN model in many different ways, thereby generating alternative functionally equivalent KPNs. All these KPNs specify the functionality of the initial application exploiting different degree of task-level parallelism. Mapping these alternative KPN specifications of an

Table 4.2: Virtex II Pro 2VP20: device utilization.

FPGA Resource	Utilization	%
Number of MULT18X18s	6 out of 88	20%
Number of RAMB16s	9 out of 88	40%
Number of SLICEs	1273 out of 9280	43%
Number of BUFGMUXs	1 out of 16	6%

application onto a multiprocessor platform can help us to explore and evaluate the performance of different systems. This allows us to select the KPN and the platform instance that meet best our performance requirements.

In our system design flow, we use algorithmic transformations *Merging* and *Unfolding* [39] [40] in combination with the COMPAAN compiler to increase or decrease the degree of task-level parallelism exploited in the KPN specification. For our matrix multiplication case, the most computational intensive processes in the application are *MultProp* and *Sum* that compute the product of the elements from the two matrices and the sum of products. We first use the *Merging* transformation to merge these two processes into one single process, see Figure 4.1. Then by applying *Unfolding*, we further partition this process into several processes that run concurrently. Thus, we can obtain a set of KPN specifications having different degree of exploited task-level parallelism. Figure 4.1 to 4.4 show four KPN specifications where the merged process of *MultProp* and *Sum* is unfolded by factor 1, 2, 3, and 4. Unfolding factor 1 means unfolding is not performed at all while factor 2, 3, or 4 means the process is partitioned into two, three, or four concurrent processes, respectively.

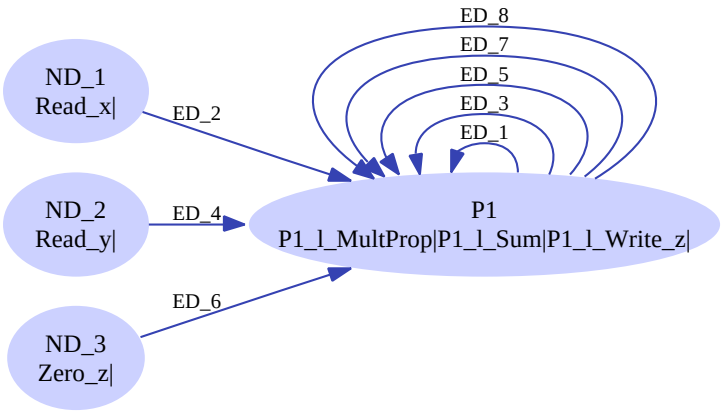

Figure 4.1: Unfolding factor = 1.

We map the four KPN specifications of the matrix multiplication in this experiment. We implement each one onto the FPGA in two different ways regarding to different bus connections, namely LMB and FSL bus connections. Then we measure the clock cycles needed for each implementation to process the two 20×20 integer matrices multiplication. The results are shown in Figure 4.5.

From the experimental results, first we can see that for each KPN specification with identical unfolding factor, the cycles consumed by the LMB bus implementation are much more than those of the FSL bus implementation. Obviously, the FSL bus is faster than the LMB bus. This is because the FLS read/write primitives shown in Figure 3.8 perform the blocking read/write in hardware with 1 assembly instruction (See lines 2 and 5 in Figure 3.7) and in total each of

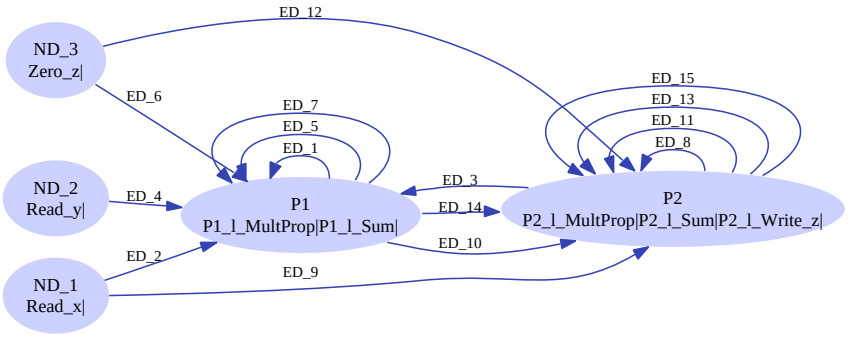

Figure 4.2: Unfolding factor = 2.

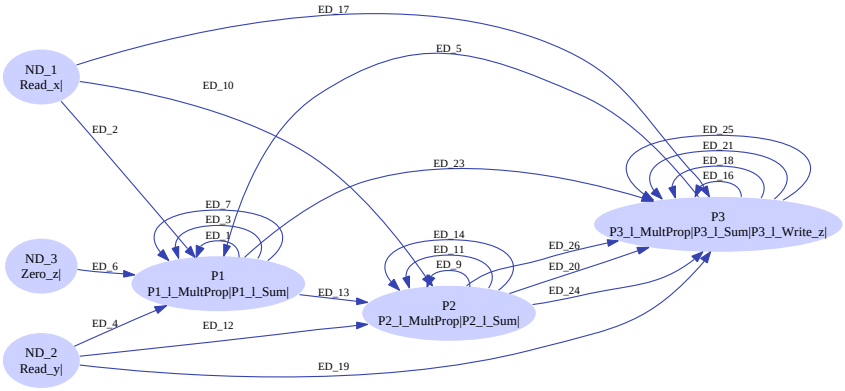

Figure 4.3: Unfolding factor = 3.

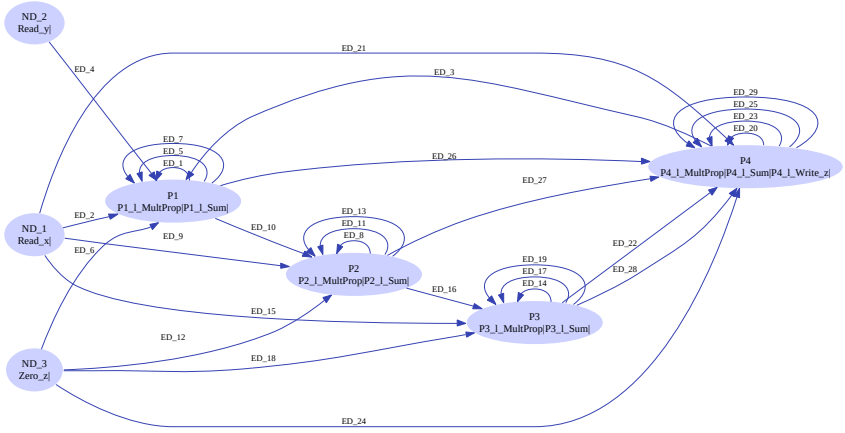

Figure 4.4: Unfolding factor = 4.

these primitives is implemented with 9 assembly instructions (1 instruction for read/write FIFO, 6 instructions for the *for* loop and 2 instructions for an end of the function call). In contrast, the LMB read/write primitives perform the blocking read/write in software, see lines 7 and 19 in Figure 3.9. This causes additional 6 assembly instructions. It means that in total each primitive for the LMB bus is implemented with 15 assembly instructions. Based on this fact, we can conclude that we get a speedup of 1.67 on average using the FSL bus as it is shown by the black boxes in Figure 4.5.

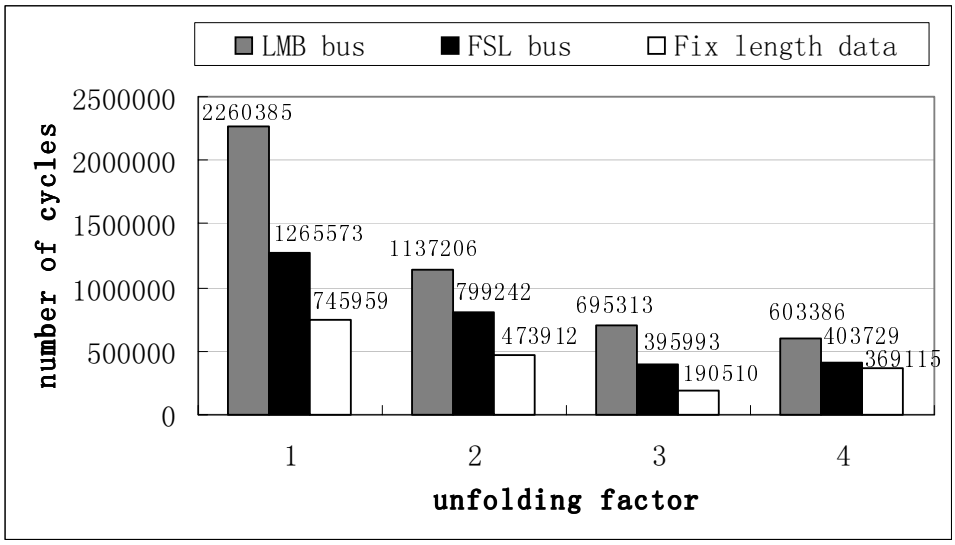

Figure 4.5: Experimental results.

From a viewpoint of different unfolding factors, speedup can also be obtained with the increment of the unfolding factor. Larger unfolding factor means the process is split into more concurrent processes, thus a higher degree of parallelism is exploited. This is especially obvious for the case of LMB bus implementation, see the gray boxes in Figure 4.5. We have a speedup of 2 from factor 1 (no unfolding) to factor 2. Unfolding by 3, we get 3 times less clock cycles. Unfolding by 4, we get nearly 4 times less clock cycles compared to the unfolding by factor 1.

The experimental results discussed above are obtained by using the FSL and LMB bus read/write primitives shown in Figure 3.8 and Figure 3.9, respectively. They are general read/write primitives in our system design method for stream oriented applications. They both have a parameter *len* that specifies the length of the data for each read/write operation. For example, the *for* loop in the FSL read primitive in line 3 - 4 of Figure 3.8 implements reading of data with different lengths. This *for*-loop control adds extra clock cycles to perform the reading. The same is valid for the LMB read/write primitives. In our matrix multiplication case, however, the parameter *len* is fixed to 1, because we always read one 32-bit data token at a time. Therefore, we can remove the *for* statement in order to obtain a further speedup as shown by the white boxes in Figure 4.5.

There is an exception in the unfolded-by-4 system. As Figure 4.5 shows the cycles consumed in FSL-bus system and Fix-length-data system are more than the corresponding cycles in the unfolded-by-3 system. We find out that the bottleneck is in the input processors. When we unfold by factor 4, the input data is split to 4 sets by using *for-if* control statements. These *for-if* control statements execute sequentially in the input processor. The additional cycles they need mask the performance improvements gained by the unfolding. We conclude that we can get the best-performance system for this application by unfolding by 3. Further unfolding does not give any improvement.

In this case study, we verify the methods and techniques implemented in our tool chain. Using our tool chain, the design time of a system can be reduced from days to hours. Thus, exploring the performance of alternative KPN specifications mapped onto instances of a multiprocessor

platform using our tool chain is feasible. Meanwhile, we explore two different bus connections, i.e., FSL and LMB. By choosing the FSL as communication bus, the system performance can be improved significantly.

4.2 Exploring the Performance of Alternative KPN Instances: an M-JPEG Case Study

In this case study, we consider a more complex application, a modified Motion JPEG (M-JPEG) encoder, to evaluate our design flow. Like traditional M-JPEG encoder, this modified M-JPEG encoder compresses a sequence of frames, applying JPEG [41] [42] compression to each frame. This encoder operates on video data in a 4:2:2 YUV formats on a per-frame basis.

The initial Matlab code is shown in Figure 4.6. It is parameterized in the number of frames to be processed (*NumFrames*) and in the vertical (*VnumBlocks*) and horizontal (*HnumBlocks*) size of a frame in number of 8×8-pixel blocks. For example, the code in line 1 specifies that the number of frames in the sequence can be any integer value from 1 to 100. The code in lines 5-9 initializes the luminance and chrominance quantization table (*QTables*) and luminance and chrominance Huffman table (*HuffTableAC*), etc. First, frames in YUV format are divided in 8×8-pixel blocks by the *VideoInMain()* function where every block is a 4:2:2 YUV block. Then a standard JPEG compression algorithm is executed for each frame. A Discrete Cosine Transform (*DCT*) is applied on every 4:2:2 YUV block - line 16, followed by quantization (*Q*) and variable-length encoding (*VLE*) - line 17-19. Function *VideoOut()* in line 20-21 adds header information to the compressed frame.

```
1   %parameter NumFrames 1 100;
2   %parameter VNumBlocks 2 100;
3   %parameter HNumBlocks 1 100;
4
5   for k = 1:1:1,
6     [LuminanceQTable, ChrominanceQTable, LuminanceHuffTableDC, ChrominanceHuffTableDC,
7      LuminanceHuffTableAC,ChrominanceHuffTableAC, LuminanceTablesInfo, ChrominanceTablesInfo
8     ] = DefaultTables();
9   end
10
11  for k = 1:1:NumFrames,
12    [ HeaderInfo ] = VideoInInit();
13    for j = 1:1:VNumBlocks,
14       for i = 1:1:HNumBlocks,
15         [ Block ] = VideoInMain();
16         [ Block ] = DCT( Block );
17         [ Block ] = Q( Block, LuminanceQTable, ChrominanceQTable );
18         [ Packets ] = VLE( Block, LuminanceHuffTableDC,ChrominanceHuffTableDC,
19                            LuminanceHuffTableAC,ChrominanceHuffTableAC );
20         [ dummy ] = VideoOut( HeaderInfo, LuminanceTablesInfo,
21                               ChrominanceTablesInfo, Packets );
22       end
23    end
24  end
```

Figure 4.6: Task-Level specification of the M-JPEG application in Matlab.

In this case study, we first convert the initial Matlab program in Figure 4.6 into several alternative KPN specifications using COMPAAN. The general partitioning strategy employed by

COMPAAN is to create a process for every function call in the initial program. By using the *Merging* transformation, we can alter the partitioning strategy and get alternative KPN specifications, thereby exploiting different degree of task-level parallelism available in the initial M-JPEG application. In this case study, we conduct three experiments to evaluate our COMPAAN/ESPAM design flow. In the first experiment, we merge all the functions into one process as shown in Figure 4.7. Actually, this KPN specification exploits no parallelism as this is the case in the initial Matlab program. We use this KPN as a refereed point for comparison. In the second experiment, we only merge the input and output functions, see Figure 4.8. In the third experiment depicted in Figure 4.9, we build a task-level pipelined network by not merging the input and output functions.

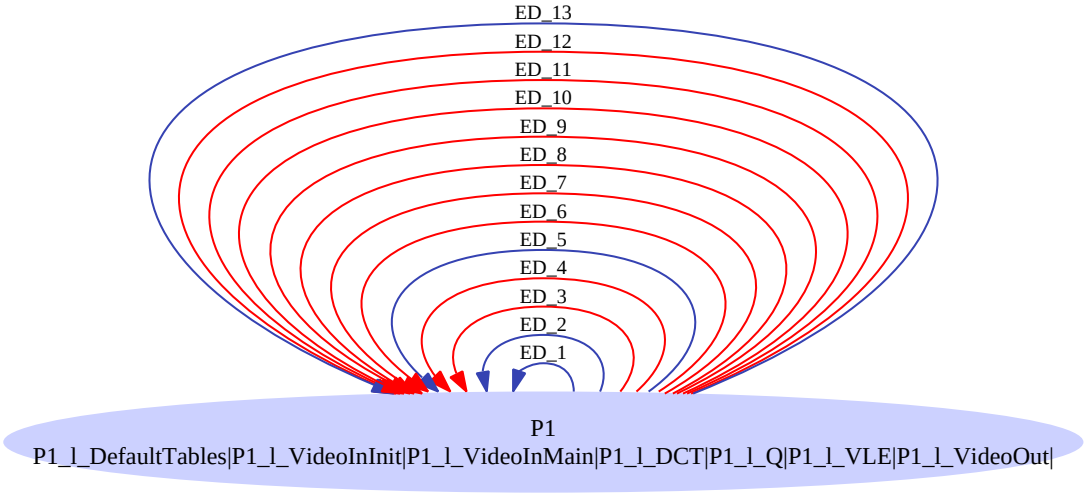

Figure 4.7: Experiment 1 - Merge all functions in to one process.

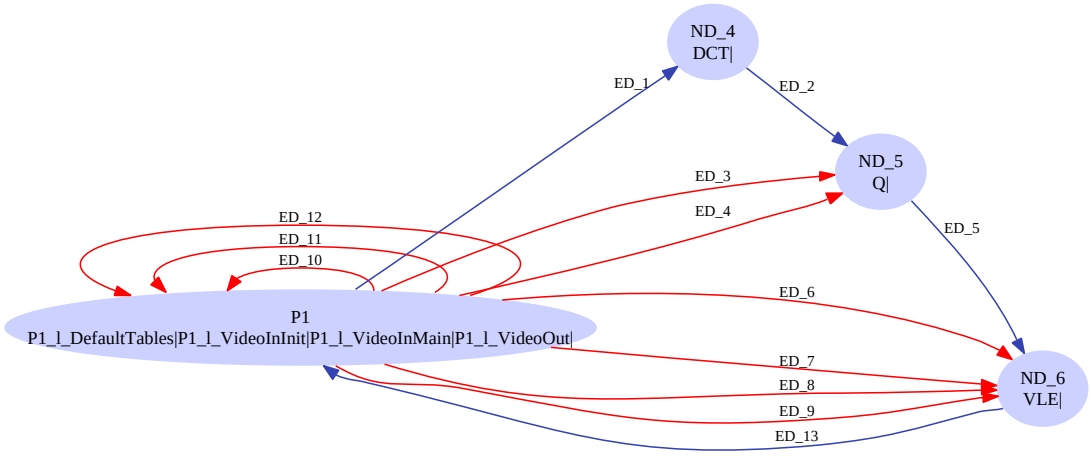

Figure 4.8: Experiment 2 - Merge only the input and output functions.

In the second step, we use ESPAM to derive three different platforms and map these three networks onto the platforms, respectively. So that we can compare their system performance. In Experiment 1, we map process *P1* to a MicroBlaze soft processor. The 13 self-loop channels are mapped to 13 FIFOs. Since the MicroBlaze processor only has 8 input and 8 output FSL ports as

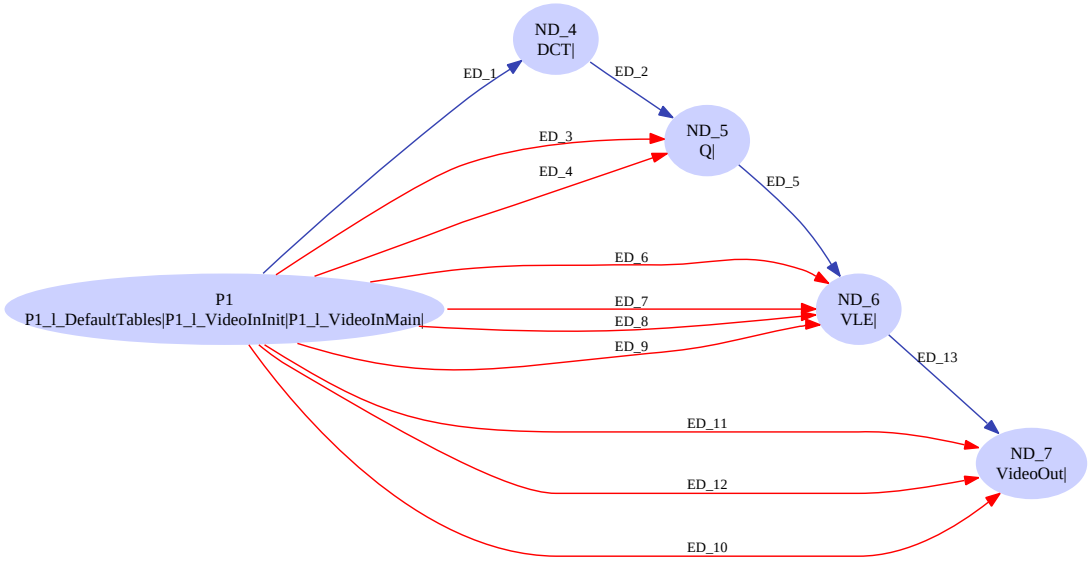

Figure 4.9: Experiment 3 - Fully pipelined M-JPEG.

has explained in Section 3.2.3, we connect the first 8 FIFOs to FSL ports and the rest 5 to LMB bus. In Experiment 2, we create a partial pipelined system, in which the input&output functions are mapped to one MicroBlaze processor and *DCT*, *Q*, and *VLE* are mapped to separate MicroBlaze processors respectively. In Experiment 3, ESPAM generates a 5-stage multi-processor pipeline system. Each stage in the pipeline is executed by a processor on which a process is mapped. See in Figure 4.9, The *VideoInMain()* process is mapped to processor *P1*, *DCT()* to *ND_4*, *Q()* to *ND_5*, *VLE()* to *ND_6*, and *VideoOut()* to *ND_7*. The performance of these three system is shown in Figure 4.10.

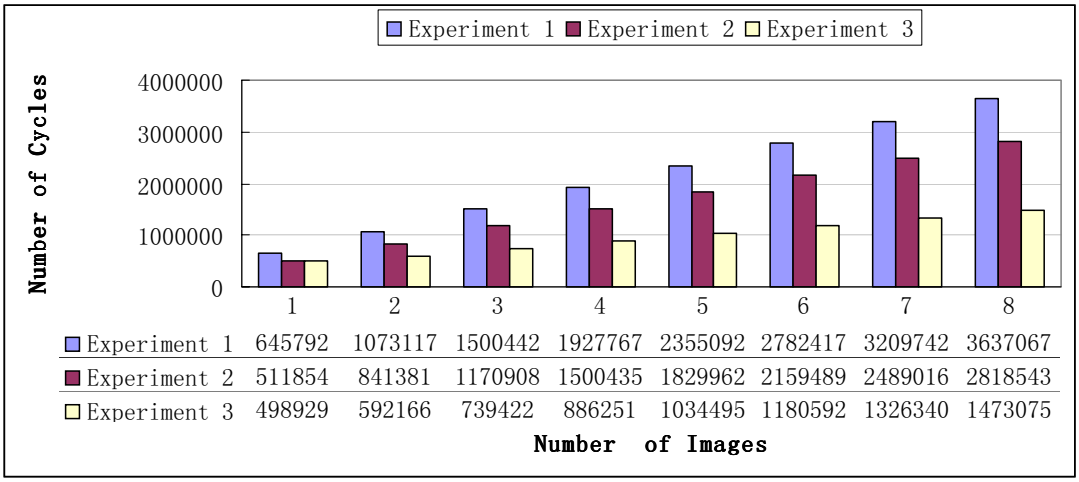

Figure 4.10: Performance of the M-JPEG application of the three experiments.

Comparing the performance of Experiment 1 and Experiment 2, the partly pipelined network in Experiment 2 cannot provide too much improvements. This is because we merge the input and output into one processor. The next frame can be put into the pipeline only after the previous one is sent to the output, which causes it perform a sequential processing the same as Experiment

1. The only difference is the bus connection. As has described above, the input and output of the upper 9-13 FIFOs of Experiment 1 are connected via the relatively low speed LMB bus. Whereas in Experiment 2, the 13 FIFOs are dispersed in the network. Only 3 output ports of processor *P1* need to connect to the LMB bus. This is the reason that Experiment 2 causes less clock cycles than Experiment 1, which is shown in the gray and black columns in Figure 4.10.

From the white boxes in Figure 4.10, we can see that if only one image is put to the pipeline, the clock cycles used are nearly the same as those in Experiment 2. There is no performance improvement. However, when a sequence of images is put to the pipeline, speedup is achieved. From a theoretical point of view, suppose the clock cycles used for one image is I, we can express the relation of the sequence of n images and clock cycles $C(n)$ for Experiment 2 as Equation (4.1) and for Experiments 3 as Equation (4.2). Then the speedup expression is given by Equation (4.3). When an infinite sequence of images is put in the pipeline, $n \rightarrow \infty$, we can theoretically achieve 5 times speedup.

$$C_2(n) = I \times n \tag{4.1}$$

$$C_3(n) = I + I \times n/5 \tag{4.2}$$

$$\lim_{n\rightarrow\infty} \frac{I \times n}{I + I \times n/5} \tag{4.3}$$

However, from Figure 4.11, we can see the actual speedup in Experiment 3 is less than the theoretical speedup. The reason for this is that the stages in the pipeline are not balanced. The total cycles for one image of 8×8 pixels is 498929 as shown in the first white box in Figure 4.10. Function *DefaultTables()*, which takes 147318 cycles, has an one time effect on the pipeline and will not be put to the total cycles. Thus, the total cycles that the pipeline runs are 351611. Table 4.3 shows the cycles and utilization percentage of each pipeline stage in Experiment 3. From the table, we can see that *DCT()* takes more than 40 percent of the whole time. If we unfold *DCT()* into two parallel processors, the speedup will be significantly increased, because we will balance the pipeline.

Table 4.3: Cycles and utilization percentage of each pipeline stage in experiment 3.

	VideoInMain	DCT	Q	VLE	VideoOut
Cycles	21,502	142,293	76,736	92,094	18,991
Percentage(%)	6.1	40.5	21.8	26.2	5.4

Another issue we need to explain is related to the input and output stages. From Table 4.3, we see that the input and output stages take only 11 percent from the total clock cycles. This is because we get an image frame from an external memory and store the result in the on-chip RAM. But for a real life streaming system, this is not the case. We might expect that the input and output stages will take more percentage. More exploration can be done to get the best pipeline performance, which is out of the scope of this experiment.

In this case study, we verify the methods and techniques implemented in our tool chain with a more complex M-JPEG application. For this complex M-JPEG application, we find out that a

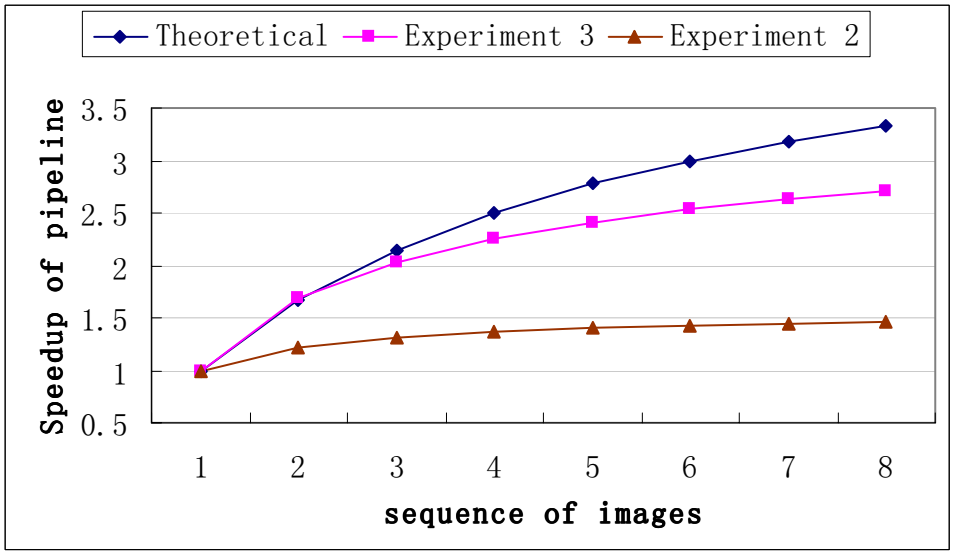

Figure 4.11: Speedup comparison of the theoretical value and actual value.

lot of new issues need to be considered. All these issues are related to the memory allocation, e.g., the stack size of each processor, the FIFO size, and the instruction/data memory allocation of each processor. In the matrix multiplication case study, the problem cannot be revealed because of the simplicity of the functions in it. Whereas, in the M-JPEG application, functions are more complex. For example, an argument of a function call may take hundreds of bytes. All these arguments will be pushed to the stack of a processor when calling this function. The stack may be filled and then broken through if its size is not enough when nested function calls occur.

We cannot predict the minimum size of the stack and memory at compile time. Therefore, we have to adjust the size of the instruction memory, data memory and stack after our automatic generation. The size of the memory and stack for the three experiments are listed in Table 4.4, 4.5, and 4.6. From these tables, we can see that the *P1* processor of all three experiments needs an extremely huge stack, which size is up to 64 Kbytes. The target board has only 88 blocks on-chip memory, which is 176 Kbytes (2k per block), thus we use an external memory for the stack.

Table 4.4: Experiment 1 - memory allocation.

	P1(VIn Vout DCT Q VLE)		
	Instruction memory	Data memory	Stack
Size (KB)	64	64	45

Table 4.5: Experiment 2 - memory allocation.

	P1 (VIn and VOut)		ND_4 (DCT)		ND_5(Q)		ND_6(VLE)	
Size(KB)	Ins/Data	Stack in external memory	Ins /Data	Stack	Ins/Data	Stack	Ins/Data	Stack
	32	64	32	18	32	9	32	19

Table 4.6: Experiment 3 - memory allocation.

	P1 (VIn)			ND_4 (DCT)		ND_5(Q)		ND_6(VLE)		ND_7(Vout)	
Size(KB)	Ins	Data	Stack in external memory	Ins /Data	Stack	Ins/Data	Stack	Ins/Data	Stack	Ins /Data	Stack
	8	16	64	32	18	16	9	32	19	32	20

Another fact needed to be discussed is the size of FIFOs in the processors network. To avoid a deadlock of the network, we need to predict the minimum FIFO size. However, our tool chain does not support this feature currently. To predict the minimum size, we need to evaluate how much tokens at most will be in a FIFO at a certain time. This is another open problem. For the three experiments in this case study, we allocate the size of each FIFO following Table 4.7.

Table 4.7: FIFO memory allocation.

	ED_1	ED_2	ED_3	ED_4	ED_5	ED_6	ED_7	ED_8	ED_9	ED_10	ED_11	ED_12	ED_13
Size(KB)	2	2	2	2	2	4	4	4	4	2	4	4	2

In this case study, we use our COMPAAN/ESPAM tool chain to design three different systems for the M-JPEG application. With the help of our tool chain, we can fast map our designs on an FPGA board and get a runnable system. Because of the complexity of the M-JPEG application, a few new issues are revealed during this case study. They are size adjustment of the stack, memory allocation, and FIFO buffer. Further research work is needed to supply these new issues. Although some features still need to be adjusted manually, our tool chain achieves great efficiency for prototyping alternative KPN specification. We believe this efficiency will release system designers from the heavy workload of prototyping a complex multiprocessor system, so that they can focus on exploring the performance of alternative systems.

Chapter 5

Getting Started: Tutorial with Example using the COMPAAN/ESPAM tool chain

In this chapter, we give a tutorial to show how an embedded system can be designed using our COMPAAN/ESPAM tool chain and the commercial synthesis tool Xilinx Platform Studio (XPS). We choose a complex application, namely the fully pipelined M-JPEG shown in Figure 4.9 as our example to explain each design step in detail.

XPS is an Integrated Development Environment (IDE) used to develop Xilinx Embedded Development Kit (EDK)-based system designs. An XPS project consists of a few description files, e.g., the Xilinx Microprocessor Project (XMP) file, the Microprocessor Hardware Specification (MHS) file, the Microprocessor Software Specification (MSS) file, and the User Constraint File (UCF). Our ESPAM tool acts as a front-end compiler of XPS. ESPAM systematically synthesizes a platform and automatically generates all necessary files of an XPS project from an application specified as a KPN.

This chapter is further organized as follows. In Section 5.1, we describe the steps of generating an XPS project as well as we explain the result of the generation in detail. In Section 5.2, we describe how to import the generated project into XPS. In Section 5.3, we discuss what is needed to be modified manually in the imported project. Finally, we show how to execute the modified project and how to get results as well as we explain some debug procedures in Section 5.4

5.1 XPS Project Generation

In this section, we briefly describe how to generate a KPN specification from an application written in Matlab and how to generate an XPS project suite from the KPN specification. We use COMPAAN to generate the KPN specification and ESPAM to generate the XPS project suite.

5.1.1 Application Source Code

The initial sequential Matlab code is shown in Figure 4.6. It is a modified Motion JPEG (M-JPEG) encoder which compresses a sequence of video frames, applying JPEG compression to each frame in the video sequence. The detail explanation of this application is given in Section 4.2.

By default, COMPAAN generates a process for each function call in the initial Matlab code. Therefore, a 7-process network will be generated for the code in Figure 4.6. To generate a 5-stage task-level pipelined system, we need a 5-process network. To get such network, we merge some of the 7 function calls using the key word $P1\_l\_$ shown in the bold lines 23, 27, and 30 in Figure 5.1. In this case, function calls $DefaultTables$, $VideoInInit$, and $VideoInMain$ will be merged into one process.

```
1    %%MJPEG_5p.mat
     %parameter NumFrames 1 100;
     %parameter VNumBlocks 2 100;
     %parameter HNumBlocks 1 100;
5
     %typedef HeaderInfo             THeaderInfo;
     %typedef LuminanceQTable        TQTables;
     %typedef ChrominanceQTable      TQTables;
     %typedef LuminanceHuffTableDC   THuffTablesDC;
10   %typedef ChrominanceHuffTableDC THuffTablesDC;
     %typedef LuminanceHuffTableAC   THuffTablesAC;
     %typedef ChrominanceHuffTableAC THuffTablesAC;
     %typedef LuminanceTablesInfo    TTablesInfo;
     %typedef ChrominanceTablesInfo  TTablesInfo;
15   %typedef Packets                TPackets;
     %typedef Block                  TBlocks;

     for k = 1:1:1,
       [ LuminanceQTable,ChrominanceQTable,
20       LuminanceHuffTableDC,ChrominanceHuffTableDC,
         LuminanceHuffTableAC,ChrominanceHuffTableAC,
         LuminanceTablesInfo, ChrominanceTablesInfo
       ] = P1_l_DefaultTables();
     end
25
     for k = 1:1:NumFrames,
       [ HeaderInfo ] = P1_l_VideoInInit();
       for j = 1:1:VNumBlocks,
         for i = 1:1:HNumBlocks,
30         [ Block ] = P1_l_VideoInMain();
           [ Block ] = DCT( Block );
           [ Block ] = Q( Block, LuminanceQTable, ChrominanceQTable );
           [ Packets ] = VLE( Block,
                              LuminanceHuffTableDC,ChrominanceHuffTableDC,
35                            LuminanceHuffTableAC,ChrominanceHuffTableAC );
           [ dummy ] = VideoOut( HeaderInfo, LuminanceTablesInfo,
                                 ChrominanceTablesInfo, Packets );
         end
       end
40   end
```

Figure 5.1: M-JPEG task-level pipeline Matlab code.

Another fact which needs to be explained is that the Matlab code in Figure 5.1 is the top-level entry for our COMPAAN/ESPAM tool chain. Our tool does not deal with the implementation of the function calls used in the initial Matlab code, it only generates empty wrappers for these function calls. To implement the application in XPS, these empty wrappers have to be replaced

in a later step. Therefore the definition and implementation of all functions in the initial Matlab code need to be considered. We declare needed data types in lines 6 - 16 in Figure 5.1, the definition of which is in file $types.h$. The $DefaultTables()$ function in line 23 is implemented in file $ControlInit.cpp$. $VideoInInit()$ and $VideoInMain()$ in lines 27 and 30 are implemented in file $Video\_in.cpp$. $DCT()$ in line 31 is implemented in file $DCT.cpp$. $Q()$ in line 32 is implemented in file $Q.cpp$. $VLE()$ in line 33 is implemented in file $VLE.cpp$. $VideoOut()$ in line 36 is implemented in file $Video\_out.cpp$. We need to import manually all these files to XPS. We describe this procedure in Section 5.3.2. The source files discussed above can be found in the CVS repository:
docs/students/KaiHuang_JiGu/experiment/projects/m-jpeg/MJPEG-Pentium.tar.gz

5.1.2 KPN Specification and XPS Project Generation

In this section, we describe how to generate the KPN specification and the XPS project from the Matlab code in Figure 5.1 using our COMPAAN/ESPAM tool chain.

Tool Chain Commands

We use three commands, shown in Figure 5.2, in the tool chain to generate the KPN specification and the XPS project. These commands have to be executed from a Linux terminal on a computer where the COMPAAN/ESPAM software is installed:

```
1) matparser --input MJPEG_5p.mat --output MJPEG_5p.sac --compile --verbose -r
2) dgparser --input MJPEG_5p.sac --output MJPEG_5p --xml -r
3) panda --input MJPEG_5p.xml -c MJPEG_5p.m -ls -lms -RP -r --clockgen --mbxps <libXPS>
```

Figure 5.2: Commands and options.

The first command starts the MATPARSER tool [43]. It transforms the initial Matlab code into a single assignment code (SAC), which resembles the dependence graph (DG) of the initial Matlab code. We explain the necessary options below:

- **--input**: This option is followed by a filename that points to a file where the initial Matlab code is stored.
- **--output**: This option is followed by a filename that points to a file where results, for example the SAC, need to be written.
- **--compile**: This option tells MATPARSER to convert the Matlab code into a SAC.
- **--verbose**: This option causes MATPARSER to produce information messages showing the progress made in the conversion.
- **-r**: This option applies a set of optimizations on a solution tree which describes data-dependencies. The optimizations include removing redundant if/else statements, removing redundant index statements, and removing redundant sub-graphs.

The second command starts the DGPARSER tool. It converts the SAC into a Polyhedral Reduced Dependence Graph (PRDG) data structure, which is a compact mathematical representation of the DG in terms of polyhedra. Figure 5.3 depicts the resultant PRDG, in which each node represents a function in the initial Matlab code and the edges represent data dependences among all function calls. To run DGPARSER, the necessary options are:

- **--input**: This option specifies the SAC file generated by MATPARSER.
- **--output**: This option specifies the output file where the PRDG data structure will be stored.
- **--xml**: This option specifies the format of the output file as XML.
- **-r**: This option manipulates the parse tree. In particular, it removes control from the index statements.

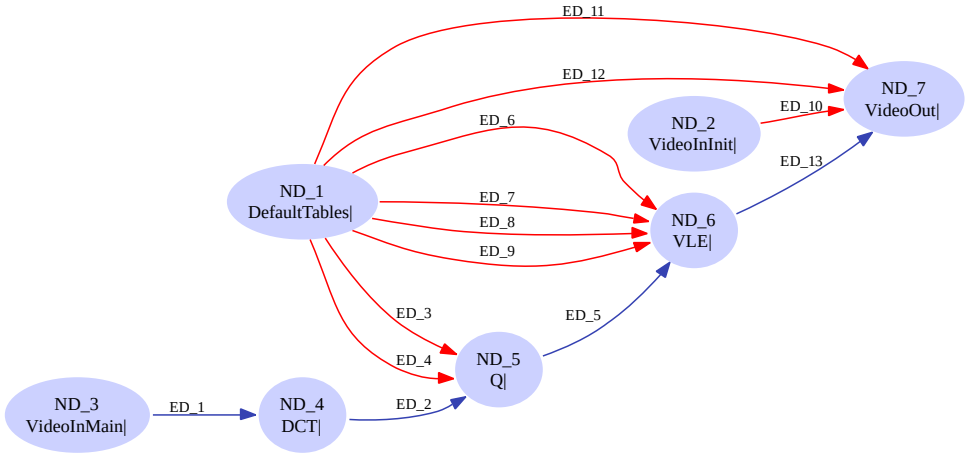

Figure 5.3: PRDG graph for the fully pipelined M-JPEG Application.

The third command starts the PANDA tool. It converts the PRDG into a KPN process network [44] [45]. For our example, the resultant KPN is shown in Figure 4.9, where nodes ND_1, ND_2, and ND_3 are merged in process $P1$. Also, this command invokes our ESPAM tool which generates the XPS project suite. The necessary options for this command are:

- **--input**: This option specifies the input PRDG XML file generated by DGPARSER.
- **-c**: This option describes a valid global schedule as a Matlab program for all the nodes in Figure 5.3. The reason that we use the initial Matlab code schedule is simply because it is available by default. Actually, we can use any other valid schedule specified as a Matlab program.
- **-ls -lms**: These options tell PANDA to select communication linearization model, since the communication is not always in order. For more details see [44] [45].
- **-RP**: This option makes sure that the number of data tokens which a producer process sends is the same as the number of tokens a consumer process needs. For more details see [44] [45].

- **-r**: This option optimizes the number of communication channels without decreasing the performance of the process network. It removes some channels which start from one and the same process and end to another process.

- **--mbxps**: This option invokes the ESPAM tool and tells ESPAM to generate an XPS project. This option has a parameter <$libXPS$>, which points to a library that stores the predefined platform components used to generate an XPS project. An XPS project consists of two parts. One part is generated at compile time, including the XMP/MHS/MSS files, program codes for each processor in the platform, and some custom IP cores. The other part is a library which consists of predefined components that are common for all projects, such as a clock IP core and a UCF file. We store this library in the CVS repository. The <$libXPS$> specifies the path to this library so that ESPAM can copy and use it during the generation of an XPS project suite. Currently, we use the following CVS repository path for this library:
.../compaan/pa/espam/libXPS

- **--clockgen**: This option tells ESPAM to generate some components used for debugging. We explain these debugging components in Section 5.4.2.

XPS Project Directory Hierarchy

After we run the three commands depicted in Figure 5.2, an XPS project is generated. Figure 5.4 shows the project directory hierarchy.

```
<PROJECT_ROOT>
|--- system.xmp
|--- system.mhs
|--- system.mss
|--- M_JPEG_5p.m
|--- loader.exe
|--- etc/
|---------- bitgen.ut
|---------- fast_runtime.opt
|--- data/
|---------- system.ucf
|--- code/
|---------- MemoryMap.h
|---------- aux_func.h
|---------- ND_4/
|---------------- ND_4.cpp
|---------------- default_link_script
|---------- ND_5/
|---------------- ND_5.cpp
|---------------- default_link_script
|---------- ND_6/
|---------------- ND_6.cpp
|---------------- default_link_script
|---------- ND_7/
|---------------- ND_7.cpp
|---------------- default_link_script
|---------- P1/
|---------------- P1.cpp
|---------------- default_link_script
|--- pcores/
|---------- fifo_if_ctrl_v1_00_a/
|---------- myCLKRST_v1_00_a/
|---------- clock_cycle_counter_v1_00_a/
|---------- counter_input_ctrl_v1_00_a/
|---------- buffers_v1_00_a/
|---------- opb_zbt_controller_v1_00_a/
```

Figure 5.4: Project directory structure.

The *system.xmp*, *system.mhs* and *system.mss* files are the corresponding XMP, MHS, and MSS files which have been introduced in section 3.4.1. The *system.mhs* file is described in detail in

Section 5.1.3. The *M_JPEG_5p.m* file stores the initial Matlab code shown in Figure 5.1. The *loader.exe* is a program used to download and run the bitstream file. We explain it in Section 5.4.1. Directory *etc* contains files *bitgen.ut* [46] and *fast_runtime.opt* [46] which store options for the Xilinx implementation tools. The directory *data* contains a file, namely *system.ucf* [37] which specifies implementation constraints such as timing, FPGA pin locations, FPGA resource specification, and IO standards.

Directory *code* stores the program code files for each processor in the platform. Each processor has a corresponding sub-directory, in which program source code files and a default linker script file are stored. We describe the program code files in Section 5.1.4 and the linker script file in Section 5.1.5.

In the top level of the *code* directory, there are two files, namely *MemoryMap.h* and *aux_func.h*. They are common for program codes of all processors. The *aux_func.h* file declares read and write primitives and the wrappers of all function calls in the initial Matlab code in lines 23, 27 and 30 - 36 in Figure 5.1. The *MemoryMap.h* file specifies the physical addresses of FIFOs connected to every processor. We explain this file in Section 5.1.6.

The *pcores* directory contains all predefined IP cores as well as IP cores generated by ESPAM. The *fifo_if_ctrl_v1_00_a* is the LMB FIFO controller described in Section 3.2.4. The *clock_cycle_counter_v1_00_a* and *counter_input_ctrl_v1_00_a* are two IP cores used for debugging. We explain them in Section 5.4.2. The *myCLKRST_v1_00_a* is a predefined IP core used for generating the system clock and reset. The *buffers_v1_00_a* and *opb_zbt_controller_v1_00_a* are also predefined IP cores used for connecting an external memory to a processor.

5.1.3 MHS File

The MHS file defines all hardware components used in a platform as well as the connections between these components. An automatically generated MHS file for our example platform is shown in Appendix B.

In code lines 1 - 16, input/output pins of the FPGA chip are specified, such as the Universal Asynchronous Receiver-Transmitter (UART) pins, clock and reset pins. In code lines 17 - 45, a bank of Zero Bus Turnaround (ZBT) SRAM memory is instantiated. This memory is connected via an OPB bus mb_opb in line 24 and its base address is set to *0xfa000000*. Every processor which is connected to bus mb_opb can access this ZBT memory by the physical address from *0xfa000000* to *0xfa3fffff*. In code lines 47 - 60, a global clock module is instantiated using the IP core *myCLKRST_v1_00_a*. It generates different clocks for different modules. In code lines 62 - 228, FSL FIFOs are instantiated corresponding to the 13 channels shown in Figure 4.9. By default, the data width is set to 32 bits and the depth is set to 512 for all FIFOs.

The rest of this file describes 5 processor sub-systems. Each sub-system is stand-alone, i.e., it has its own instruction/data memory and memory controller. Each processor sub-system corresponds to a process in Figure 4.9. For example, in code lines 230 - 314, all components of processor $P1$ which corresponds to process $P1$ in Figure 4.9 are specified. Processor $P1$ itself is described in lines 242 - 261. It has 10 output FIFO connections. The first 8 FIFOs are connected to FSL ports, see lines 247-254. The rest two are connected via a LMB bus in line 257 using the *fifo_ctrl_P1* component which is instantiated in lines 230 - 240. One dual-port

on-chip BRAM, which is in lines 299 - 304, is allocated for processor $P1$ with a default size of 16 Kbytes. The processor uses two controllers, i.e., *dlmb_cntlr_P1* and *ilmb_cntlr_P1* in lines 279 - 297 to access the BRAM.

5.1.4 Processor Program Code

In directory $code$ of an XPS project, each processor has a corresponding sub-directory, in which program code files and a default linker script file are stored.

Figure 5.5 depicts the program code for processor $P1$. This is a sequential program that describes the internal behavior of processor $P1$. In code lines 0 - 4, some basic definitions of the environment are included. Lines 8 - 19 declare local variables that are used later in the code. The real work is done in the body of the *for* loops in lines 22 - 67. In lines 23, 51, and 58, empty wrappers corresponding to function calls *P1_1_DefaultTables*, *P1_1_VideoInMain*, and *P1_1_VideoInInit* are placed, which have to implement the main computational tasks of processor $P1$. These empty wrappers are defined in file $aux\_func.h$. We explain how to import the implementation of these wrappers in Section 5.3.2.

The rest of the code in the body of the *for* loops controls the sequence of executions of the three wrappers above. Also, it controls from/to which ports the input/output arguments of these wrappers are read/written using a read/write primitives. For example, in line 26, argument *out_0ND_1* is written to the output port *ND_1_OP_1_ED_3* using the write primitive *writeFSL()*. The data size of argument *out_0ND_1* in 32-bit words is computed by the expression: $(sizeof(tED\_3) + (sizeof(tED\_3)\%4) + 3)/4$.

In line 66, the variable *counter_flag* is used for debugging. We explain this in Section 5.4.2.

5.1.5 Linker Script

In the program codes directory of each processor, there is a default linker script file [31]. The linker script consists of two parts. The first part defines a memory layout, i.e., specifies the start address and size of different memory regions. The second part specifies a location in a memory region for each section of an executable file by defining the start address of each section. If the address map of a processor occupy contiguous areas of memory, its default linker script need not to be changed. Else, we have to modify its default linker script according to [47].

A default linker script is shown in Figure 5.6. In lines 1 - 3, we use the $MEMORY$ command to define the memory layout. All parameters for a memory region come from the MHS file. For example, in line 2, we specify a memory region $INSTRUCTION\_DATA\_MEM$, of which the base address is set to $0x00000000$ and the length is set to 16 Kbytes. These parameters are based on lines 282, 285, 291 and 295 in Appendix B, which define parameter values for memory controllers $lmb\_bram\_if\_cntlr$ connected to processor $P1$.

In lines 10 - 80 of the default linker script file, we specify a location in this memory region for each section of an ELF executable file [48] by defining the start address of each section. For example, section $.text$ is set to the beginning of the memory block in lines 10 - 16 and section $.rodata$ is set to the following region in lines 18 - 25. The $.text$ section is always set to launch

```
0   #include "xparameters.h"
    #include "stdio.h"
    #include "stdlib.h"
    #include "aux_func.h"
    #include "MemoryMap.h"
5

    int main(){
      // Input Arguments
      // Output Arguments
10    tED_3 out_0ND_1;
      tED_4 out_1ND_1;
      tED_6 out_2ND_1;
      tED_7 out_3ND_1;
      tED_8 out_4ND_1;
15    tED_9 out_5ND_1;
      tED_11 out_6ND_1;
      tED_12 out_7ND_1;
      tED_10 out_0ND_2;
      tED_1 out_0ND_3;
20

      for ( int k =  ceil1(1) ; k <=  floor1(1 ) ; k += 1 ) {
        P1_l_DefaultTables(&out_0ND_1, &out_1ND_1, &out_2ND_1, &out_3ND_1, &out_4ND_1, &out_5ND_1, &out_6ND_1, &out_7ND_1);

25      // Variable: LuminanceQTable_1(k)
        writeFSL(ND_1_OP_1_ED_3, &out_0ND_1, (sizeof(tED_3)+(sizeof(tED_3)%4)+3)/4);

        // Variable: ChrominanceQTable_1(k)
        writeFSL(ND_1_OP_2_ED_4, &out_1ND_1, (sizeof(tED_4)+(sizeof(tED_4)%4)+3)/4);
30
        // Variable: LuminanceHuffTableDC_1(k)
        writeFSL(ND_1_OP_3_ED_6, &out_2ND_1, (sizeof(tED_6)+(sizeof(tED_6)%4)+3)/4);

        // Variable: ChrominanceHuffTableDC_1(k)
35      writeFSL(ND_1_OP_4_ED_7, &out_3ND_1, (sizeof(tED_7)+(sizeof(tED_7)%4)+3)/4);

        // Variable: LuminanceHuffTableAC_1(k)
        writeFSL(ND_1_OP_5_ED_8, &out_4ND_1, (sizeof(tED_8)+(sizeof(tED_8)%4)+3)/4);

40      // Variable: ChrominanceHuffTableAC_1(k)
        writeFSL(ND_1_OP_6_ED_9, &out_5ND_1, (sizeof(tED_9)+(sizeof(tED_9)%4)+3)/4);

        // Variable: LuminanceTablesInfo_1(k)
        writeFSL(ND_1_OP_7_ED_11, &out_6ND_1, (sizeof(tED_11)+(sizeof(tED_11)%4)+3)/4);
45
        // Variable: ChrominanceTablesInfo_1(k)
        writeFSL(ND_1_OP_8_ED_12, &out_7ND_1, (sizeof(tED_12)+(sizeof(tED_12)%4)+3)/4);
      } // for k

50    for ( int k =  ceil1(1) ; k <=  floor1(NumFrames ) ; k += 1 ) {
        P1_l_VideoInInit(&out_0ND_2);

        // Variable: HeaderInfo_1(k)
        writeLMB(ND_2_OP_1_ED_10, &out_0ND_2, (sizeof(tED_10)+(sizeof(tED_10)%4)+3)/4);
55
        for ( int j =  ceil1(1) ; j <=  floor1(VNumBlocks ) ; j += 1 ) {
          for ( int i =  ceil1(1) ; i <=  floor1(HNumBlocks ) ; i += 1 ) {
            P1_l_VideoInMain(&out_0ND_3);

60          // Variable: Block_1(k,j,i)
            writeLMB(ND_3_OP_1_ED_1, &out_0ND_3, (sizeof(tED_1)+(sizeof(tED_1)%4)+3)/4);

          } // for i
        } // for j
65    } // for k
      *counter_flag = 1;
    }
```

Figure 5.5: Program code for processor $P1$.

from the physical address $0x00000000$, which is required by the MicroBlaze core definition. The $.bss$ section is located at the end of the memory block in lines 67 - 79, where the heap must be set to the beginning of this section and the stack set to the end of this section.

5.1.6 Memory Map

The *MemoryMap.h* file in the top level of directory $code$ stores the physical address of each port for every processor. This is shown in Figure 5.7. Via these addresses, a processor can access a FIFO for a read/write operation using a read/write primitive. For example, lines 2 - 14 define

```
0  /* OUTPUT_FORMAT("elf32-microblaze", "", "") */
   MEMORY {
     INSTRUCTION_DATA_MEM: ORIGIN=0x00000000, LENGTH=0x3fff /*16k*/
   }

5  ENTRY(_start)
   SECTIONS {
     _TEXT_START_ADDR = 0x0;
     . = _TEXT_START_ADDR;

10   _ftext  =  .;
     .text : {
       *(.text)
       *(.text.*)
       *(.gnu.linkonce.t*)
15   } > INSTRUCTION_DATA_MEM
     _etext  =  .;

     . = ALIGN(4);
     _frodata = . ;
20   .rodata : {
       *(.rodata)
       *(.gnu.linkonce.r*)
       CONSTRUCTORS; /* Is this needed? */
     } > INSTRUCTION_DATA_MEM
25   _erodata = .;

   . = ALIGN(8);
     _ssrw = .;
     .sdata2 : {
30     *(.sdata2)
     } > INSTRUCTION_DATA_MEM

     . = ALIGN(8);
     _essrw = .;
35   _ssrw_size = _essrw - _ssrw;
     PROVIDE (_SDA2_BASE_ = _ssrw + (_ssrw_size/2));
     . = ALIGN(4);

     _fdata = .;
40     .data : {
       *(.data)
       *(.gnu.linkonce.d*)
       *(.eh_frame)
       CONSTRUCTORS;
45   } > INSTRUCTION_DATA_MEM
     _edata = . ;

     . = ALIGN(8);
      _ssro = .;
50   .sdata : {
       *(.sdata)
     } > INSTRUCTION_DATA_MEM

     . = ALIGN(4);
55   .sbss : {
       PROVIDE (__sbss_start = .);
       *(.sbss)
       PROVIDE (__sbss_end = .);
     } > INSTRUCTION_DATA_MEM
60
     . = ALIGN(8);
     _essro = .;
     _ssro_size = _essro - _ssro;
     PROVIDE (_SDA_BASE_ = _ssro + (_ssro_size / 2 ));
65   . = ALIGN(4);

     _fbss = .;
     .bss : {
       PROVIDE (__bss_start = .);
70     *(.bss)
       *(COMMON)
       . = ALIGN(4);
       PROVIDE (__bss_end = .);
       _heap = .;
75     _STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE:0x1000;
       . += _STACK_SIZE;
       . = ALIGN(8);
       _stack = .;
     } > INSTRUCTION_DATA_MEM
80   _end = .;
   }
```

Figure 5.6: Default linker script.

the physical addresses of all ports for processor $P1$. Since $P1$ has no input ports, only output addresses are defined in lines 5 - 14. These addresses are used to output data to other processors, see lines 26, 29, 32, 35, 38, 41, 44, 47, 54, and 61 in the program code of $P1$ in Figure 5.5.

There are two kinds of addresses, namely FSL bus address and LMB bus address. If a FIFO is connected to a FSL port, we assign a FSL address. The MicroBlaze processor that we use to build a multiprocessor platform has 8 FSL write ports and 8 FSL read ports. To every FSL port, only one FIFO can be connected. The addresses of these ports are numbered from 0 to 7 for both read and write ports. For example, in line 26 of Figure 5.5, a write FSL primitive is used for data communication via a FIFO. This means that the corresponding FIFO is connected to a FSL port (see line 247 in Appendix B) which address is $ND\_1\_OP\_1\_ED\_3$. In line 5 of Figure 5.7, this address is defined and its value is 0.

If a FIFO is connected to a LMB port, we assign LMB address. The MicroBlaze processor has one data LMB port. This port defines a data LMB bus to which many components can be connected. These components are accessed by the processor via addresses from the processor's data memory address space. For our multiprocessor platform, FIFO components can be connected to the LMB bus of each processor. The address space for FIFOs connected to a processor begins from address $0x08000000$, see line 237 in Appendix B. Each FIFO uses 8 bytes from the address space. For example, in lines 54 and 61 of Figure 5.5, write LMB primitives are used for data communication via FIFOs. This means that the corresponding FIFOs are connected to the LMB bus of processor $P1$, see lines 231 and 232 in Appendix B. These FIFOs are accessed via addresses $ND\_2\_OP\_1\_ED\_10$ and $ND\_3\_OP\_1\_ED\_1$ that are defined in lines 13 and 14 of Figure 5.7.

In summary, our ESPAM tool tries to connect FIFOs to the FSL ports of each processor, because FSL connection is faster than LMB connection. However, the number of FSL ports for each processor is limited to 8 FSL ports for read and 8 FSL ports for write. If extra FIFOs have to be connected to a processor, ESPAM uses LMB connection. For example, processor $P1$ has 10 FIFOs to be connected. We assign the first 8 to FSL ports , which are shown in lines 5 - 12 of Figure 5.7. The rest two are assigned to LMB bus address in lines 13 and 14.

The last two lines 57 and 58 of Figure 5.7 define addresses corresponding to components used for debugging We explain them in Section 5.4.2.

```
0   #ifndef __MEMORYMAP_H_
    #define __MEMORYMAP_H_
    //P1 FIFOs
    // InPut to FIFOs
    // OutPut to FIFOs
5   #define ND_1_OP_1_ED_3 0 //write to edgeED_3 address
    #define ND_1_OP_2_ED_4 1 //write to edgeED_4 address
    #define ND_1_OP_3_ED_6 2 //write to edgeED_6 address
    #define ND_1_OP_4_ED_7 3 //write to edgeED_7 address
    #define ND_1_OP_5_ED_8 4 //write to edgeED_8 address
10  #define ND_1_OP_6_ED_9 5 //write to edgeED_9 address
    #define ND_1_OP_7_ED_11 6 //write to edgeED_11 address
    #define ND_1_OP_8_ED_12 7 //write to edgeED_12 address
    #define ND_2_OP_1_ED_10 0x08000000 //write to edgeED_10 address
    #define ND_3_OP_1_ED_1 0x08000008 //write to edgeED_1 address
15

    //ND_4 FIFOs
    // InPut to FIFOs
    #define ND_4_IP_1_ED_1 0 //read from edgeED_1 address
20
    // OutPut to FIFOs
    #define ND_4_OP_1_ED_2 0 //write to edgeED_2 address

25  //ND_5 FIFOs
    // InPut to FIFOs
    #define ND_5_IP_2_ED_2 0 //read from edgeED_2 address
    #define ND_5_IP_3_ED_3 1 //read from edgeED_3 address
    #define ND_5_IP_4_ED_4 2 //read from edgeED_4 address
30
    // OutPut to FIFOs
    #define ND_5_OP_1_ED_5 0 //write to edgeED_5 address


35  //ND_6 FIFOs
    // InPut to FIFOs
    #define ND_6_IP_5_ED_5 0 //read from edgeED_5 address
    #define ND_6_IP_6_ED_6 1 //read from edgeED_6 address
    #define ND_6_IP_7_ED_7 2 //read from edgeED_7 address
40  #define ND_6_IP_8_ED_8 3 //read from edgeED_8 address
    #define ND_6_IP_9_ED_9 4 //read from edgeED_9 address

    // OutPut to FIFOs
#define ND_6_OP_1_ED_13 0 //write to edgeED_13 address
45

    //ND_7 FIFOs
    // InPut to FIFOs
    #define ND_7_IP_10_ED_10 0 //read from edgeED_10 address
50  #define ND_7_IP_11_ED_11 1 //read from edgeED_11 address
    #define ND_7_IP_12_ED_12 2 //read from edgeED_12 address
    #define ND_7_IP_13_ED_13 3 //read from edgeED_13 address

    // OutPut to FIFOs
55

    #define counter_flag (volatile int*)0x09000000
    #define counter_addr (volatile int*)0x0a000000
    #endif
```

Figure 5.7: Memory map file.

5.2 Importing the Project to XPS

After an XPS project is generated as described in Section 5.1.2, we have to import this project into XPS. To start XPS, we use the start menu of Windows: **start->Xilinx Platform Studio->Xilinx Platform Studio**. In the XPS tool, select the menu option: **File->Open Project**. Then in the new dialog box **Microprocessor Project files**, we select the XMP file $system.xmp$ by double clicking on it. Thereby, the project is loaded to XPS automatically. We can get a visual view of all the settings by clicking on the dialog box **Project->Add/Edit Cores...(Dialog)**. All the components, buses, ports, and parameters are listed separately in the tabs **Peripherals**, **Bus Connections**, **Ports**, and **Parameters**, respectively. More details of how to use the XPS tool can be found in [49].

5.3 Custom Modification

After we import the project in XPS, we still need to make some manual modifications on both the hardware description and the program code files. As discussed in Chapter 4, the size of the

on-chip memory is not sufficient for our five-processor system. We need to refine the memory allocation as well as we need to use the external on-board memory as a data memory. At the same time, we need to import the definition and implementation of all function calls used in program code files of each processor.

5.3.1 Hardware Modifications

We first describe the hardware modifications. The hardware specification is captured in the MHS file shown in Appendix B. We can modify manually each entry respecting the syntax rules of the MHS file as described in Section 3.4.1.

All modifications are related to the memory allocation, because there are a lot of limits for the on-chip memory allocation in XPS. The minimum size of a memory block which can be allocated is 2 Kbytes, and the size can be enlarged by power of 2, which means we can allocate memory per block with sizes 2K, 4K, 8K, 16K, 32K, etc. Moreover, since the use of memory can not be fully determined at compile time, it is hard to find an automated procedure to allocate the memory efficiently. A typical case is the memory allocation for the stack. Its size changes at runtime according to the execution flow of a program which in many case is data depended, i.e., unpredictable at compile time. Using worst-case scenario to estimate the minimum size of the stack is not a good option, because our total on-chip memory is only 176 Kbytes. Therefore, currently the memory allocation is more like an art than a science. Efficient memory allocation cannot be done without the skills and knowledge of the designer.

FIFO Size Adjustment

The first step in our manual modification procedure is to adjust the size of FIFOs. By default, we allocate 2048 bytes (512×32) for each FIFO, for example see lines 67 and 68 of Appendix B. The 512 is the data depth of a FIFO and the 32 is the data width of a FIFO. However, when we explore the initial M-JPEG code, we find out that the size of structures *THuffTablesAC*, *THuffTablesDC* and *TTablesInfo* is larger than 2048 bytes, all of which will be put into certain FIFOs. Therefore, the default size of the corresponding FIFOs is not sufficient. We need to enlarge the corresponding FIFO buffer sizes to 4096 bytes (1024×32). According to Table 4.7, we enlarge the size of FIFOs ED_6, ED_7, ED_8, ED_9, ED_11, and ED_12 to 4096 bytes. A sample modification of the size of FIFO ED_6 is shown in line 133 of Figure 5.8. In the same way, the rest of the FIFOs in the MHS file are modified. After these modifications, 38 Kbytes of the on-chip memory are allocated for FIFO buffers.

Memory Allocation

The second step is to refine the data and instruction memory of each process. By default, we allocate a 16 Kbytes BRAM for both the instruction and data memory. For example, lines 495 and 505 in Appendix B shows the default value for processor ND_6. However, this is not sufficient for our M-JPEG system. Because of the complexity of the function calls in the M-JPEG application, a larger stack is need for the context switch between two function calls.

```
127    BEGIN fsl_v20
        PARAMETER HW_VER = 2.00.a
        PARAMETER C_USE_CONTROL = 0
130     PORT FSL_Clk = sys_clk_s
        PARAMETER INSTANCE = sync_fifo_ED_6_ND_1_to_ND_6
        PARAMETER C_FSL_DWIDTH = 32
        PARAMETER C_FSL_DEPTH = 1024
        PARAMETER C_EXT_RESET_HIGH = 0
135     PARAMETER C_IMPL_STYLE = 1
        PORT SYS_Rst = sys_rst_s
        PARAMETER C_ASYNC_CLKS = 0
       END
```

Figure 5.8: Set the size of FIFO ED_6 to 4 Kbytes.

We refine the memory allocation following Table 4.6. For example, Figure 5.9 shows how we modify the MHS file to allocate sufficient memory for processor ND_6. We enlarge its memory size to 32 Kbytes as shown in lines 495 and 505 below.

```
490  BEGIN lmb_bram_if_cntlr
490   BUS_INTERFACE BRAM_PORT = conn_d_ND_6
      PARAMETER HW_VER = 1.00.b
      PARAMETER C_BASEADDR = 0x00000000
      PARAMETER INSTANCE = dlmb_cntlr_ND_6
      BUS_INTERFACE SLMB = dlmb_ND_6
495   PARAMETER C_HIGHADDR = 0x00007fff
      PARAMETER C_MASK = 0xff000000
     END
```

```
      BEGIN lmb_bram_if_cntlr
500    BUS_INTERFACE BRAM_PORT = conn_i_ND_6
       PARAMETER HW_VER = 1.00.b
       PARAMETER C_BASEADDR = 0x00000000
       PARAMETER INSTANCE = ilmb_cntlr_ND_6
       BUS_INTERFACE SLMB = ilmb_ND_6
505    PARAMETER C_HIGHADDR = 0x00007fff
       PARAMETER C_MASK = 0xff000000
      END
```

Figure 5.9: Processor ND_6 memory allocation.

Another example is Processor $P1$. After the memory allocation for FIFOs and other processors, there are 26 Kbytes on-chip BRAM left for processor $P1$. To efficiently use these 26 Kbytes on-chip BRAM, we use separate memory blocks for the instruction memory and data memory for processor $P1$. Figure 5.10 shows how this is done by modifying the MHS file. We allocate 8 Kbytes for the instruction memory in lines 302 - 303 and 16 Kbytes for the data memory in line 291 - 292. This is done according to the memory values for processor $P1$ listed in Table 4.6. Since two separate physical memory blocks are needed, we add a new memory module to the MHS file, which is shown in lines 315 - 319.

Now we have used 24 Kbytes of the 26 Kbytes on-chip memory left for processor $P1$. According to Table 4.6, processor $P1$ needs 64 Kbytes for its stack. This huge stack cannot be placed in the on-chip memory, because we have only 2Kbytes on-chip memory available. To resolve this problem, we put the stack and heap to an external memory on the prototyping board which is connected to the OPB bus, see lines 17 - 45 in Appendix B. Now processor $P1$ has three separate physical memory regions. They are mapped to different addresses. The instruction memory begins from address $0x00000000$ and its length is 8 Kbytes. The data memory begins from address $0x01000000$ and its length is 16 Kbytes. The external memory is mapped from address $0xfa000000$ to address $0xfa3fffff$. Because the address space is changed, we need to modify the default linker script of processor $P1$ accordingly. We discuss this in the next section.

```
289  BEGIN lmb_bram_if_cntlr
290   BUS_INTERFACE BRAM_PORT = conn_d_P1
      PARAMETER C_BASEADDR = 0x01000000
      PARAMETER C_HIGHADDR = 0x01003fff
      PARAMETER INSTANCE = dlmb_cntlr_P1
      BUS_INTERFACE SLMB = dlmb_P1
295   PARAMETER HW_VER = 1.00.b
      PARAMETER C_MASK = 0xff000000
     END

     BEGIN lmb_bram_if_cntlr
300   BUS_INTERFACE BRAM_PORT = conn_i_P1
      PARAMETER HW_VER = 1.00.b
      PARAMETER C_BASEADDR = 0x00000000
      PARAMETER C_HIGHADDR = 0x00001fff
      PARAMETER INSTANCE = ilmb_cntlr_P1
305   BUS_INTERFACE SLMB = ilmb_P1
      PARAMETER C_MASK = 0xff000000
     END

     BEGIN bram_block
310   PARAMETER HW_VER = 1.00.a
      PARAMETER INSTANCE = dlmb_bram_P1
      BUS_INTERFACE PORTA = conn_d_P1
     END

315  BEGIN bram_block
      PARAMETER HW_VER = 1.00.a
      PARAMETER INSTANCE = lmb_bram_P1
      BUS_INTERFACE PORTA = conn_i_P1
     END
```

Figure 5.10: P1 memory allocation.

Linker Script Adjustment

We modify the default linker script shown in Figure 5.6 for processor $P1$ according to the memory changes described in the previous section. The modifications are shown in bold font in Figure 5.11. In lines 2 - 6, we define the starting address and length of the three separate memory blocks as we described at the end of the previous section. Then, we specify the location of each section of the ELF executable file. In lines 13 - 19, the $.text$ section is always allocated in the instruction memory and begins from address $0x00000000$. All the other sections are allocated to the data memory continuously except the $.bss$ section. In lines 70-82, the $.bss$ section is placed to the external memory $ZBTMM$ in which the stack and heap are allocated.

5.3.2 Program Code Modifications

After the hardware modifications, we need to modify the program code for each processor.

The first step is to import the implementation of the function calls. In our automatic code generation, we create an empty wrapper for each function call presented in the initial M-JPEG code shown in Figure 5.1. As explained in Section 5.1.1, our tool chain does not deal with the actual implementation of these function calls. Therefore, we need to import manually the implementation of each function call.

In the **Applications** tab of XPS, five software projects can be found, one for each processor. We need to import files *ControlInit.cpp* and *Video_in.cpp* for processor $P1$, *DCT.cpp* for processor ND_4, $Q.cpp$ for processor ND_5, $VLE.cpp$ for processor ND_6 and $Video\_out.cpp$ for processor ND_7. We can import these files by double clicking on the **source** entry of a software project in the **Applications** tab.

After we import these files, the second step is to add the function declaration and replace each empty wrapper with a method call. This is done by modifying the program code of each processor. For example, the modified program code of processor $P1$ is shown in Figure 5.12. The bold lines in this code highlight the differences between this code and the initially generated code shown in Figure 5.5. In lines 18 - 19, we define two instances vin and $cinit$. Then in lines 23, 52, and 60, we replace the empty wrappers with the actual method calls. In the same way,

```
/* OUTPUT_FORMAT("elf32-microblaze", "", "") */

MEMORY {
   INSTM: ORIGIN=0x00000000, LENGTH=0xffff   /* 8k   */
   DATAM: ORIGIN=0x01000000, LENGTH=0x3fff   /* 16k  */
   ZBTMM: ORIGIN=0xfa000000, LENGTH=0x1ffff /* 128k */
}

ENTRY(_start)
SECTIONS {
  _TEXT_START_ADDR = 0x0;
  . = _TEXT_START_ADDR;

  _ftext  =  .;
  .text : {
    *(.text)
    *(.text.*)
    *(.gnu.linkonce.t*)
  } > INSTM
  _etext  =  .;

  . = ALIGN(4);
  _frodata = . ;
  .rodata : {
    *(.rodata)
    *(.gnu.linkonce.r*)
    CONSTRUCTORS; /* Is this needed? */
  } > DATAM
  _erodata = .;

  . = ALIGN(8);
  _ssrw = .;
  .sdata2 : {
    *(.sdata2)
  } > DATAM

  . = ALIGN(8);
  _essrw = .;
  _ssrw_size = _essrw - _ssrw;
  PROVIDE (_SDA2_BASE_ = _ssrw + (_ssrw_size/2));
  . = ALIGN(4);

  _fdata = .;
  .data : {
    *(.data)
    *(.gnu.linkonce.d*)
    *(.eh_frame)
    CONSTRUCTORS;
  } > DATAM
  _edata = . ;

  . = ALIGN(8);
   _ssro = .;
  .sdata : {
    *(.sdata)
  } > DATAM

  . = ALIGN(4);
  .sbss : {
    PROVIDE (__sbss_start = .);
    *(.sbss)
    PROVIDE (__sbss_end = .);
  } > DATAM

  . = ALIGN(8);
  _essro = .;
  _ssro_size = _essro - _ssro;
  PROVIDE (_SDA_BASE_ = _ssro + (_ssro_size / 2 ));
  . = ALIGN(4);

  _fbss = .;
  .bss : {
    PROVIDE (__bss_start = .);
    *(.bss)
    *(COMMON)
    . = ALIGN(4);
    PROVIDE (__bss_end = .);
    _heap = .;
    _STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE: 0xf000;
    . += _STACK_SIZE;
    . = ALIGN(8);
    _stack = .;
  } > ZBTMM
  _end = .;
}
```

Figure 5.11: Modified linker script of process P1 .

all the program codes of the other processors are modified. The complete modified project can be found in the CVS repository:
docs/students/KaiHuang_JiGu/experiment/projects/m-jpeg/M_JPEG_5p_zbt.tgz

5.4 XPS project Execution and Results

Once we have adjusted all components for our five-processor task-level pipelined M-JPEG system, we are ready to use XPS to generate the final bitstream file. The bitstream file is used to configure the FPGA chip such that it implements our M-JPEG system specified by the XPS project. We use the following commands to generate the bitstream file step by step. All these commands can be found in the menu option **Tools** in the XPS tool.

- **Generate Libraries**: This command invokes the library building tool $LibGen$ with the correct MSS file as input to create the Board Support Packet (BSP) which includes device drivers, libraries, STDIN/STDOUT configurations, and interrupt handlers associated with the design.
- **Compile Program Source**: This command invokes the cross compiler $mc - gcc$. This compiler generates several ELF executable files, one for each processor in the system, by compiling the program code of each processor. If $LibGen$ has not been executed, this command first invokes $LibGen$.

```
#include "xparameters.h"
#include "aux_func.h"
#include "MemoryMap.h"

int main() {
  // Input Arguments
  // Output Arguments
  tED_3 out_0ND_1;
  tED_4 out_1ND_1;
  tED_6 out_2ND_1;
  tED_7 out_3ND_1;
  tED_8 out_4ND_1;
  tED_9 out_5ND_1;
  tED_11 out_6ND_1;
  tED_12 out_7ND_1;
  tED_10 out_0ND_2;
  tED_1 out_0ND_3;

  Video_in vin (VNumBlocks, 2 * HNumBlocks);
  ControlInit cinit;

  for (int k = ceil1 (1); k <= floor1 (1); k += 1) {
    //P1_1_DefaultTables(&out_0ND_1, &out_1ND_1, &out_2ND_1, &out_3ND_1, &out_4ND_1, &out_5ND_1, &out_6ND_1, &out_7ND_1);
    cinit.main (out_0ND_1, out_1ND_1, out_2ND_1, out_3ND_1, out_4ND_1, out_5ND_1, out_6ND_1, out_7ND_1);

    // Variable: LuminanceQTable_1(k)
    writeFSL (ND_1_OP_1_ED_3, &out_0ND_1, (sizeof (tED_3) + (sizeof (tED_3) % 4) + 3) / 4);

    // Variable: ChrominanceQTable_1(k)
    writeFSL (ND_1_OP_2_ED_4, &out_1ND_1, (sizeof (tED_4) + (sizeof (tED_4) % 4) + 3) / 4);

    // Variable: LuminanceHuffTableDC_1(k)
    writeFSL (ND_1_OP_3_ED_6, &out_2ND_1, (sizeof (tED_6) + (sizeof (tED_6) % 4) + 3) / 4);

    // Variable: ChrominanceHuffTableDC_1(k)
    writeFSL (ND_1_OP_4_ED_7, &out_3ND_1, (sizeof (tED_7) + (sizeof (tED_7) % 4) + 3) / 4);

    // Variable: LuminanceHuffTableAC_1(k)
    writeFSL (ND_1_OP_5_ED_8, &out_4ND_1, (sizeof (tED_8) + (sizeof (tED_8) % 4) + 3) / 4);

    // Variable: ChrominanceHuffTableAC_1(k)
    writeFSL (ND_1_OP_6_ED_9, &out_5ND_1, (sizeof (tED_9) + (sizeof (tED_9) % 4) + 3) / 4);

    // Variable: LuminanceTablesInfo_1(k)
    writeFSL (ND_1_OP_7_ED_11, &out_6ND_1, (sizeof (tED_11) + (sizeof (tED_11) % 4) + 3) / 4);

    // Variable: ChrominanceTablesInfo_1(k)
    writeFSL (ND_1_OP_8_ED_12, &out_7ND_1, (sizeof (tED_12) + (sizeof (tED_12) % 4) + 3) / 4);
  }// for k

  for (int k = ceil1 (1); k <= floor1 (NumFrames); k += 1) {
    //P1_1_VideoInInit(&out_0ND_2);
    vin.init (out_0ND_2);

    // Variable: HeaderInfo_1(k)
    writeLMB (ND_2_OP_1_ED_10, &out_0ND_2, (sizeof (tED_10) + (sizeof (tED_10) % 4) + 3) / 4);

    for (int j = ceil1 (1); j <= floor1 (VNumBlocks); j += 1) {
      for (int i = ceil1 (1); i <= floor1 (HNumBlocks); i += 1) {
       //P1_1_VideoInMain(&out_0ND_3);
       vin.main (out_0ND_3);

       // Variable: Block_1(k,j,i)
       writeLMB (ND_3_OP_1_ED_1, &out_0ND_3, (sizeof (tED_1) + (sizeof (tED_1) % 4) + 3) / 4);

      } // for i
    } // for j
  } // for k
  *counter_flag = 1;
}
```

Figure 5.12: Modified program code for processor $P1$.

- **Generate Netlist**: This command invokes the platform building tool $PlatGen$ with the correct MHS file as input. It produces system netlist files in NGC format.

- **Generate Bitstream**: This command invokes the $xflow$ tool with the NGC netlist file as input. The $fast\_runtime.opt$ and $bitgen.ut$ files residing in the etc directory of the project are used to set some options of the $xflow$ tool. The $xflow$ tool generates the bitstream file for the FPGA. This file is located in directory *implementation/system.bit*.

- **Update Bitstream**: This invokes the tool $bitinit$. This is the stage where the hardware and the software flows come together. If above four commands have not been executed,

this command will first invoke them, respectively. At the end of this stage, the resultant $download.bit$ file is located in the $implementation$ directory and it contains FPGA configuration information regarding both the software and the hardware part of the design.

5.4.1 How to Get Results

The input and output capabilities of our prototyping board are limited. It has only a serial port controlled by a Universal Asynchronous Receiver-Transmitter (UART) component which is connected to processor ND_7. We create a hyper terminal in the host machine, which is connected to the serial port of the prototyping board, to get the output of our M-JPEG system. To create the hyper terminal, we use the start menu of Windows: **start->Accessories->Communication->Hyper Terminal**. The settings of the hyper terminal have to be the same as the settings of the UART component, see lines 530 - 533 in Appendix B. According to the UART settings in the MHS file, we have to set the hyper terminal as follows:

```
Bits per second: 9600
Data bits:       8
Parity:          None
Stop bits:       1
Flow control:    None
```

Now, we can run the system by executing the $loader.exe$ program in the Xygwin shell command line interface (Xygwin can be started from the menu option **Tools** in XPS). The $loader.exe$ program, which can be found on the top level of the generated project suite in Figure 5.4, is a program used to download and run the bitstream file. The sources of the $loader$ can be found in the CVS repository:
docs/students/KaiHuang_JiGu/experiment/accessary/loader.

The $loader$ program has the following options:

- **--help**: This option shows all the options and their explanation.
- **--file**: This option is followed by a filename that specifies the path of the bitstream file. The default value is *implementation/download.bit*.
- **--lclk**: This option is followed by an integer value that specifies a clock frequency in MHz. The default value is 100 MHz. This frequency is the system clock of the multiprocessor platform. Note: the parameter value of the UART component shown in line 529 of Appendix B must be equal to this $lclk$.
- **--mclk**: This option is followed by an integer that specifies the user clock frequency in MHz. The relation between the $mclk$ and the $lclk$ is: $f_{mclk} = 2 \times f_{lclk}$.

While the system runs, we can get messages on the hyper terminal. If not, see the conclusion in [49] for some hints.

5.4.2 IP Cores for Debugging

To evaluate the performance of our system, we count the number of clock cycles for compressing a certain number of image frames. We attach a hardware counter module to the processor to which the UART component is connected. Each processor in the processors network controls a 1-bit flag which is observed by the counter module. When a processor finishes its execution, it sets its flag to 1. Meanwhile, the counter module counts the cycles from the beginning and stops until all the flags are set to 1. When the counter module stops counting, it stores the cycles that our system has run.

Two custom IP cores, namely *clock_cycle_counter_v1_00_a* and *counter_input_ctrl_v1_00_a* are developed for evaluating the performance as described above, see the description of processor ND_7 in lines 550 - 561 and lines 624 - 633 in Appendix B. The *counter_input_ctrl_v1_00_a* core controls the 1-bit flag. An instance of this IP core is attached to each processor of the processors network via the processor's data memory bus. When a processor finishes its computation, it announces its completion by setting its flag to 1. The setting is acquired by writing a 1 to the absolute address *counter_flag* which is defined in line 57 in Figure 5.7. The *clock_cycle_counter_v1_00_a* core is the hardware counter module attached to the processor that has the output interface. It maintains a hardware counter, which stores the cycles from the beginning. The value of the counter can be accessed by the absolute address *counter_addr* which is defined in line 58 of Figure 5.7.

An example is given in Figure 5.13 to show the program code in which the debugging operations have been implemented. The processor ND_7 controls the output interface. Once it finishes its computation task, it sets its flag to 1 in line 53 to announce its finish. Then in line 54, we get the value of the hardware counter by reading the absolute address *counter_addr*.

5.5 Conclusion and Discussion

In this chapter, we described step by step how to design an embedded system from an application written in Matlab. We first use COMPAAN to generate a KPN specification of the application. Then we use ESPAM to generate the XPS project suite from the KPN specification. Finally, we use a commercial tool, namely Xilinx Platform Studio to generate the configuration bitstream file. Our tool chain can fully automate the conversion from Matlab code to KPN specification and the generation from KPN specification to the XPS project suite. However, for complex applications like M-JPEG, a lot of issues still need to be considered regarding to the automation.

- We need to import the declaration and implementation of each function call in the initial Matlab code to XPS, then replace the empty wrappers with method calls.
- We need to refine the FIFO size, memory size and stack size.
- We even need to redefine the instruction and data memory for some of the processors in our special fully pipelined 5-processor case.

```
#include "xparameters.h"
#include "stdio.h"
#include "stdlib.h"
#include "aux_func.h"
#include "MemoryMap.h"

int main (){
  // Input Arguments
  tED_10 in_0ND_7;
 tED_11 in_1ND_7;
 tED_12 in_2ND_7;
  tED_13 in_3ND_7;
  // Output Arguments
  double out_0ND_7;

 Video_out vout;

  for ( int k =  ceil1(1) ; k <=  floor1(NumFrames ) ; k += 1 ) {
    for ( int j =  ceil1(1) ; j <=  floor1(VNumBlocks ) ; j += 1 ) {
      for ( int i =  ceil1(1) ; i <=  floor1(HNumBlocks ) ; i += 1 ) {
        // BEGIN Broadcast In Order Read Tree
        if ( j-1 == 0) {
          if ( i-1 == 0) {
            readFSL(ND_7_IP_10_ED_10, &in_0ND_7, (sizeof(tED_10)+(sizeof(tED_10)%4)+3)/4);
          }
        }
        // END Broadcast In Order Read Tree
        // BEGIN Broadcast In Order Read Tree
        if ( k-1 == 0) {
          if ( j-1 == 0) {
            if ( i-1 == 0) {
              readFSL(ND_7_IP_11_ED_11, &in_1ND_7, (sizeof(tED_11)+(sizeof(tED_11)%4)+3)/4);
            }
          }
        }
        // END Broadcast In Order Read Tree
        // BEGIN Broadcast In Order Read Tree
        if ( k-1 == 0) {
          if ( j-1 == 0) {
            if ( i-1 == 0) {
              readFSL(ND_7_IP_12_ED_12, &in_2ND_7, (sizeof(tED_12)+(sizeof(tED_12)%4)+3)/4);
            }
          }
        }
        // END Broadcast In Order Read Tree
        readFSL(ND_7_IP_13_ED_13, &in_3ND_7, (sizeof(tED_13)+(sizeof(tED_13)%4)+3)/4);

        //P1_l_VideoOut(in_0ND_7, in_1ND_7, in_2ND_7, in_3ND_7, &out_0ND_7);
        vout.main(in_0ND_7, in_1ND_7, in_2ND_7, in_3ND_7);

      } // for i
    } // for j
  } // for k
  *counter_flag = 1;
  volatile int counter = *counter_addr;
  xil_printf("count = %d\n\r", counter);
}
```

Figure 5.13: Source file of process ND_7.

Importing the declaration and implementation of function calls is easy to automate. There is not a theoretical problem. What is difficult is the refinement of the FIFO size, memory size, and stack size. For example, the stack size cannot be precisely decided at compile time. It changes at runtime according to the execution flow of a program which in many cases is data depended. We can use a worst-case scenario to estimate the minimum value of the stack size. However this is not a good option because of the limited memory resource for the on-chip embedded system. See in our target prototyping board, the total on-chip memory is only 176 Kbytes. Efficient memory allocation cannot be done without the skills and knowledge of the designer. A more appropriate option is to find an accurate modeling in order to change the refinement from an art to a science.

Chapter 6

Summary and Conclusions

In this thesis we propose a system design methodology that allows efficient and effective mapping of a class of multimedia and signal processing applications onto multiprocessor platforms in a systematic and automated way. We have implemented this methodology in a system design flow by integrating two tools, namely COMPAAN and ESPAM. First, we use COMPAAN to convert a sequential application specified in Matlab to an equivalent KPN specification. Then, we use ESPAM to synthesize a multiprocessor platform and map the KPN specification onto it.

Essential to the design flow is the use of the Kahn Process Network (KPNs) model of computation. This model inherently expresses applications in terms of distributed computation, control, and memory. Because of this, the KPN model matches the emerging multiprocessor platforms where concurrent processing components are connected via distributed memory buffers. Therefore, the mapping of KPN specifications of applications onto multiprocessor platforms can be done in a transparent, systematic, and automated way.

In Chapter 1, we discussed that the performance demands of many of today and future applications can only be satisfied by embedded systems based on multiprocessor platforms. Such applications and platforms are so complex that the current traditional design methodologies can no longer deal with such complexity. This is because these methodologies are based on Register Transfer Level (RTL) system (application/platform) models created in programing and HDL languages by hand. The RTL level is too low for a complex system design because RTL models are very detailed, therefore creating them by hand for complex systems becomes error-prone and time-consuming. We can conclude that the use of a RTL-level specification as a starting point of multiprocessor system design methodologies is the bottleneck to achieve a reasonable design time and effort. Therefore, the system should be designed at a higher level of abstraction. Moving up from the detailed RTL-level specification to a more abstract specification opens a gap that we call Implementation Gap because currently we do not have mature methodologies and tools to go back from a high-level specification to an implementation. In response to that, we propose our techniques implemented in the ESPAM tool as a systematic and automated way to effectively and efficiently convert a high-level system specification to a RTL-level specification, thereby closing the Implementation Gap in a particular way.

In Chapter 2, we elaborated in detail on the techniques and tools integrated in our design flow. An important part of the flow is the definition of a component library that consists of generic

parameterized system components from which platform models can be composed. We describe these components at a high level of abstraction in order to conceal implementation details, thereby allowing easy construction of a variety of platform instances. These components are used by ESPAM to synthesize a multiprocessor platform from a KPN specification and to map this KPN onto the platform. A key feature in our design flow is that currently ESPAM performs a one-to-one mapping, i.e., each process in a KPN is mapped onto a processor component and each channel onto a FIFO component. As a consequence of this, the obtained multiprocessor platform has the same topology as the initial KPN, thereby exploiting the full parallelism offered by this KPN. We are aware that the one-to-one mapping is expensive in the sense that it requires a lot of computation and communication resources. Fortunately, the Moore's law predicts that chips in 2010 will count over 4 billion transistors, which means that the computation and communication resources available on a single chip will be of less concern. More important will be the time performance of a multiprocessor system and the ability to program such system in a systematic and automated way. In this respect, currently ESPAM delivers high performance multiprocessor systems using a one-to-one mapping as well as an automated programming of these systems is supported. ESPAM generates fully automatically program code files for each processor in the system as well as the synchronization between processors is realized by implementing read/write primitives based on the KPN communication semantic.

In Chapter 3, we presented how we targeted our methodology and tools to a particular implementation technology, namely Field Programmable Gate Arrays (FPGAs). We use a single Xilinx Virtex-II Pro FPGA chip to prototype multiprocessor systems generated by our ESPAM tool. To do this we use as a back-end of ESPAM, the commercial synthesis tool Xilinx Platform Studio (XPS). The output of ESPAM can be targeted to other implementation technologies by using other back-end synthesis tools. ESPAM is constructed such that it supports easy integration with such tools. Our current choice to target ESPAM to Xilinx FPGA chips is motivated by the following facts. First, prototyping alternative multiprocessor systems using Xilinx FPGA technology can be done relatively easy because of the flexible reconfigurable feature of the FPGA. Second, the Xilinx FPGA chips offer a lot of computation and communication resources needed for our multiprocessor systems, such as embedded soft processor cores (MicroBlaze and PicoBlaze) and FIFO based communication buffers. The number of processors and FIFOs that we can implement on a given FPGA is only limited by the size of the FPGA itself. Third, connecting FIFOs to the MicroBlaze soft processor core can be done very easy because MicroBlaze supports 16 dedicated ports which can be used for very fast connection to FIFOs. In general, not all existing processors have dedicated FIFO ports. Therefore, ESPAM can generate a controller that can be used as an interface to connect FIFOs to the general data bus of a processor. Such connection has the disadvantage that in many cases it is slower than a connection via dedicated FIFO ports. However, it is general enough to be applied to any processor because always a processor has a general data bus.

In Chapter 4, we use two real-life applications, namely a benchmark Matrix Multiplication application and an industrially-relevant Motion-JPEG encoder application, to verify the functionality of our system design flow and tools. Several experiments were conducted to evaluate the efficiency and usefulness of our tools. From the results we have obtained during the experiments, we can conclude the following. First, all the experiments show that the design time of a complex multiprocessor system can be reduced from months to hours by using our tools. The main reason is the great time performance of the COMPAAN compiler and our ESPAM tool.

COMPAAN/ESPAM take only a few seconds to derive a multiprocessor platform and to map an application specified as a KPN onto this multiprocessor platform. The rest of the design time (a few hours) is taken by the back-end commercial synthesis tool XPS to generate the final FPGA implementation. An interesting observation is that the time taken by XPS is strongly dependent on the complexity of the application to be implemented whereas the time taken by our tools is weakly dependent. This is supported by the fact that for a simple application like Matrix Multiplication and for a complex application like M-JPEG encoder, the time performance of COMPAAN/ESPAM stays in the range of a few seconds whereas the time performance of XPS varies from several minutes to a few hours. Second, for both applications we have generated multiprocessor systems with high performance. However, generating a multiprocessor platform for the Matrix Multiplication application is not very efficient from a point of view of resource utilization. This is because the Matrix Multiplication application can only be partitioned to a concurrent tasks with a very fine granularity. This very fine-grain tasks do not require powerful resources such as processors to be mapped on. When the complexity of applications increases, as in the case of the M-JPEG encoder, such applications can be partitioned to coarse-grain concurrent tasks which require powerful computational resources. Therefore, mapping such tasks onto processors is justified and our experiments proved that in this case it is efficient.

In Chapter 5, a step-by-step tutorial is given to demonstrate the complete design flow in action. First, we use COMPAAN to generate a KPN specification of an application. Then, we use ESPAM to generate an XPS project suite from the KPN specification. Finally, we use the XPS tool to generate a bitstream file to configure the FPGA. This chapter acts as a starting point for users as well as other people who will continue developing our system design flow.

As a general conclusion, we would like to mention that our ESPAM tool together with the COMPAAN compiler and the XPS tool can systematically and automatically implement multiprocessor systems for an application in a relatively short amount of time (a few hours). Thus, a very accurate exploration of the performance of alternative multiprocessor systems becomes feasible.

Appendix A

XMP File

A.1 XMP file Global field

UsePeriphRepos	Specifies whether to use any specified PeriphReposDir or not.
PeriphReposDir	PeriphReposDir.
MHS File	If theMHSFile does not exist in the project directory with same base name as project name, then XPS copies that MHS file into this name and location.
MSS File	MSS Files are created by XPS in the project directory with project name as base.
MVS File	MVS Files are created by XPS in the project directory with project name as base.
UseProjNav	Specifies whether the XPS project should use Project Navigator or Xflow for implementation tools.
AddToNPL	Specifies that if using ProjNav.
NPL File	The Project Navigator Project (NPL) file location is specified by NPL File field.
Architecture	The valid strings for target architecture families are virtex2, spartan2, spartan2e, virtex, virtexe, and virtex2p.
Device	Specifies the target device name.
Package	Specifies the device package.
SpeedGrade	Specifies the speed grade of the device.
HierMode	HierMode corresponds to the PlatGen option of whether to generate netlist in hierarchical mode or flat mode.
SynProj	Specifies which synthesis tool script file is to be generated.
InsertNoPads	Specifies the design hierarchy.
TopInst	Specifies the instance name give to this design in the top-level module.

A.2 XMP file Processor Instance Specific

Processor	This field should be used at least once before specifying any other processor specific fields.
Header	Specifies a source file for the current processor instance.
Source	Specifies a header file for the current processor instance.
CompilerFlow	Specifies how far the compiler flow should be run.
CompilerOptLevel	Specifies the compiler optimization level.
HardMul	Specifies to use hard multiplier available on Virtex2 and Virtex2P devices for MicroBlaze instance.
GlobPtrOpt	Specifies whether to use Global Pointer Optimization during compilation of program sources for the current processor.
DebugSym	Specifies whether to compile the program sources with debugging information or not.
SearchComp	This field specifies various directories (separated by space) for compiler search path (-B option)
SearchLibs	This field specifies various directories (separated by space) where the linker should look for libraries for the program sources (-L option).
SearchIncl	This filed specifies various directories (separated by space) where the compiler should look for various include files for the program sources (-I option).
PrepOpt	This field specifies various options (separated by space) to be passed on to the preprocessor (-Wp option).
AsmOpt	This field specifies various options (separated by space) to be passed to the assembler (-Wa option).
LinkOpt	This field specifies various options (separated by space) to be passed to linker (-Wl option).
ProgStart	This field specifies the program start address for your application software.
StackSize	This field specifies the stack size for your application software.
HeapSize	This field specifies the heap size for your application software.
LinkerScript	This field specifies the linker script file to be used for compiling program sources.
ProgCCFlags	This field specifies various options to be passed to the top-level compiler wrapper.

A.3 XMP file for Fully Pipelined M-JPEG System

```
#Please do not modify this file by hand
XmpVersion: 6.2
IntStyle: default
MHS File: system.mhs
MSS File: system.mss
NPL File: projnav/system.npl
Architecture: virtex2p
Device: xc2vp20
Package: ff896
SpeedGrade: -6
UseProjNav: 0
AddToNPL: 0
PNImportBitFile: projnav/system.bit
PNImportBmmFile: implementation/system_stub.bmm
UserCmd1:
UserCmd2:
SynProj: xst
ReloadPbde: 0
MainMhsEditor: 0
InsertNoPads: 0
HdlLang: VHDL
Simulator: mti
SimModel: BEHAVIORAL
SimXLib:
SimEdkLib:
Processor: mb_P1
BootLoop: 0
XmdStub: 0
Processor: mb_ND_4
BootLoop: 0
XmdStub: 0
Processor: mb_ND_5
BootLoop: 0
XmdStub: 0
Processor: mb_ND_6
BootLoop: 0
XmdStub: 0
Processor: mb_ND_7
BootLoop: 0
XmdStub: 0
SwProj: mb_P1
Processor: mb_P1
Executable: mb_P1/executable.elf
Source: code/P1/P1.cpp
Source: code/P1/Video_in.cpp
Source: code/P1/ControlInit.cpp
Header: code/aux_func.h
Header: code/MemoryMap.h
DefaultInit: EXECUTABLE
InitBram: 1
Active: 1
CompilerOptLevel: 2
GlobPtrOpt: 0
DebugSym: 0
ProfileFlag: 0
AsmOpt:
LinkOpt:
ProgStart:
StackSize: 64000
HeapSize:
LinkerScript: code/my_linker_script
ProgCCFlags:
SwProj: mb_ND_4
Processor: mb_ND_4
Executable: mb_ND_4/executable.elf
Source: code/ND_4/ND_4.cpp
Source: code/ND_4/DCT.cpp
Header: code/aux_func.h
Header: code/MemoryMap.h
DefaultInit: EXECUTABLE
InitBram: 1
Active: 1
CompilerOptLevel: 2
GlobPtrOpt: 0
DebugSym: 0
ProfileFlag: 0
AsmOpt:
LinkOpt:
ProgStart:
StackSize: 18000
HeapSize:
LinkerScript:
ProgCCFlags:
SwProj: mb_ND_5
Processor: mb_ND_5
Executable: mb_ND_5/executable.elf
Source: code/ND_5/ND_5.cpp
Source: code/ND_5/Q.cpp
Header: code/aux_func.h
Header: code/MemoryMap.h
DefaultInit: EXECUTABLE
InitBram: 1
Active: 1
CompilerOptLevel: 2
GlobPtrOpt: 0
DebugSym: 0
ProfileFlag: 0
AsmOpt:
LinkOpt:
ProgStart:
StackSize: 9000
HeapSize:
LinkerScript:
ProgCCFlags:
SwProj: mb_ND_6
Processor: mb_ND_6
Executable: mb_ND_6/executable.elf
Source: code/ND_6/ND_6.cpp
Source: code/ND_6/VLE.cpp
Header: code/aux_func.h
Header: code/MemoryMap.h
DefaultInit: EXECUTABLE
InitBram: 1
Active: 1
CompilerOptLevel: 2
GlobPtrOpt: 0
DebugSym: 0
ProfileFlag: 0
AsmOpt:
LinkOpt:
ProgStart:
StackSize: 19000
HeapSize:
LinkerScript:
ProgCCFlags:
SwProj: mb_ND_7
Processor: mb_ND_7
Executable: mb_ND_7/executable.elf
Source: code/ND_7/ND_7.cpp
Source: code/ND_7/Video_out.cpp
Header: code/aux_func.h
Header: code/MemoryMap.h
DefaultInit: EXECUTABLE
InitBram: 1
Active: 1
CompilerOptLevel: 2
GlobPtrOpt: 0
DebugSym: 0
ProfileFlag: 0
AsmOpt:
LinkOpt:
ProgStart:
StackSize: 20000
HeapSize:
LinkerScript:
ProgCCFlags:
```

Appendix B

MHS File for M-JPEG System

```
PARAMETER VERSION = 2.1.0
PORT RS232_Uart_1_RX = RS232_Uart_1_RX, DIR = I
PORT RS232_Uart_1_TX = RS232_Uart_1_TX, DIR = O
PORT lclk = lclk, DIR = I
PORT mclk = mclk, DIR = I
PORT mgt_clk = mgt_clk, DIR = I
PORT mgt_clkb = mgt_clkb, DIR = I
PORT clk_fb = clk_fb, VEC = [1:0], DIR = I
PORT clk_gen = clk_gen, VEC = [1:0], DIR = O
PORT lreset_l = sys_rst_s, DIR = IN
# PORT sys_clk = sys_clk_s, DIR = IN, SIGIS = CLK
# PORT system_reset = sys_rst_s, DIR = IN
# Begin:zbt
PORT zbt_ad = zbt_ad, VEC = [0:19], DIR = O
PORT zbt_dq = zbt_dq, VEC = [0:31], DIR = IO
PORT zbt_ctrl = zbt_ctrl, VEC = [0:11], DIR = O
BEGIN opb_zbt_controller
 PARAMETER INSTANCE = opb_zbt_controller_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xFA000000
 PARAMETER C_HIGHADDR = 0xFA3FFFFF
 PARAMETER C_EXTERNAL_DLL = 1
 PARAMETER C_ZBT_ADDR_SIZE = 20
 BUS_INTERFACE SOPB = mb_opb
 BUS_INTERFACE DESIGN_BUFF_PORT = buffer_zbt_ctrl
 PORT RA_O = zbt_ad
 PORT RC_O = zbt_ctrl
END
BEGIN buffers
 PARAMETER INSTANCE = buffers_0
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE BUFF_ZBT_CTRL_PORT = buffer_zbt_ctrl
 PORT rd0 = zbt_dq
 PORT O1 = net_vcc
 PORT O2 = net_vcc
 PORT O3 = net_vcc
 PORT O4 = net_vcc
 PORT O5 = net_vcc
 PORT T1 = net_vcc
 PORT T2 = net_vcc
 PORT T3 = net_vcc
 PORT T4 = net_vcc
 PORT T5 = net_vcc
END
# End: zbt

BEGIN myclkrst
 PARAMETER INSTANCE = myclkrst_1
 PARAMETER HW_VER = 1.00.a
 PARAMETER num_clock = 2
 PORT lreset_l = sys_rst_s
 PORT lclk = lclk
 PORT mclk = mclk
 PORT mgt_clk = mgt_clk
 PORT mgt_clkb = mgt_clkb
 PORT clk_fb = clk_fb
 PORT ctrl = net_gnd
 PORT lclk_gen = sys_clk_s
 PORT clk_gen = clk_gen
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = sys_clk_s
 PARAMETER INSTANCE = sync_fifo_ED_1_ND_3_to_ND_4
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_IMPL_STYLE = 1
 PORT SYS_Rst = sys_rst_s
 PARAMETER C_ASYNC_CLKS = 0
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = sys_clk_s
 PARAMETER INSTANCE = sync_fifo_ED_2_ND_4_to_ND_5
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_IMPL_STYLE = 1
 PORT SYS_Rst = sys_rst_s
 PARAMETER C_ASYNC_CLKS = 0
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = sys_clk_s
 PARAMETER INSTANCE = sync_fifo_ED_3_ND_1_to_ND_5
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_IMPL_STYLE = 1
 PORT SYS_Rst = sys_rst_s
 PARAMETER C_ASYNC_CLKS = 0
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = sys_clk_s
 PARAMETER INSTANCE = sync_fifo_ED_4_ND_1_to_ND_5
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_IMPL_STYLE = 1
 PORT SYS_Rst = sys_rst_s
 PARAMETER C_ASYNC_CLKS = 0
END
```

```
    BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
     PARAMETER C_USE_CONTROL = 0
     PORT FSL_Clk = sys_clk_s
     PARAMETER INSTANCE = sync_fifo_ED_5_ND_5_to_ND_6
     PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
     PARAMETER C_EXT_RESET_HIGH = 0
     PARAMETER C_IMPL_STYLE = 1
     PORT SYS_Rst = sys_rst_s
     PARAMETER C_ASYNC_CLKS = 0
 END

    BEGIN fsl_v20
     PARAMETER HW_VER = 2.00.a
     PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = sys_clk_s
     PARAMETER INSTANCE = sync_fifo_ED_6_ND_1_to_ND_6
     PARAMETER C_FSL_DWIDTH = 32
     PARAMETER C_FSL_DEPTH = 512
     PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_IMPL_STYLE = 1
     PORT SYS_Rst = sys_rst_s
     PARAMETER C_ASYNC_CLKS = 0
    END

 BEGIN fsl_v20
     PARAMETER HW_VER = 2.00.a
     PARAMETER C_USE_CONTROL = 0
     PORT FSL_Clk = sys_clk_s
     PARAMETER INSTANCE = sync_fifo_ED_7_ND_1_to_ND_6
 PARAMETER C_FSL_DWIDTH = 32
     PARAMETER C_FSL_DEPTH = 512
     PARAMETER C_EXT_RESET_HIGH = 0
     PARAMETER C_IMPL_STYLE = 1
     PORT SYS_Rst = sys_rst_s
 PARAMETER C_ASYNC_CLKS = 0
    END

    BEGIN fsl_v20
     PARAMETER HW_VER = 2.00.a
 PARAMETER C_USE_CONTROL = 0
     PORT FSL_Clk = sys_clk_s
     PARAMETER INSTANCE = sync_fifo_ED_8_ND_1_to_ND_6
     PARAMETER C_FSL_DWIDTH = 32
     PARAMETER C_FSL_DEPTH = 512
 PARAMETER C_EXT_RESET_HIGH = 0
     PARAMETER C_IMPL_STYLE = 1
     PORT SYS_Rst = sys_rst_s
     PARAMETER C_ASYNC_CLKS = 0
   END

 BEGIN fsl_v20
     PARAMETER HW_VER = 2.00.a
     PARAMETER C_USE_CONTROL = 0
     PORT FSL_Clk = sys_clk_s
     PARAMETER INSTANCE = sync_fifo_ED_9_ND_1_to_ND_6
 PARAMETER C_FSL_DWIDTH = 32
     PARAMETER C_FSL_DEPTH = 512
     PARAMETER C_EXT_RESET_HIGH = 0
     PARAMETER C_IMPL_STYLE = 1
     PORT SYS_Rst = sys_rst_s
 PARAMETER C_ASYNC_CLKS = 0
    END

    BEGIN fsl_v20
     PARAMETER HW_VER = 2.00.a
 PARAMETER C_USE_CONTROL = 0
     PORT FSL_Clk = sys_clk_s
     PARAMETER INSTANCE = sync_fifo_ED_10_ND_2_to_ND_7
     PARAMETER C_FSL_DWIDTH = 32
     PARAMETER C_FSL_DEPTH = 512
 PARAMETER C_EXT_RESET_HIGH = 0
     PARAMETER C_IMPL_STYLE = 1
     PORT SYS_Rst = sys_rst_s
     PARAMETER C_ASYNC_CLKS = 0
    END
 BEGIN fsl_v20
     PARAMETER HW_VER = 2.00.a
     PARAMETER C_USE_CONTROL = 0
     PORT FSL_Clk = sys_clk_s
 PARAMETER INSTANCE = sync_fifo_ED_11_ND_1_to_ND_7
     PARAMETER C_FSL_DWIDTH = 32
     PARAMETER C_FSL_DEPTH = 512
     PARAMETER C_EXT_RESET_HIGH = 0
     PARAMETER C_IMPL_STYLE = 1
 PORT SYS_Rst = sys_rst_s
     PARAMETER C_ASYNC_CLKS = 0
    END

    BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
     PARAMETER C_USE_CONTROL = 0
     PORT FSL_Clk = sys_clk_s
     PARAMETER INSTANCE = sync_fifo_ED_12_ND_1_to_ND_7
     PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
     PARAMETER C_EXT_RESET_HIGH = 0
     PARAMETER C_IMPL_STYLE = 1
     PORT SYS_Rst = sys_rst_s
     PARAMETER C_ASYNC_CLKS = 0
 END

    BEGIN fsl_v20
     PARAMETER HW_VER = 2.00.a
     PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = sys_clk_s
     PARAMETER INSTANCE = sync_fifo_ED_13_ND_6_to_ND_7
     PARAMETER C_FSL_DWIDTH = 32
     PARAMETER C_FSL_DEPTH = 512
     PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_IMPL_STYLE = 1
     PORT SYS_Rst = sys_rst_s
     PARAMETER C_ASYNC_CLKS = 0
    END

 BEGIN fifo_if_ctrl
     BUS_INTERFACE FIFO_WRITE_1 = sync_fifo_ED_10_ND_2_to_ND_7
     BUS_INTERFACE FIFO_WRITE_2 = sync_fifo_ED_1_ND_3_to_ND_4
     PARAMETER C_AB = 8
     BUS_INTERFACE SLMB = dlmb_P1
 PARAMETER INSTANCE = fifo_ctrl_P1
     PARAMETER C_FIFO_WRITE = 2
     PARAMETER C_BASEADDR = 0x08000000
     PARAMETER C_HIGHADDR = 0x080007f
     PARAMETER HW_VER = 1.00.a
 END

    BEGIN microblaze
     PARAMETER INSTANCE = mb_P1
     PARAMETER HW_VER = 2.10.a
 PORT CLK = sys_clk_s
     BUS_INTERFACE DOPB = mb_opb
     BUS_INTERFACE MFSL0 = sync_fifo_ED_3_ND_1_to_ND_5
     BUS_INTERFACE MFSL1 = sync_fifo_ED_4_ND_1_to_ND_5
     BUS_INTERFACE MFSL2 = sync_fifo_ED_6_ND_1_to_ND_6
 BUS_INTERFACE MFSL3 = sync_fifo_ED_7_ND_1_to_ND_6
     BUS_INTERFACE MFSL4 = sync_fifo_ED_8_ND_1_to_ND_6
     BUS_INTERFACE MFSL5 = sync_fifo_ED_9_ND_1_to_ND_6
     BUS_INTERFACE MFSL6 = sync_fifo_ED_11_ND_1_to_ND_7
     BUS_INTERFACE MFSL7 = sync_fifo_ED_12_ND_1_to_ND_7
 PARAMETER C_NUMBER_OF_PC_BRK = 1
     BUS_INTERFACE ILMB = ilmb_P1
     BUS_INTERFACE DLMB = dlmb_P1
     PARAMETER C_FSL_LINKS = 8
     PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
    END

    BEGIN lmb_v10
     PARAMETER HW_VER = 1.00.a
 PARAMETER INSTANCE = ilmb_P1
     PORT LMB_Clk = sys_clk_s
     PARAMETER C_EXT_RESET_HIGH = 0
     PORT SYS_Rst = sys_rst_s
    END

    BEGIN lmb_v10
     PARAMETER HW_VER = 1.00.a
     PARAMETER INSTANCE = dlmb_P1
     PORT LMB_Clk = sys_clk_s
 PARAMETER C_EXT_RESET_HIGH = 0
     PORT SYS_Rst = sys_rst_s
    END

    BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_d_P1
     PARAMETER HW_VER = 1.00.b
     PARAMETER C_BASEADDR = 0x00000000
     PARAMETER INSTANCE = dlmb_cntlr_P1
     BUS_INTERFACE SLMB = dlmb_P1
 PARAMETER C_HIGHADDR = 0x00003fff
     PARAMETER C_MASK = 0xff000000
    END

    BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_i_P1
     PARAMETER HW_VER = 1.00.b
     PARAMETER C_BASEADDR = 0x00000000
     PARAMETER INSTANCE = ilmb_cntlr_P1
     BUS_INTERFACE SLMB = ilmb_P1
 PARAMETER C_HIGHADDR = 0x00003fff
     PARAMETER C_MASK = 0xff000000
    END
```

```
BEGIN bram_block
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTB = conn_d_P1
 PARAMETER INSTANCE = lmb_bram_P1
 BUS_INTERFACE PORTA = conn_i_P1
END

BEGIN counter_input_ctrl
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x09000000
 PARAMETER INSTANCE = counter_input_ctrl_P1
 BUS_INTERFACE CounterFlag = P1_count_conn
 PORT LMB_Clk = sys_clk_s
 BUS_INTERFACE SLMB = dlmb_P1
 PARAMETER C_HIGHADDR = 0x09000003
END

BEGIN microblaze
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 PARAMETER HW_VER = 2.10.a
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 PARAMETER C_FSL_LINKS = 2
 PARAMETER INSTANCE = mb_ND_4
 BUS_INTERFACE DLMB = dlmb_ND_4
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE SFSL0 = sync_fifo_ED_1_ND_3_to_ND_4
 BUS_INTERFACE MFSL0 = sync_fifo_ED_2_ND_4_to_ND_5
 BUS_INTERFACE ILMB = ilmb_ND_4
 PORT CLK = sys_clk_s
END

BEGIN lmb_v10
 PARAMETER HW_VER = 1.00.a
 PARAMETER INSTANCE = ilmb_ND_4
 PORT LMB_Clk = sys_clk_s
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst_s
END

BEGIN lmb_v10
 PARAMETER HW_VER = 1.00.a
 PARAMETER INSTANCE = dlmb_ND_4
 PORT LMB_Clk = sys_clk_s
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst_s
END

BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_d_ND_4
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER INSTANCE = dlmb_cntlr_ND_4
 BUS_INTERFACE SLMB = dlmb_ND_4
 PARAMETER C_HIGHADDR = 0x00003fff
 PARAMETER C_MASK = 0xff000000
END

BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_i_ND_4
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER INSTANCE = ilmb_cntlr_ND_4
 BUS_INTERFACE SLMB = ilmb_ND_4
 PARAMETER C_HIGHADDR = 0x00003fff
 PARAMETER C_MASK = 0xff000000
END

BEGIN bram_block
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTB = conn_d_ND_4
 PARAMETER INSTANCE = lmb_bram_ND_4
 BUS_INTERFACE PORTA = conn_i_ND_4
END

BEGIN counter_input_ctrl
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x09000000
 PARAMETER INSTANCE = counter_input_ctrl_ND_4
 BUS_INTERFACE CounterFlag = ND_4_count_conn
 PORT LMB_Clk = sys_clk_s
 BUS_INTERFACE SLMB = dlmb_ND_4
 PARAMETER C_HIGHADDR = 0x09000003
END

BEGIN microblaze
 PARAMETER HW_VER = 2.10.a
 PORT CLK = sys_clk_s
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE SFSL0 = sync_fifo_ED_2_ND_4_to_ND_5
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 BUS_INTERFACE ILMB = ilmb_ND_5
 BUS_INTERFACE MFSL0 = sync_fifo_ED_5_ND_5_to_ND_6
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 BUS_INTERFACE SFSL1 = sync_fifo_ED_3_ND_1_to_ND_5
 PARAMETER C_FSL_LINKS = 4
 BUS_INTERFACE DLMB = dlmb_ND_5
 PARAMETER INSTANCE = mb_ND_5
 BUS_INTERFACE SFSL2 = sync_fifo_ED_4_ND_1_to_ND_5
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
END

BEGIN lmb_v10
 PARAMETER HW_VER = 1.00.a
 PARAMETER INSTANCE = ilmb_ND_5
 PORT LMB_Clk = sys_clk_s
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst_s
END

BEGIN lmb_v10
 PARAMETER HW_VER = 1.00.a
 PARAMETER INSTANCE = dlmb_ND_5
 PORT LMB_Clk = sys_clk_s
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst_s
END

BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_d_ND_5
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER INSTANCE = dlmb_cntlr_ND_5
 BUS_INTERFACE SLMB = dlmb_ND_5
 PARAMETER C_HIGHADDR = 0x00003fff
 PARAMETER C_MASK = 0xff000000
END

BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_i_ND_5
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER INSTANCE = ilmb_cntlr_ND_5
 BUS_INTERFACE SLMB = ilmb_ND_5
 PARAMETER C_HIGHADDR = 0x00003fff
 PARAMETER C_MASK = 0xff000000
END

BEGIN bram_block
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTB = conn_d_ND_5
 PARAMETER INSTANCE = lmb_bram_ND_5
 BUS_INTERFACE PORTA = conn_i_ND_5
END

BEGIN counter_input_ctrl
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x09000000
 PARAMETER INSTANCE = counter_input_ctrl_ND_5
 BUS_INTERFACE CounterFlag = ND_5_count_conn
 PORT LMB_Clk = sys_clk_s
 BUS_INTERFACE SLMB = dlmb_ND_5
 PARAMETER C_HIGHADDR = 0x09000003
END

BEGIN microblaze
 PARAMETER HW_VER = 2.10.a
 PORT CLK = sys_clk_s
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE SFSL0 = sync_fifo_ED_5_ND_5_to_ND_6
 BUS_INTERFACE MFSL0 = sync_fifo_ED_13_ND_6_to_ND_7
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 BUS_INTERFACE ILMB = ilmb_ND_6
 BUS_INTERFACE SFSL4 = sync_fifo_ED_9_ND_1_to_ND_6
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 BUS_INTERFACE SFSL1 = sync_fifo_ED_6_ND_1_to_ND_6
 PARAMETER C_FSL_LINKS = 6
 BUS_INTERFACE DLMB = dlmb_ND_6
 PARAMETER INSTANCE = mb_ND_6
 BUS_INTERFACE SFSL2 = sync_fifo_ED_7_ND_1_to_ND_6
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 BUS_INTERFACE SFSL3 = sync_fifo_ED_8_ND_1_to_ND_6
END

BEGIN lmb_v10
 PARAMETER HW_VER = 1.00.a
 PARAMETER INSTANCE = ilmb_ND_6
 PORT LMB_Clk = sys_clk_s
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst_s
END

BEGIN lmb_v10
 PARAMETER HW_VER = 1.00.a
 PARAMETER INSTANCE = dlmb_ND_6
 PORT LMB_Clk = sys_clk_s
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst_s
END
```

```
BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_d_ND_6
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER INSTANCE = dlmb_cntlr_ND_6
 BUS_INTERFACE SLMB = dlmb_ND_6
 PARAMETER C_HIGHADDR = 0x00003fff
 PARAMETER C_MASK = 0xff000000
END

BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_i_ND_6
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER INSTANCE = ilmb_cntlr_ND_6
 BUS_INTERFACE SLMB = ilmb_ND_6
 PARAMETER C_HIGHADDR = 0x00003fff
 PARAMETER C_MASK = 0xff000000
END

BEGIN bram_block
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTB = conn_d_ND_6
 PARAMETER INSTANCE = lmb_bram_ND_6
 BUS_INTERFACE PORTA = conn_i_ND_6
END

BEGIN counter_input_ctrl
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x09000000
 PARAMETER INSTANCE = counter_input_ctrl_ND_6
 BUS_INTERFACE CounterFlag = ND_6_count_conn
 PORT LMB_Clk = sys_clk_s
 BUS_INTERFACE SLMB = dlmb_ND_6
 PARAMETER C_HIGHADDR = 0x09000003
END

BEGIN opb_uartlite
 PARAMETER HW_VER = 1.00.b
 PARAMETER INSTANCE = RS232_Uart_ND_7
 PARAMETER C_CLK_FREQ = 100000000
 PARAMETER C_DATA_BITS = 8
 PARAMETER C_BAUDRATE = 9600
 PARAMETER C_ODD_PARITY = 0
 PARAMETER C_USE_PARITY = 0
 BUS_INTERFACE SOPB = mb_opb
 PARAMETER C_HIGHADDR = 0xfffe21ff
 PARAMETER C_BASEADDR = 0xfffe2100
 PORT OPB_Clk = sys_clk_s
 PORT RX = RS232_Uart_1_RX
 PORT TX = RS232_Uart_1_TX
END

BEGIN opb_v20
 PARAMETER HW_VER = 1.10.b
 PORT OPB_Clk = sys_clk_s
 PARAMETER INSTANCE = mb_opb
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst_s
END

BEGIN clock_cycle_counter
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE CounterFlag_3 = ND_5_count_conn
 BUS_INTERFACE CounterFlag_2 = ND_4_count_conn
 BUS_INTERFACE CounterFlag_5 = ND_7_count_conn
 PARAMETER C_BASEADDR = 0x0a000000
 PARAMETER INSTANCE = clock_cycle_counter_ND_7
 PORT LMB_Clk = sys_clk_s
 BUS_INTERFACE CounterFlag_1 = P1_count_conn
 BUS_INTERFACE SLMB = dlmb_ND_7
 BUS_INTERFACE CounterFlag_4 = ND_6_count_conn
 PARAMETER C_HIGHADDR = 0x0a000003
END

BEGIN microblaze
 PARAMETER HW_VER = 2.10.a
 PORT CLK = sys_clk_s
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE SFSL0 = sync_fifo_ED_10_ND_2_to_ND_7
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 BUS_INTERFACE ILMB = ilmb_ND_7
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 BUS_INTERFACE SFSL1 = sync_fifo_ED_11_ND_1_to_ND_7
  PARAMETER C_FSL_LINKS = 5
 BUS_INTERFACE DLMB = dlmb_ND_7
 PARAMETER INSTANCE = mb_ND_7
 BUS_INTERFACE SFSL2 = sync_fifo_ED_12_ND_1_to_ND_7
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 BUS_INTERFACE SFSL3 = sync_fifo_ED_13_ND_6_to_ND_7
END

BEGIN lmb_v10
 PARAMETER HW_VER = 1.00.a
 PARAMETER INSTANCE = ilmb_ND_7
 PORT LMB_Clk = sys_clk_s
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst_s
END

BEGIN lmb_v10
 PARAMETER HW_VER = 1.00.a
 PARAMETER INSTANCE = dlmb_ND_7
 PORT LMB_Clk = sys_clk_s
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst_s
EN

BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_d_ND_7
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER INSTANCE = dlmb_cntlr_ND_7
 BUS_INTERFACE SLMB = dlmb_ND_7
 PARAMETER C_HIGHADDR = 0x00003fff
 PARAMETER C_MASK = 0xff000000
END

BEGIN lmb_bram_if_cntlr
 BUS_INTERFACE BRAM_PORT = conn_i_ND_7
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER INSTANCE = ilmb_cntlr_ND_7
 BUS_INTERFACE SLMB = ilmb_ND_7
 PARAMETER C_HIGHADDR = 0x00003fff
 PARAMETER C_MASK = 0xff000000
END

BEGIN bram_block
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTB = conn_d_ND_7
 PARAMETER INSTANCE = lmb_bram_ND_7
 BUS_INTERFACE PORTA = conn_i_ND_7
END

BEGIN counter_input_ctrl
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x09000000
 PARAMETER INSTANCE = counter_input_ctrl_ND_7
 BUS_INTERFACE CounterFlag = ND_7_count_conn
 PORT LMB_Clk = sys_clk_s
 BUS_INTERFACE SLMB = dlmb_ND_7
 PARAMETER C_HIGHADDR = 0x09000003
END
```

Bibliography

[1] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*. 2001.

[2] http://www.picochip.com.

[3] http://www.xilinx.com.

[4] Paul Stravers and Jan Hoogerbrugge. Homogeneous multiprocessoring and the future of silicon design paradigms. In *Proceedings of the Int. Symposium on VLSI Technology, Systems, and Applications*, April 2001.

[5] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, USA, May 3-5 2000.

[6] Edward Lee and Alberto Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[7] E.A. Lee et al. PtolemyII: Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California at Berkeley, 1999. UCB/ERL M99/40.

[8] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[9] Alexandru Turjan and Bart Kienhuis. Storage Management in Process Networks using the Lexicographically Maximal Preimage. In *Proc. of the IEEE 14th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'03)*, The Hague, The Netherlands, January 24-26 2003.

[10] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A Technique to Determine Interprocess Communication in the Polyhedral Model. In *Proc. Int. Workshop on Compilers for Parallel Computers (CPC'03)*, Amsterdam, The Netherlands, January 8-10 2003.

[11] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, September 1-3 2003.

[12] Song Peng, David Fang, John Teifel, and Rajit Manohar. Automated Synthesis for Asynchronous FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International symposium on Field Programmable Logic and Applications*, Monterey, California, USA, February 2005.

[13] Andre Nieuwland, Jeffrey Kang, O. P. Gangwal, R. Sethuraman, N. Busa, K Goosens, R. P. Llopis, and P. Lippens. *C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems*. Kluwer Academic Publishers, 2002.

[14] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Synthesis of application specific multiprocessor architectures for process networks. In *Proc. 17th International Conference on VLSI Design (VLSI-2004)*, Mumbai, India, January 2004.

[15] Rolf Enzler, Marco Platzner, Christian Plessl, Lothar Thiele, and Gerhard Troster. Reconfigurable pprocessors for Handhelds and Wearables: Application Analysis. In *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications III (Proceedings of ITCom'01)*, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, 2001.

[16] Kees Goossens, John Dielissen, Jef van Meerbergen, Peter. Poplavko, Andrei. Radulescu, Edwin Rijpkema, Erwin. P. Waterlander, and Paul. Wielage. Guaranteeing the Quality Of Services in Networks On Chip. In *Networks on Chip*, pages 61–82. Kluwer Academic Publishers, 2003.

[17] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, and Ed Deprettere. System Level Design with SPADE: an M-JPEG Case Study. In *Proc. Int. Conference on Computer Aided Design (ICCAD'01)*, pages 31–38, San Jose CA, USA, November 4-8 2001.

[18] Joe Coffland and Andy Pimentel. A Software Framework for Efficient System-level Pe rformance Evaluation of Embedded Systems. In *Proc. of the 18th ACM Symposium on Applied Computing, Embedded Systems track*, pages 666–671, Melbourne, Florida, USA, March 2003.

[19] Erwin de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. 15th Int. Symposium on System Synthesis (ISSS'2002)*, pages 68–73, Kyoto, Japan, October 2-4 2002.

[20] F. Balarin, E. Sentovich, M Chiodo, P. Giusto, H. Hsieh, B Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS approach*. Kluwer Academic Publishers, 1997.

[21] H. Muller. *Simulating computer architectures*. PhD thesis, Dept. of Computer Science, Univ. of Amsterdam, 1993.

[22] E.A. de Kock et al. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5-9 2000.

[23] "http://www.criticalblue.com". CriticalBlue Limited.

[24] "http://www.tensilica.com". Tensilica Inc.

[25] http://www.celoxica.com/products/agility/default.asp. Celoxica, Inc.

[26] Shawn McCloud. Algorithmic c synthesis optimizes esl design flows. *Xcell Journal*, August 2004.

[27] http://www.accelchip.com/.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[29] R. Sethi, A.V. Aho, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[30] Platform studio user guide: Xilinx, inc. http://www.xilinx.com/ise/embedded/edk6_2docs/platform_studio_ug.pdf.

[31] Microblaze software reference guide: Xilinx, inc. http://www.xilinx.com/ipcenter/processor_central/microblaze/doc/swref.pdf.

[32] Embedded system tools guide: Xilinx, inc. http://www.xilinx.com/ise/embedded/edk6_2docs/est_guide.pdf.

[33] http://www.alpha-data.com/adm-xrc-ii.html. Alpha Data Parallel Systems, Ltd.

[34] Local memory bus (lmb) v1.0, xilinx, inc. http://www.xilinx.com/ise/embedded/edk_docs.htm.

[35] 64-bit on-chip peripheral bus, architectural specifications, version 2.0. http://www.xilinx.com/ise/embedded/edk_docs.htm.

[36] Fast simplex link (fsl) bus (v2.00a), xilinx, inc. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/FSL_V20.pdf.

[37] User constraint file: Xilinx, inc. toolbox.xilinx.com/docsan/xilinx7/books/docs/sim/sim.pdf.

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[39] Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Algorithmic Transformation Techniques for Efficient E xploration of Alternative Application Instances. In *Proc. 10th International Symposium on Hardware/Software Codesi gn (CODES'02)*, pages 7–12, Estes Park, Colorado, USA, May 6-8 2002.

[40] Todor Stefanov. *Converting Weakly Dynamic Programs to Equivalent Process Network Specifications*. PhD dissertation, Universiteit Leiden., 2004.

[41] Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards; Algorithms and Architectures*. Kluwer Academic Publishers, 1995.

[42] W.B. Pennebacker and J.L. Mitchel. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, 1993.

[43] Bart Kienhuis. MatParser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, 2000. UCB/ERL M00/9.

[44] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Translating Affine Nested-loop Programs to Process Networks. In *Proc. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'04)*, Washington D.C., USA, September 23-25 2004.

[45] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. The Compaan Communication Model Selection. In *Proc. of the IEEE 15th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'04)*, Galveston, Texas, USA, September 27-29 2004.

[46] http://toolbox.xilinx.com/docsan/xilinx7/de/dev/xflow.pdf.

[47] http://www.gnu.org/software/binutils/manual/ld-2.9.1/.

[48] Elf: Executable and linkable format. ftp://ftp.intel.com/pub/tis, 1998.

[49] Ji Gu Kai Huang. Mapping process networks onto microblaze based multiprocessor platforms : Research project report. Technical report, LIACS, the Netherlands, 2005.