Embedded System-Level Platform Synthesis and Application Mapping for Heterogeneous and Hierarchical Multiprocessor Systems

MASTER'S THESIS

by

Wei Zhong
wzhong@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

First of all, I would like to express my gratitude to Prof. Ed Deprettere who gives me the opportunity to do my Master's research at the Leiden Embedded Research Center of the Leiden Institute of Advanced Computer Science (LIACS) - Leiden University.

A very special thanks go to my supervisor, Todor Stefanov, for his guidance and support throughout my Master's research and during the completion of my Master thesis. He was always patient to help me to solve difficult problems during my Master's research. Without his assistance, motivation, and encouragement I could not successfully finish my Master thesis.

I would also acknowledge Hristo Nikolov for the techical support he provided during my research. I would also like to thank all the people in my group for the help and advice they give me during my Master's research.

Wei Zhong
Leiden, The Netherlands
May 16, 2006

Chapter 1

Introduction

Nowadays, modern embedded applications are becoming complex. Such complex embedded applications lead to a single processor embedded system architecture can no longer meet the performance requirements of these applications. Therefore, in order to meet the performance requirements of the complex applications, the emerging embedded system platforms are increasingly becoming multiprocessor architectures. Fortunately, 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]. Thus, the modern embedded System-on-Chip platforms have enough resources to support to map the modern complex applications onto multiprocessor architectures.

Because of the fact which has been discussed above, several challenges emerge. The first challenge is how to specify an application. The suitable specification format of applications which makes the mapping of these applications onto multiprocessor architectures easy is parallel model of computation. But at present, applications that need to execute on embedded system architectures are typically specified using a sequential model of computation, such as sequential programs written in C or Matlab. What is needed is a methodology or tool that can exploit inherent parallelism available in the applications and convert the sequential specifications into parallel specifications.

The second challenge is how to design multiprocessor embedded systems. There are several issues in this challenge. The first issue is most of the current design methodologies and tools are based on Register Transfer Level (RTL) and most of the designers create such level by hand. Because complexity of multiprocessor embedded system architectures, the RTL level is too low to design such system and these methodologies for creating multiprocessor embedded system architectures are error-prone and time consuming. Therefore, a methodology and techniques which can systematically and automatically design multiprocessor embedded systems are needed. The second issue is the modern application always include several processes. If we map the processes of an application onto the homogeneous architecture which means we map the processes of an application onto the same type of components, maybe some of the processes can not meet the performance requirements. We need to map the processes onto the suitable components which are the different types. Therefore, in order to meet the performance requirements of the processes of an application, an embedded system should be heterogeneous architecture. The third issue is some of the processes of an application maybe very complex. If we map a complex process onto a single component, the process may not meet the required performance. In such case, we need to map the complex process onto several components in order to meet the required performance. These several components form a sub-network on an embedded system platform. Therefore we call the system hierarchical architecture. Therefore, an embedded system also should be hierarchical architecture.

The third challenge is the applications which are mapped onto embedded system platforms always need to communicate with the outside world. The challenge is how to make an efficient interface of an embedded system with the outside world.

This thesis focuses on the second, third challenges discussed above. The efficient solutions to these two challenges are presented. We propose a methodology implemented in a tool called ESPAM for systematic and automated multiprocessor embedded system design. Also, we prove that it is possible to implement an embedded system as heterogeneous and hierarchical architecture systematically and automatically using the ESPAM technology. We implement an efficient interface of an embedded system with the outside world.

In Section 1.1, detailed problem description is given. In Section 1.2, the methodology and techniques which are used to solve the problems described in Section 1.1 are presented. In Section 1.3, related work is discussed. The contributions of this thesis are stated in Section 1.4. Finally, in Section 1.5, the organization of this thesis are described.

1.1 Problem Description

Due to the complexity of modern applications, such as high throughput multimedia, imaging and digital signal processing which usually include complicated algorithms, a single processor embedded system architecture on an embedded system platform is inadequate. In order to meet the required performance for such complex applications, multiprocessor embedded system architectures have to be implemented on embedded system platforms. Therefore, exploiting parallelism available in such applications is important for current embedded system design. However, most of the applications are usually specified using the sequential model of computation, such as sequential programs written in C or Matlab. The sequential model of computation makes an application be easy to reason about a program, as only a single memory and a single thread of control need to be considered. But such sequential model of computation can not exploit the internal parallelism which is available in an application. This means mapping such application onto a multiprocessor embedded system architecture is difficult because the way the application is specified does not match the way the multiprocessor embedded system architecture operates. Thus, the suitable specification format of the applications which makes the mapping of the applications onto the multiprocessor embedded system architectures easy is the parallel model of computation.

Currently, the task of mapping the complex applications which are specified in sequential model of computation onto the multiprocessor embedded system platforms is usually done by hand. This means this mapping task depends much on the expertise of the designers and it is error-prone and time consuming. Therefore, a methodology and tool that can exploit inherent parallelism available in the complex applications and convert the sequential specifications into parallel specifications is needed. For example, the COMPAAN tool [2] can automatically transforms an application which is specified in sequential model of computation into the abstract concurrent model which consists of several concurrent tasks making the task-level parallelism available in an application explicit.

Now another problem emerges which is how to efficiently and effectively map the concurrent model of the applications onto the multiprocessor embedded system platforms in a systematic and automated way. In the realm of modern embedded system, most of the design and implementation methodology are still based on Register Transfer Level (RTL) platform/application descriptions which are created manually, such as very high speed integrated circuit hardware description language (VHDL) and C language. Such methodologies were effective in the past. Due to the complexity of the modern applications and platforms which are used in many of today's new system designs, the traditional design methodology is inadequate now. Creating such RTL descriptions of the complex multiprocessor platforms is 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. Furthermore, using traditional logic simulation to verify a large design represented in RTL is computationally expensive and extremely slow. From what have been discussed above, we can conclude that using the RTL system specification as a starting point of multiprocessor embedded system design methodology is the bottleneck. 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, we believe that a system should be specified at a higher level of abstraction called System-level. However, the embedded system design methodology which moves 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 an 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 today's embedded systems forces us to move to higher levels of abstraction when designing a embedded system, but currently we do not have mature methodologies, techniques, and tools to go back from the high-level specification to an implementation. Therefore, the *Implementation Gap* has to be closed by devising a systematic and automated way to convert effectively and efficiently a System-level specification to a RTL-level specification.

From what have been discussed above, it is clear that in order to map a complex application onto a multiprocessor embedded system platform, the application has to be transformed into an abstract concurrent model which consists of several concurrent processes. At present, multiprocessor embedded systems as homogeneous architectures can no longer meet the applications' requirements. An embedded system as homogeneous architecture means all of the concurrent processes of an application are executed by the same type of components on an embedded system platform. For example, all the processes of an application are executed by the same type of processor cores. The problem is that different types of processes are suitable for being executed by different types of components on an embedded system platform. For example, it is better to use the types of processor cores which are good at floating point computation to execute the processes which contain the floating point computation. And the other example is that it is better to use the dedicated hardware IP cores to execute the most complicated processes of the applications in order to reach the good performance of execution time. Therefore, an embedded system as heterogeneous architecture has to be implemented in order to meet the requirement performances of various applications. The problem is how to implement an embedded system as heterogeneous architecture systematically and automatically. What's more, an application always consists of several processes and some of the processes of an application maybe very complex. If we map a complex process onto a single component, the process may not meet the required performance. In such case, we need to map the complex process onto several components in order to meet the required performance. These several components form a sub-network on an embedded system platform. Therefore we call the system hierarchical architecture. Thus, such an embedded system as hierarchical architecture also has to be implemented. The problem is how to implement an embedded system as hierarchical architecture systematically and automatically.

The applications of modern embedded systems in the realm of high throughput multimedia, imaging, and digital signal processing, always need to exchange the data with the outside world. Thus modern embedded systems need an interface of an embedded system with the outside world. If the interface of an embedded system with the outside world is not efficient, that will intensely restrict embedded systems to reach the high performances. Due to this reason, an efficient interface of an embedded system with the outside world must be implemented to let the applications which are mapped onto the embedded system platforms can efficiently communicate with the outside world. The problem is how to construct an efficient interface of an embedded system with the outside world.

1.2 Solution Approach

Based on the problems which have been described above, the general description of the solution approaches for these problems is given in this section.

1.2.1 Closing the Implementation Gap

First in order to successfully close the *Implementation Gap* between the *System-level* specification of multiprocessor systems and the *RTL-level* specification of multiprocessor systems, we have developed a tool called ESPAM (Embedded System-level Platform synthesis and Application Mapping). This tool can systematically and automatically convert the *System-level* specification to the *RTL-level* specification. ESPAM allows the designers to specify a multiprocessor embedded system at a high level of abstraction (*System-level*), then it refines such specification and systematically and automatically convert this specification to a *RTL-level* specification. Figure 1.1 shows our system design flow which includes the ESPAM tool.

In Figure 1.1, we see that there are three levels of specifications in our system design flow. They are *System-level* specification, *RTL-level* specification and *Gate-level* specification.

The *System-level* specification consists of three parts which are *Platform Specification*, *Application Specification* and *Mapping Specification*. *Platform Specification* specifies the topology of a platform using our system level platform model which includes generic parameterized sys-

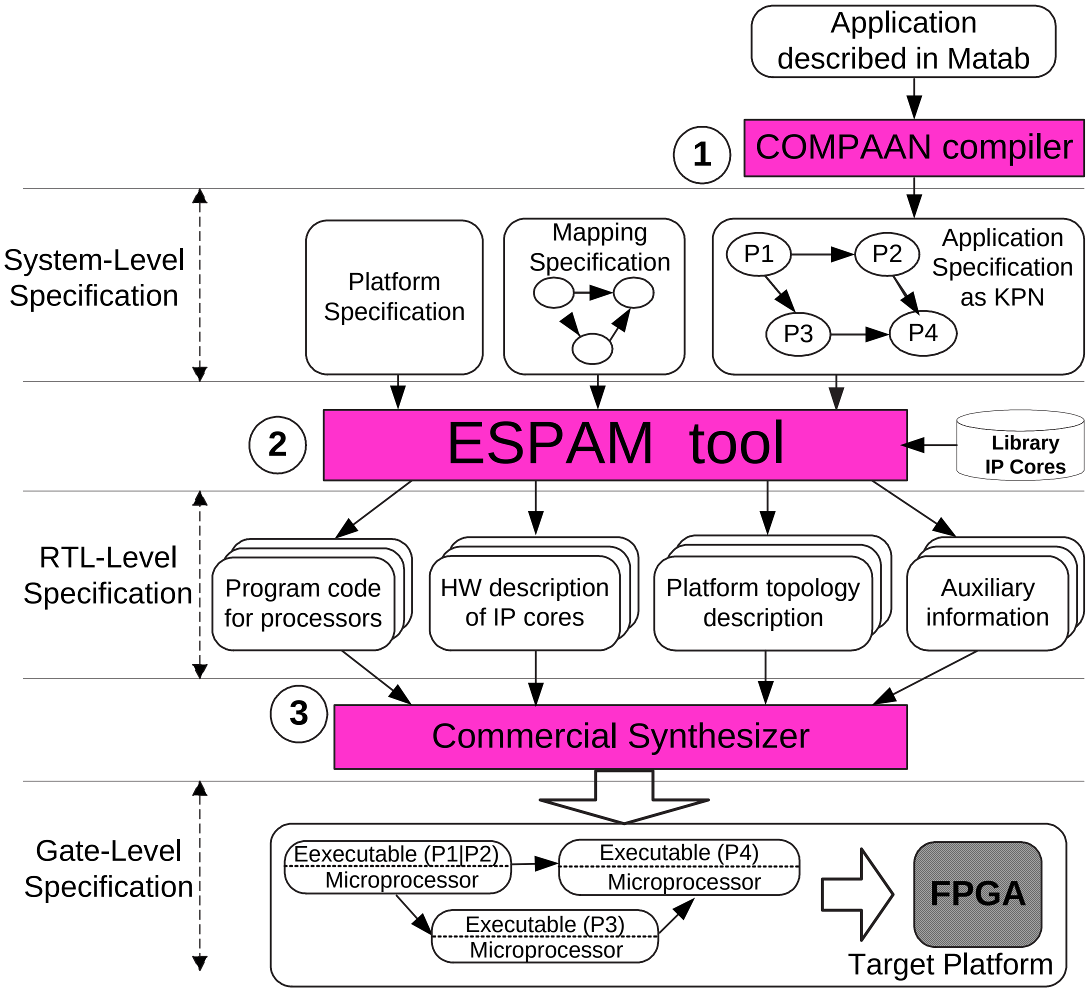


Figure 1.1: System design flow.

tem components. *Application Specification* specifies an application as a Kahn Process Network (KPN) which is a network of concurrent processes communicating via FIFO channels. Such KPN specification reveals the task-level parallelism available in an application. In order to migrate from a sequential specification of an application to an equivalent KPN specification, we use the COMPAAN compiler [2] [3] [4] which automates the transformation of Matlab code into KPN specification. The applications that COMPAAN can handle as the input have to be specified as parameterized static affine nested loop programs, which is a subset of the Matlab language. *Mapping Specification* specifies the relation between all processes and FIFO channels in *Application Specification* and all components in *Platform Specification*.

In Figure 1.1, the *System-level* specification is the input to the ESPAM tool. In our case, besides one-to-one mapping [5], ESPAM also supports many-to-one mapping. That means the number of processor components in *Platform Specification* can be less or equal to the number of processes in *Application Specification*. In other words, one or more than one processes in *Application Specification* can be mapped onto one processor in *Platform Specification*. For the channels in *Application Specification* and the FIFO components in *Platform Specification*, we still consider one-to-one mapping. This means that one channel in *Application Specification* is mapped onto one FIFO component in *Platform Specification* and one FIFO component has only one channel mapped onto it. Therefore, in our case we need all of three specification – *Platform Specification*, *Application Specification* and *Mapping Specification* as the input of ESPAM tool.

Our ESPAM tool systematically and automatically converts a *System-level* specification to a *RTL-level* specification thereby closing the *Implementation Gap* described in Section 1.1. First, ESPAM constructs a platform instance according to *Platform Specification* and runs a consistency check on this instance. This platform instance is an abstract model and at this step no information about the target physical platform is taken into account. Such platform instance consists of generic parameterized system components. Second, ESPAM refines the abstract platform model to an elaborate parameterized RTL model which is ready for an implementation on a target physical platform. Finally, ESPAM generates program code for each processor on the multiprocessor embedded system platform according to *Application Specification* and *Mapping Specification*. Our ESPAM tool will be described in detail in Chapter 2.

The output of ESPAM is a *RTL-level* specification of an embedded system which consists of four parts – *Platform topology description*, *Hardware description of IP cores*, *Program code for processors* and *Auxiliary information*. *Platform topology description* gives in great detail description of a multiprocessor platform. *Hardware description of IP cores* includes all predefined IP cores and reconfigurable IP cores which are used in *Platform topology description*. *Program code for processors* contains the program source code for each processor component on a multiprocessor platform. ESPAM can generate the program source code in C/C++ language for each processor component according to the behavior of the corresponding process in *Application Specification*. *Auxiliary information* includes supply files which give tight control of the overall specifications, such as defining precise timing requirements and prioritizing signal constrains.

A commercial synthesizer can be used to convert the *RTL-level* specification of an embedded system to the *Gate-level* specification of an embedded system. In the bottom part of Figure 1.1, we see that such commercial synthesizer can be used to generate the target platform gate-level netlist which is actually the system implementation.

1.2.2 Heterogeneous and Hierarchical Architecture Implementation

In order to meet the requirement performances of various applications an embedded system as heterogeneous and hierarchical architecture has to be implemented systematically and automatically. In this thesis, we give the procedure which explains how to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture which contains processor components and a dedicated hardware IP core. In our case, the processor components use FIFOs to communicate with each other. In order to make the dedicated hardware IP core can communicate with the processor components, the dedicated hardware IP core should has the FIFO input and output interfaces. The dedicated hardware IP core can be designed by hand. But this method is error-prone and time consuming. In our case, we use the LAURA tool [6] which has been developed at the Leiden Embedded Research Center (LERC) to generate the dedicated hardware IP core which contains the FIFO input and output interfaces. In this heterogeneous and hierarchical architecture, we use the dedicated hardware IP core to execute the most complicated process of an application repetitively and use the processor components to execute the other processes of an application in order to reach the good performance of execution time. In this way, we can prove that it is possible to implement systematically and automatically an embedded system as heterogenous and hierarchical architecture using the ESPAM technology.

1.2.3 Interface of an Embedded System with the Outside World Construction

As we have presented in Section 1.1, we need to construct an efficient interface of an embedded system which can make the applications which are mapped onto the embedded system platforms can communicate with the outside world efficiently. In our case, we have constructed an efficient interface to make embedded systems can communicate with the outside world by using several memories. Our interface uses the memories as the buffers to exchange data between embedded systems and the outside world. First embedded systems or the outside world writes data to the memories, then the outside world or embedded systems read data from the memories. Because embedded systems and the outside world may have different data transfer speeds, by using the memories as the buffers to exchange data embedded systems and the outside world do not need to wait for each other. In this way, we can speed up data transfer between embedded systems and the outside world. As we have discussed before, the embedded systems are becoming multiprocessor architecture. If we just use one memory as buffer to exchange data between an embedded system and the outside world, the processors of an embedded system cannot access to the memory concurrently. This means that every time just one processor can exchange data with the outside world. This is not efficient. Thus we use several memories as the buffers to exchange data between embedded systems and the outside world. In this way, each processor of an embedded system can access to one of the memories. This means the processors of an embedded system can exchange data with the outside world concurrently. By using several memories, we also can speed up data transfer between embedded systems and the outside world.

1.3 Related Work

Mapping application to architecture systematically and automatically has been widely studied in the research community. The closest work to our work is the LAURA tool [6] which has been developed at the Leiden Embedded Research Center (LERC). The LAURA tool accepts the Kahn Process Network (KPN) specification and transforms the KPN specification together with predefined non-programmable IP cores into design implementations described as synthesizable VHDL. The KPN specification is automatically generated by COMPAAN from the Matlab code. The IP cores are needed preemptively as they implement the functionality of the functions used in the initial Matlab code. However, our ESPAM tool map the KPN Specification together with Platform Specification and Mapping Specification onto multiprocessor platforms. The functions used in the initial Matlab code 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 targeted for high-performance asynchronous FPGA (AFPGA) architectures has been described in [7]. This method transforms sequential programs as well as high-level descriptions of asynchronous circuits into fine-grain asynchronous process netlists suitable for an AFPGA. The resulting circuits are inherently pipelined, and can be physically mapped onto an AFPGA with standard partitioning and place-and-route algorithms. The input to the synthesis is a sequential program written in CHP notation which is a hardware description language. Their automated synthesis of asynchronous computations is limited onto an pipelined AFPGA architecture. In contrast, in our design methodology, 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 with various abstraction levels called C-HEAP [8] is introduced which starts with a high-level executable specification and converges towards a silicon implementation. A major task in the design process is to ensure that all components (hardware and software) communicate with each other correctly. In their design methodology, seven abstraction levels that are traversed throughout the design process have been identified. They propose a heterogenous multi-processor architecture template based on distributed shared memory and present an efficient and transparent protocol for communication and (re)configuration. Our design methodology is similar to this. There are four levels in our design flow, e.g., application level, system level, RTL level and Gate level. We traverse them from application level to system level using COMPAAN tool, from system level to RTL level using our ESPAM tool then to Gate level using a commercial synthesis tool. Another major difference is that our platform model uses distributed memory instead of a shared memory. Another similar work which is focus on synthesis of application specific multiprocessor System-on-Chip architectures for process networks of streaming applications has been presented in [9]. In their methodology, 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. On the contrary, 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.

Many research works have been done for architecture development for embedded system in order to meet the required performance. A microcode-based microarchitecture has been described in [10]. They propose a microarchitecture based on reconfigurable hardware emulation to allow high-speed reconfiguration and execution. They implement a microarchitecture on the Virtex II Pro with the embedded PowerPC 405 serving as the core processor. On the contrary, in our case we implement systematically and automatically the embedded system as heterogeneous and hierarchical architecture which contains different types of components in the embedded system, such as processors and dedicated hardware IP cores. A next generation architecture for heterogeneous embedded systems has been presented in [11]. In their methodology, the Software Communications Architecture (SCA), a mandatory specification for Software Radio implementations by the Joint Tactical Radio System (JTRS), defines a Common Object Request Broker Architecture (CORBA) based component model for building portable applications in a heterogeneous environment. They use the SCA revisions to address the key scalable embedded processing issue – interchangeability of software and heterogeneous hardware components. In our case, the heterogeneous and hierarchical architecture contains the programmable processors, which are used to execute the software programs, and dedicated hardware IP cores. This heterogeneous and hierarchical architecture is able to meet the required performance of various applications. A heterogeneous evolutional architecture has been described in [12]. In they methodology, heterogeneous architecture means the architecture involves some combination of several single styles. They believe that the heterogenous architecture they need is that one group of components can be aggregated to form a subsystem in a particular architectural style, while another group of components can form a second subsystem in a completely different architectural style. Our heterogeneous and hierarchical architecture is similar to this. The difference is that our heterogeneous and hierarchical architecture means the components which form a subsystem also can be completely different types.

1.4 Research Contributions

The main research contributions of this thesis are:

- The gap between the *System-level* specification of multiprocessor systems and the *RTL-level* specification of multiprocessor systems has been successfully closed. In this thesis, we present our design methods and techniques for mapping applications onto multiprocessor platforms. We also introduce our ESPAM tool which allows the system designers to specify a multiprocessor system at a high level of abstraction – *System-level* specification in a short amount of time and it can systematically and automatically convert a *System-level* specification to a *RTL-level* specification for a multiprocessor platform.

- We have proved that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical multiprocessor architecture using the ESPAM technology. We give the procedure which explains how to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture which contains processor components and dedicated hardware IP core. With the heterogeneous architecture, different processes of an application can be executed by different types of components on an embedded system platform. With the hierarchical architecture, the complex process of an application can be mapped onto several components which compose a sub-network on an embedded system platform. By systematically and automatically implementing an embedded system as heterogeneous and hierarchical architecture, it is easy to meet the requirement performances of various applications.

- We have developed an efficient interface of an embedded system with the outside world using several memories. With this interface, the applications which are mapped onto the embedded system platforms can efficiently exchange data with the outside world via several memories.

1.5 Thesis Organization

The organization of the following part of this thesis is described as follows. Chapter 2 introduces our system design methodology and gives a detailed description of ESPAM tool we have developed. In this chapter, first the application model is introduced. Second, the platform model and platform synthesis is presented. Third, the mapping techniques are described. Fourth, programming multiprocessor platforms is explained. Finally, project generation for Xilinx Platform Studio (XPS) [13] is introduced.

Chapter 3 proves that it is possible to implement an embedded system as heterogeneous and hierarchical architecture systematically and automatically. First, we give a brief introduction to what we mean as heterogeneous and hierarchical architecture. Second, we give the procedure which explains how to implement an embedded system as heterogeneous and hierarchical architecture thereby proving that it is possible to implement an embedded system as heterogeneous and hierarchical architecture systematically and automatically.

Chapter 4 introduces the implementation of an efficient interface of an embedded system with the outside world. First, we describe the target FPGA platform. Second, the components including in the interface are introduced. Third, the steps about how to make the ESPAM automatically generate our interface when it maps applications onto multiprocessor platforms are presented.

In Chapter 5 two case studies are presented. The first one is mapping the M-JPEG encoder application onto a multiprocessor embedded system platform with homogeneous architecture. The second one is mapping the M-JPEG encoder application onto a multiprocessor embedded system platform with heterogenous and hierarchical architecture. The analysis of the results obtained from the experiments is also given in these case studies.

In Chapter 6 a tutorial on how to map the M-JPEG encoder application onto an embedded system platform with heterogenous and hierarchical architecture using the COMPAAN/ESPAM tools and the commercial synthesis tool – Xilinx Platform Studio (XPS) is presented.

In the last chapter, the summary and conclusions are given. The suggestions for the future work are also presented in this chapter.

Chapter 2

Embedded System-level Platform Synthesis and Application Mapping

In this chapter, a detailed description of our system design methodology which is implemented in our ESPAM tool – Embedded System-level Platform Synthesis and Application Mapping is presented. The structure of our system design flow has already been shown in Figure 1.1. In Figure 1.1, we can see that the input of our ESPAM tool is the *System-level* specification: *Application Specification*, *Platform Specification* and *Mapping Specification*. The output of our ESPAM tool is the *RTL-level* specification: *Platform topology description*, *Hardware description of IP cores*, *Program code for processors* and *Auxiliary information*. By describing our system design methodology we explain how the ESPAM tool bridges the *Implementation Gap* between the *System-level* specification of an embedded system and the *RTL-level* specification of this embedded system.

In Section 2.1, we introduce the Kahn Process Networks (KPN) model of computation which is used for the *Application Specification*. We also explain the COMPAAN tool that converts a sequential specification of an application to an equivalent KPN specification. In Section 2.2, the platform model is described first and an example of a *Platform Specification* is given. Then the synthesis of a platform is explained in detail. In Section 2.3, the mapping procedure which is used to bind the application and platform models together is described. Also, an example is given to explain clearly this procedure. Section 2.4 explains how to generate program code for each processor on a platform. Section 2.5 describes the mechanism of project generation for Xilinx Platform Studio (XPS).

2.1 Application Model

As discussed in Chapter 1, the suitable specification format for applications which makes the mapping of the applications onto multiprocessor embedded system architectures easy is the parallel model of computation. Therefore, exploiting parallelism available in such applications is important in embedded system design. In our ESPAM design methodology, we use the Kahn Process Network [14] (KPN) model of computation for *Application Specification*. We use the

COMPAAN tool [2] to automatically transform an application which is specified in sequential model of computation into KPN model of computation making the task-level parallelism available in an application explicit.

2.1.1 Kahn Process Networks

We believe that the Kahn Process Network model is an appropriate parallel model of computation for *Application Specification*. The reason is that in order to use parallel resources available in a multiprocessor platform, we need to program them in a way that we exploit distributed control and distributed memory. Kahn Process Networks inherently express applications in terms of distributed control and memory.

The KPN model of computation [14] 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. Each process in the network is specified as a sequential program that executes concurrently with other processes. A simple example of the KPN model is shown in Figure 2.1. There are three processes in this KPN model. They are processes P1, P2, and P3. These three processes are connected by the FIFO channels CH1, CH2, and CH3. In Figure 2.1 we see that process P1 first reads data from its input port, executes some computations and then writes the resulting data to processes P2 and P3 via CH1 and CH2 respectively. Process P2 first reads data from CH1, executes some computations and then writes the resulting data to process P3 via CH3. Process P3 first reads data from CH2 and CH3, executes some computations and then writes the resulting data to its output port.

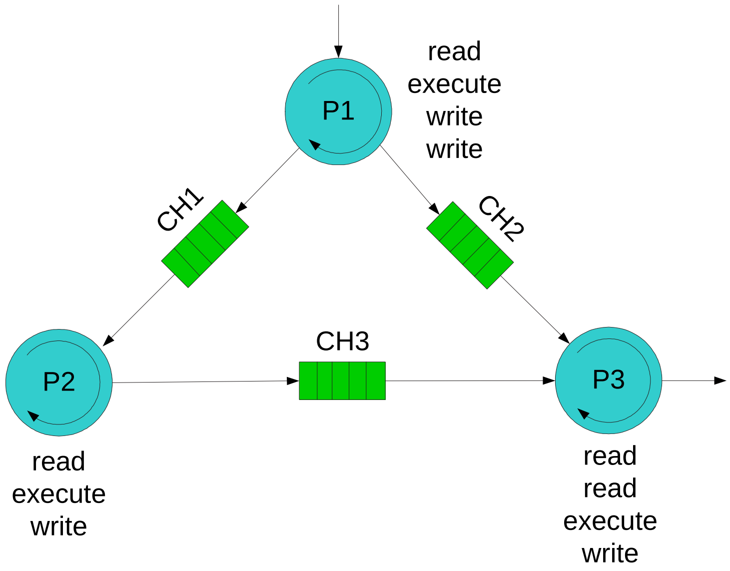


Figure 2.1: A simple KPN model.

The KPN has the following favorable characteristics [15]:

- The KPN model is deterministic, which means that irrespective of the schedule chosen to evaluate the network, always the same input/output relation exists. This gives us a lot of scheduling freedom that we can exploit when mapping processes to hardware or software.

- The inter-process synchronization is done by a blocking read. This is a very simple synchronization protocol that can be realized easily and efficiently in hardware and software.
- Processes run autonomously and synchronize via the blocking read. When mapping processes on hardware like an FPGA, you get autonomous islands on the FPGA that are only synchronized via blocking read.
- As control is completely distributed to the individual processes, there is no global scheduler present. As a consequence, partitioning a KPN over a number of reconfigurable components such as microprocessors is a simple task.
- As the exchange of data has been distributed over the FIFOs, there is no notion of a global memory that has to be accessed by multiple processes. Therefore, resource contention does not occur.

Due to the characteristics of the KPN described above, we believe that the KPN parallel processing model matches our system design methodology very well and the mapping of KPN specifications onto our multiprocessor platforms can be done in a systematic and automated way using our ESPAM tool.

2.1.2 The COMPAAN tool

Nowadays, most of the applications are written using a sequential model of computation. The sequential model of computation makes it easy to reason about an application, as only a single memory and a single thread of control need to be considered. But such sequential model of computation can not exploit the inherent parallelism available in an application. In order to automatically transform the application which is specified in sequential model of computation into KPN model of computation making the task-level parallelism available in an application explicit, we use the COMPAAN tool chain [2] [3] [4].

COMPAAN fully automates the transformation of Matlab code into Kahn Process Network (KPN). The applications, COMPAAN can handle, have to be specified as parameterized static affine nested loop programs, which is a subset of the Matlab language. The COMPAAN tool 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 associating a process with each node of the PRDG. The parallel processes communicate with each other according to the data-dependency given in the DG.

2.2 Platform Model and Synthesis

Here we introduce the platform model and synthesis in our system design methodology. In our ESPAM tool, the platform model is an abstract model of a multiprocessor platform onto which we map a KPN specification. Such abstract model is constructed by using a set of generic parameterized components. In the ESPAM tool there are four groups of generic parameterized components which are listed below. These components are generic parameterized modules that can specify a large number of concrete components.

- *Processing Components*: Currently, our system level platform model supports only one type of processing component, namely a programmable processor. It has several parameters such as *type*, *number of I/O ports*, *speed*, etc.
- *Memory Components*: Two types of memory components are defined and supported. One is used for specifying the processors' local program and data memories and the other is so called ”Communication Memory”. It is used to specify data communication storage (buffer) between processors. Important memory component parameters are *type*, *size*, *number of I/O ports*.
- *Communication Components*: They are a point-to-point network, a crossbar switch, and a shared bus. These components specify the network topology of a multiprocessor platform.
- *Auxiliary Components*: This group consists of two components, namely a controller and a link. The controller component is used to specify an interface between processing, memory, and communication components (if necessary). The link component is used to connect any two components in our system level platform model.

Using our platform model, the embedded system designer can easily construct many alternative multiprocessor platforms by instantiating generic parameterized components from the platform model and interconnecting these components. Each component in the platform model has several parameters which need to be set when such component has to be instantiated. Each parameter of generic component in the platform model has a range of values and the range is determined by resource limitations of the physical platform technology onto which our multiprocessor platforms are implemented. For example, if we use the Xilinx VirtexII-Pro FPGA as the physical platform technology onto which our multiprocessor platforms are implemented, the parameter *type* of the *Processing Components* can be set to *MicroBlaze* and *PowerPC* which are the two types of processor supported by Xilinx. Moreover, each platform specification can have many *MicroBlaze Processing Components* but it cannot have more than four *PowerPC Processing Components* according to the resource limitations of the Xilinx VirtexII-Pro FPGA. In order to guarantee correct-by-construction automated platform synthesis and implementation, ESPAM tool runs a consistency check on the platform specification which is specified by the designer. The consistency check includes checking whether the connections between platform components are correct and whether the parameter values of the platform components are set correctly. Moreover, the designer can leave parameter values undefined and let the ESPAM tool to set them automatically in the model refinement and synthesis procedure.

In the ESPAM tool, we use XML format for a platform specification because it is an easy way to specify a platform instance using the platform model. Figure 2.2 shows an example of a platform specification. In Figure 2.2 we see that there are three processors – *MB_1*, *MB_2* and *MB_3* in this platform specification and the types of these three processors are all *MicroBlaze*. We also set the size of the data memory and program memory for each processor. In this platform specification, we do not have to specify the memory structures, interface controllers, and

```
<platform name="myPlatform">

<processor name="MB_1" type="MB" data_memory="16384" program_memory="8192">
</processor>

<processor name="MB_2" type="MB" data_memory="16384" program_memory="8192">
</processor>

<processor name="MB_3" type="MB" data_memory="16384" program_memory="8192">
</processor>

</platform>
```

Figure 2.2: An example of the platform specification.

communication and synchronization protocols. Our ESPAM tool automatically specifies these in the platform synthesis which is described as follows. First, our tool instantiate the processing and the communication components following the platform specification. Second, it automatically attaches memories to each processor. In our case, one or two (data and program) memory modules have to be instantiated as the local memories along with each processor and the memory controllers have to be instantiated as the interfaces between each processor and its local memories. The memory generation is controlled by parameters within the platform specification. For example, in Figure 2.2 we have specified the size of the three processors' memories such as the data memories and the program memories. The size of the data memories and the program memories which are generated for the three processors are controlled by the parameters which are specified in Figure 2.2. Third, our tool automatically synthesizes, instantiates, and connects all necessary communication memories and communication controllers to allow efficient and safe data communication and synchronization between the components. In our case, a FIFO buffer has to be instantiated for each channel in the KPN model. A bus has to be instantiated for a connection between any two components of processor, FIFO, FIFO controller, memory and memory controller. Finally, our tool sets proper values of the parameters of each component.

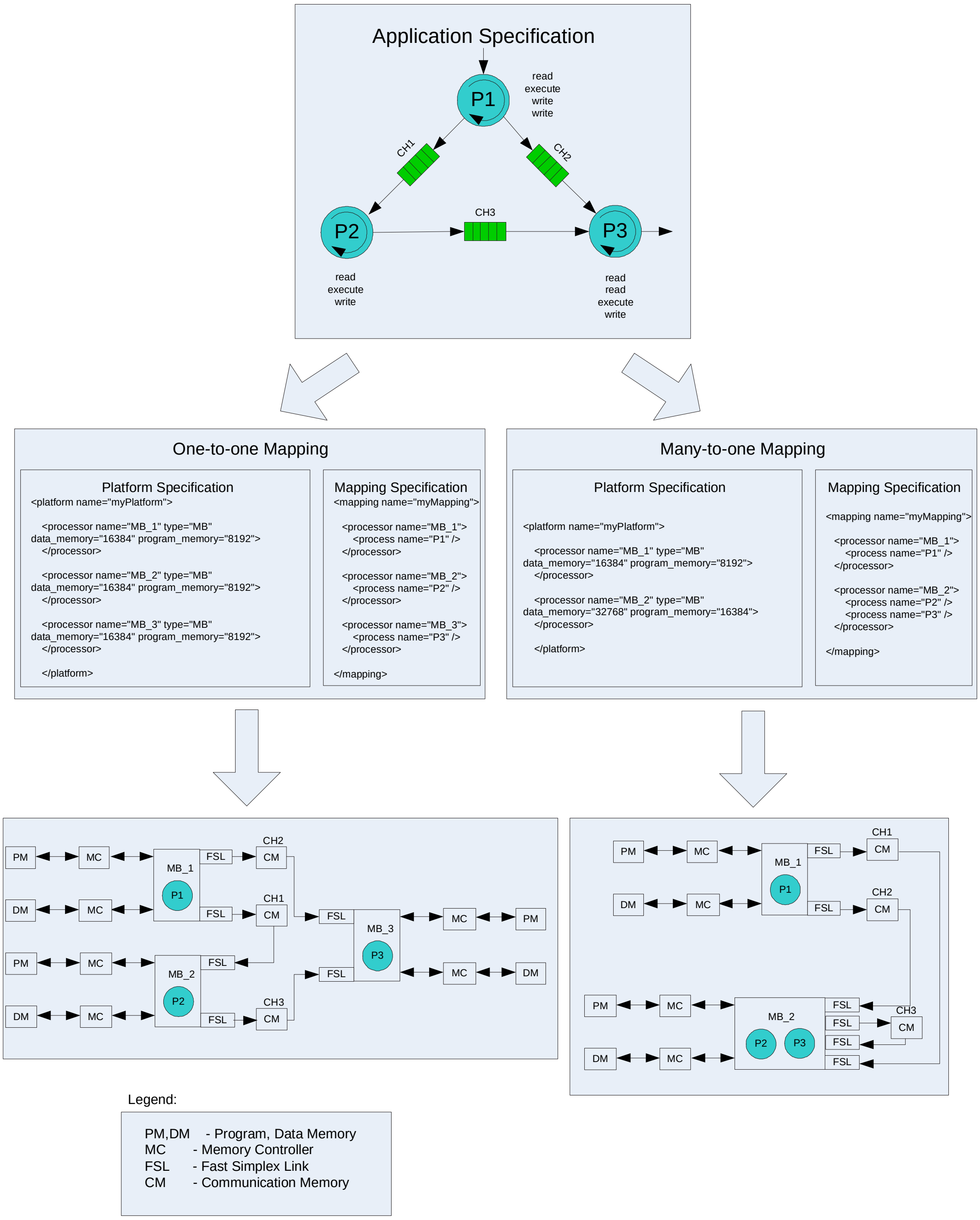
In ESPAM, a communication memory is organized as FIFO buffers. This organization is because: 1) The applications which we map onto our multiprocessor platforms are specified as KPNs where the data communication is realized via FIFO channels; 2) the inter-processor synchronization in a platform can be implemented in a very simple and efficient way by blocking read/write operations on empty/full FIFO buffers. When a processor has to write data to its local communication memory, it first checks if there is room in the corresponding FIFO. If the FIFO is full, the processor blocks. Otherwise, it sends the data to this FIFO buffer. When a processor has to read from a communication memory, it first checks if there is any data in the corresponding FIFO. The processor blocks if the FIFO is empty, otherwise it reads the data. This mechanism which is described above is called *blocking* read/write. There are two methods to implement the *blocking* read/write. The first method is that some processors have dedicated embedded hardware that can be used to stall the processors. The second method is that the blocking is realized in software by executing empty loops. There are different advantages in each of the methods. For the first one, the *blocking* read/write implemented in hardware is faster than the second method in which the *blocking* read/write is implemented in software. For the second one, the *blocking* read/write implemented in software is more general than the first method in which the *blocking* read/write is implemented in hardware because the different processors are stalled in hardware in different ways. In our case, we use both of the methods to implement the *blocking* read/write.

2.3 Mapping of Application Model onto Platform Model

In Figure 1.1, there is a specification named *Mapping Specification* in the *System-level* specification of our ESPAM tool. Based on the *Mapping Specification*, our ESPAM tool executes the mapping process which is a process of binding the application and platform models together. In the *Mapping Specification*, the relation between the channels and processes in the *Application Specification* and all the components in the *Platform Specification* is given.

Currently, our ESPAM tool supports two types of mapping. They are *one-to-one* mapping and *many-to-one* mapping. *One-to-one* mapping means that: 1) the number of processing components in the *Platform Specification* is equal to the number of processes in the *Application Specification*. Each process is mapped onto only one processor and each processor has only one process mapped onto it; 2) the number of communication memories in the *Platform Specification* is equal to the number of channels in the *Application Specification*. A channel in the *Application Specification* is mapped onto a communication memory in the *Platform Specification* and each communication memory has only one channel mapped onto it, so that all the connections are point-to-point connections. *Many-to-one* mapping means that: 1) the number of processing components in the *Platform Specification* is less than the number of processes in the *Application Specification*. Two or more processes are mapped onto only one processor; 2) the number of communication memories in the *Platform Specification* is still equal to the number of channels in the *Application Specification*. A channel in the *Application Specification* is mapped onto a communication memory in the *Platform Specification* and each communication memory has only one channel mapped onto it. Therefore, in order to obtain different alternative implementations for an application we just need to change the *Platform Specification* and the *Mapping Specification* of this application.

Figure 2.3 shows an example of both the *one-to-one* and *many-to-one* mapping processes. The top part of Figure 2.3 shows the *Application Specification* of this example. There are three processes in this KPN model. They are processes P1, P2, and P3. These three processes are connected by the FIFO channels CH1, CH2, and CH3. The left part of Figure 2.3 shows the *one-to-one* mapping process. The middle-left part of Figure 2.3 shows the *Platform Specification* and the *Mapping Specification* for the *one-to-one* mapping. In the *Platform Specification* of the *one-to-one* mapping, we see that there are three processors – *MB_1*, *MB_2* and *MB_3* in this platform specification and the types of these three processors are all *MicroBlaze*. We also set the size of the data memory and program memory for each processor. The number of the processors is equal to the number of processes in the *Application Specification*. In the *Mapping Specification*, we see that process P1 is mapped onto processor *MB_1*, process P2 is mapped onto processor *MB_2*, process P3 is mapped onto processor *MB_3*. Notice that mapping of channels is not specified in the *Mapping Specification*. This is not necessary because each communication memory (CM) may has only one channel mapped onto it according to the definition of the *one-to-one* mapping. Therefore, each channel in the *Application Specification* is mapped onto a communication memory which is organized as FIFO buffer with standard FIFO input and output interface signals. We use Fast Simplex Link (FSL) to connect a FIFO buffer



```
<platform name="myPlatform">

   <processor name="MB_1" type="MB"
data_memory="16384" program_memory="8192">
   </processor>

   <processor name="MB_2" type="MB"
data_memory="16384" program_memory="8192">
   </processor>

   <processor name="MB_3" type="MB"
data_memory="16384" program_memory="8192">
   </processor>

   </platform>
```

```
<mapping name="myMapping">

  <processor name="MB_1">
     <process name="P1" />
  </processor>

  <processor name="MB_2">
     <process name="P2" />
  </processor>

  <processor name="MB_3">
     <process name="P3" />
  </processor>

</mapping>
```

```
<platform name="myPlatform">

   <processor name="MB_1" type="MB"
data_memory="16384" program_memory="8192">
   </processor>

   <processor name="MB_2" type="MB"
data_memory="32768" program_memory="16384">
   </processor>

   </platform>
```

```
<mapping name="myMapping">

  <processor name="MB_1">
     <process name="P1" />
  </processor>

  <processor name="MB_2">
     <process name="P2" />
     <process name="P3" />
  </processor>

</mapping>
```

Figure 2.3: An example of *one-to-one* mapping and *many-to-one* mapping.

to a *MicroBlaze* processor. In Section 2.2, we explained that ESPAM automatically attaches memories to each processor. In this example, data (DM) and program (PM) memory modules are instantiated as local memories along with each processor and the memory controllers (MC) are instantiated as interfaces between each processor and its local memories. The size of the memories is controlled by parameters within the *Platform Specification*. The final elaborate platform of the *one-to-one* mapping example is shown in the bottom-left part of Figure 2.3.

The right part of Figure 2.3 shows the *many-to-one* mapping process. The middle-right part of Figure 2.3 shows the *Platform Specification* and the *Mapping Specification* for the *many-to-one* mapping. In the *Platform Specification*, we see that there are two processors – *MB_1* and *MB_2* in this *Platform Specification* and the types of these two processors are *MicroBlaze*. We also set the size of the data memory and program memory for each processor. In the *Platform Specification*, we see that the number of the processors is less than the number of processes in the *Application Specification*. In the *Mapping Specification*, we see that process P1 is mapped onto processor *MB_1*, process P2 and process P3 are mapped onto processor *MB_2*. Notice that mapping of channels is also not specified in the *Mapping Specification*. The reason is the same as in the *one-to-one* mapping. Our ESPAM also automatically attaches data and program memory and memory controllers to each processor. The size of the memories is controlled by parameters within the *Platform Specification*. The final elaborate platform of the *many-to-one* mapping example is shown in the bottom-right part of Figure 2.3.

2.4 Programming Multiprocessor Platforms

The synthesized multiprocessor platform has to be programmed in order to execute an application. Programming the multiprocessor platform means generating program code for each processor in the platform using high level programming languages like C/C++.

In this thesis we use the MicroBlaze soft processor core as the processor in multiprocessor platforms. The MicroBlaze soft processor core is programmed by GNU tools that generate standard Executable and Linkable Format (ELF) [16] [17]. 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 ELF executable files. Our methodology implemented in the ESPAM tool is able to generate program code for MicroBlaze processors. We use the software engineering technique called *Visitor* [18] to generate C program code for each MicroBlaze processor. The brief explanation of the program code generation for each processor follows. As discussed earlier, we model an application as a Kahn Process Network (KPN) and map processes of the KPN onto the processors of a multiprocessor platform. Therefore, the processors must be programmed according to the behaviors of the corresponding processes in the KPN. The process in the KPN 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 [19]. The advantage 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. Then we use the software engineering technique called *Visitor* to traverse a syntax tree and to generate program code. The program code can be expressed in any programming language for which a compiler support exists for the processors used in a platform. We use the MicroBlaze soft processor core as the processor in multiprocessor platforms and 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 ELF executable files. Therefore, we use the *Visitor* technique to traverse a syntax tree and to generate C program code.

2.5 Project Generation for Xilinx Platform Studio

In this section, we introduce the methodology implemented in ESPAM to generate Xilinx Platform Studio (XPS) projects. Xilinx Platform Studio (XPS) is a system design Integrated Development Environment (IDE) that supports open interfaces making tool integration easy and painless and it is used to develop Xilinx Embedded Development Kit (EDK) - based system designs. XPS provides a common fully integrated hardware/software development environment that supports the complete range of Xilinx's processor solutions. XPS is the graphical user interface technology that integrates all of the processes from design entry to design debug and verification. Embedded Development Kit (EDK) is a series of software tools for designing embedded processor systems on programmable logic, and supports the IBM PowerPC hard processor core and the Xilinx MicroBlaze soft processor core. Including in the EDK, the scalable Platform Studio enables designers to easily develop, integrate and debug their entire embedded system. In this thesis, we mainly use the configurable MicroBlaze embedded soft processor core. The MicroBlaze embedded soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx field programmable gate arrays (FPGAs). It is highly configurable, allowing users to select a specific set of features required by their design. As the MicroBlaze 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. Due to this reason, the MicroBlaze embedded soft processor core is suitable for constructing our multiprocessor embedded systems.

However, directly using Xilinx Platform Studio (XPS) to design a multiprocessor embedded system is extremely time-consuming and the parallelism implicit in an application can only be depicted manually. Due to these reasons, generation of a complex multiprocessor embedded system in XPS takes lots of time. In order to reduce the design time, the XPS tool can be used as a back-end tool of our ESPAM tool. Our ESPAM tool can systematically synthesize a platform and automatically generate all necessary files for an XPS project according to *Platform Specification*, *Application Specification* and *Mapping Specification* which are shown in Figure 1.1. Therefore, using our ESPAM tool as the front-end tool and XPS tool as the back-end tool a designer can design a multiprocessor embedded system on a specific FPGA board efficiently and effectively.

2.5.1 Introduction to XPS Project Specification

In a Xilinx Platform Studio (XPS) project, all of the project information is stored in four files: Xilinx Microprocessor Project (XMP) file [20], Microprocessor Hardware Specification (MHS) file [20], Microprocessor Software Specification (MSS) file [20] and User Constraint File (UCF) [21]. An Xilinx Microprocessor Project (XMP) file is the top-level project file for an EDK design. It stores the project options. A Microprocessor Hardware Specification (MHS) file defines the configuration of an embedded processor system including buses, peripherals, processors, connectivity, and address space. A Microprocessor Software Specification (MSS) file contains directives for customizing libraries, drivers, and file systems. An User Constraint File (UCF) contains pin information for the physical implementation in a selected FPGA device.

An Xilinx Microprocessor Project (XMP) file includes the XMP version number, the location of MHS and MSS files, the FPGA architecture family and the device type for which the XPS hardware tool flow needs to run and the software setting for this project.

A Microprocessor Hardware Specification (MHS) file defines the hardware component used in a platform as well as the connections between these components. A MHS file defines the configuration of an embedded processor system, and includes the following: 1) Bus architecture; 2) Peripherals; 3) Processor; 4) Connectivity; 5) Address space. A MHS file uses the following format at the beginning of a component definition: *BEGIN peripheral_name*. The *BEGIN* keyword signifies the beginning of a new peripheral. It uses the following format for assignment commands: *command name = value*. It uses the following format to end a peripheral definition: *END*. There are three assignment commands: 1) *BUS_INTERFACE*; 2) *PARAMETER*; 3) *PORT*.

A Microprocessor Software Specification (MSS) file contains directives for customizing operating systems (OS), libraries, and drivers. A MSS file has a dependency on a MHS file. The keywords that are used in a MSS file are as follows: *BEGIN*, *END* and *Parameter*. The *BEGIN* keyword starts a driver, processor, or file system definition block. The begin keyword should be followed by *driver*, *processor* or *filesys* keywords. The *END* keyword signifies the end of a definition block. A MSS file has a simple *name = value* format for most statements. The *Parameter* keyword is required before every such NAME, VALUE pairs. The format for assigning a value to a parameter is *parameter name = value*. If the parameter is within a *begin-end* block, it is a local assignment, otherwise it is a global (system level) assignment.

An User Constraint File (UCF) contains pin information for the physical implementation in a selected FPGA device. It contains constrains such as FPGA pin locations, timing, FPGA resource specification and I/O standards.

2.5.2 Project Suite Generation

Our ESPAM tool can systematically synthesize a platform and automatically generate all necessary files for an XPS project according to *Platform Specification*, *Application Specification* and *Mapping Specification* that have been discussed before. The project suite is shown in Figure 2.4.

It includes the *system.xmp*, *system.mhs*, *system.mss* files and *code*, *etc*, *data*, *pcores* directories. The *system.xmp*, *system.mhs*, *system.mss* files are the MHS, MSS, XMP files of the project which have been discussed above. In the *code* directory, the software program code files for processors are stored. In the top level of the *code* directory, there are two files named *aux_func.h*, *MemoryMap.h*. They are the common files for all of the processors. The *aux_func.h* file declares read and write primitives and wrappers of all function calls in the initial code of an application. The *MemoryMap.h* file specifies physical addresses of the components in a platform. The program code for each processor is stored in the corresponding subdirectory named after the processors. The *etc* directory stores the optional files for the XPS implementation tools. There are four files in this directory: *bitgen.ut*, *bitgen_spartan3.ut*, *fast_runtime.opt* and *download*. In the *data* directory, the UCF file is stored. According to different FPGA boards, several UCF files are generated by our ESPAM tool. The *pcores* directory stores the customized IP cores for the EDK project. This is the ESPAM library of components depicted in Figure 1.1.

```
<PROJECT_SUITE>
|--- system.xmp
|--- system.mhs
|--- system.mss
|--- code/: software program code
|---------- aux_func.h
|---------- MemoryMap.h
|---------- P_1/: program code for processor P_1
|---------------- P_1.cpp
|---------- P_2/: program code for processor P_2
|---------------- P_2.cpp
|--- etc/: optional files for implementation tools
|--------- bitgen.ut
|--------- bitgen_spartan3.ut
|--------- fast_runtime.opt
|--------- download.cmd
|--- data/: UCF files
|---------- system.ucf
|---------- system_ADMXRCII.ucf
|---------- system-default.ucf
|---------- system-zbt.ucf
|--- pcores/: customized IP cores for the EDK project
|------------ buffers_v1_00_a/
|------------ cb_wrapper_v1_00_a/
|------------ fifo_if_ctrl_v1_00_a/
|------------ fin_ctrl_v1_00_a/
|------------ host_design_ctrl_v1_00_a/
|------------ LMB_VB_CTRL_v1_00_a/
|------------ mux_v1_00_a/
|------------ myCLKRST_v1_00_a/
|------------ opb_zbt_controller_v1_00_a/
|------------ VB_Wrapper_v1_00_a/
|------------ zbt_main_v1_00_a/
```

Figure 2.4: The project suite automatically generated by our ESPAM tool.

2.5.3 Visitor Pattern Mechanism

In this section, first we briefly introduce the *Visitor Pattern* and then we explain the *Visitor Pattern* mechanism which has been used in our ESPAM tool to generate the XPS project.

The *Visitor Pattern* [18] represent an operation to be performed on the elements of an object structure. The *Visitor Pattern* lets we define a new operation without changing the classes of the elements on which it operates. The *Visitor Pattern* turns the tables on our object-oriented model and creates an external class to act on data in other classes. This is useful if there are a fair number of instances of a small number of classes and we want to perform some operation that involves all or most of them. There are several participants in the *Visitor Pattern*: 1) *Visitor* declares a Visit operation for each class of *ConcreteElement* in the object structure. 2) *ConcreteVisitor* implements each operation declared by *Visitor*. 3) *Element* defines an Accept operation that takes a *Visitor* as an argument. 4) *ConcreteElement* implements an Accept operation that takes a *Visitor* as an argument. 5) *ObjectStructure* can enumerate its elements, may provide a high-level interface to allow the *Visitor* to visit its elements and may either be a composite or a collection such as a list or a set. The implementation of the *Visitor Pattern* is described as follows: Each *ObjectStructure* will have an associated *Visitor* class. This abstract *Visitor* class declares a *VisitConcreteElement* operation for each class of *ConcreteElement* defining the *ObjectStructure*. Each Visit operation on the *Visitor* declares its argument to be a particular *ConcreteElement*, allowing the *Visitor* to access the interface of the *ConcreteElement* directly. *ConcreteElement* classes override each Visit operation to implement visitor-specific behavior for the corresponding *ConcreteElement* class.

The visitor classes hierarchy in our ESPAM tool is shown in Figure 2.5. We use the *Visitor* technique which has been introduced above to generate all necessary files for an XPS project.

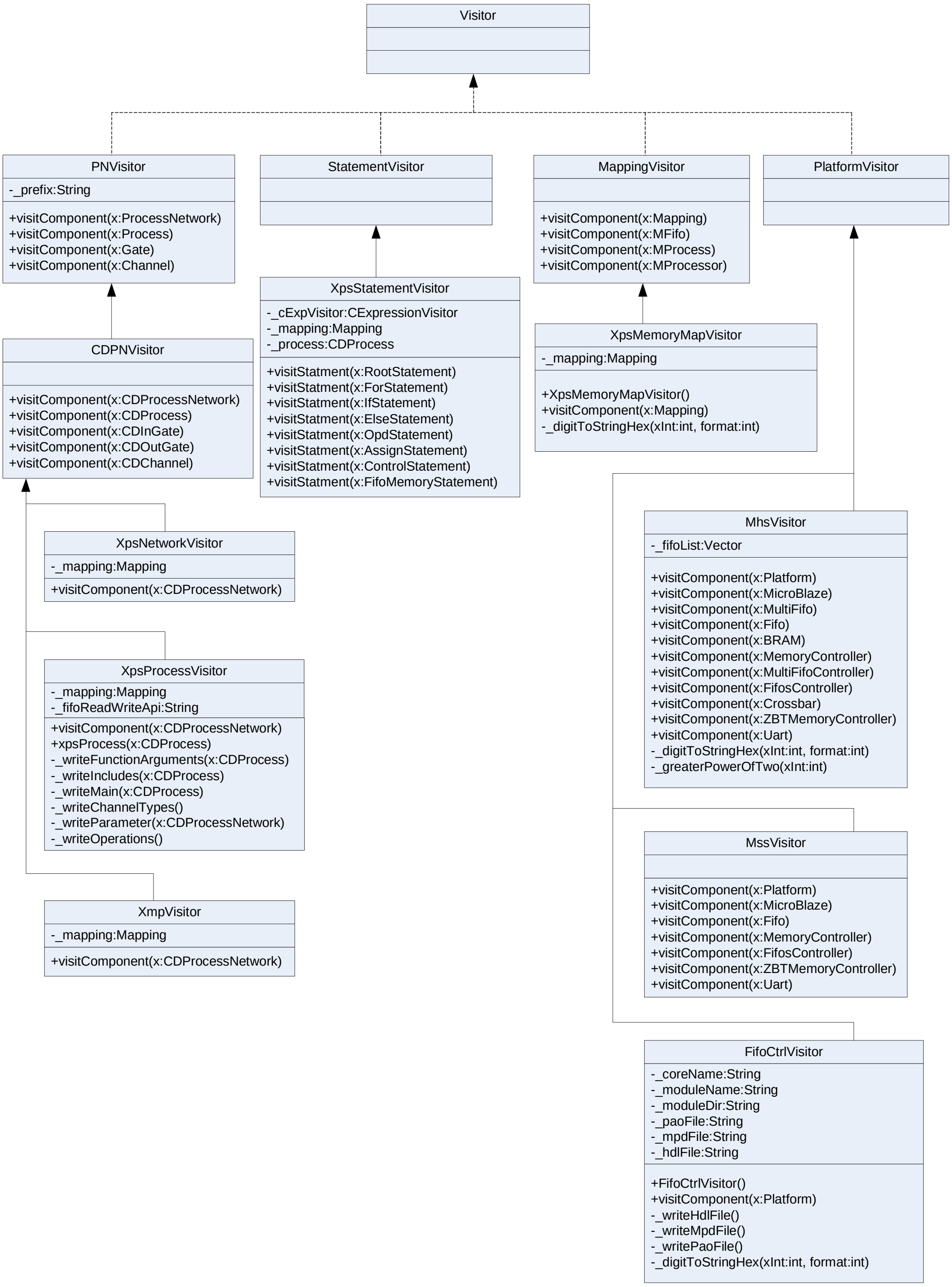


Figure 2.5: The visitor classes hierarchy in the ESPAM tool.

In Figure 2.5, the top level in our visitor classes hierarchy is an interface class called *Visitor* which is defined to traverse the data model. The data model includes all of the information which is given by *Platform Specification*, *Application Specification* and *Mapping Specification*. Four classes *PNVisitor*, *StatementVisitor*, *MappingVisitor* and *PlatformVisitor* implement the interface class. The *PNVisitor* class is an abstract class for a visitor that is used to generate a Process Network description. The *StatementVisitor* class is an abstract class for a visitor to traverse a processor's syntax tree. The *MappingVisitor* class is an abstract class for a visitor that is used to generate Mapping information. The *PlatformVisitor* class is an abstract class for a visitor that is used to generate a Platform description.

CDPNVisitor is an abstract class that extents *PNVisitor* class and it is used to generate Compaan Dynamic Process Network (CDPN) description. Three concrete classes named *XpsNetworkVisitor*, *XpsProcessVisitor* and *XmpVisitor* extend abstract class *CDPNVisitor*. *XpsNetworkVisitor* class is used to copy all of the predefined IP cores and the other necessary project files such as optional files and UCF files which have been introduced in Section 2.5.2 into an XPS project. *XpsNetworkVisitor* class is also used to call the *XpsProcessVisitor* class. *XpsProcessVisitor* class is used to generate the global program code file *aux_func.h* and it is also used to call the *XpsStatementVisitor* in order to traverse the syntax tree of each processor to generate program code for each processor. *XmpVisitor* class is used to generate the Xilinx Microprocessor Project (XMP) file for an XPS project.

The concrete class named *XpsStatementVisitor* which extends abstract class *StatementVisitor* is used to traverse the syntax tree of each processor and generate C code for each processor.

The concrete class named *XpsMemoryMapVisitor* which extends abstract class *MappingVisitor* is used to generate the global program code file *MemoryMap.h*.

Three concrete classes named *MhsVisitor*, *MssVisitor* and *FifoCtrlVisitor* extend abstract class *PlatformVisitor*. *MhsVisitor* class is used to generate Microprocessor Hardware Specification (MHS) file for an XPS project. *MssVisitor* class is used to generate Microprocessor Software Specification (MSS) file for an XPS project. *FifoCtrlVisitor* class is used to generate a custom IP core named Fifo Controller for an XPS project.

Chapter 3

Embedded System as Heterogeneous and Hierarchical Architecture

In this chapter, we introduce an embedded system as heterogeneous and hierarchical architecture and prove that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture using the ESPAM technology. In Chapter 2 we explained that an application always consists of several concurrent processes and these processes can be mapped onto the components on an embedded system platform. An embedded system as homogeneous architecture means the processes of an application are mapped onto the same type of components on an embedded system platform such as the same type of processors. However, as we know different types of components are suitable for implementing different types of processes. For example, some types of processors do not support floating point computation, this means if we map the processes which include the floating point computation onto such types of processors, the processors will spend a lot of time to evaluate floating point computation and the results will not be good enough. But if we map the floating point computation processes onto the processors or the dedicated hardware IP cores which support the floating point computation, it will save a lot of time and the results will be much better. Due to similar reasons an embedded system as heterogeneous architecture has to be implemented in order to meet the required performance of various applications. The problem is how to implement an embedded system as heterogeneous architecture systematically and automatically. What's more, an application always consists of several processes and some of the processes of the application maybe very complex. If we map a complex process onto a single component, the process may not meet the required performance. In such case, we need to map the complex process onto several components in order to meet the required performance. These several components form a sub-network on an embedded system platform. Therefore we call the system hierarchical architecture. Thus, an embedded system as hierarchical architecture also has to be implemented. The problem is how to implement an embedded system as hierarchical architecture systematically and automatically. In this chapter, we give the procedure to explain how to implement an embedded system as heterogeneous and hierarchical architecture in order to prove that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture using the ESPAM technology.

This chapter is organized as follows. In Section 3.1, we first give a general introduction to an embedded system as heterogeneous and hierarchical architecture. In this section, we give an example to describe the structure of the heterogeneous and hierarchical architecture and explain the differences between the homogeneous architecture and the heterogeneous and hierarchical architecture. In Section 3.2, we give the procedure that explains how to implement an embedded system as heterogeneous and hierarchical architecture using Xilinx VirtexII FPGA as the physical platform in order to prove that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture using the ESPAM technology.

3.1 Introduction to Heterogeneous and Hierarchical Architecture

As described above, an embedded system as homogeneous architecture means that all of the components which compose an embedded system platform are of the same type. For example, if an embedded system platform is a multiprocessor embedded system platform, a homogeneous architecture means all of the processors on the platform have the same attributes. Due to the complexity of modern applications, such as high throughput multimedia, imaging and digital signal processing which usually include complicated algorithms, an embedded system as homogeneous architecture is no longer suitable for modern applications. As what have been explained earlier, in order to meet the required performance of various applications we need to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture.

Figure 3.1 give examples which describe the structures of a homogeneous architecture, and a heterogeneous and hierarchical architecture. In the top of Figure 3.1, there is an example of a homogeneous architecture. This example is a multiprocessor embedded system. It consists of five processors and all of the processors are of the same type. They use a communication structure to communicate with each other. This means all of the processes of an application are mapped onto the same type of components. Because only one type of processors is not suitable for different kinds of processes, such homogeneous architecture is difficult to meet the required performance of various processes.

In the middle of Figure 3.1, there is an example of a heterogeneous and hierarchical architecture. This architecture includes different types of components — four different types of processors and one dedicated hardware IP core. They also use a communication structure to communicate with each other. That means the processes of an application can be mapped onto different types of components which are suitable for different types of processes of an application. Moreover, it is better to map the process which is the most complicated or which runs most frequently onto the dedicated hardware IP core. Because the process which is executed by a dedicated hardware IP core is much faster than the process which is executed by the software program of a processor. Thus, by using an embedded system as heterogeneous architecture, an application can reach higher performances. The bottom part of Figure 3.1 shows what we call hierarchical architecture in this example. The hierarchical architecture shows that the dedicated hardware IP core is not a single component. The dedicated hardware IP core is a sub-network which consists of four different hardware components — HW1, HW2, HW3 and HW4. The sub-network of

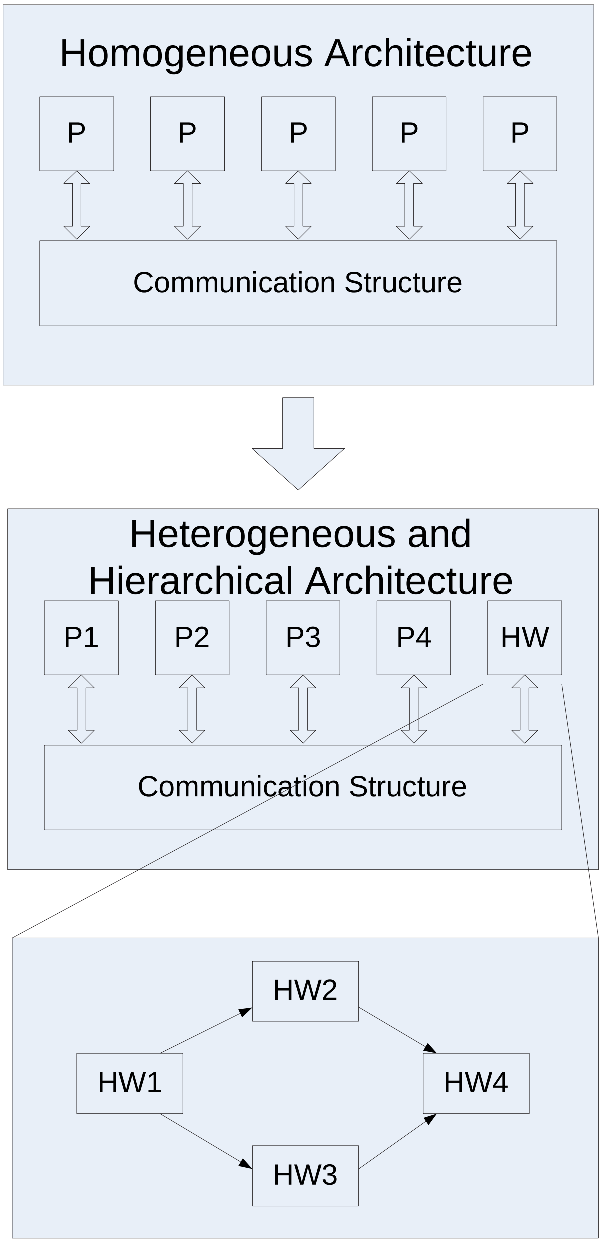


Figure 3.1: The structures of a homogeneous architecture, and a heterogeneous and hierarchical architecture.

the four hardware components implement the complex process of an application. This means we map the complex process of an application onto several components in order to meet the required performance. These several components form a sub-network on the embedded system platform. Therefore we call the system hierarchical architecture. In the next section, we will prove that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture using the ESPAM technology.

3.2 Heterogeneous and Hierarchical Architecture Implementation

In this section, we prove that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture using the ESPAM technology. In this section we use an example to explain how to implement an embedded system as heterogeneous and hierarchical architecture in order to prove that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture using the ESPAM technology. This example maps the same application onto two architectures — one is a homogeneous architecture and the other is a heterogeneous and hierarchical architecture. Also, we compare the application performances between these two architectures. This section is organized as follows. In Section 3.2.1, we create a system with an embedded system as homogeneous architecture. In Section 3.2.2, we create a system with an embedded system as heterogeneous and hierarchical architecture which has the same functionality with the system in 3.2.1. In Section 3.2.3, we do some tests on the system with the embedded system as heterogeneous and hierarchical architecture and compare the results with the system with the embedded system as homogeneous architecture.

3.2.1 Creating a System with Homogeneous Architecture

In this section, we use our ESPAM tool to automatically generate a system with homogeneous architecture. This system is used to implement the Discrete Cosine Transform (DCT). Many digital image and video compression applications usually use the Discrete Cosine Transform (DCT) as the transform coding step [22]. First images are always spatially divided into blocks, usually 8x8 pixels. Then DCT can process each block which includes 8x8 pixels. In our case, the system uses Xilinx VirtexII FPGA as the physical platform. The architecture of the system is shown in Figure 3.2.

Figure 3.2: The system with homogeneous architecture.

This system includes three *MicroBlaze* processors — MB1, MB2 and MB3 and two FIFOs — FIFO1 and FIFO2. Processor MB1 first generates the initial block and then writes the block to processor MB2 using FIFO1. Processor MB2 first reads the block from FIFO1, applies the DCT on this block and then writes the resulting block to processor MB3 using FIFO2. Processor MB3 first reads the resulting block from FIFO2 and then writes the resulting block to an off-chip memory. The main software code of these three processors is shown in Figure 3.3.

In Figure 3.3 we see that in our case we use an image which is in 4:2:2 YUV format. Thus the image block includes four 8x8 sub-blocks — Y1 sub-block, Y2 sub-block, U1 sub-block and V1 sub-block. In order to transfer the data between the processors, we use the FIFO components. The *MicroBlaze* processor gets data from other processor via a hardware FIFO buffer

```
int main (){

    TBlocks blocks = {
     ... ,
     ... ,
     ... ,
     ...
    };

    writeFSL(0, &blocks, (sizeof(blocks)+(sizeof(blocks)%4)+3)/4);

    ...
} // main
```

MB1

```
int main (){

    TBlocks blocks_in;

    TBlocks blocks_out;

    DCT dct;

    readFSL(0, &blocks_in, (sizeof(blocks_in)+(sizeof(blocks_in)%4)+3)/4);

    dct.main(blocks_in, blocks_out);

    writeFSL(0, &blocks_out, (sizeof(blocks_out)+(sizeof(blocks_out)%4)+3)/4);

    ...
} // main
```

MB2

```
int main (){

    volatile long *compImage = (volatile long *)0xf0000000;

    TBlocks blocks;

    readFSL(0, &blocks, (sizeof(blocks)+(sizeof(blocks)%4)+3)/4);

    for( int k =  0; k < 64; k += 1 ) {

        compImage[k] = (volatile long) blocks.Y1.pixel[k];
        compImage[k+64] = (volatile long) blocks.Y2.pixel[k];
        compImage[k+64*2] = (volatile long) blocks.U1.pixel[k];
        compImage[k+64*3] = (volatile long) blocks.V1.pixel[k];

    }

    ...
} // main
```

MB3

Figure 3.3: The main software code of the three processors.

using a read primitive and sends data to other processor via a hardware FIFO buffers using a write primitive. Because the hardware FIFO buffers in our platform are bounded, the read/write operation is blocking. In our example, we use Fast Simplex Link (FSL) [23] bus to communicate with the FIFO buffers. The code in Figure 3.3 show that we use *readFSL* and *writeFSL* functions to implement the blocking read/write FIFO mechanism. The FSL primitives implement the blocking read/write mechanism in hardware controlled by two *MicroBlaze* specific assembly instructions, namely *put* and *get* [24]. The *MicroBlaze* specific assembly instructions are shown in Figure 3.4. The *readFSL* and *writeFSL* functions are the wrappers for these assembly instructions which are shown in Figure 3.5. The variable *pos* denotes a port number for a FSL bus of a *MicroBlaze* processor. The variable *value* is used to store the data to be read or written. The variable *len* denotes the length (measured in 32-bit words) of the data to be read or written. When performing the read operation, a *MicroBlaze* processor gets data from one of its FSL input ports and stores data into the variable *value*. When performing the write operation, the *MicroBlaze* processor puts data stored in the variable *value* to one of its FSL output ports.

```
#define microblaze_bread_datafsl(val, id) \
            asm(’’get %0, %1’’ : ’’=d’’ (##val##) : ’’m’’ (rfsl##id##))

#define microblaze_bwrite_datafsl(val, id) \
            asm(’’put %0, %1’’ : ’’=d’’ (##val##) : ’’m’’ (rfsl##id##))
```

Figure 3.4: The *MicroBlaze* specific FSL bus read/write assembly instructions.

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

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

Figure 3.5: The *MicroBlaze* FSL bus read/write primitives.

When we ran the system in Figure 3.2 which is an embedded system with homogeneous architecture, we found out that the time performance of the DCT process is not very good. The reason is that in this homogeneous architecture the DCT process is run as software on a *MicroBlaze* processor. It is hard for the system to reach the good time performance by running the software DCT process on the processor.

3.2.2 Creating a system with Heterogeneous and Hierarchical Architecture

In this section, we introduce the procedure to create a system with embedded system as heterogeneous and hierarchical architecture. This system has the same functionality as the system presented in Section 3.2.1. It is also used to implement the Discrete Cosine Transform (DCT). The architecture of this system is shown in the top part of Figure 3.6. We see that this heterogeneous and hierarchical architecture also includes three components. The difference between this system and the system of homogeneous architecture is that we use a dedicated hardware IP core to implement the Discrete Cosine Transform (DCT). In this system, there are two *MicroBlaze* processors — MB1 and MB3 which have the same function as the MB1 and MB3 processors of the system which has been explained in Section 3.2.1. Instead of processor MB2 in the system explained in Section 3.2.1, we use a dedicated hardware IP core to implement the

DCT process. Therefore this system is a heterogeneous architecture. *MicroBlaze* processors use FIFOs to communicate with each other. In order to make the dedicated hardware IP core can communicate with *MicroBlaze* processors, the dedicated hardware IP core should has the FIFO input and output interfaces. In Figure 3.6 we see that the dedicated hardware IP core uses the FIFO input interface to read data from FIFO1, and uses the FIFO output interface to write data to FIFO2. The data flow of this system is: processor MB1 first generates the initial block and then writes the block to the hardware IP core using FIFO1. The hardware IP core first reads the block from FIFO1, applies the DCT on this block and then writes the resulting block to processor MB3 using FIFO2. Processor MB3 first reads the resulting block from FIFO2 and then writes the resulting block to an off-chip memory.

In order to create the system which has been described above, the first step we need to do is to generate the dedicated hardware IP core which can implement the DCT process. In general, we can design this dedicated hardware IP core by hand. But this method is error-prone and time consuming. In our case, we use the LAURA tool [6], which has been developed at the Leiden Embedded Research Center (LERC), to generate this dedicated hardware IP core which contains the FIFO input and output interfaces. There is a tool chain called COMPAAN/LAURA that allows us to map fast and efficiently applications written in Matlab onto reconfigurable platforms. In this chain, first the Matlab code is converted automatically to executable Kahn Process Network (KPN) specification. Then the tool called LAURA accepts this specification and transforms the specification into design implementations described as synthesizable VHDL. With the help of LAURA, we can fast prototype the DCT process directly in hardware — synthesizable VHDL code.

The bottom part of Figure 3.6 shows the sub-network of the hardware IP core for the DCT process which is generated by the LAURA tool. This sub-network includes four components — Node 1(ND_1), Node 2(ND_2), Node 3(ND_3), and Node 4(ND_4). They use the FIFO components to communicate with each other. However, this sub-network of the hardware IP core for the DCT process each time can only process one image block which includes four 8x8 sub-blocks — Y1 sub-block, Y2 sub-block, U1 sub-block and V1 sub-block. We need to use a reset signal of this sub-network to repetitively reset the sub-network in order to execute the DCT process for many image blocks. The architecture which shows how we use the reset signal is shown in the middle of Figure 3.6. When the sub-network finishes processing the DCT for one image block, it sends a stop signal to Flipflop1. Flipflop1 is used to store the stop signal. There is a logic element which is used to delay the stop signal for a certain time. The *Counter* component in the middle of Figure 3.6 is this logic element. Then Flipflop2 is set by the stop signal to start the sub-network of the hardware IP core for the DCT process. This system is a hierarchical architecture, because there are three components on this embedded system platform — two *MicroBlaze* processors — MB1, MB3 and one dedicated hardware IP core for the DCT process, and the dedicated hardware IP core for the DCT process is a sub-network on this embedded system platform which includes four components — Node 1(ND_1), Node 2(ND_2), Node 3(ND_3) and Node 4(ND_4). What's more, this sub-network is reset repetitively to apply the DCT operation on many image blocks.

Because we use Xilinx Platform Studio (XPS) as a back-end tool of our ESPAM tool, after we have got the hardware — synthesizable VHDL code of the DCT process, we still need to generate the pcore of the DCT process in order to make this hardware IP core can be used in XPS. There are two directories named *data* and *hdl* in the pcore directory of the DCT process.

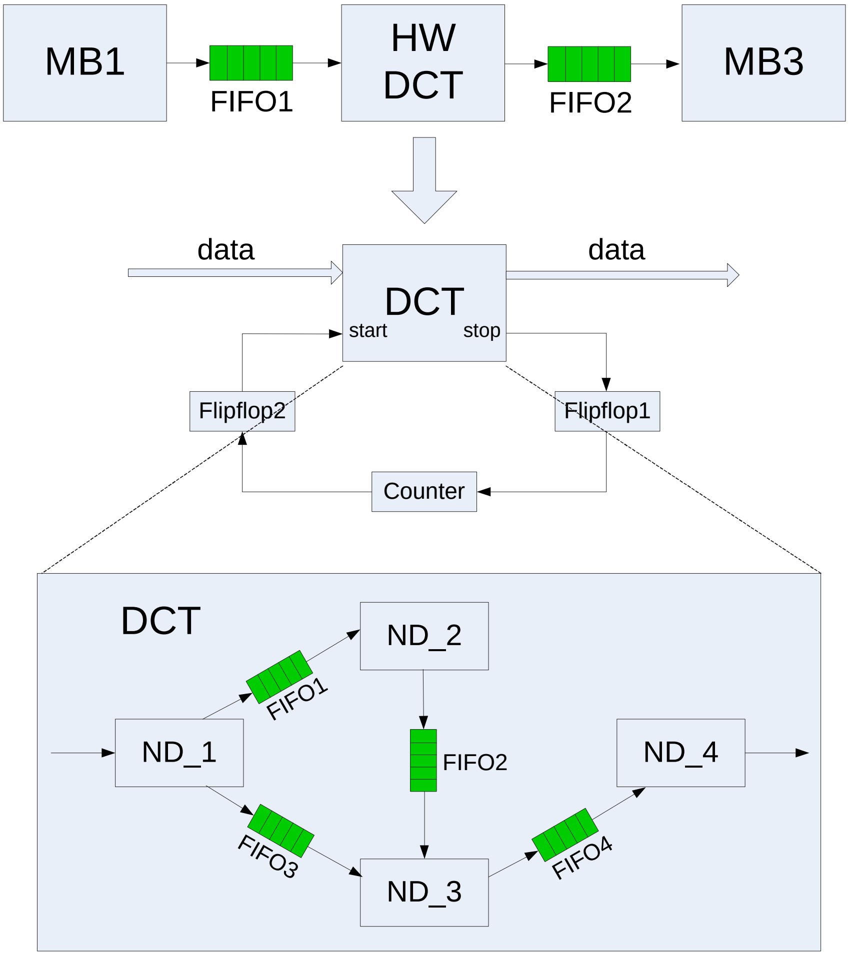


Figure 3.6: The heterogeneous and hierarchical architecture with one dedicated hardware IP core for the DCT process.

We need to generate two files for the pcore of the DCT process which are stored in the *data* directory: a Microprocessor Peripheral Definition (MPD) file [25] and a Peripheral Analyze Order (PAO) file [25]. The MPD file defines the interface of the DCT and the PAO file contains a list of HDL files that are needed for synthesis, and defines the analyze order for compilation. The VHDL source code files of the DCT process are stored in the *hdl* directory. The main contents of the MPD file and the PAO file are shown in Figure 3.7. In the MPD file the ports *RD_CLK*, *RD_EN*, *RD_CONTROL*, *RD_DATA* and *RD_EXISTS* are used to read data from a FIFO. The ports *WR_CLK*, *WR_EN*, *WR_CONTROL*, *WR_DATA* and *WR_FULL* are used to write data to a FIFO. The port *STATUS* is used to indicate that the DCT processes are finished. The port *CLK* and *RST* are the clock signal and reset signal. In the PAO file we have all the VHDL files that are needed for the DCT process and the analyze order for compilation. We can see that the top level of the DCT process is the VHDL file called *dct*. Thus all the top level architecture information of the DCT process is stored in this file. The pcore for the DCT process can be found in the CVS repository:
docs/students/WeiZhong/experiment/DCTpcore.zip

When we finish generating the pcore of the DCT process, based on the system with homoge-

```
BEGIN kpn

## Peripheral Options
...

## Bus Interfaces
BUS_INTERFACE BUS = SFSL, BUS_STD = FSL, BUS_TYPE = SLAVE
BUS_INTERFACE BUS = MFSL, BUS_STD = FSL, BUS_TYPE = MASTER

## Generics for VHDL or Parameters for Verilog

## Ports
PORT RD_CLK = FSL_S_Clk, DIR = out, SIGIS = CLK, BUS = SFSL
PORT RD_EN = FSL_S_Read, DIR = out, BUS = SFSL
PORT RD_CONTROL = FSL_S_Control, DIR = in, BUS = SFSL
PORT RD_DATA = FSL_S_Data, DIR = in, VEC = [31:0], BUS = SFSL, ENDIAN = LITTLE
PORT RD_EXISTS = FSL_S_Exists, DIR = in, BUS = SFSL
PORT WR_CLK = FSL_M_Clk, DIR = out, SIGIS = CLK, BUS = MFSL
PORT WR_EN = FSL_M_Write, DIR = out, BUS = MFSL
PORT WR_CONTROL = FSL_M_Control, DIR = out, BUS = MFSL
PORT WR_DATA = FSL_M_Data, DIR = out, VEC = [31:0], BUS = MFSL, ENDIAN = LITTLE
PORT WR_FULL = FSL_M_Full, DIR = in, BUS = MFSL
PORT STATUS = "", DIR = O
PORT CLK = "", DIR = I
PORT RST = "", DIR = I

END
```

MPD file

```
lib kpn_v1_00_a counter vhdl
lib kpn_v1_00_a decode_5 vhdl
lib kpn_v1_00_a fifo_cam_cntrl_c vhdl
lib kpn_v1_00_a fifo_cam_cntrl_p vhdl
...
lib kpn_v1_00_a dct vhdl
```

PAO file

Figure 3.7: The main contents of the MPD file and the PAO file of the pcore for the DCT process.

neous architecture which has been created in Section 3.2.1, we just need to copy the pcore of the dedicated hardware IP core for the DCT process to the system and replace the processor MB2 with this dedicated hardware IP core for the DCT process in the system by hand. It is possible for our ESPAM tool to automatically implement the work which is described above. In this thesis, we just focus on showing the procedure about how to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture. The implementation in our ESPAM tool is straightforward and it is out of the scope of this thesis. After this step, we have already finish creating the system with heterogeneous and hierarchical architecture.

3.2.3 Testing the System with Heterogeneous and Hierarchical Architecture

In this section, we explain how to test the system and compare the performance of this system with the performance of the system which is the homogeneous architecture.

```
int main (){

    TBlocks blocks = {
     ... ,
     ... ,
     ... ,
     ...
    };

    for (int i = 0; i < 6; i++) {

        writeFSL(0, &blocks, (sizeof(blocks)+(sizeof(blocks)%4)+3)/4);

    }

    ...
} // main
```

MB1

```
int main (){

    volatile long *compImage = (volatile long *)0xf0000000;

    TBlocks blocks;

    for (int i = 0; i < 6; i++) {

        readFSL(0, &blocks, (sizeof(blocks)+(sizeof(blocks)%4)+3)/4);

        for( int k =  0; k < 64; k += 1 ) {

            compImage[k + i * 64 * 4] = (volatile long) blocks.Y1.pixel[k];
            compImage[k + 64 + i * 64 * 4] = (volatile long) blocks.Y2.pixel[k];
            compImage[k + 64*2 + i * 64 * 4] = (volatile long) blocks.U1.pixel[k];
            compImage[k + 64*3 + i * 64 * 4] = (volatile long) blocks.V1.pixel[k];

        }
    }

    ...
} // main
```

MB3

Figure 3.8: The main code of MB1 and MB3 for testing many image blocks.

In order to compare the performances of the two systems, we use the same image block which is used in the system with homogeneous architecture in the system with heterogeneous and hierarchical architecture. We use MB1 to generate the image block and send this image block to the dedicated hardware IP core for the DCT process. When the hardware IP core finishes, it sends the resulting data to MB3 and then MB3 writes the data to the off-chip memory. This procedure is almost the same as the procedure which is done in Section 3.2.1. The only difference is that in this procedure we use the dedicated hardware IP core instead of *MicroBlaze* processor to do the DCT process. As a result, we can get the correct resulting data and we find that this system with heterogeneous and hierarchical architecture needs less time to do the DCT process than the system which is the homogeneous architecture. Using the dedicated hardware IP core for the DCT process the system with heterogeneous and hierarchical architecture can get better time performance than the system with homogeneous architecture.

The other test we need to do is to test whether the system with heterogeneous and hierarchical architecture can do the DCT process for more than one image blocks. Thus we need to change the software code of MB1 and MB3 to send several image blocks to the dedicated hardware IP core for the DCT process and receive the resulting image blocks from the dedicated hardware IP core for the DCT process. The main code of MB1 and MB3 which has been changed is shown in Figure 3.8. In Figure 3.8 we see that MB1 sends 6 image blocks to the dedicated hardware IP core for the DCT process and MB3 receives 6 resulting image blocks from the dedicated hardware IP core for the DCT process and writes the 6 resulting image blocks to the off-chip memory. The test result shows that we can get 6 correct resulting image blocks. This means the system with heterogeneous and hierarchical architecture can do the DCT process with more than one image blocks. After these two tests, we have proven that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture and this heterogeneous and hierarchical architecture can get better performance than the homogeneous architecture. A more complex system with heterogeneous and hierarchical architecture example is given in Chapter 5.

Chapter 4

Interface of an Embedded System with the Outside World

Applications of modern embedded systems, such as the high throughput multimedia, imaging, and digital signal processing, always need to exchange data with the outside world. Due to this reason an efficient interface of an embedded system with the outside world is necessary for modern embedded system. In this chapter, we explain how to construct an efficient interface by using several memories and the approach about how to make the ESPAM tool be able to automatically generate the interface when it maps an application onto a multiprocessor platform.

In Section 4.1, we describe the target FPGA platform which our interface of an embedded system with the outside world is based on. In Section 4.2, we explain the construction of the interface and introduce the components included in the interface. In Section 4.3, the approach about how to make ESPAM automatically generate our interface when it maps an application onto a multiprocessor platform is presented.

4.1 Target FPGA platform

The target FPGA platform on which we implement our interface of an embedded system with the outside world is the ADM-XRC-II board that is developed by Alpha Data Parallel Systems Ltd [26]. The ADM-XRC-II is a high performance PCI Mezzanine Card (PMC) format device designed for supporting development of applications using the Virtex-II series of FPGAs from Xilinx. The architecture of the ADM-XRC-II board is shown in Figure 4.1.

The ADM-XRC-II supports high performance PCI operations without the need to integrate proprietary cores into the FPGA. A PLX PCI9656 provides a rich set of PCI resources including two high-speed DMA controllers. We can use this PCI interface to communicate with outside host processors via the PCI bus. The features of the ADM-XRC-II board are listed below:

- Physically compatible to IEEE P1386 Common Mezzanine Card standard
- High performance PCI and DMA controllers

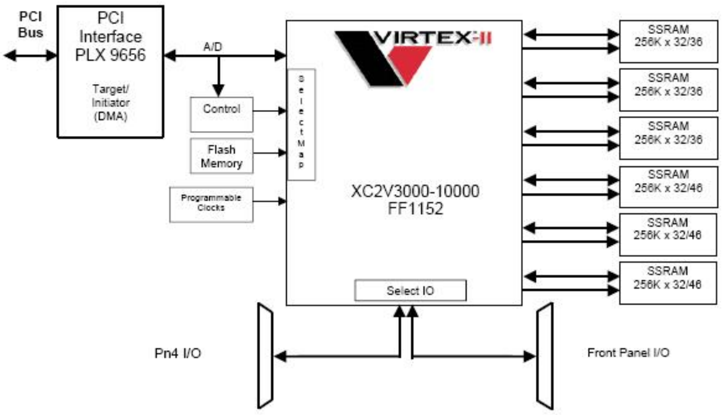


Figure 4.1: The architecture of the ADM-XRC-II board.

- Local bus speeds of up to 66MHz
- Six banks of 256k/512kx32/36 ZBT SSRAM
- User clock programmable between 0.5MHz and 100MHz
- User front panel adapter with up to 146 free IO signals
- User rear panel PMC connector with 64 free IO signals
- Supports 3.3V and 5V PCI signalling levels (VI/O)

From the specification, we see that this FPGA board has six banks of ZBT SSRAM which are off-chip memories. This type of off-chip memory is the Zero Bus Turnaround (ZBT) SSRAM that employs high-speed, low-power CMOS designs using an advanced CMOS process. These SSRAMs are optimized for 100 percent bus utilization, eliminating any turnaround cycles for READ to WRITE, or WRITE to READ, transitions. All synchronous inputs pass through registers controlled by a positive-edge-triggered single clock input (CLK). Our interface uses these six banks ZBT SSRAM which are off-chip memories to communicate with the outside world. In order to make the processors in a multiprocessor platform can access the off-chip ZBT SSRAM, we need to develop a custom controller to connect the processor to the off-chip ZBT SSRAM which is introduced in the next section.

4.2 Structure of the Interface of an Embedded System with the Outside World

In this section, we introduce the construction of an interface of an embedded system with the outside world by using several off-chip memories. The block diagram of the interface is shown in Figure 4.2. In Figure 4.2, we show that the interface of an embedded system with the outside world consists of four main parts — Host Interface, Function Design, Multiplexer and Buffer. The Function Design is a multiprocessor system which is used to implement different types of embedded system applications. Besides these four main parts, our interface still need two connection parts. One connection part is a custom controller for a processor in the Function Design to connect to the off-chip ZBT SSRAM which is the block *B1* in Figure 4.2. The other connection part includes two components which are used to transfer control signals and status signals between the Host Interface and the Function Design which are the block *B2* and block *B3* in Figure 4.2. All components included in the interface are introduced one by one in Section 4.2.1 to Section 4.2.5. The more detailed explanation about the components included in the interface is given in [27].

As described above, this interface can be used to communicate data between embedded systems and the outside world, such as an outside host processor, via the off-chip memories. For example, this interface can be used in this way: first an outside host processor, such as Pentium, can store data in the off-chip memories using the Host Interface. Then an application which has been mapped onto an embedded system platform, which is the Function Design, can read the data from the off-chip memories using the custom controllers (*B1*) and execute the tasks. At last, when the application finishes the tasks it can store the resulting data in the off-chip memories using the custom controllers (*B1*) and the outside host processor can read the resulting data back from the off-chip memories using the Host Interface.

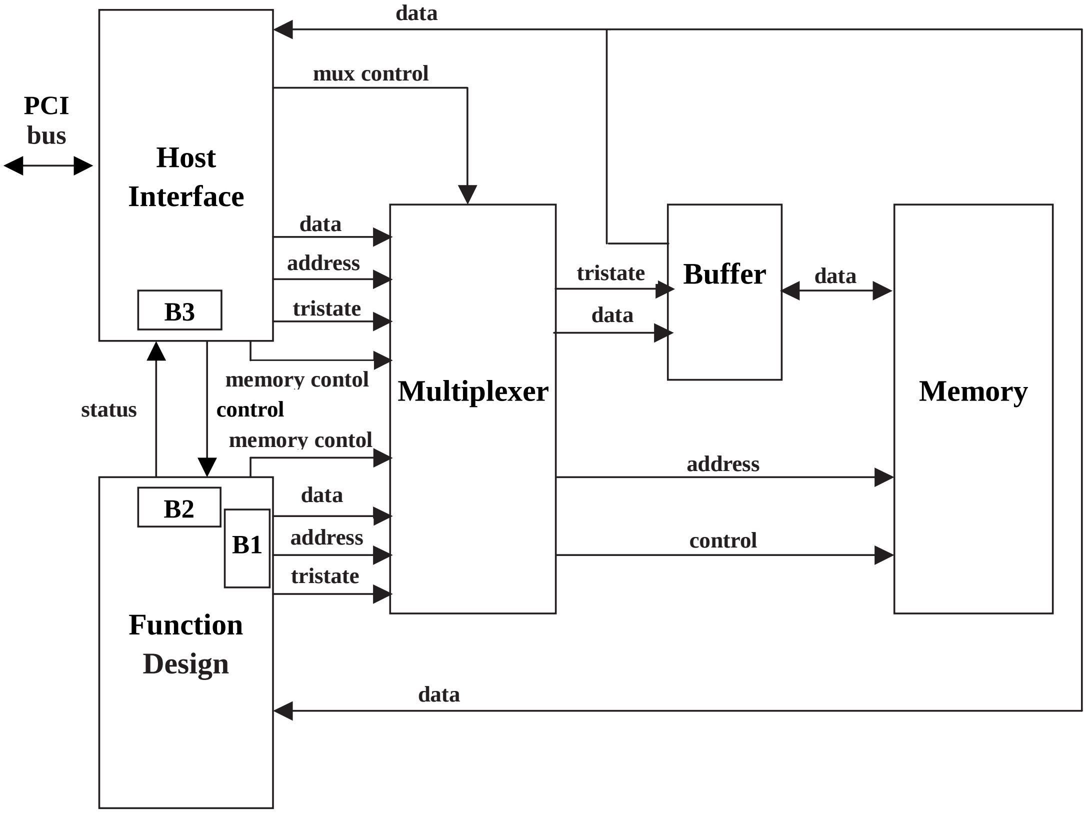


Figure 4.2: An interface of an embedded system with the outside world.

4.2.1 Host Interface

The Host Interface component uses PCI interface PLX 9656 to connect to an outside host processor, such as Pentium, with a PCI bus. An outside host processor uses the Host Interface component to write data to the off-chip ZBT SSRAMs and read data from the off-chip ZBT SSRAMs. The Host Interface component generates control signals to tell the Multiplexer component which part (the outside host processor or the Function Design component) needs to be connected to the off-chip ZBT SSRAMs. It also generates control signals to tell the Function Design component to start running and receive status signals from the Function Design component that indicates that the Function Design component has already finished the tasks.

In order to be able to use the Host Interface component in an XPS project, we need to create a pcore for the Host Interface component. We create a *zbt_main_v1_00_a* directory that includes all the files which the pcore of the Host Interface component requires, such as a Microprocessor Peripheral Definition (MPD) file, a Peripheral Analyze Order (PAO) file and VHDL source code files. In order to make the Host Interface component connectivity interface simpler, we add some bus interfaces in the MPD file. The main code of the MPD file of the Host Interface component is shown in Figure 4.3. As Figure 4.3 shows, we add a bus named *HOST_MUX_PORT* to bundle the signals that the Host Interface component uses to connect to the Multiplexer component and add the buses named *HOST_BUFF_0_PORT*, *HOST_BUFF_1_PORT*, *HOST_BUFF_2_PORT*, *HOST_BUFF_3_PORT*, *HOST_BUFF_4_PORT*, and *HOST_BUFF_5_PORT* to bundle the signals that the Host Interface component uses to connect to the Buffer component. The port *COMMAND_REG* is used to send control signals to the Multiplexer component or the Function Design component. The port *DESIGN_STAT_REG* is used to receive status signals from the Function Design component.

4.2.2 Multiplexer

The function of the Multiplexer component is to switch signals from the Host Interface component or signals from the Function Design component according to the control signals given by the Host Interface component. We need to create a pcore for the Multiplexer component. We create a *mux_v1_00_a* directory that includes all the files and directories which the pcore of the Multiplexer component requires, such as a Microprocessor Peripheral Definition (MPD) file, a Peripheral Analyze Order (PAO) file and VHDL source code files. We also need to add some bus interfaces in the MPD file of the Multiplexer component in order to make the Multiplexer component connectivity interface simpler. The main code of the MPD file of the Multiplexer component is shown in Figure 4.4. As Figure 4.4 shows, we add a bus named *MUX_HOST_PORT* to bundle the signals that the Multiplexer component uses to connect to the Host Interface component. We add the buses named *MUX_DESIGN_0_PORT*, *MUX_DESIGN_1_PORT*, *MUX_DESIGN_2_PORT*, *MUX_DESIGN_3_PORT*, *MUX_DESIGN_4_PORT*, and *MUX_DESIGN_5_PORT* to bundle the signals that the Multiplexer component uses to connect to the Function Design component and add a bus named *MUX_BUFF_PORT* to bundle the signals that the Multiplexer component uses to connect to the Buffer component. The port *CNTRL* is used to receive control signals from the Host Interface component. In the MPD file of the Multiplexer component we also add a parameter named *N_MUX* which is used to tell the Multiplexer component how many multiplexer units it needs to generate. The maximum value

```
BEGIN zbt_main

## Peripheral Options
...

## Bus Interfaces
BUS_INTERFACE BUS = HOST_MUX_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = HOST_BUFF_0_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = HOST_BUFF_1_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = HOST_BUFF_2_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = HOST_BUFF_3_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = HOST_BUFF_4_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = HOST_BUFF_5_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF

## Generics for VHDL or Parameters for Verilog
...

## Ports
...

PORT H_Data_R0 = RD_I, DIR = I, VEC = [31:0], ENDIAN = LITTLE, BUS = HOST_BUFF_0_PORT
PORT H_Data_W0 = H_DW0, DIR = O, VEC = [31:0], ENDIAN = LITTLE, BUS = HOST_MUX_PORT
PORT H_Tristate_0 = H_TRI0, DIR = O, VEC = [31:0], ENDIAN = LITTLE, BUS = HOST_MUX_PORT
PORT H_RA0 = H_AD0, DIR = O, VEC = [19:0], ENDIAN = LITTLE, BUS = HOST_MUX_PORT
PORT H_RC0 = H_CO0, DIR = O, VEC = [8:0], ENDIAN = LITTLE, BUS = HOST_MUX_PORT
...

PORT COMMAND_REG = "", DIR = O, VEC = [31:0], ENDIAN = LITTLE
...

PORT DESIGN_STAT_REG = "", DIR = I, VEC = [31:0], ENDIAN = LITTLE

END
```

Figure 4.3: The main code of the MPD file of the Host Interface component.

of parameter *N_MUX* is 6.

4.2.3 Buffer

The function of the Buffer component is to transfer data between the ZBT SSRAM memory and the Function Design component or the Host Interface component. We need to create a pcore for the Buffer component. We create a *buffers_v1_00_a* directory that includes all the files and directories which the pcore of the Buffer component requires, such as a Microprocessor Peripheral Definition (MPD) file, a Peripheral Analyze Order (PAO) file and VHDL source code files. We also need to add some bus interfaces in the MPD file of the Buffer component in order to make the Buffer component connectivity interface simpler. The main code of the MPD file of the Buffer component is shown in Figure 4.5. As Figure 4.5 shows, we add a bus named *BUFF_MUX_PORT* to bundle the signals that the Buffer component uses to connect to the Multiplexer component and the buses named *BUFF_RD_0_PORT*, *BUFF_RD_1_PORT*, *BUFF_RD_2_PORT*, *BUFF_RD_3_PORT*, *BUFF_RD_4_PORT*, and *BUFF_RD_5_PORT* to bundle the signals that the Buffer component uses to connect to the Host Interface component or the Function Design component.

```
BEGIN mux

## Peripheral Options
...

## Bus Interfaces
BUS_INTERFACE BUS = MUX_HOST_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = MUX_DESIGN_0_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = MUX_DESIGN_1_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = MUX_DESIGN_2_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = MUX_DESIGN_3_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = MUX_DESIGN_4_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = MUX_DESIGN_5_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = MUX_BUFF_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF

## Generics for VHDL or Parameters for Verilog
PARAMETER N_MUX = 1, DT = integer

## Ports
PORT H_DW0 = H_DW0, DIR = I, VEC = [31:0], ENDIAN = LITTLE, BUS = MUX_HOST_PORT, DEFAULT = H_DW0
PORT H_TRI0 = H_TRI0, DIR = I, VEC = [31:0], ENDIAN = LITTLE, BUS = MUX_HOST_PORT, DEFAULT = H_TRI0
PORT H_AD0 = H_AD0, DIR = I, VEC = [19:0], ENDIAN = LITTLE, BUS = MUX_HOST_PORT, DEFAULT = H_AD0
PORT H_CO0 = H_CO0, DIR = I, VEC = [8:0], ENDIAN = LITTLE, BUS = MUX_HOST_PORT, DEFAULT = H_CO0

PORT D_DW0 = D_DW, DIR = I, VEC = [31:0], ENDIAN = LITTLE, BUS = MUX_DESIGN_0_PORT, DEFAULT = D_DW
PORT D_TRI0 = D_TRI, DIR = I, VEC = [31:0], ENDIAN = LITTLE, BUS = MUX_DESIGN_0_PORT, DEFAULT = D_TRI
PORT D_AD0 = D_AD, DIR = I, VEC = [19:0], ENDIAN = LITTLE, BUS = MUX_DESIGN_0_PORT, DEFAULT = D_AD
PORT D_CO0 = D_CO, DIR = I, VEC = [8:0], ENDIAN = LITTLE, BUS = MUX_DESIGN_0_PORT, DEFAULT = D_CO

PORT DW0 = DW0, DIR = O, VEC = [31:0], ENDIAN = LITTLE, BUS = MUX_BUFF_PORT, DEFAULT = DW0
PORT TRI0 = TRI0, DIR = O, VEC = [31:0], ENDIAN = LITTLE, BUS = MUX_BUFF_PORT, DEFAULT = TRI0
PORT ra0 = "", DIR = O, VEC = [19:0], ENDIAN = LITTLE
PORT rc0 = "", DIR = O, VEC = [8:0], ENDIAN = LITTLE
...

PORT CNTRL = "", DIR = I, VEC = [31:0], ENDIAN = LITTLE

END
```

Figure 4.4: The main code of the MPD file of the Multiplexer component.

4.2.4 Custom Memory Controller

The custom memory controller which is the block *B1* in Figure 4.2 is used as an interface between a *MicroBlaze* processor and the ZBT SSRAM. Because we choose the IBM's On-chip Peripheral Bus (OPB) [28] as the bus interface of a *MicroBlaze* processor to connect to the off-chip ZBT SSRAM, the custom memory controller translates the OPB bus protocol into the ZBT SSRAM special protocol. In order to make our custom memory controller as a consistent interface to connect a *MicroBlaze* processor to the ZBT SSRAM, we also write a wrapper for our custom memory controller. Finally, we have got two VHDL files for our custom memory controller — *opb_zbt_controller_core.vhd* (the core VHDL file) and *opb_zbt_controller.vhd* (the wrapper VHDL file).

We need to create a pcore for our custom memory controller. We create a *opb_zbt_controller_v1_00_a* directory that includes all the files and directories which the pcore of the custom memory controller requires, such as a Microprocessor Peripheral Definition (MPD) file, a Peripheral Analyze Order (PAO) file and VHDL source code files. We also need to add some bus interfaces in the MPD file of the custom memory controller in order to make the custom memory controller connectivity interface simpler. The main code of the MPD file of the custom memory

```
BEGIN buffers

## Peripheral Options
...

## Bus Interfaces
BUS_INTERFACE BUS = BUFF_MUX_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = BUFF_RD_0_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = BUFF_RD_1_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = BUFF_RD_2_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = BUFF_RD_3_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = BUFF_RD_4_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = BUFF_RD_5_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF

## Generics for VHDL or Parameters for Verilog

## Ports
PORT I0 = RD_I, DIR = O, VEC = [31:0], ENDIAN = LITTLE, BUS = BUFF_RD_0_PORT, DEFAULT = RD_I
PORT O0 = DW0, DIR = I, VEC = [31:0], ENDIAN = LITTLE, BUS = BUFF_MUX_PORT
PORT T0 = TRI0, DIR = I, VEC = [31:0], ENDIAN = LITTLE, BUS = BUFF_MUX_PORT
PORT rd0 = "", DIR = IO, VEC = [31:0], ENDIAN = LITTLE, THREE_STATE=FALSE, IOB_STATE=BUF
...

END
```

Figure 4.5: The main code of the MPD file of the Buffer component.

controller is shown in Figure 4.6. As Figure 4.6 shows, we add a bus named *SOPB* to bundle the signals that the custom memory controller uses to connect to the OPB bus, add a bus named *DESIGN_BUFF_PORT* to bundle the signals that the custom memory controller uses to connect to the Buffer component and a bus named *DESIGN_MUX_PORT* to bundle the signals that the custom memory controller uses to connect to the Multiplexer component.

4.2.5 Transfer Components

In order to transfer control signals and status signals between the Host Interface component and the Function Design component, we need to develop two components — *fin_ctrl* component which is the block *B2* in Figure 4.2 and *host_design_ctrl* component which is the block *B3* in Figure 4.2. The *fin_ctrl* component is used to connect the *host_design_ctrl* component to *MicroBlaze* processors in the Function Design component using the Local Memory Buses (LMB) [29]. When a *MicroBlaze* processor finishes its tasks, it sends a finish signal to the *host_design_ctrl* component through the *fin_ctrl* component. The *host_design_ctrl* component is used to connect the Host Interface component to the Function Design component. The function of the *host_design_ctrl* component is to send the start signal to the Function Design component that is used to tell *MicroBlaze* processors to start to work. The *host_design_ctrl* component is also used to collect all the finish signals sent by *MicroBlaze* processors through the *fin_ctrl* components and when all of the *MicroBlaze* processors have already sent the finish signals to it, it will sent a final finish signal to the Host Interface component to tell an outside host processor that the Function Design component has already finished the tasks. We need to create the pcores for the *fin_ctrl* component and the *host_design_ctrl* component. We create a *fin_ctrl_v1_00_a* directory that include all the files and directories which the pcore of the *fin_ctrl* component requires, such as a MPD file, a PAO file and VHDL source code files and a *host_design_ctrl_v1_00_a* directory that include all the files and directories which the pcore

```
BEGIN opb_zbt_controller

## Peripheral Options
...

## Bus Interfaces
BUS_INTERFACE BUS = SOPB, BUS_STD = OPB, BUS_TYPE = SLAVE
BUS_INTERFACE BUS = DESIGN_BUFF_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF
BUS_INTERFACE BUS = DESIGN_MUX_PORT, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF

## Generics for VHDL or Parameters for Verilog
...

## Ports
PORT OPB_Clk = "", DIR = IN, SIGIS = CLK, BUS = SOPB, DEFAULT =
PORT OPB_Rst = OPB_Rst, DIR = IN, BUS = SOPB, DEFAULT = OPB_Rst
PORT OPB_ABus = OPB_ABus, DIR = IN, VEC = [0:31], BUS = SOPB, DEFAULT = OPB_ABus
PORT OPB_BE = OPB_BE, DIR = IN, VEC = [0:3], BUS = SOPB, DEFAULT = OPB_BE
PORT OPB_RNW = OPB_RNW, DIR = IN, BUS = SOPB, DEFAULT = OPB_RNW
PORT OPB_select = OPB_select, DIR = IN, BUS = SOPB, DEFAULT = OPB_select
PORT OPB_seqAddr = OPB_seqAddr, DIR = IN, BUS = SOPB, DEFAULT = OPB_seqAddr
PORT OPB_DBus = OPB_DBus, DIR = IN, VEC = [0:31], BUS = SOPB, DEFAULT = OPB_DBus
PORT ZBT_DBus = Sl_DBus, DIR = OUT, VEC = [0:31], BUS = SOPB, DEFAULT = Sl_DBus
PORT ZBT_errAck = Sl_errAck, DIR = OUT, BUS = SOPB, DEFAULT = Sl_errAck
PORT ZBT_retry = Sl_retry, DIR = OUT, BUS = SOPB, DEFAULT = Sl_retry
PORT ZBT_toutSup = Sl_toutSup, DIR = OUT, BUS = SOPB, DEFAULT = Sl_toutSup
PORT ZBT_xferAck = Sl_xferAck, DIR = OUT, BUS = SOPB, DEFAULT = Sl_xferAck

...

PORT RC_O = D_CO, DIR = O, VEC = [0:8], BUS = DESIGN_MUX_PORT
PORT RA_O = D_AD, DIR = OUT, VEC = [0:C_ZBT_ADDR_SIZE-1], BUS = DESIGN_MUX_PORT
PORT RD_I = RD_I, DIR = I, VEC = [0:31], BUS = DESIGN_BUFF_PORT
PORT RD_O = D_DW, DIR = O, VEC = [0:31], BUS = DESIGN_MUX_PORT
PORT T_RD = D_TRI, DIR = O, VEC = [0:31], BUS = DESIGN_MUX_PORT

END
```

Figure 4.6: The main code of the MPD file of the custom memory controller.

of the *host_design_ctrl* component requires, such as a MPD file, a PAO file and VHDL source code files. The main code of the MPD files of the *fin_ctrl* component and the main code of the MPD files of the *host_design_ctrl* component are shown in Figure 4.7 and Figure 4.8. As Figure 4.7 shows, in the *fin_ctrl* component we add a bus named *SLMB* to bundle the signals that the *fin_ctrl* component uses to connect to the LMB bus. The port *Sl_FinOut* is used to send the finish signal to the *host_design_ctrl* component. As Figure 4.8 shows, in the *host_design_ctrl* component the port *COMMAND_REG* is used to receive the control signals from the Host Interface component. The ports from *FIN_REG_0* to *FIN_REG_19* are used to receive the finish signals from the *fin_ctrl* components. The port *RST_OUT* is used to reset the Function Design component, in other words it is used to tell the Function Design component to start to work. The port *STATUS_REG* is used to send the final finish signal to the Host Interface component to tell an outside host processor that the Function Design component has already finished the tasks. We also add a parameter named *N_FIN* which is used to tell the *host_design_ctrl* component how many *fin_ctrl* components need to connect to it. The maximum number of the parameter *N_FIN* is twenty.

```
BEGIN fin_ctrl

## Peripheral Options
...

## Bus Interfaces
BUS_INTERFACE BUS = SLMB, BUS_TYPE = SLAVE, BUS_STD = LMB

## Generics for VHDL or Parameters for Verilog
...

## Ports
PORT LMB_Clk = "", DIR = I, BUS = SLMB
PORT LMB_Rst = LMB_Rst, DIR = I, BUS = SLMB
PORT LMB_ABus = LMB_ABus, DIR = I, VEC = [0:(C_LMB_AWIDTH-1)], BUS = SLMB
PORT LMB_WriteDBus = LMB_WriteDBus, DIR = I, VEC = [0:(C_LMB_DWIDTH-1)], BUS = SLMB
PORT LMB_AddrStrobe = LMB_AddrStrobe, DIR = I, BUS = SLMB
PORT LMB_ReadStrobe = LMB_ReadStrobe, DIR = I, BUS = SLMB
PORT LMB_WriteStrobe = LMB_WriteStrobe, DIR = I, BUS = SLMB
PORT LMB_BE = LMB_BE, DIR = I, VEC = [0:((C_LMB_DWIDTH/8)-1)], BUS = SLMB
PORT Sl_DBus = Sl_DBus, DIR = O, VEC = [0:(C_LMB_DWIDTH-1)], BUS = SLMB
PORT Sl_Ready = Sl_Ready, DIR = O, BUS = SLMB

PORT Sl_FinOut = "", DIR = O

END
```

Figure 4.7: The main code of the MPD files of the *fin_ctrl* component.

```
BEGIN host_design_ctrl

## Peripheral Options
...

## Bus Interfaces

## Generics for VHDL or Parameters for Verilog
PARAMETER N_FIN = 1, DT = integer

## Ports
...
PORT COMMAND_REG = "", DIR = I, VEC = [31:0], ENDIAN = LITTLE
PORT FIN_REG_0 = "", DIR = I
...
PORT FIN_REG_19 = "", DIR = I
PORT RST_OUT = "", DIR = O
PORT STATUS_REG = "", DIR = O, VEC = [31:0], ENDIAN = LITTLE

END
```

Figure 4.8: The main code of the MPD files of the *host_design_ctrl* component.

4.3 Generating the Interface of an Embedded System with the Outside World

In Section 4.2, we introduced the structure of the interface of an embedded system with the outside world. This interface can be used for data exchange between the embedded system and the outside world, such as an outside host processor, via the off-chip memories. In this section, we explain the approach about how to make the ESPAM tool be able to automatically generate the interface when it maps an application onto a multiprocessor platform.

First, we need to add a new group of generic parameterized components named *Peripheral Com-*

ponents in the platform model of our ESPAM tool. In the *Peripheral Components* we need to add our custom memory controller which is used as an interface between a *MicroBlaze* processor and the ZBT SSRAM. For the sake of making a processor communicate with an outside terminal, in the *Peripheral Components* we also need to add the UART (Universal Asynchronous Receiver Transmitter) which can control the serial port of the FPGA board to communicate with an outside terminal. Moreover, we need to add the OPB (IBM's On-chip Peripheral Bus) port which is used by our custom memory controller and the UART in the platform model. In order to add such components in the platform model of our ESPAM tool, we need to create a new class named *Peripheral*, a new class named *ZBTMemoryController* which is the class for our custom memory controller and it extents *Peripheral* class, a new class named *Uart* which is the class for the Universal Asynchronous Receiver Transmitter and it also extents *Peripheral* class, a new class named *OPBPort* which is the class for the OPB port in the data model of our ESPAM tool. The second step is to modify the platform specification parser of our ESPAM tool. In this step, we need to modify the platform specification parser to make it parse the peripheral components such as our custom memory controller and the UART when we specify such peripheral components in a platform specification. The third step is to modify our *MhsVisitor* class which is used to generate a Microprocessor Hardware Specification (MHS) file for an XPS project and our *MssVisitor* class which is used to generate a Microprocessor Software Specification (MSS) file for an XPS project. In the *MhsVisitor* class, first we generate the external port for our interface to connect to the PCI bus in a MHS file. Second, every time we generate a processor component in the MHS file we also generate a *fin_ctrl* component. Third, we make the *MhsVisitor* class visit the data model to get the information of our custom memory controllers and the UARTs and generate these two types of components in the MHS file. Fourth, when we generate our custom memory controllers in the MHS file we also generate the Host Interface component, the Multiplexer component, the Buffer component, and the *host_design_ctrl* component in the MHS file. But the Host Interface component, the Multiplexer component, the Buffer component, and the *host_design_ctrl* component are just generated once in the MHS file. In the *MssVisitor* class, first every time we generate a processor component in a MSS file we also generate a *fin_ctrl* component. Second, we also make this class can visit the data model to get the information of our custom memory controllers and the UARTs and generate these two types of components in the MSS file. Third, when we generate our custom memory controllers in the MSS file we also generate the Host Interface component, the Multiplexer component, the Buffer component, and the *host_design_ctrl* component in the MSS file. But also the Host Interface component, the Multiplexer component, the Buffer component, and the *host_design_ctrl* component are just generated once in the MSS file.

By implementing the steps explained above, our ESPAM tool can automatically generate the interface of an embedded system with the outside world when it maps an application onto a multiprocessor platform. For example, when we give the platform specification shown in Figure 4.9, our ESPAM tool can automatically generate the interface. In this platform specification, we specify three *MicroBlaze* processors (*MB_1*, *MB_2* and *MB_3*), one UART (*RS232_Uart_1*) and three custom memory controllers (*ZBT_CTRL_1*, *ZBT_CTRL_2* and *ZBT_CTRL_3*) which are used as the interfaces between the *MicroBlaze* processors and the ZBT SSRAMs. Each processor has 16K data memory, 8K program memory and the OPB port. *MB_1* uses the link *mb_opb_1* to connect to the *ZBT_CTRL_1* and *RS232_Uart_1* via the OPB bus. *MB_2* uses the link *mb_opb_2* to connect to the *ZBT_CTRL_2* via the OPB bus. *MB_3* uses the link *mb_opb_3*

```
<platform name="myPlatform">

<processor name="MB_1" type="MB" data_memory="16384" program_memory="8192">
    <port name="OPB_1" type="OPBPort"/>
</processor>

<processor name="MB_2" type="MB" data_memory="16384" program_memory="16384">
    <port name="OPB_2" type="OPBPort"/>
</processor>

<processor name="MB_3" type="MB" data_memory="8192" program_memory="8192">
    <port name="OPB_3" type="OPBPort"/>
</processor>

<peripheral name="ZBT_CTRL_1" type="ZBTCTRL" size="1000000">
    <port name="IO_1" type="OPBPort"/>
</peripheral>

<peripheral name="RS232_Uart_1" type="UART" size="256">
    <port name="UARTIO_1" type="OPBPort"/>
</peripheral>

<peripheral name="ZBT_CTRL_2" type="ZBTCTRL" size="1000000">
    <port name="IO_2" type="OPBPort"/>
</peripheral>

<peripheral name="ZBT_CTRL_3" type="ZBTCTRL" size="1000000">
    <port name="IO_3" type="OPBPort"/>
</peripheral>

<link name="mb_opb_1">
    <resource name="MB_1" port="OPB_1"/>
    <resource name="ZBT_CTRL_1" port="IO_1"/>
    <resource name="RS232_Uart_1" port="UARTIO_1"/>
</link>

<link name="mb_opb_2">
    <resource name="MB_2" port="OPB_2"/>
    <resource name="ZBT_CTRL_2" port="IO_2"/>
</link>

<link name="mb_opb_3">
    <resource name="MB_3" port="OPB_3"/>
    <resource name="ZBT_CTRL_3" port="IO_3"/>
</link>

</platform>
```

Figure 4.9: An example of a platform specification.

to connect to the *ZBT_CTRL_3* via the OPB bus. Our ESPAM tool automatically generates the components which the interface needs, such as the Host Interface component, the Multiplexer component, the Buffer component, the *fin_ctrl* components, and the *host_design_ctrl* component. More complex example which generates the interface using our ESPAM tool is given in Chapter 5.

Chapter 5

Case Studies

In this chapter, we present two case studies. The first case study is about M-JPEG multiprocessor system with homogeneous architecture which is used to evaluate the design methodology in our ESPAM tool presented in Chapter 2 and to validate the interface of an embedded system with the outside world explained in Chapter 4. The second case study is about M-JPEG multiprocessor system with heterogeneous and hierarchical architecture which is used to validate the procedure of implementing an embedded system as heterogeneous and hierarchical architecture and to evaluate the heterogeneous and hierarchical architecture introduced in Chapter 3. Based on the results which are obtained from the experiments in these two case studies, we present an analysis and comments on these results.

5.1 M-JPEG Homogeneous Multiprocessor System

In this case study, we use a complex application, namely a modified Motion JPEG (M-JPEG) encoder which is mapped onto multiprocessor embedded system platform with homogeneous architecture. Just as the traditional M-JPEG encoder, this modified M-JPEG encoder compresses a sequence of video frames, using JPEG [30] [31] picture compression in each frame of the video. This modified M-JPEG encoder processes video data which is in the 4:2:2 YUV format.

Figure 5.1 shows the initial Matlab code of this M-JPEG encoder application. In line 1 to line 3, it specifies the parameters which are named *NumFrames*, *VNumBlocks* and *HNumBlocks*. The parameter *NumFrames* stands for the number of frames to be processed and it ranges from 1 to 100. The parameter *VNumBlocks* stands for the vertical size of a frame in number of 8×8-pixel blocks and it ranges from 2 to 100. The parameter *HNumBlocks* stands for the horizontal size of a frame in number of 8×8-pixel blocks and it ranges from 1 to 100. Lines 5-15 define some types of data which are used in the code. Lines 17-23 initialize the luminance and chrominance quantization table (*QTables*) and luminance and chrominance Huffman table (*HuffTableAC*) and so on. First, the *VideoInMain()* function divides the frames in YUV format in 8×8-pixel blocks. Thus, every block is a 4:2:2 YUV block. After that each frame is compressed using the standard JPEG compression algorithm. The Discrete Cosine Transform (*DCT*) is applied on every 4:2:2 YUV block - line 30, followed by quantization (*Q*) and variable-length encoding (*VLE*) - lines 31-34. Function *VideoOut()* in lines 35-36 is used to add the header information to the compressed frame.

```
1   %parameter NumFrames 1 100;
2   %parameter VNumBlocks 2 100;
3   %parameter HNumBlocks 1 100;
4
5   %typedef HeaderInfo             THeaderInfo;
6   %typedef LuminanceQTable        TQTables;
7   %typedef ChrominanceQTable      TQTables;
8   %typedef LuminanceHuffTableDC   THuffTablesDC;
9   %typedef ChrominanceHuffTableDC THuffTablesDC;
10  %typedef LuminanceHuffTableAC   THuffTablesAC;
11  %typedef ChrominanceHuffTableAC THuffTablesAC;
12  %typedef LuminanceTablesInfo    TTablesInfo;
13  %typedef ChrominanceTablesInfo  TTablesInfo;
14  %typedef Packets                TPackets;
15  %typedef Block                  TBlocks;
16
17  for k = 1:1:1,
18    [  LuminanceQTable,     ChrominanceQTable,
19       LuminanceHuffTableDC,ChrominanceHuffTableDC,
20       LuminanceHuffTableAC,ChrominanceHuffTableAC,
21       LuminanceTablesInfo, ChrominanceTablesInfo
22    ] = DefaultTables();
23  end
24
25  for k = 1:1:NumFrames,
26    [  HeaderInfo  ] = VideoInInit();
27    for j = 1:1:VNumBlocks,
28      for i = 1:1:HNumBlocks,
29        [ Block ] = VideoInMain();
30        [ Block ] = DCT( Block );
31        [ Block ] = Q( Block, LuminanceQTable, ChrominanceQTable );
32        [ Packets ] = VLE( Block,
33                           LuminanceHuffTableDC,ChrominanceHuffTableDC,
34                           LuminanceHuffTableAC,ChrominanceHuffTableAC );
35        [  dummy  ] = VideoOut( HeaderInfo, LuminanceTablesInfo,
36                                ChrominanceTablesInfo, Packets );
37      end
38    end
39  end
```

Figure 5.1: The initial Matlab code of the M-JPEG encoder application.

First, we need to convert the initial Matlab code which is shown in Figure 5.1 into a KPN specification. We use the Compaan tool [2] to automatically transform the Matlab code of the M-JPEG encoder application which is specified in a sequential model of computation into a KPN model of computation making the task-level parallelism available in the M-JPEG encoder application explicit. The KPN of the M-JPEG encoder application which is generated by Compaan is shown in Figure 5.2. In this KPN specification of the M-JPEG encoder application, there are seven processes — *ND_1*, *ND_2*, *ND_3*, *ND_4*, *ND_5*, *ND_6* and *ND_7*. *ND_1* is the *DefaultTables()* process. *ND_2* is the *VideoInInit()* process. *ND_3* is the *VideoInMain()* process. *ND_4* is the *DCT()* process. *ND_5* is the *Q()* process. *ND_6* is the *VLE()* process. *ND_7* is the *VideoOut()* process. In this case study, we conduct two experiments to evaluate the design methodology in our Espam tool and validate the interface of an embedded system with the outside world.

In the first experiment, we map the M-JPEG encoder application onto the one-processor embedded system platform shown in Figure 5.3. In this case, actually there is no task-level par-

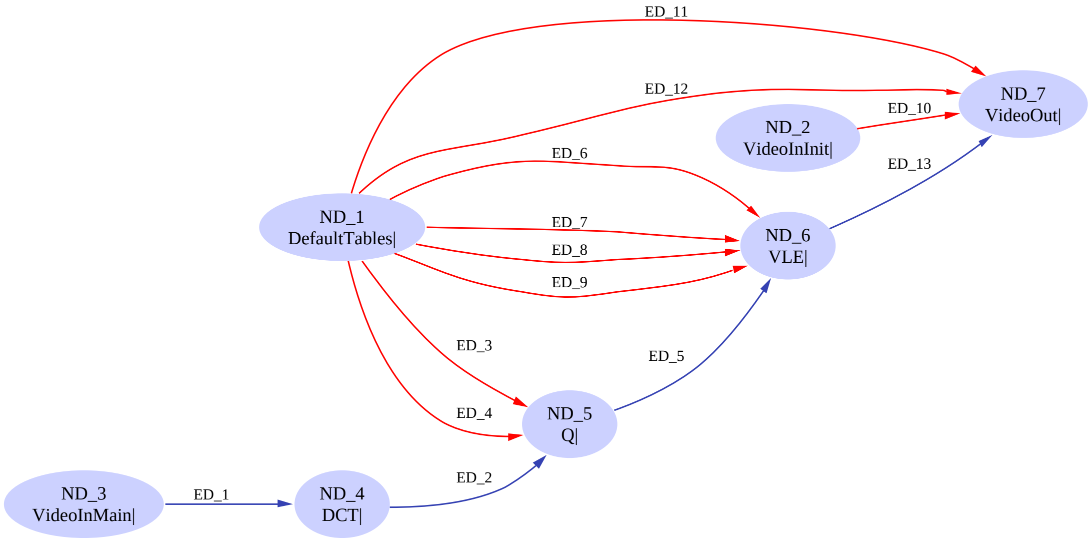


Figure 5.2: The KPN of the M-JPEG encoder application.

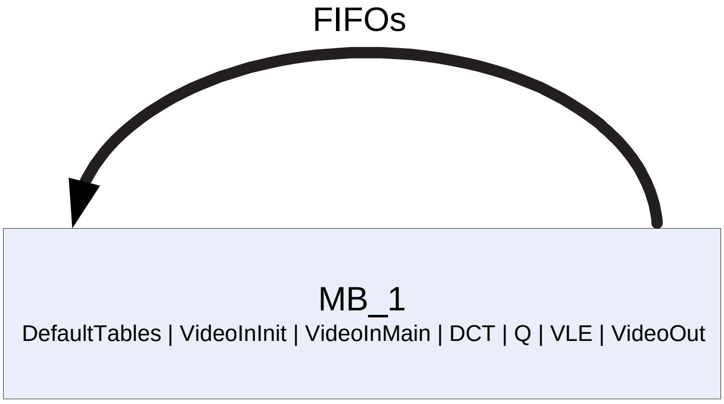


Figure 5.3: One-processor embedded system platform for M-JPEG encoder application.

allelism exploited in this embedded system as this is the case in the initial Matlab program. Based on the design methodology in our ESPAM tool, we still need to write the *Platform Specification* and the *Mapping Specification* shown in Figure 5.4 and Figure 5.5. In the *Platform Specification*, we see that there are one *MicroBlaze* processor (*MB_1*) and two custom memory controllers (*ZBT_CTRL_1* and *ZBT_CTRL_2*) which are used as the interfaces between the *MicroBlaze* processors and the ZBT SSRAMs in this embedded system platform. Because we use the ADM-XRC-II board as the target FPGA platform, there are six banks of ZBT SSRAM which are the off-chip memories on this FPGA board. The *MB_1* uses the two custom memory controllers — *ZBT_CTRL_1* and *ZBT_CTRL_2* to connect to two banks of ZBT SSRAM. *ZBT_CTRL_1* is used to read the initial video data from ZBT SSRAM and *ZBT_CTRL_2* is used to write the resulting video data to the ZBT SSRAM. We also set the data memory and program memory of *MB_1* to 64K. In the *Mapping Specification*, we map all of the processes which include *DefaultTables()* process (ND_1), *VideoInInit()* process (ND_2), *VideoInMain()* process (ND_3), *DCT()* process (ND_4), *Q()* process (ND_5), *VLE()* process (ND_6) and *VideoOut()* process (ND_7) in the KPN specification which is shown in Figure 5.2 onto one *MicroBlaze* processor — *MB_1*. In this experiment we use one video frame which size is 128×128 pixels to test the M-JPEG encoder embedded system. In order to make the embedded system be able to exchange video data with the outside world, we need to use the interface presented in Chapter 4 to communicate with an outside host processor and to store the video data in the off-chip memories. First, we use the outside host processor to store the video data in the first bank of ZBT SSRAM. Then processor *MB_1* uses the controller *ZBT_CTRL_1* to read the video data from this bank and starts to execute the M-JPEG process on this video frame. When processor *MB_1* finishes all of the tasks, it stores the resulting video data in the second bank of ZBT SSRAM using controller *ZBT_CTRL_2*. Finally, the outside host processor uses the interface to read back the resulting video data from such bank of ZBT SSRAM.

```
<platform name="myPlatform">

<processor name="MB_1" type="MB" data_memory="64000" program_memory="64000">
    <port name="OPB_1" type="OPBPort"/>
</processor>

<peripheral name="ZBT_CTRL_1" type="ZBTCTRL" size="1000000">
    <port name="IO_1" type="OPBPort"/>
</peripheral>

<peripheral name="ZBT_CTRL_2" type="ZBTCTRL" size="1000000">
    <port name="IO_2" type="OPBPort"/>
</peripheral>

<link name="mb_opb_1">
    <resource name="MB_1" port="OPB_1"/>
    <resource name="ZBT_CTRL_1" port="IO_1"/>
    <resource name="ZBT_CTRL_2" port="IO_2"/>
</link>

</platform>
```

Figure 5.4: *Platform Specification* for one-processor embedded system platform.

```
<mapping name="myMapping">

<processor name="MB_1">
    <process name="ND_1" />
    <process name="ND_2" />
    <process name="ND_3" />
    <process name="ND_4" />
    <process name="ND_5" />
    <process name="ND_6" />
    <process name="ND_7" />
</processor>

</mapping>
```

Figure 5.5: *Mapping Specification* for one-processor embedded system platform.

In the second experiment, we map the M-JPEG encoder application onto a five-processor embedded system platform shown in Figure 5.6. In this case, there are five parallel tasks which are executed concurrently in this embedded system platform. The *Platform Specification* and the *Mapping Specification* for this five-processor embedded system platform are shown in Figure 5.7 and Figure 5.8. In the *Platform Specification*, we see that there are five *MicroBlaze* processors (*MB_1*, *MB_2*, *MB_3*, *MB_4*, and *MB_5*) and five custom memory controllers (*ZBT_CTRL_1*, *ZBT_CTRL_2*, *ZBT_CTRL_3*, *ZBT_CTRL_4*, and *ZBT_CTRL_5*) which are used as the interfaces between the *MicroBlaze* processors and the ZBT SSRAMs in this embedded system platform.

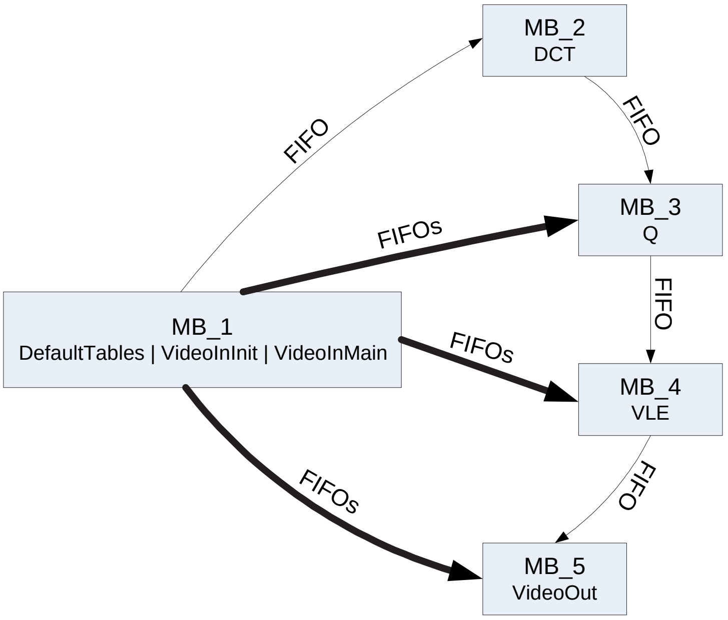


Figure 5.6: Five-processor embedded system platform for M-JPEG encoder application.

In this platform, each *MicroBlaze* processor uses one of the custom memory controllers to connect to one bank of ZBT SSRAM on the target FPGA platform. The *MB_1* uses *ZBT_CTRL_1* to read the initial video data from the ZBT SSRAM and the *MB_5* uses *ZBT_CTRL_5* to write the resulting video data to the ZBT SSRAM. We also set the data memory size and program memory size for each processor in the *Platform Specification*. In the *Mapping Specification*, we map *DefaultTables()* process (ND_1), *VideoInInit()* process (ND_2) and *VideoInMain()* process (ND_3) onto processor *MB_1*, *DCT()* process (ND_4) onto processor *MB_2*, *Q()* process (ND_5) onto processor *MB_3*, *VLE()* process (ND_6) onto processor *MB_4* and *VideoOut()* process (ND_7) onto processor *MB_5*. In this experiment we also use one video frame which size is 128×128 pixels to test the M-JPEG encoder embedded system. In order to make the embedded system be able to exchange video data with the outside world, we need to use the interface explained in Chapter 4 to communicate with an outside host processor and to store the video data in the off-chip memories. First, we use the outside host processor to store the video data in the first bank of ZBT SSRAM using the interface. Then processor *MB_1* uses the controller *ZBT_CTRL_1* to read the video data from this bank of ZBT SSRAM and the five *MicroBlaze* processors start to execute the M-JPEG process on this video frame. When all of the five processors finish all of the tasks, processor *MB_5* stores the resulting video data in the fifth bank of ZBT SSRAM using the controller *ZBT_CTRL_5*. Finally, the outside host processor uses the interface to read back the resulting video data from this bank of ZBT SSRAM.

In these two experiments, we use one video frame which size is 128×128 pixels to test these two M-JPEG encoder embedded systems. The performances of these two M-JPEG encoder embedded systems is shown in Figure 5.9. The frequency of the processors in this case study is 100MHz. Comparing the performances of these two experiments, we see that the second experiment which maps the M-JPEG encoder application onto five-processor embedded system platform is about 2 times faster than the first experiment which maps the M-JPEG encoder application onto one-processor embedded system platform. The first experiment uses one processor

```
<platform name="myPlatform">

<processor name="MB_1" type="MB" data_memory="65536" program_memory="32768">
    <port name="OPB_1" type="OPBPort"/>
</processor>
<processor name="MB_2" type="MB" data_memory="16384" program_memory="16384">
    <port name="OPB_2" type="OPBPort"/>
</processor>
<processor name="MB_3" type="MB" data_memory="8192" program_memory="8192">
    <port name="OPB_3" type="OPBPort"/>
</processor>
<processor name="MB_4" type="MB" data_memory="16384" program_memory="16384">
    <port name="OPB_4" type="OPBPort"/>
</processor>
<processor name="MB_5" type="MB" data_memory="16384" program_memory="16384">
    <port name="OPB_5" type="OPBPort"/>
</processor>

<peripheral name="ZBT_CTRL_1" type="ZBTCTRL" size="1000000">
    <port name="IO_1" type="OPBPort"/>
</peripheral>
<peripheral name="ZBT_CTRL_2" type="ZBTCTRL" size="1000000">
    <port name="IO_2" type="OPBPort"/>
</peripheral>
<peripheral name="ZBT_CTRL_3" type="ZBTCTRL" size="1000000">
    <port name="IO_3" type="OPBPort"/>
</peripheral>
<peripheral name="ZBT_CTRL_4" type="ZBTCTRL" size="1000000">
    <port name="IO_4" type="OPBPort"/>
</peripheral>
<peripheral name="ZBT_CTRL_5" type="ZBTCTRL" size="1000000">
    <port name="IO_5" type="OPBPort"/>
</peripheral>

<link name="mb_opb_1">
    <resource name="MB_1" port="OPB_1"/>
    <resource name="ZBT_CTRL_1" port="IO_1"/>
</link>
<link name="mb_opb_2">
    <resource name="MB_2" port="OPB_2"/>
    <resource name="ZBT_CTRL_2" port="IO_2"/>
</link>
<link name="mb_opb_3">
    <resource name="MB_3" port="OPB_3"/>
    <resource name="ZBT_CTRL_3" port="IO_3"/>
</link>
<link name="mb_opb_4">
    <resource name="MB_4" port="OPB_4"/>
    <resource name="ZBT_CTRL_4" port="IO_4"/>
</link>
<link name="mb_opb_5">
    <resource name="MB_5" port="OPB_5"/>
    <resource name="ZBT_CTRL_5" port="IO_5"/>
</link>

</platform>
```

Figure 5.7: *Platform Specification* for five-processor embedded system platform.

to execute the M-JPEG encoder application and the second experiment uses five processors which run concurrently to execute the M-JPEG encoder application. Thus, the platform in the second experiment should be 5 times faster than the first experiment theoretically. However, in Figure 5.9 we see that actually the second experiment is just 2 times faster than the first experiment. The first reason is that the tasks which are executed in each processor in the second experiment are not balanced. Table 5.1 shows how many clock cycles and the utilization percent

```
<mapping name="myMapping">

<processor name="MB_1">
    <process name="ND_1" />
    <process name="ND_2" />
    <process name="ND_3" />
</processor>

<processor name="MB_2">
    <process name="ND_4" />
</processor>

<processor name="MB_3">
    <process name="ND_5" />
</processor>

<processor name="MB_4">
    <process name="ND_6" />
</processor>

    <processor name="MB_5">
    <process name="ND_7" />
</processor>

</mapping>
```

Figure 5.8: *Mapping Specification* for five-processor embedded system platform.

of each process, which is executed by one processor in the second experiment, need to take in order to process one block image which includes 8×8 pixels. We see that the processes which are executed by the five processors are not balanced. The *DCT()* process takes more than 50 percent of the whole time, but the *VideoInMain()* process just takes 4.1 percent and *VideoOut()* process just takes 0.7 percent of the whole time. Thus, in this case the *DCT()* process is the bottleneck of the whole system. The second reason is that in the second experiment, the five processors have to spend time in communicating with each other. In contrast, the first experiment just includes one processor and it saves lots of time in the communication.

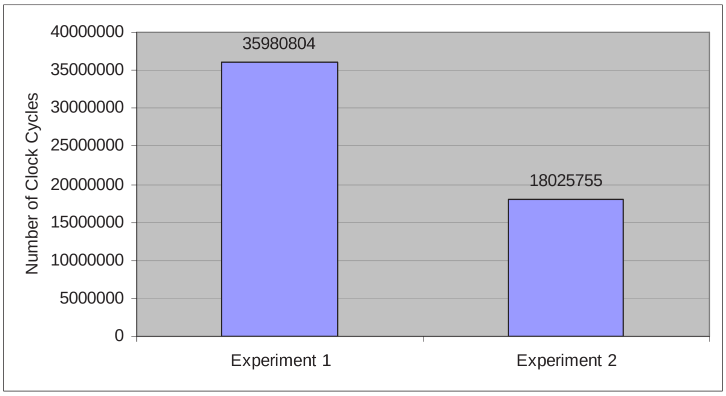


Figure 5.9: The performances of the two M-JPEG encoder embedded systems.

Table 5.2 shows the device utilization summary for the second experiment. In this experiment, the number indicates that 13 percent of the FPGA resources are used. However, we see that there are 123 out of 144 *RAMB16s* of the on-chip memories are used. This means 85 percent of the on-chip memories are used. Because a *MicroBlaze* processor is a soft core, based on the requirement of an application we can map the application onto any number of *MicroBlaze*

Table 5.1: Cycles and utilization percentage of each process in experiment 2.

	VideoInMain	DCT	Q	VLE	VideoOut
Cycles	10,837	135,036	68,314	49,683	1,727
Percentage(%)	4.1	50.8	25.7	18.7	0.7

processors embedded system platform. The only limitation is whether the target FPGA board has enough on-chip memories and reconfigurable resources.

Table 5.2: Virtex2 xc2v6000: device utilization summary for experiment 2.

FPGA Resource	Utilization	%
Number of MULT18X18s	15 out of 144	10%
Number of RAMB16s	123 out of 144	85%
Number of SLICEs	4664 out of 33792	13%
Number of BUFGMUXs	2 out of 16	12%

In this case study, we verify the design methodology in our ESPAM tool by mapping the M-JPEG encoder application onto two types of embedded system platform and compare the performances of a multiprocessor embedded system with a single processor embedded system. With the help of our ESPAM tool we can map an application onto a multiprocessor embedded system platform easily and quickly. We prove that with mapping the same application onto a multiprocessor embedded system gives better time performance compared a single processor embedded system. We also validate the interface of our embedded systems with the outside world explained in Chapter 4. In this case study, we find out that there are still several tasks we need to do manually after the system as XPS project automatic generation using our ESPAM tool. The main tasks are related to the memory allocation. According to different applications, we need to manually set the size of some FIFOs, the stack size of each processor or even the data/program memory allocation of each processor. The other tasks are about importing the implementations of function calls in processors and changing function calls in processors' program code and so on. All these custom tasks which we need to do manually will be explained in Chapter 6.

5.2 M-JPEG Heterogeneous and Hierarchical Multiprocessor System

In this case study, we use the same application M-JPEG encoder, but we map this application onto a multiprocessor embedded system platform with heterogeneous and hierarchical architecture which is shown in Figure 5.10. We see that this heterogeneous and hierarchical multiprocessor system includes four *MicroBlaze* processors and one dedicated hardware IP core for the *DCT()* process in the M-JPEG encoder application.

In order to generate this multiprocessor embedded system, the first step is to convert the Matlab code shown in Figure 5.1 to a KPN specification. We use the COMPAAN tool to automatically transform the code to a KPN specification. Because of the four *MicroBlaze* processors together with one dedicated hardware IP core, there are five parallel tasks which are executed concurrently in this embedded system platform in this case. Thus, the second step is to write the *Platform Specification* and the *Mapping Specification* for a five-processor embedded system

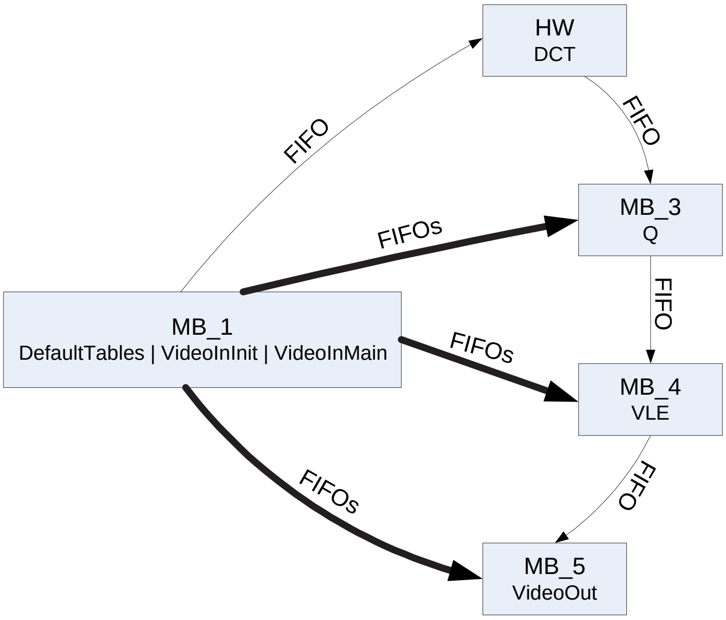


Figure 5.10: The multiprocessor embedded system platform with heterogeneous and hierarchical architecture for M-JPEG encoder application.

platform which are the same as Figure 5.7 and Figure 5.8. In the *Platform Specification*, there are five *MicroBlaze* processors (*MB_1*, *MB_2*, *MB_3*, *MB_4*, and *MB_5*) and five custom memory controllers (*ZBT_CTRL_1*, *ZBT_CTRL_2*, *ZBT_CTRL_3*, *ZBT_CTRL_4*, and *ZBT_CTRL_5*) which are used as the interfaces between the *MicroBlaze* processors and the ZBT SSRAMs in this embedded system platform. In the *Mapping Specification*, we map *DefaultTables()* process (ND_1), *VideoInInit()* process (ND_2) and *VideoInMain()* process (ND_3) onto processor *MB_1*, *DCT()* process (ND_4) onto processor *MB_2*, *Q()* process (ND_5) onto processor *MB_3*, *VLE()* process (ND_6) onto processor *MB_4* and *VideoOut()* process (ND_7) onto processor *MB_5*. In the third step we use our ESPAM tool to map the M-JPEG encoder application onto this five-processor embedded system platform. Because the *DCT()* process has been mapped onto processor *MB_2*, the fourth step is to use the dedicated hardware IP core for the *DCT()* process which was generated in Section 3.2.2 to replace processor *MB_2*. The detailed steps of replacing processor *MB_2* with the dedicated hardware IP core for the *DCT()* process will be explained in Chapter 6. It is possible for our ESPAM tool to automatically implement the work which is described above. In this thesis, we just focus on showing the procedure about how to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture. The implementation in our ESPAM tool is straightforward and it is out of the scope of this thesis.

In this case study, we use one video frame which size is 128×128 pixels to test this M-JPEG encoder heterogeneous and hierarchical embedded system. In order to make the embedded system be able to exchange video data with the outside world, we still need to use the interface which is explained in Chapter 4 to communicate with an outside host processor and to store the video data in the off-chip memories. Figure 5.11 shows the performances of this M-JPEG encoder heterogeneous and hierarchical embedded system together with the M-JPEG encoder homogeneous embedded systems — the one-processor embedded system and five-processor embedded system. The frequency of the processors in this case study is 100MHz. In Figure 5.11 we see that the M-JPEG encoder heterogeneous and hierarchical embedded system is around 2 times faster than the five-processor homogeneous embedded system and it is around 4 times faster than the one-processor homogeneous embedded system. In Table 5.1, we see that in the five-processor homogeneous embedded system the *DCT()* process is the bottleneck of the system. In the five-processor homogeneous embedded system, the *DCT()* process takes 50.8 percent of the whole time and it is around 2 times slower than the *Q()* process which takes 25.7 percent of the whole time. For this heterogeneous and hierarchical embedded system, Table 5.3 shows how many clock cycles and the utilization percent of each process, which is executed by one processor or the dedicated hardware IP core, need to take in order to process one block image. We see that the *Q()* process takes the longest time in the processes of the M-JPEG encoder application. The *Q()* process takes 50.3 percent of the whole time and now it is the bottleneck of the system. Comparing with the *Q()* process, the *DCT()* process takes around 0 percent of the whole time. In the five-processor homogeneous embedded system the *DCT()* process is the bottleneck of the whole system and it is around 2 times slower than the *Q()* process, but in this heterogeneous and hierarchical embedded system the *Q()* process is the bottleneck of the whole system and comparing with the *Q()* process the *DCT()* process takes around 0 percent of the whole time. Due to this reason the M-JPEG encoder heterogeneous and hierarchical embedded system is 2 times faster than the five-processor homogeneous embedded system. In Table 5.3, we also see that the *VLE()* process and the *VideoOut()* process in this case take different clock cycles from the *VLE()* process and the *VideoOut()* process in the five-processor homogeneous embedded system. The reason is that the precision of the resulting data that we get from the *DCT()* process executed by the dedicated hardware IP core is different from the the resulting data when the *DCT()* process is executed by the *MicroBlaze* processor. Because the *VLE()* process and the *VideoOut()* process are sensitive to the precision of the data, the clock cycles spent on the *VLE()* process and the *VideoOut()* process in this case are different from the *VLE()* process and the *VideoOut()* process in the five-processor homogeneous embedded system. Because the *VideoInMain()* process and the *Q()* process are insensitive to the precision of the data, the clock cycles spent on the *VideoInMain()* process and the *Q()* process in this case are almost the same as the *VideoInMain()* process and the *Q()* process in the five-processor homogeneous embedded system.

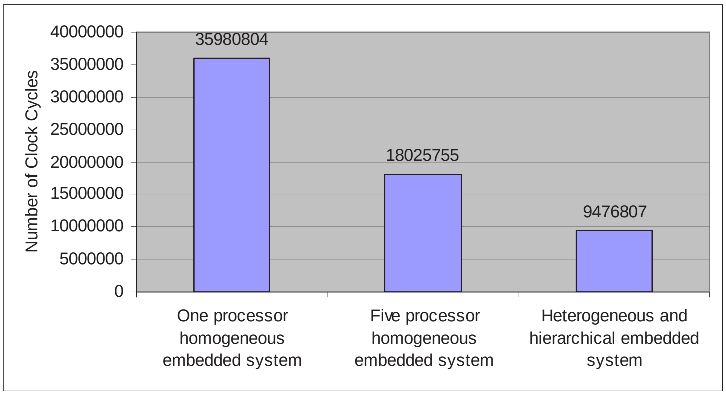


Figure 5.11: The performances of the three M-JPEG encoder embedded systems.

Table 5.4 shows the device utilization summary for this heterogeneous and hierarchical em-

Table 5.3: Cycles and utilization percentage of each process.

	VideoInMain	DCT	Q	VLE	VideoOut
Cycles	10,837	400	68,972	54,210	2,795
Percentage(%)	7.9	0.3	50.3	39.5	2

bedded system. We see that there are 111 out of 144 (77 percent) *RAMB16s* which are the on-chip memories used. This heterogeneous and hierarchical embedded system needs less on-chip memories than the five-processor homogeneous embedded system. The reason is that we use a dedicated hardware IP core for the *DCT()* process and it doesn't need any data memories or program memories comparing to a *MicroBlaze* processor.

Table 5.4: Virtex2 xc2v6000: device utilization summary.

FPGA Resource	Utilization	%
Number of MULT18X18s	20 out of 144	13%
Number of RAMB16s	111 out of 144	77%
Number of SLICEs	5675 out of 33792	16%
Number of BUFGMUXs	2 out of 16	12%

In this case study, we validate the procedure of implementing an embedded system as heterogeneous and hierarchical architecture and evaluate the heterogeneous and hierarchical architecture introduced in Chapter 3. Also we compare the performances of the heterogeneous and hierarchical embedded system with the homogeneous embedded systems. We prove that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture, and with mapping the same application a heterogeneous and hierarchical embedded system has better time performance comparing with a homogeneous embedded system. For this M-JPEG encoder application, we use a dedicated hardware IP core for the *DCT()* process. Then the *Q()* process becomes the bottleneck of the whole system. In Table 5.3, we see that the *Q()* process and the *VLE()* process take much longer time than the *VideoInMain()* process and *VideoOut()* process. If we want to improve the time performance further, we have to use dedicated hardware IP cores for the *Q()* process and the *VLE()* process. Then in such heterogeneous and hierarchical embedded system, we just use *MicroBlaze* processors to execute the *VideoInMain()* process and *VideoOut()* process and the other processes are all executed by the dedicated hardware IP cores. In such case, we can get real-time performance.

Chapter 6

Getting Started: Tutorial on Heterogeneous and Hierarchical System Design

In this chapter, we give a tutorial with example of heterogeneous and hierarchical embedded system design. This tutorial gives the detailed steps for how to design a heterogeneous and hierarchical embedded system using the COMPAAN tool, our ESPAM tool and the commercial synthesis tool Xilinx Platform Studio (XPS). We use the M-JPEG encoder heterogeneous and hierarchical embedded system presented in Section 5.2 to explain in detail the design steps. In order to design the heterogeneous and hierarchical embedded system for the M-JPEG encoder application, first we need to use the COMPAAN tool and our ESPAM tool to generate a five-processor homogeneous embedded system for the M-JPEG encoder application and generate systematically and automatically all of the necessary files of an XPS project for the M-JPEG encoder homogeneous embedded system. Then we need to change this XPS project to heterogeneous and hierarchical embedded system manually. Finally, we import this XPS project into XPS and use XPS to generate the final bitstream file which is used to configure the FPGA chip to implement the M-JPEG encoder application.

This chapter is organized as follows. In Section 6.1, we explain how to generate the XPS project with homogeneous embedded system for the M-JPEG encoder application. In Section 6.2, we describe how to change this XPS project to heterogeneous and hierarchical embedded system by hand. In Section 6.3 we explain how to import the project into XPS and use XPS to generate the final bitstream file. In this section, we also describe how to use a software program in an outside host processor to download the final bitstream file onto the target FPGA board and test the heterogeneous and hierarchical embedded system to get the resulting data, and how to debug the M-JPEG encoder heterogeneous and hierarchical embedded system.

6.1 Generation of Homogeneous Embedded System

In this section, we explain how to generate an XPS project with homogeneous embedded system for the M-JPEG encoder application. First, we need to use the COMPAAN tool to automatically transform the initial Matlab code of the M-JPEG encoder application into KPN specification.

Second, we need to create the *Platform Specification* and the *Mapping Specification* for this five-processor homogeneous embedded system. Then, we use our Espam tool to automatically generate all of the necessary files of an Xps project for this M-JPEG encoder system. Finally, we need to manually do some modifications in the Xps project.

6.1.1 KPN Specification Generation Using the Compaan tool

In this section, we describe how to generate the KPN specification from the initial Matlab code of the M-JPEG encoder application using the Compaan tool. The initial Matlab code of the M-JPEG encoder application is shown in Figure 6.1. This M-JPEG encoder compresses a sequence of video frames, using JPEG picture compression in each frame of the video. The detailed explanation of this Matlab code was given in Section 5.1.

```
%parameter NumFrames 1 100;
%parameter VNumBlocks 2 100;
%parameter HNumBlocks 1 100;

%typedef HeaderInfo             THeaderInfo;
%typedef LuminanceQTable        TQTables;
%typedef ChrominanceQTable      TQTables;
%typedef LuminanceHuffTableDC   THuffTablesDC;
%typedef ChrominanceHuffTableDC THuffTablesDC;
%typedef LuminanceHuffTableAC   THuffTablesAC;
%typedef ChrominanceHuffTableAC THuffTablesAC;
%typedef LuminanceTablesInfo    TTablesInfo;
%typedef ChrominanceTablesInfo  TTablesInfo;
%typedef Packets                TPackets;
%typedef Block                  TBlocks;

for k = 1:1:1,
  [  LuminanceQTable,     ChrominanceQTable,
     LuminanceHuffTableDC,ChrominanceHuffTableDC,
     LuminanceHuffTableAC,ChrominanceHuffTableAC,
     LuminanceTablesInfo, ChrominanceTablesInfo
  ] = DefaultTables();
end

for k = 1:1:NumFrames,
  [  HeaderInfo  ] = VideoInInit();
  for j = 1:1:VNumBlocks,
    for i = 1:1:HNumBlocks,
      [ Block ] = VideoInMain();
      [ Block ] = DCT( Block );
      [ Block ] = Q( Block, LuminanceQTable, ChrominanceQTable );
      [ Packets ] = VLE( Block,
                         LuminanceHuffTableDC,ChrominanceHuffTableDC,
                         LuminanceHuffTableAC,ChrominanceHuffTableAC );
      [  dummy  ] = VideoOut( HeaderInfo, LuminanceTablesInfo,
                              ChrominanceTablesInfo, Packets );
    end
  end
end
```

Figure 6.1: The initial Matlab code of the M-JPEG encoder application.

In this Matlab code, we see that there are seven function calls named *DefaultTables()*, *VideoInInit()*, *VideoInMain()*, *DCT()*, *Q()*, *VLE()*, and *VideoOut()*. When we use the Compaan tool to generate the KPN specification, by default it generates a process for each function call in the initial Matlab code. Thus, the Compaan tool will generate seven processes in the KPN specifica-

tion. Notice that the COMPAAN tool and our ESPAM tool do not deal with the implementations of the function calls in the initial Matlab code, they just generate empty wrappers for these function calls. In order to implement the M-JPEG encoder application in the XPS, we need to change these empty wrappers which is explained in Section 6.1.3. We need to generate the implementations for all of the function calls in the initial Matlab code. The needed data types are declared in lines 5-15 lines in Figure 6.1 and the definitions of these data types are in the file *types.h*. The implementation of *DefaultTables()* function in line 22 is in file *ControlInit.cpp*. The implementations of *VideoInInit()* function in line 26 and *VideoInMain()* function in line 29 are in file *Video_in.cpp*. The implementation of *DCT()* function in line 30 is in file *DCT.cpp*. The implementation of *Q()* function in line 31 is in file *Q.cpp*. The implementation of *VLE()* function in line 32 is in file *VLE.cpp*. The implementation of *VideoOut()* function in line 35 is in file *Video_out.cpp*. We need to manually import all of these files to the XPS project which we will generate later and this step will be explained in Section 6.3.1. The source files discussed above can be found in the CVS repository :
docs/students/WeiZhong/experiment/MJPEG-Pentium.zip

```
1) matparser --input M_JPEG.m --output M_JPEG.sac --compile --verbose -r

2) dgparser --input M_JPEG.sac --output M_JPEG --xml -r

3) panda --input M_JPEG.xml -c M_JPEG.m --xml -ls --lms -RP -r
```

Figure 6.2: The three commands of the COMPAAN tool.

We need to use three commands of the COMPAAN tool to generate a KPN specification for the M-JPEG encoder application. The three commands are shown in Figure 6.2. The first command uses the MATPARSER tool [32] to transform the initial Matlab code into a single assignment code (SAC), which resembles the dependence graph (DG) of the initial Matlab code. The --*input* option is followed by a filename that points to a file where the initial Matlab code is stored. The --*output* option is followed by a filename that points to a file where results, for example the SAC, need to be written. The --*compile* option tells MATPARSER to convert the Matlab code into a SAC. The --*verbose* option causes MATPARSER to produce information messages showing the progress made in the conversion. The *-r* option applies a set of optimizations on the 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 uses the DGPARSER tool to convert 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 --*input* option specifies the SAC file generated by MATPARSER. The --*output* option specifies the output file where the PRDG data structure will be stored. The --*xml* option specifies the format of the output file to be XML. The *-r* option manipulates the parse tree. In particular, it removes control from the index statements.

The third command uses the PANDA tool to convert the PRDG into a KPN process network [33] [34]. The --*input* option specifies the input PRDG XML file generated by DGPARSER. The *-c* option describes a valid global schedule as a Matlab program for all the nodes in the PRDG graph. The --*xml* option specifies the format of the output file to be XML. The *-ls* and --*lms* options tell PANDA to select communication linearization model, since the communication is not always in order. For more details see [33] [34]. The *-RP* 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 [33] [34]. The *-r* 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.

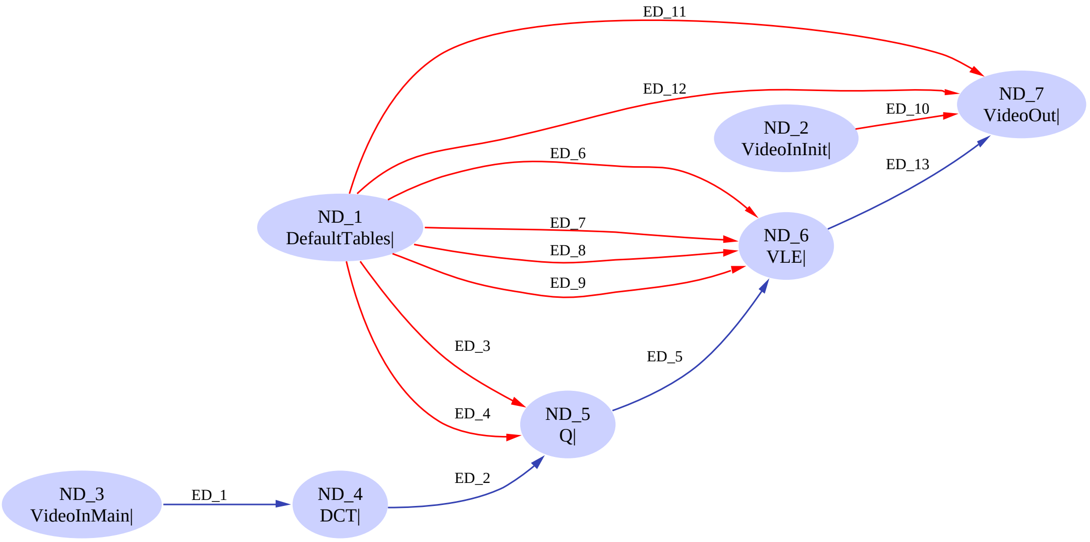


Figure 6.3: The KPN of the M-JPEG encoder application.

After executing the three commands described above, we can get the KPN specification in XML format. The KPN of the M-JPEG encoder application which is generated by COMPAAN is shown in Figure 6.3. In this KPN specification of the M-JPEG encoder application, there are seven processes — *ND_1*, *ND_2*, *ND_3*, *ND_4*, *ND_5*, *ND_6* and *ND_7*. *ND_1* is the *DefaultTables()* process. *ND_2* is the *VideoInInit()* process. *ND_3* is the *VideoInMain()* process. *ND_4* is the *DCT()* process. *ND_5* is the *Q()* process. *ND_6* is the *VLE()* process. *ND_7* is the *VideoOut()* process.

6.1.2 Generating Homogeneous Embedded System Using the ESPAM tool

In Figure 1.1, we see that the inputs of our ESPAM tool are *Application Specification*, *Platform Specification* and *Mapping Specification*. Thus, after we get the KPN specification which is the *Application Specification* from the initial Matlab code of the M-JPEG encoder application using the COMPAAN tool, we still need to create the *Platform Specification* and the *Mapping Specification*. The *Platform Specification* and the *Mapping Specification* for the M-JPEG encoder five processors homogeneous embedded system are shown in Figure 6.4 and Figure 6.5. In the *Platform Specification*, there are five *MicroBlaze* processors (*MB_1*, *MB_2*, *MB_3*, *MB_4*, and *MB_5*) and five custom memory controllers (*ZBT_CTRL_1*, *ZBT_CTRL_2*, *ZBT_CTRL_3*, *ZBT_CTRL_4*, and *ZBT_CTRL_5*) which are used as the interfaces between the *MicroBlaze* processors and the ZBT SSRAMs in this embedded system platform. In the *Mapping Specification*, we map *DefaultTables()* process (ND_1), *VideoInInit()* process (ND_2) and *VideoInMain()* process (ND_3) onto processor *MB_1*, *DCT()* process (ND_4) onto processor *MB_2*, *Q()* process (ND_5) onto processor *MB_3*, *VLE()* process (ND_6) onto processor *MB_4* and *VideoOut()* process (ND_7) onto processor *MB_5*. The detailed description for these *Platform Specification* and *Mapping Specification* are given in Section 5.1.

```
<platform name="myPlatform">

<processor name="MB_1" type="MB" data_memory="65536" program_memory="32768">
    <port name="OPB_1" type="OPBPort"/>
</processor>
<processor name="MB_2" type="MB" data_memory="16384" program_memory="16384">
    <port name="OPB_2" type="OPBPort"/>
</processor>
<processor name="MB_3" type="MB" data_memory="8192" program_memory="8192">
    <port name="OPB_3" type="OPBPort"/>
</processor>
<processor name="MB_4" type="MB" data_memory="16384" program_memory="16384">
    <port name="OPB_4" type="OPBPort"/>
</processor>
<processor name="MB_5" type="MB" data_memory="16384" program_memory="16384">
    <port name="OPB_5" type="OPBPort"/>
</processor>

<peripheral name="ZBT_CTRL_1" type="ZBTCTRL" size="1000000">
    <port name="IO_1" type="OPBPort"/>
</peripheral>
<peripheral name="ZBT_CTRL_2" type="ZBTCTRL" size="1000000">
    <port name="IO_2" type="OPBPort"/>
</peripheral>
<peripheral name="ZBT_CTRL_3" type="ZBTCTRL" size="1000000">
    <port name="IO_3" type="OPBPort"/>
</peripheral>
<peripheral name="ZBT_CTRL_4" type="ZBTCTRL" size="1000000">
    <port name="IO_4" type="OPBPort"/>
</peripheral>
<peripheral name="ZBT_CTRL_5" type="ZBTCTRL" size="1000000">
    <port name="IO_5" type="OPBPort"/>
</peripheral>

<link name="mb_opb_1">
    <resource name="MB_1" port="OPB_1"/>
    <resource name="ZBT_CTRL_1" port="IO_1"/>
</link>
<link name="mb_opb_2">
    <resource name="MB_2" port="OPB_2"/>
    <resource name="ZBT_CTRL_2" port="IO_2"/>
</link>
<link name="mb_opb_3">
    <resource name="MB_3" port="OPB_3"/>
    <resource name="ZBT_CTRL_3" port="IO_3"/>
</link>
<link name="mb_opb_4">
    <resource name="MB_4" port="OPB_4"/>
    <resource name="ZBT_CTRL_4" port="IO_4"/>
</link>
<link name="mb_opb_5">
    <resource name="MB_5" port="OPB_5"/>
    <resource name="ZBT_CTRL_5" port="IO_5"/>
</link>

</platform>
```

Figure 6.4: *Platform Specification* for the five processors homogeneous embedded system.

When we get the *Application Specification*, *Platform Specification* and *Mapping Specification*, we can start to run our ESPAM tool to automatically generate all of the necessary files of the XPS project for this M-JPEG encoder five-processor homogeneous embedded system. The command of our ESPAM tool we need to execute is shown in Figure 6.6.

```
<mapping name="myMapping">

<processor name="MB_1">
    <process name="ND_1" />
    <process name="ND_2" />
    <process name="ND_3" />
</processor>

<processor name="MB_2">
    <process name="ND_4" />
</processor>

<processor name="MB_3">
    <process name="ND_5" />
</processor>

<processor name="MB_4">
    <process name="ND_6" />
</processor>

<processor name="MB_5">
    <process name="ND_7" />
</processor>

</mapping>
```

Figure 6.5: *Mapping Specification* for the five processors homogeneous embedded system.

```
espam --platform M_JPEG.pla --kpn M_JPEG.kpn --mapping M_JPEG.map --scheduler M_JPEG.m --xps
--libxps <libXPS> --debugger
```

Figure 6.6: The command of the ESPAM tool.

By executing this command, our ESPAM tool can automatically generate all of the necessary files of the XPS project for this M-JPEG encoder five processors homogeneous embedded system according to the *Application Specification*, *Platform Specification* and *Mapping Specification*. The --*platform* option specifies the *Platform Specification* file. The --*kpn* option specifies the *Application Specification* file. The --*mapping* option specifies the *Mapping Specification* file. The --*scheduler* option specifies a file which is used to describe a valid global schedule among the processes in the *Application Specification*. The --*xps* option is used to tell our ESPAM tool to generate all necessary files of an XPS project. The --*libxps* option specifies a library that stores the predefined platform components used to generate an XPS project. An XPS project always consists of two parts. One part is generated at compile time, including the XMP/MHS/MSS files, the program code for each processor in a 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 some common custom IP cores, the UCF file and some optional files for XPS implementation tools. We store this library in the CVS repository. The <$libXPS$> specifies the path to this library so that our ESPAM tool can copy and use it during the generation of an XPS project suite. Currently, we use the following CVS repository path for this library: *.../espam/src/espam/libXPS*. The --*debugger* option is used to tell our ESPAM tool to generate component used for debugging. We explain this debugging component in Section 6.3.3.

After we run this command of our ESPAM tool, an XPS project for the M-JPEG encoder five-processor homogeneous embedded system is generated. Figure 6.7 shows the XPS project directory hierarchy.

The *system.xmp*, *system.mhs* and *system.mss* files are the corresponding XMP, MHS and MSS

```
<PROJECT_ROOT>
|--- system.xmp
|--- system.mhs
|--- system.mss
|--- loader.exe
|--- etc/
|------------- bitgen.ut
|------------- bitgen_spartan3.ut
|------------- fast_runtime.opt
|------------- download.cmd
|--- data/
|------------- system.ucf
|------------- system_ADMXRCII.ucf
|------------- system-default.ucf
|------------- system-zbt.ucf
|--- code/
|------------- aux_func.h
|------------- MemoryMap.h
|------------- P_1/
|----------------------- P_1.cpp
|------------- P_2/
|----------------------- P_2.cpp
|------------- P_3/
|----------------------- P_3.cpp
|------------- P_4/
|----------------------- P_4.cpp
|------------- P_5/
|----------------------- P_5.cpp
|--- pcores/
|------------- buffers_v1_00_a/
|------------- cb_wrapper_v1_00_a/
|------------- clock_cycle_counter_v1_00_a/
|------------- fifo_if_ctrl_v1_00_a/
|------------- fin_ctrl_v1_00_a/
|------------- host_design_ctrl_v1_00_a/
|------------- LMB_VB_CTRL_v1_00_a/
|------------- mux_v1_00_a/
|------------- myCLKRST_v1_00_a/
|------------- opb_zbt_controller_v1_00_a/
|------------- VB_Wrapper_v1_00_a/
|------------- zbt_main_v1_00_a/
```

Figure 6.7: XPS project directory hierarchy for the M-JPEG encoder embedded system.

files which have been explained in Section 2.5.1. The MHS file — *system.mhs* and the MSS file — *system.mss* which are automatically generated by our ESPAM tool are shown in Appendix A and Appendix B. The *loader.exe* file is a program used to download and run the bitstream file. The *etc* directory contains four files — *bitgen.ut* [35], *bitgen_spartan3.ut*, *fast_runtime.opt* [35] and *download.cmd*. They are the files with options for setting XPS implementation tools. The *data* directory contains several UCF files according to the different FPGA devices. In our case, we use the *system_ADMXRCII.ucf* UCF file which contains pin information for the physical implementation in the selected FPGA device. In the *code* directory, the software program code files for processors are stored. In the top level of the *code* directory, there are two files named *aux_func.h* and *MemoryMap.h*. They are the common files for all of the processors. The *aux_func.h* file declares read and write primitives and wrappers of all function calls in the initial code of the application. The *MemoryMap.h* file specifies physical addresses of the components in the platform. The program code for each processors is stored in the corresponding subdirectory named after the processors. The *pcores* directory contains all predefined IP cores and the IP cores generated by our ESPAM tool. The *buffers_v1_00_a*, *fin_ctrl_v1_00_a*, *host_design_ctrl_v1_00_a*, *mux_v1_00_a*, *opb_zbt_controller_v1_00_a* and *zbt_main_v1_00_a* are the IP cores for the interface of an embedded system with the outside world which have been explained in Section 4.2. The *fifo_if_ctrl_v1_00_a* is the LMB FIFO controller. The detailed description about this controller can be found in [5]. The *clock_cycle_counter_v1_00_a* is the IP core for debugging. The *myCLKRST_v1_00_a* is the IP core which is used to generate the system clock and reset and it is not used in our case. The *cb_wrapper_v1_00_a*, *LMB_VB_CTRL_v1_00_a* and *VB_Wrapper_v1_00_a* are the IP cores for the crossbar communication component and they are not used in our case.

6.1.3 Custom Modification for the XPS Project

After we get the XPS project which is automatically generated by our ESPAM tool, we still need to do some modifications for both the hardware and software in this XPS project.

Hardware Modification

As discussed in Section 5.1, the main purpose of the hardware modification is related to the memory allocation. The main task for the memory allocation modification is the FIFOs size adjustment. We need to adjust the size of the FIFOs in the MHS file. By default, our ESPAM tool set 2048 bytes (512×32) for each FIFO. The 512 is the data depth of a FIFO and the 32 is the data width of a FIFO. Lines 496 and 497 of Appendix A show the example of FIFO size setting in the MHS file. However, in 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. Thus, the corresponding FIFOs' size is not sufficient. We need to enlarge the corresponding FIFOs' size to 4096 bytes (1024×32) [5]. In the MHS file which is shown in Appendix A, we need to enlarge the size of FIFOs *FIFO_MB_1_Out_4*, *FIFO_MB_1_Out_5*, *FIFO_MB_1_Out_6*, *FIFO_MB_1_Out_7*, *FIFO_MB_1_Out_9* and *FIFO_MB_1_Out_10* to 4096 bytes. An example modification of the size of FIFO *FIFO_MB_1_Out_4* is shown in line 562 of Figure 6.8. The other FIFOs' size can be modified in the same way. Other task for the memory allocation modification is the stack size adjustment of each processor which will be explained in Section 6.3.1.

```
554    BEGIN fsl_v20
555     PARAMETER HW_VER = 2.00.a
        PARAMETER INSTANCE = FIFO_MB_1_Out_4
        PARAMETER C_EXT_RESET_HIGH = 0
        PARAMETER C_ASYNC_CLKS = 0
        PARAMETER C_IMPL_STYLE = 1
560     PARAMETER C_USE_CONTROL = 0
        PARAMETER C_FSL_DWIDTH = 32
        PARAMETER C_FSL_DEPTH = 1024
        PORT FSL_Clk = sys_clk_s
        PORT SYS_Rst = net_design_rst
565    END
```

Figure 6.8: Set the size of FIFO *FIFO_MB_1_Out_4* to 4 Kbytes.

The second thing we need to modify is the UCF file name. In the *data* directory of our XPS project, there are several UCF files. When we import the project to XPS, the XPS will automatically recognize the UCF file which is named *system.ucf*. Thus, we need to change the name of the UCF file which we need to use to *system.ucf* in the *data* directory. In our case, the UCF file we need in the *data* directory is *system_ADMXRCII.ucf*. However, there is already a UCF file named *system.ucf* in the *data* directory. We need to change the name of original *system.ucf* file to *system_old.ucf*, then change the name of *system_ADMXRCII.ucf* file to *system.ucf*.

The third thing we need to modify is related to file *fast_runtime.opt*. The XPS project generated by our ESPAM tool is based on XPS version 6.3, but later we will import our XPS project to XPS version 7.1. When we import our XPS project to XPS version 7.1, XPS will automatically upgrade XPS project to adapt to version 7.1. However just one thing XPS can not upgrade automatically is in the *fast_runtime.opt* file which is stored in the *etc* directory of our XPS project. In the *fast_runtime.opt* file there is an option for place and route named *-ol* which is used to set the overall effort level. In XPS version 6.3, it can be set to number 1 to 5. But in XPS version 7.1, it just can be set to *std*, *med* and *high*. By default our ESPAM tool set this option to number 5. In our case, we need to manually change this number 5 to *std*.

Software Modification

The first thing for the software modification is that we need to copy all of the header files which the program code for processors needs to the *code* directory of our XPS project. Also we need to copy the implementation program code files for each processor's program code to the corresponding subdirectory named after the processors in the *code* directory. In our case, we need to copy *ControlInit.cpp* and *Video_in.cpp* files to *P_1* subdirectory, *DCT.cpp* file to *P_2* subdirectory, *Q.cpp* file to *P_3* subdirectory, *VLE.cpp* file to *P_4* subdirectory, and *Video_out.cpp* file to *P_5* subdirectory. After this step, we still need to manually import all of these header files and implementation program code files to the XPS project which will be explained in Section 6.3.1.

The second task we need to do is to add the function declarations and replace each empty wrapper with a function call in each processor program code. As an example, the modified program code of processor *P_1* is shown in Figure 6.9. The bold lines in the code highlight the modification which we need to do manually. In lines 26 and 27, we define two instances *vin* and *cinit*. In lines 31, 59 and 66, we replace the empty wrappers with the actual function calls. The program code of the other processors can be modified in the same way. In Figure 6.9, we see that there is one more place we need to modify is in line 75. In line 75 we store a variable to the ZBT memory which is used for debugging and it will be explained in Section 6.3.3.

The third thing we need to change is to modify the *aux_func.h* file. The modified *aux_func.h* file is shown in Figure 6.10. The bold lines in the code highlight the modification which we need to do manually. In lines 6-11, we include all of the header files which are used in the processors' program code. In lines 28-30, we can change the three parameters — *NumFrames*, *VNumBlocks* and *HNumBlocks* based on how many frames we need to process and the size of the video frame. Because later we will use one video frame which size is 128×128 pixels to test the M-JPEG encoder embedded system, we set the parameter *NumFrames* to 1, *VNumBlocks* to 16 and *HNumBlocks* to 8. Because we have already replaced the empty wrappers with the actual function calls in program code of each processor, we need to comment the empty wrapper declarations in lines 33-59.

The fourth thing we need to change is to modify the *MemoryMap.h* file. The modified *Mem-*

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


int main (){

int clk_num;
*clk_cntr = 0;

    // Input Arguments

    // Output Arguments
    tCH_3 out_0ND_1;
    tCH_4 out_1ND_1;
    tCH_6 out_2ND_1;
    tCH_7 out_3ND_1;
    tCH_8 out_4ND_1;
    tCH_9 out_5ND_1;
    tCH_11 out_6ND_1;
    tCH_12 out_7ND_1;
    tCH_10 out_0ND_2;
    tCH_1 out_0ND_3;

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

    for( int k =  ceil1(1); k <=  floor1(1 ); k += 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);

        writeFSL(ND_1_OG_2_CH_3, &out_0ND_1, (sizeof(tCH_3)+(sizeof(tCH_3)%4)+3)/4);


        writeFSL(ND_1_OG_3_CH_4, &out_1ND_1, (sizeof(tCH_4)+(sizeof(tCH_4)%4)+3)/4);


        writeFSL(ND_1_OG_4_CH_6, &out_2ND_1, (sizeof(tCH_6)+(sizeof(tCH_6)%4)+3)/4);


        writeFSL(ND_1_OG_5_CH_7, &out_3ND_1, (sizeof(tCH_7)+(sizeof(tCH_7)%4)+3)/4);


        writeFSL(ND_1_OG_6_CH_8, &out_4ND_1, (sizeof(tCH_8)+(sizeof(tCH_8)%4)+3)/4);


        writeFSL(ND_1_OG_7_CH_9, &out_5ND_1, (sizeof(tCH_9)+(sizeof(tCH_9)%4)+3)/4);


        write(ND_1_OG_9_CH_11, &out_6ND_1, (sizeof(tCH_11)+(sizeof(tCH_11)%4)+3)/4);


        write(ND_1_OG_10_CH_12, &out_7ND_1, (sizeof(tCH_12)+(sizeof(tCH_12)%4)+3)/4);

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

        writeFSL(ND_2_OG_8_CH_10, &out_0ND_2, (sizeof(tCH_10)+(sizeof(tCH_10)%4)+3)/4);

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

            writeFSL(ND_3_OG_1_CH_1, &out_0ND_3, (sizeof(tCH_1)+(sizeof(tCH_1)%4)+3)/4);

        } // for i
        } // for j
    } // for k

    clk_num = *clk_cntr;
    *(ZBT_MEMORY) = (volatile long)clk_num;
    *FIN_SIGNAL = (volatile long)0x00000001;
} // main
```

Figure 6.9: Modified program code of processor *P_1*.

oryMap.h file is shown in Figure 6.11. The bold lines in the code highlight the modification which we need to do manually. In line 149 we need to add the physical address for our custom memory controllers which are used as the interfaces between the *MicroBlaze* processors and the ZBT SSRAMs in this embedded system platform. The complete modified project can be found in the CVS repository:

```
#ifndef __AUX_FUNC_H__
#define __AUX_FUNC_H__

#include <math.h>
#include "mb_interface.h"

#include ”Video_in.h”
#include ”Video_out.h”
#include ”ControlInit.h”
#include ”DCT.h”
#include ”Q.h”
#include ”VLE.h”

typedef TBlocks tCH_1;
typedef TBlocks tCH_2;
typedef TQTables tCH_3;
typedef TQTables tCH_4;
typedef TBlocks tCH_5;
typedef THuffTablesDC tCH_6;
typedef THuffTablesDC tCH_7;
typedef THuffTablesAC tCH_8;
typedef THuffTablesAC tCH_9;
typedef THeaderInfo tCH_10;
typedef TTablesInfo tCH_11;
typedef TTablesInfo tCH_12;
typedef TPackets tCH_13;

// Parameters
#define NumFrames 1
#define VNumBlocks 16
#define HNumBlocks 8

/*
inline
void  DefaultTables( tCH_3 *out_0, tCH_4 *out_1, tCH_6 *out_2, tCH_7 *out_3, tCH_8 *out_4, tCH_9 *out_5, tCH_11 *out_6, tCH_12 *out_7 )
}

inline
void  VideoInInit( tCH_10 *out_0 ) {
}

inline
void  VideoInMain( tCH_1 *out_0 ) {
}

inline
void  DCT( tCH_1 in_0, tCH_2 *out_0 ) {
}

inline
void  Q( tCH_2 in_0, tCH_3 in_1, tCH_4 in_2, tCH_5 *out_0 ) {
}

inline
void  VLE( tCH_5 in_0, tCH_6 in_1, tCH_7 in_2, tCH_8 in_3, tCH_9 in_4, tCH_13 *out_0 ) {
}

inline
void  VideoOut( tCH_10 in_0, tCH_11 in_1, tCH_12 in_2, tCH_13 in_3, char *out_0 ) {
}
*/

#define min(a,b) ((a)<=(b))?(a):(b)
#define max(a,b) ((a)>=(b))?(a):(b)

...

#endif
```

Figure 6.10: Modified *aux_func.h* file.

docs/students/WeiZhong/experiment/M_JPEG_5p.zip

6.2 Generation of Heterogeneous and Hierarchical Embedded System

After we get the XPS project with homogeneous embedded system for the M-JPEG encoder application in Section 6.1, we can change this XPS project to heterogeneous and hierarchical embedded system.

```
#ifndef __MEMORYMAP_H_
#define __MEMORYMAP_H_

#define PCTRL_BRAM1_MB_1 0x00000000 //read from PCTRL_BRAM1_MB_1 address for MB_1

#define PCTRL_BRAM1_MB_1 0x00000000 //write to PCTRL_BRAM1_MB_1 address for MB_1

#define PCTRL_BRAM2_MB_1 0x00004000 //read from PCTRL_BRAM2_MB_1 address for MB_1

#define PCTRL_BRAM2_MB_1 0x00004000 //write to PCTRL_BRAM2_MB_1 address for MB_1

//MB_1 FIFOs
#define ND_1_OG_9_CH_11 0xc0800000 //write to CDChannelCH_11 address for MB_1

//MB_1 FIFOs
#define ND_1_OG_10_CH_12 0xc0800008 //write to CDChannelCH_12 address for MB_1

#define DCTRL_BRAM1_MB_1 0x00000000 //read from DCTRL_BRAM1_MB_1 address for MB_1

#define DCTRL_BRAM1_MB_1 0x00000000 //write to DCTRL_BRAM1_MB_1 address for MB_1

#define DCTRL_BRAM2_MB_1 0x00004000 //read from DCTRL_BRAM2_MB_1 address for MB_1

#define DCTRL_BRAM2_MB_1 0x00004000 //write to DCTRL_BRAM2_MB_1 address for MB_1

#define ZBT_CTRL_1 0xf0000000 //read from ZBT_CTRL_1 address for MB_1

#define ZBT_CTRL_1 0xf0000000 //write to ZBT_CTRL_1 address for MB_1

//MB_1 FIFOs
#define ND_3_OG_1_CH_1 0 //write to CDChannelCH_1 address for MB_1

//MB_1 FIFOs
#define ND_1_OG_2_CH_3 1 //write to CDChannelCH_3 address for MB_1

//MB_1 FIFOs
#define ND_1_OG_3_CH_4 2 //write to CDChannelCH_4 address for MB_1

//MB_1 FIFOs
#define ND_1_OG_4_CH_6 3 //write to CDChannelCH_6 address for MB_1

//MB_1 FIFOs
#define ND_1_OG_5_CH_7 4 //write to CDChannelCH_7 address for MB_1

//MB_1 FIFOs
#define ND_1_OG_6_CH_8 5 //write to CDChannelCH_8 address for MB_1

//MB_1 FIFOs
#define ND_1_OG_7_CH_9 6 //write to CDChannelCH_9 address for MB_1

//MB_1 FIFOs
#define ND_2_OG_8_CH_10 7 //write to CDChannelCH_10 address for MB_1

...

#define ZBT_MEMORY (volatile long *)0xf0000000
#define clk_cntr (volatile int *)0xf8000000
#define FIN_SIGNAL (volatile long *)0xf9000000

#endif
```

Figure 6.11: Modified *MemoryMap.h* file.

The first step is that we need to copy the pcore for the *DCT()* process which has been described in Section 3.2.2 to the *pcores* directory of our XPS project. Later we will introduce how to use this dedicated hardware IP core to replace the *MicroBlaze* processor — *MB_2* in the XPS project of homogeneous embedded system which is also used to execute the *DCT()* process. The detailed steps for the pcore for the *DCT()* process generation was given in Section 3.2.2. The pcore for the DCT process can be found in the CVS repository:
docs/students/WeiZhong/experiment/DCTpcore.zip

In the second step we start to replace the *MicroBlaze* processor — *MB_2* in the XPS project of homogeneous embedded system with the dedicated hardware IP core for the *DCT()* process. In this step, we need to replace the *MB_2* with the dedicated hardware IP core for the *DCT()* process in the MHS file. First, we need to comment *MB_2* and the components which belong to *MB_2* in the MHS file. In the MHS file which is shown in Appendix A, we need to comment *PBUS_MB_2* component in lines 99-105, *DBUS_MB_2* component in lines 107-113, *mb_opb_2* component in lines 115-121, *fin_ctrl_P2* component in lines 123-131, *clock_cycle_counter_P2* component in lines 133-140, *MB_2 MicroBlaze* processor in lines 142-155, the *BUS_INTERFACE MUX_DESIGN_1_PORT = mux_design_1* of *multiplexer* component in line 390, *ZBT_CTRL_2* component in lines 441-451, *BRAM1_MB_2* component in lines 725-730, *DCTRL_BRAM1_MB_2* component in lines 732-740 and *PCTRL_BRAM1_MB_2* component in lines 742-750. Then we need to add the dedicated hardware IP core for the *DCT()* process in the MHS file which is shown in Figure 6.12.

```
BEGIN kpn
PARAMETER INSTANCE = KPN_DCT
PARAMETER HW_VER = 1.00.a
BUS_INTERFACE MFSL = FIFO_MB_2_Out_1
BUS_INTERFACE SFSL = FIFO_MB_1_Out_1
PORT STATUS = net_fin_signal_P2
PORT CLK = sys_clk_s
PORT RST = net_design_rst
END
```

Figure 6.12: The dedicated hardware IP core for the *DCT()* process in the MHS file.

In the third step, we need to remove the *MB_2* and the components which belong to *MB_2* in the MSS file. Because the dedicated hardware IP core for the *DCT()* process use the generic driver and XPS can automatically add this generic driver for it, we do not need to add the driver for the dedicated hardware IP core for the *DCT()* process in the MSS file. In the MSS file which is shown in Appendix B, we need to comment *MB_2 MicroBlaze* processor in lines 35-47, *mb_opb_2* component in lines 49-53, *fin_ctrl_P2* component in lines 55-59, *clock_cycle_counter_P2* component in lines 61-65, *ZBT_CTRL_2* component in lines 193-197, *DCTRL_BRAM1_MB_2* component in lines 325-329 and *PCTRL_BRAM1_MB_2* component in lines 331-335.

In the fourth step, we need to change the software in the XPS project. First, we need to delete the software project for *MicroBlaze* processor — *MB_2* in XPS which will be introduced in Section 6.3.1. Second, in order to get the resulting video frame for the M-JPEG encoder application we need to change some program code for the processor *P_1* and processor *P_3*. The modified program code of processor *P_1* is shown in Figure 6.13. The bold lines in the code highlight the modification. In 68-74, we linearize the packet of the video data which is used to preprocess the video data for the dedicated hardware IP core for the *DCT()* process. The modified program code of processor *P_3* is shown in Figure 6.14. The bold lines in the code highlight the modification. In lines 31-45, we need to convert the negative 9-bit numbers to 32-bit negative numbers for the video data. In lines 47-55, we need to transpose the video data blocks in the packet. The processes of these lines are used to postprocess the video data for the dedicated hardware IP core for the *DCT()* process. After these steps, finally we get the XPS project of heterogeneous and hierarchical embedded system which consists of four *MicroBlaze* processors and one dedicated hardware IP core for the *DCT()* process.

6.3 Import Project to XPS and XPS Project Execution and Results

In this section, we explain how to import our project of heterogeneous and hierarchical embedded system to XPS and there are still some modifications we need to do in XPS. Then we

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


int main (){

int clk_num;
*clk_cntr = 0;

    // Input Arguments

    // Output Arguments
    tCH_3 out_0ND_1;
    tCH_4 out_1ND_1;
    tCH_6 out_2ND_1;
    tCH_7 out_3ND_1;
    tCH_8 out_4ND_1;
    tCH_9 out_5ND_1;
    tCH_11 out_6ND_1;
    tCH_12 out_7ND_1;
    tCH_10 out_0ND_2;
    tCH_1 out_0ND_3;

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

    for( int k =  ceil1(1); k <=  floor1(1 ); k += 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);

        writeFSL(ND_1_OG_2_CH_3, &out_0ND_1, (sizeof(tCH_3)+(sizeof(tCH_3)%4)+3)/4);



        writeFSL(ND_1_OG_3_CH_4, &out_1ND_1, (sizeof(tCH_4)+(sizeof(tCH_4)%4)+3)/4);


        writeFSL(ND_1_OG_4_CH_6, &out_2ND_1, (sizeof(tCH_6)+(sizeof(tCH_6)%4)+3)/4);


        writeFSL(ND_1_OG_5_CH_7, &out_3ND_1, (sizeof(tCH_7)+(sizeof(tCH_7)%4)+3)/4);


        writeFSL(ND_1_OG_6_CH_8, &out_4ND_1, (sizeof(tCH_8)+(sizeof(tCH_8)%4)+3)/4);


        writeFSL(ND_1_OG_7_CH_9, &out_5ND_1, (sizeof(tCH_9)+(sizeof(tCH_9)%4)+3)/4);


        write(ND_1_OG_9_CH_11, &out_6ND_1, (sizeof(tCH_11)+(sizeof(tCH_11)%4)+3)/4);


        write(ND_1_OG_10_CH_12, &out_7ND_1, (sizeof(tCH_12)+(sizeof(tCH_12)%4)+3)/4);

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

        writeFSL(ND_2_OG_8_CH_10, &out_0ND_2, (sizeof(tCH_10)+(sizeof(tCH_10)%4)+3)/4);

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

            // linearize the packet
            for (int l = 0; l < 64; l++) {
                out_0ND_3.Y1.pixel[l] = (unsigned int)(out_0ND_3.Y1.pixel[l]/2);
                out_0ND_3.Y2.pixel[l] = (unsigned int)(out_0ND_3.Y2.pixel[l]/2);
                out_0ND_3.U1.pixel[l] = (unsigned int)(out_0ND_3.U1.pixel[l]/2);
                out_0ND_3.V1.pixel[l] = (unsigned int)(out_0ND_3.V1.pixel[l]/2);
            }

            writeFSL(ND_3_OG_1_CH_1, &out_0ND_3, (sizeof(tCH_1)+(sizeof(tCH_1)%4)+3)/4);

        } // for i
        } // for j
    } // for k

    clk_num = *clk_cntr;
    *(ZBT_MEMORY) = (volatile long)clk_num;
    *FIN_SIGNAL = (volatile long)0x00000001;
} // main
```

Figure 6.13: Modified program code of processor *P_1*.

introduce how to execute the project in XPS, get the result from this heterogeneous and hierar-

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


int main (){

int clk_num;
*clk_cntr = 0;

    // Input Arguments
    tCH_2 in_0ND_5;
    tCH_3 in_1ND_5;
    tCH_4 in_2ND_5;

    // Output Arguments
    tCH_5 out_0ND_5;

    tCH_2 tmp;

    Q  q;

    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 ) {

            //readFSL(ND_5_IG_1_CH_2, &in_0ND_5, (sizeof(tCH_2)+(sizeof(tCH_2)%4)+3)/4);
            readFSL(ND_5_IG_1_CH_2, &tmp, (sizeof(tCH_2)+(sizeof(tCH˙2)%4)+3)/4);

            //convert the negative 9-bit numbers to 32-bit negative numbers
            for (int l = 0; l < 64; l++) {
                if (tmp.Y1.pixel[l] >= 256) {
                    tmp.Y1.pixel[l] = tmp.Y1.pixel[l] | 0xfffffe00U;
                }
                if (tmp.Y2.pixel[l] >= 256) {
                    tmp.Y2.pixel[l] = tmp.Y2.pixel[l] | 0xfffffe00U;
                }
                if (tmp.U1.pixel[l] >= 256) {
                    tmp.U1.pixel[l] = tmp.U1.pixel[l] | 0xfffffe00U;
                }
                if (tmp.V1.pixel[l] >= 256) {
                    tmp.V1.pixel[l] = tmp.V1.pixel[l] | 0xfffffe00U;
                }
            }

            // transpose the data blocks in the packet
            for (int t = 0; t < 8; t++) {
                for (int q = 0; q < 8; q++) {
                in_0ND_5.Y1.pixel[t*8+q] = (int)tmp.Y1.pixel[q*8+t]*2;
                in_0ND_5.Y2.pixel[t*8+q] = (int)tmp.Y2.pixel[q*8+t]*2;
                in_0ND_5.U1.pixel[t*8+q] = (int)tmp.U1.pixel[q*8+t]*2;
                in_0ND_5.V1.pixel[t*8+q] = (int)tmp.V1.pixel[q*8+t]*2;
                }
            }

            if( k-1 == 0 ) {
                if( j-1 == 0 ) {
                if( i-1 == 0 ) {

                    readFSL(ND_5_IG_2_CH_3, &in_1ND_5, (sizeof(tCH_3)+(sizeof(tCH_3)%4)+3)/4);

                }
                }
            }
            if( k-1 == 0 ) {
                if( j-1 == 0 ) {
                if( i-1 == 0 ) {

                    readFSL(ND_5_IG_3_CH_4, &in_2ND_5, (sizeof(tCH_4)+(sizeof(tCH_4)%4)+3)/4);

                }
                }
            }
            //Q(in_0ND_5, in_1ND_5, in_2ND_5, &out_0ND_5) ;
            q.main(in_0ND_5, in_1ND_5, in_2ND_5, out_0ND_5);

            writeFSL(ND_5_OG_1_CH_5, &out_0ND_5, (sizeof(tCH_5)+(sizeof(tCH_5)%4)+3)/4);

        } // for i
        } // for j
    } // for k

    clk_num = *clk_cntr;
    *(ZBT_MEMORY) = (volatile long)clk_num;
    *FIN_SIGNAL = (volatile long)0x00000001;
} // main
```

Figure 6.14: Modified program code of processor *P_3*.

chical embedded system and debug this heterogeneous and hierarchical embedded system.

6.3.1 Import Project to XPS

In order to import our project of heterogeneous and hierarchical embedded system into XPS, first we need to start XPS. We use the start menu of the Windows: **start->Xilinx Platform Studio 7.1i->Xilinx Platform Studio**. In XPS, we select the menu option: **File->Open Project**. In the new dialog box, we select the XMP file — *system.xmp* of our XPS project. In our case we use XPS version 7.1. Because our XPS project is based on XPS version 6.3, XPS automatically upgrade our XPS project to adapt to version 7.1 and import our XPS project into XPS. We can get a view of all the components and settings in our XPS project by selecting the menu option: **Project->Add/Edit Cores...(dialog)**. All the components, buses, addresses, ports, and parameters are listed separately in the tabs **Peripherals**, **Bus Connections**, **Addresses**, **Ports**, and **Parameters**.

After we import our project to XPS, we still need to do some modifications for our project in XPS. First, we need to set our target FPGA board. In the *System* tab of XPS, there is a *Project Options*. In the *Project Options*, there is an option for *Device*. Double click the *Device* option, then in the new dialog box we set the target device to : *Architecture: virtex2*, *Device Size: xc2v6000*, *Package: ff1152*, *Grade: -5*. When we set the target device, we can click the *OK* button and then XPS set the target device for our project.

Second, we need to set the stack size for each processor of our project and import all of the header files and implementation program code files for each processor of our project in XPS. Also, we need to delete the software project for processor *MB_2*, because it has already been replaced with the dedicated hardware IP core for the *DCT()* process. In the *Applications* tab of XPS, there are five *Software Projects*: *Proj_MB_1*, *Proj_MB_2*, *Proj_MB_3*, *Proj_MB_4* and *Proj_MB_5*. Right click the *Proj_MB_2* and select the *Delete Project ...* option to delete the software project for processor *MB_2*. In each software project, there is a *Compiler Options*. Double click the *Compiler Options*, then in the new dialog box we can set the *Stack Size* for each processor. We need to set 64000 for *Stack Size* of *Proj_MB_1*, 9000 for *Stack Size* of *Proj_MB_3*, 19000 for *Stack Size* of *Proj_MB_4*, and 20000 for *Stack Size* of *Proj_MB_5*. Now we need to import all of the header files and implementation program code files for each processor of our project. In each software project, there are a *Sources* option and a *Headers* option. Double click the *Sources* option, then in the new dialog box we can add the implementation program code files for each processor. Double click the *Headers* option, then in the new dialog box we can add the head files for each processor. For *Proj_MB_1*, in *Sources* option we need to add *Video_in.cpp* and *ControlInit.cpp* files, and in *Headers* option we need to add *Video_in.h*, *ControlInit.h*, *csize.h*, *marker.h*, *param.h*, *tables.h*, and *types.h* files. For *Proj_MB_3*, in *Sources* option we need to add *Q.cpp* file, and in *Headers* option we need to add *Q.h*, *csize.h*, *marker.h*, *param.h*, *tables.h*, and *types.h* files. For *Proj_MB_4*, in *Sources* option we need to add *VLE.cpp* file, and in *Headers* option we need to add *VLE.h*, *csize.h*, *marker.h*, *param.h*, *tables.h*, and *types.h* files. For *Proj_MB_5*, in *Sources* option we need to add *Video_out.cpp* file, and in *Headers* option we need to add *Video_out.h*, *csize.h*, *marker.h*, *param.h*, *tables.h*, and *types.h* files. The complete modified project of this heterogeneous and hierarchical embedded system can be found in the CVS repository:

docs/students/WeiZhong/experiment/M_JPEG_4p_DCTHW.zip

6.3.2 XPS Project Execution and Results

When we finish with importing our project to XPS and all of the modifications for our project, we start to use XPS to generate the final bitstream file. The bitstream file is used to configure the FPGA chip to implement the M-JPEG encoder application. We use the following commands that can be found in the menu option **Tools** in XPS to generate the final bitstream file step by step.

- **Generate Netlist**: This command uses the platform building tool *PlatGen* with the MHS file as input. It produces system netlist files in NGC format.
- **Generate Bitstream**: This command uses the *xflow* tool with the NGC netlist files as input. The *fast_runtime.opt* and *bitgen.ut* files in the *etc* directory of our project are used to set some options of the *xflow* tool. The *xflow* tool generates the bitstream file — *system.bit* for the FPGA. This file is located in directory *implementation* of our project.
- **Generate Libraries**: This command uses 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.
- **Build All User Applications**: This command uses the cross compiler *mc-gcc*. This compiler generates several ELF executable files, one for each processor in the system, by compiling the program code for each processor. If *LibGen* has not been executed, this command first executes *LibGen*.
- **Update Bitstream**: This command uses the tool *bitinit*. This is the stage where the hardware and the software flows are merged. If the above commands have not been executed, this command will execute them one by one. Finally, we can get the final bitstream file — *download.bit* file in the *implementation* directory of our project that contains the entire FPGA configuration information including both the software and the hardware information of our heterogeneous and hierarchical embedded system.

In order to download the final bitstream file onto the target FPGA board and test our heterogeneous and hierarchical embedded system to get the resulting data, we need to use a software program in an outside host processor to communicate with our target board — ADM-XRC-II board. The software program uses the ADM-XRC application-programming interface (API) to takes care of open, close and device I/O control calls to the driver of the ADM-XRC-II board. We compile and run the software program with Microsoft Visual C++ 6.0. The main code of the software program is shown in Figure 6.15. In our case, we use one video frame which size is 128×128 pixels to test our M-JPEG encoder heterogeneous and hierarchical embedded system. First in line 24 of Figure 6.15, the outside host processor writes the initial video data into the off-chip memory. Then in lines 31-37, our M-JPEG encoder heterogeneous and hierarchical embedded system reads the initial video data from the off-chip memory, executes

```
0   void FPGA::MJPEG() {

        // Initialization
        fh1 = mropen("nonint.Y");
        fh2 = mropen("nonint.U");
5       fh3 = mropen("nonint.V");

        for (int n=0; n<imageH*imageV; n++) {
            rambuf[n] = (DWORD)bgetc(fh1);
        }
10      for (n=0; n<imageH*imageV/2; n++) {
            rambuf[imageH*imageV + n] = (DWORD)bgetc(fh2);
        }
        for (n=0; n<imageH*imageV/2; n++) {
            rambuf[imageH*imageV + imageH*imageV/2 + n] = (DWORD)bgetc(fh3);
15      }

        mclose(fh1);
        mclose(fh2);
        mclose(fh3);
20
        // write the packet into to Bank1 of the FPGA board
        fpgaSpace[COMMAND_REG] = cmd_Initialize; // initialise memory mode + access to banks to host
        fpgaSpace[COMMAND_REG];
        status = writeSSRAM(rambuf , 0, imageH*imageV*2, dma);
25      if (status != ADMXRC2_SUCCESS) {
            printf("exiting
n");
            exit(0);
        }

30      // process the packet in the FPGA
        fpgaSpace[COMMAND_REG] = cmd_Execute; // execute mode + access to banks to design
        fpgaSpace[COMMAND_REG];
        WORD temp;
        while(1){
35          temp = fpgaSpace[STATUS_REG];
            if (temp == stat_Finished) break;
        }

        // read the packet from Banks of the FPGA board
40      fpgaSpace[COMMAND_REG] = cmd_Read; // read memory mode + access to banks to host
        fpgaSpace[COMMAND_REG];
        DWORD index;
        DWORD clk1;
        DWORD clk3;
45      DWORD clk4;
        DWORD clk5;

        readSSRAM(&clk1, 0, 1, dma);
        readSSRAM(&clk3, bankSize + bankSize, 1, dma);
50      readSSRAM(&clk4, bankSize + bankSize + bankSize, 1, dma);
        readSSRAM(&index, bankSize + bankSize + bankSize + bankSize, 1, dma);
        readSSRAM(&clk5, bankSize + bankSize + bankSize + bankSize + index + 1, 1, dma);
        status = readSSRAM(rambuf + bankSize, bankSize + bankSize + bankSize + bankSize + 1, index, dma);
        if (status != ADMXRC2_SUCCESS) {
55      printf("Error: failed to read SSRAM
n");
        exit(1);
        }

        // Store the jpeg image
60      fh4 = mwopen("nonint.jpg");

        for (int k = 0; k < index; k++) {
        bputc(rambuf[bankSize + k],fh4);
        }
65
        mclose(fh4);

        printf("%i",(int)clk1);
        printf("");
70      printf("%i",(int)clk3);
        printf("");
        printf("%i",(int)clk4);
        printf("");
        printf("%i",(int)clk5);
75      printf("");

        return;
    }
```

Figure 6.15: The main code of the software program in the host processor.

the M-JPEG application for the initial video data and writes the resulting video data into the off-chip memory. Finally, in lines 48-53, the outside host processor reads back the resulting video data from the off-chip memory. Meanwhile, the outside host processor reads back the debugging information from the off-chip memory for our M-JPEG encoder heterogeneous and hierarchical embedded system which will be explained in Section 6.3.3. Therefore, in order to download the final bitstream file onto the target FPGA board and test our heterogeneous and hierarchical embedded system to get the resulting data, we just need to copy the final bitstream file which has been generated before to the directory of this software program. Then we compile and run this software program with Microsoft Visual C++ 6.0. We can get the resulting video data in the outside host processor. This software program can be found in the CVS repository: *docs/students/WeiZhong/experiment/PentiumProgram.zip*

6.3.3 Debugging the Heterogeneous and Hierarchical Embedded System

In order to debug our M-JPEG encoder heterogeneous and hierarchical embedded system and evaluate the time performance of our system, we need to count the number of the clock cycles of each processor for processing the video frame with the M-JPEG application.

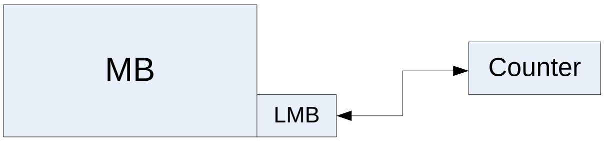


Figure 6.16: *MicroBlaze* processor connect to *Counter* component via LMB bus.

We need a custom IP core named *clock_cycle_counter_v1_00_a* for counting the number of the clock cycles of each processor. Figure 6.16 shows that a *MicroBlaze* processor use LMB bus to connect to a *clock_cycle_counter_v1_00_a* component in order to count the number of the clock cycles. As an example, we use the component *clock_cycle_counter_P1* in lines 69-76 of the MHS file shown in Appendix A. In order to make an outside host processor get the number of the clock cycles, we also need to store the number of the clock cycles in the off-chip memories. Because we add the --*debugger* option when we run our ESPAM tool, this --*debugger* option tells our ESPAM tool to generate the component which is used for debugging. Our ESPAM tool automatically copies the pcore of *clock_cycle_counter_v1_00_a* to the *pcores* directory of our project and store the number of the clock cycles in a variable in the program code of each processor. However, we still need to manually do the modification in the program code of each processor for storing the number of the clock cycles in the off-chip memories. As an example, we can see the modified program code of processor *P_1* which is shown in Figure 6.9. In lines 9-10, we define a variable for storing the number of the clock cycles and initialize the *clock_cycle_counter_v1_00_a* component by setting the initial value to 0. In lines 74-75, first we store the number of the clock cycles in the variable which is defined before and then store the value of this variable in the off-chip memory. Finally, by using the software program in an outside host processor which has been explained in Section 6.3.2, the outside host processor can read back the number of the clock cycles of each processor from the off-chip memories. Lines 48-52 in Figure 6.15 show how an outside host processor read back the number of the clock cycles of each processor from the off-chip memories.

Chapter 7

Summary and Conclusions

In this thesis, first we propose a system design methodology which is used to close the *Implementation Gap* between the *System-level* specification of multiprocessor embedded systems and the *RTL-level* specification of multiprocessor embedded systems. We have developed a tool called ESPAM (Embedded System-level Platform synthesis and Application Mapping) to implement this system design methodology. Our ESPAM tool allows designers to specify a multiprocessor embedded system at a high level of abstraction (*System-level*), then it refines this specification and systematically and automatically converts this specification to a *RTL-level* specification. Second, we introduce our view on an embedded system with heterogeneous and hierarchical architecture and prove that it is possible to implement systematically and automatically such embedded system as heterogeneous and hierarchical architecture using the ESPAM technology. Third, we introduce the construction of an interface of an embedded system with the outside world which can be used to efficiently communicate between the system and the outside world, such as an outside host processor, via off-chip memories. We also have explained the approach about how to make the ESPAM tool automatically generate the interface when it maps an application onto a multiprocessor platform.

In Chapter 1, we have explained that modern complex embedded applications lead to the situation that a single processor embedded system architecture can no longer meet the performance requirements of these applications. Because of this fact, several problems emerge. The first problem is how to design systematically and automatically a multiprocessor embedded system. The second problem is how to implement an embedded system as heterogeneous and hierarchical architecture systematically and automatically. The third problem is how to construct an efficient interface of an embedded system with the outside world. First, we need to develop a system design methodology to efficiently and effectively map the concurrent model of an application onto a multiprocessor embedded system platform in a systematic and automated way. Second, we need to give the procedure which explains how to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture. Third, we need to construct an efficient interface of an embedded system with the outside world.

In Chapter 2, we have given a detailed description of our system design methodology which is implemented in our ESPAM tool – Embedded System-level Platform Synthesis and Application Mapping. The description of our system design methodology follows the process of how the ESPAM tool bridge the *Implementation Gap* between the *System-level* specification of an embedded system and the *RTL-level* specification of an embedded system. The *System-level* specification consists of three parts which are *Platform Specification*, *Application Specification* and *Mapping Specification*. In our ESPAM design methodology, we use the Kahn Process Networks (KPN) model of computation for *Application Specification*. We use the COMPAAN tool to automatically transforms an application which is specified in a sequential model of computation into a KPN model of computation making the task-level parallelism available in an application explicit. First, ESPAM constructs a platform instance according to a *Platform Specification* and runs a consistency check on this instance. This platform instance is an abstract model and at this step no information about the target physical platform is taken into account. Such platform instance consists of the generic parameterized system components. At the second step, ESPAM refines the abstract platform model to an elaborate parameterized RTL model which is ready for an implementation on a target physical platform. At last, ESPAM generates the program code for each processor in the multiprocessor embedded system platform according to the *Application Specification* and *Mapping Specification*. At present, our ESPAM tool can systematically synthesize a platform and automatically generate all necessary files for an XPS project according to *Platform Specification*, *Application Specification* and *Mapping Specification*. In our ESPAM tool, the *Visitor Pattern* mechanism is used to generate an XPS project.

In Chapter 3, we introduce a heterogeneous and hierarchical architecture, and the differences between a homogeneous architecture and a heterogeneous and hierarchical architecture, and prove that it is possible to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture using the ESPAM technology. A homogeneous architecture means all of the components which compose an embedded system platform belong to the same type. A heterogeneous architecture means different types of processes are executed by different types of components which compose an embedded system platform. The hierarchical architecture which we have defined earlier means the complex process of an application is mapped onto several components which compose a sub-network on an embedded system platform. Due to the complexity of modern applications, such as high throughput multimedia, imaging and digital signal processing which usually include complicated algorithms, different types of processes of an application are suitable for being executed by different types of components on an embedded system platform. Therefore, an embedded system as homogeneous architecture is no longer suitable for modern applications. In order to meet the required performance of various applications we need to implement systematically and automatically a heterogeneous and hierarchical architecture on an embedded system platform. In this chapter, we give the procedure which explains how to implement systematically and automatically an embedded system as heterogeneous and hierarchical architecture which contains processor components and a dedicated hardware IP core. In our case, the processor components use FIFOs to communicate with each other. In order to make the dedicated hardware IP core can communicate with the processor components, the dedicated hardware IP core should has the FIFO input and output interfaces. We use the LAURA tool [6] which has been developed at the Leiden Embedded Research Center (LERC) to generate the dedicated hardware IP core which contains the FIFO input and output interfaces. In this heterogeneous and hierarchical architecture, we use the dedicated hardware IP core to execute the most complicated process of an application repetitively and use the processor components to execute the other processes of the application in order to get good performance of execution time. In this way, we can prove that it is possible to implement systematically and automatically an embedded system as heterogenous and hierarchical architecture using the ESPAM technology.

In Chapter 4, we explain how to construct an efficient interface of an embedded system with the outside world step by step. This interface can be used to communicate between embedded systems and the outside world via off-chip memories. The target FPGA platform on which we implement our interface of an embedded system with the outside world is the ADM-XRC-II board which supports high performance PCI operation without the need to integrate proprietary cores into the FPGA. The interface of an embedded system with the outside world consists of four main parts — Host Interface, Function Design, Multiplexer and Buffer. The Function Design is a multiprocessor system which is used to implement different types of embedded system applications. Besides these four main parts, our interface has two connection parts. One connection part is a custom controller for a processor in the Function Design to connect to the off-chip ZBT SSRAM. The other connection part includes two components which are used to transfer control signals and status signals between the Host Interface and the Function Design. We also make the ESPAM tool can automatically generate the interface when it maps an application onto a multiprocessor platform.

In Chapter 5, two case studies are given. The first case study is about a M-JPEG multiprocessor system with homogeneous architecture which is used to evaluate the design methodology in our ESPAM tool presented in Chapter 2 and validate the interface of an embedded system with the outside world explained in Chapter 4. The second case study is about a M-JPEG multiprocessor system with heterogeneous and hierarchical architecture which is used to validate the procedure of implementing an embedded system as heterogeneous and hierarchical architecture and evaluate the heterogeneous and hierarchical architecture introduced in Chapter 3. From the result of the first case study, we prove that with mapping the same application a multiprocessor embedded system has better time performance than a single processor embedded system. We find out that based on requirement of an application we can map the application onto any number of *MicroBlaze* processors embedded system platform. The only limitation is whether the target FPGA board has enough on-chip memories and reconfigurable resources. We also find out that there are still several tasks we need to do after the XPS project automatic generation using our ESPAM tool, such as modifying the memory allocation, importing implementations of the function calls in processors and changing function calls in processors' program code and so on. From the result of the second case study, we prove that with mapping the same application a heterogeneous and hierarchical embedded system has better time performance than a homogeneous embedded system.

In Chapter 6, a tutorial with example of heterogeneous and hierarchical embedded system design is given. This tutorial gives the detailed steps for how to design a heterogeneous and hierarchical embedded system using the COMPAAN tool, our ESPAM tool and the commercial synthesis tool Xilinx Platform Studio (XPS). First, we generate an XPS project of homogeneous embedded system for an application. Second, we change the XPS project of homogeneous embedded system to heterogeneous and hierarchical embedded system by hand. Third, we import the project into XPS and use XPS to generate the final bitstream file. At last, we use a software program in an outside host processor to download the final bitstream file onto the target FPGA board and test the heterogeneous and hierarchical embedded system to get the resulting data.

In conclusion, by using our ESPAM tool, designers can easily design multiprocessor embedded systems for various applications. By implementing embedded systems as heterogeneous and hierarchical architecture, we can make embedded systems of various applications meet required performance. By using our interface of an embedded system with the outside world, an embedded system can efficiently communicate with the outside world. Because of time limitations related to the preparation of this thesis, currently our ESPAM tool can not generate automatically an embedded system as heterogeneous and hierarchical architecture. However, this is only an implementation issue that has to be addressed in the future. In this thesis we have already proven that it is possible for our ESPAM tool to generate systematically and automatically an embedded system as heterogeneous and hierarchical architecture by giving a detailed procedure. Therefore, in the future the people who continue developing our ESPAM tool can work on the implementation issue related to the automatic generation of heterogeneous and hierarchical systems.

Appendix A

MHS File for M-JPEG Encoder Five Processors Homogeneous Embedded System

```
PARAMETER VERSION = 2.1.0
PORT lclk = lclk, DIR = IN
PORT mclk = mclk, DIR = IN
PORT ramclki = ramclki, VEC = [1:0], DIR = IN
PORT ramclko = ramclko, VEC = [1:0], DIR = OUT
PORT lreseto_l = lreseto_l, DIR = IN
PORT lwrite = lwrite, DIR = IN
PORT lads_l = lads_l, DIR = IN
PORT lblast_l = lblast_l, DIR = IN
PORT lbterm_l = lbterm_l, DIR = INOUT
PORT ld = ld, VEC = [31:0], DIR = INOUT
PORT la = la, VEC = [23:2], DIR = IN
PORT lreadyi_l = lreadyi_l, DIR = OUT
PORT lbe_l = lbe_l, VEC = [3:0], DIR = IN
PORT fholda = fholda, DIR = IN
PORT ra0 = ra0, VEC = [19:0], DIR = OUT
PORT rd0 = rd0, VEC = [31:0], DIR = INOUT
PORT rc0 = rc0, VEC = [8:0], DIR = OUT
PORT ra1 = ra1, VEC = [19:0], DIR = OUT
PORT rd1 = rd1, VEC = [31:0], DIR = INOUT
PORT rc1 = rc1, VEC = [8:0], DIR = OUT
PORT ra2 = ra2, VEC = [19:0], DIR = OUT
PORT rd2 = rd2, VEC = [31:0], DIR = INOUT
PORT rc2 = rc2, VEC = [8:0], DIR = OUT
PORT ra3 = ra3, VEC = [19:0], DIR = OUT
PORT rd3 = rd3, VEC = [31:0], DIR = INOUT
PORT rc3 = rc3, VEC = [8:0], DIR = OUT
PORT ra4 = ra4, VEC = [19:0], DIR = OUT
PORT rd4 = rd4, VEC = [31:0], DIR = INOUT
PORT rc4 = rc4, VEC = [8:0], DIR = OUT
PORT ra5 = ra5, VEC = [19:0], DIR = OUT
PORT rd5 = rd5, VEC = [31:0], DIR = INOUT
PORT rc5 = rc5, VEC = [8:0], DIR = OUT

BEGIN lmb_v10
 PARAMETER INSTANCE = PBUS_MB_1
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END

BEGIN lmb_v10
 PARAMETER INSTANCE = DBUS_MB_1
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END

BEGIN opb_v20
 PARAMETER INSTANCE = mb_opb_1
 PARAMETER HW_VER = 1.10.c
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT OPB_Clk = sys_clk_s
END

BEGIN fin_ctrl
 PARAMETER INSTANCE = fin_ctrl_P1
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf9000000
 PARAMETER C_HIGHADDR = 0xf900000f
 PARAMETER C_AB = 8
 BUS_INTERFACE SLMB = DBUS_MB_1
 PORT Sl_FinOut = net_fin_signal_P1
END

BEGIN clock_cycle_counter
 PARAMETER INSTANCE = clock_cycle_counter_P1
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf8000000
 PARAMETER C_HIGHADDR = 0xf8000003
 BUS_INTERFACE SLMB = DBUS_MB_1
 PORT LMB_Clk = sys_clk_s
END

BEGIN microblaze
 PARAMETER INSTANCE = MB_1
 PARAMETER HW_VER = 4.00.a
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 BUS_INTERFACE MFSL0 = FIFO_MB_1_Out_1
 BUS_INTERFACE MFSL1 = FIFO_MB_1_Out_2
 BUS_INTERFACE MFSL2 = FIFO_MB_1_Out_3
 BUS_INTERFACE MFSL3 = FIFO_MB_1_Out_4
 BUS_INTERFACE MFSL4 = FIFO_MB_1_Out_5
 BUS_INTERFACE MFSL5 = FIFO_MB_1_Out_6
 BUS_INTERFACE MFSL6 = FIFO_MB_1_Out_7
 BUS_INTERFACE MFSL7 = FIFO_MB_1_Out_8
 BUS_INTERFACE DLMB = DBUS_MB_1
 BUS_INTERFACE ILMB = PBUS_MB_1
 BUS_INTERFACE DOPB = mb_opb_1
 PARAMETER C_FSL_LINKS = 8
 PORT CLK = sys_clk_s
END

BEGIN lmb_v10
 PARAMETER INSTANCE = PBUS_MB_2
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END
```

```
BEGIN lmb_v10
 PARAMETER INSTANCE = DBUS_MB_2
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END

BEGIN opb_v20
 PARAMETER INSTANCE = mb_opb_2
 PARAMETER HW_VER = 1.10.c
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT OPB_Clk = sys_clk_s
END

BEGIN fin_ctrl
 PARAMETER INSTANCE = fin_ctrl_P2
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf9000000
 PARAMETER C_HIGHADDR = 0xf900000f
 PARAMETER C_AB = 8
 BUS_INTERFACE SLMB = DBUS_MB_2
 PORT Sl_FinOut = net_fin_signal_P2
END

BEGIN clock_cycle_counter
 PARAMETER INSTANCE = clock_cycle_counter_P2
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf8000000
 PARAMETER C_HIGHADDR = 0xf8000003
 BUS_INTERFACE SLMB = DBUS_MB_2
 PORT LMB_Clk = sys_clk_s
END

BEGIN microblaze
 PARAMETER INSTANCE = MB_2
 PARAMETER HW_VER = 4.00.a
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 BUS_INTERFACE MFSL0 = FIFO_MB_2_Out_1
 BUS_INTERFACE SFSL0 = FIFO_MB_1_Out_1
 BUS_INTERFACE DLMB = DBUS_MB_2
 BUS_INTERFACE ILMB = PBUS_MB_2
 BUS_INTERFACE DOPB = mb_opb_2
 PARAMETER C_FSL_LINKS = 1
 PORT CLK = sys_clk_s
END

BEGIN lmb_v10
 PARAMETER INSTANCE = PBUS_MB_3
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END

BEGIN lmb_v10
 PARAMETER INSTANCE = DBUS_MB_3
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END

BEGIN opb_v20
 PARAMETER INSTANCE = mb_opb_3
 PARAMETER HW_VER = 1.10.c
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT OPB_Clk = sys_clk_s
END

BEGIN fin_ctrl
 PARAMETER INSTANCE = fin_ctrl_P3
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf9000000
 PARAMETER C_HIGHADDR = 0xf900000f
 PARAMETER C_AB = 8
 BUS_INTERFACE SLMB = DBUS_MB_3
 PORT Sl_FinOut = net_fin_signal_P3
END

BEGIN clock_cycle_counter
 PARAMETER INSTANCE = clock_cycle_counter_P3
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf8000000
 PARAMETER C_HIGHADDR = 0xf8000003
 BUS_INTERFACE SLMB = DBUS_MB_3
 PORT LMB_Clk = sys_clk_s
END

BEGIN microblaze
 PARAMETER INSTANCE = MB_3
 PARAMETER HW_VER = 4.00.a
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 BUS_INTERFACE MFSL0 = FIFO_MB_3_Out_1
 BUS_INTERFACE SFSL0 = FIFO_MB_2_Out_1
 BUS_INTERFACE SFSL1 = FIFO_MB_1_Out_2
 BUS_INTERFACE SFSL2 = FIFO_MB_1_Out_3
 BUS_INTERFACE DLMB = DBUS_MB_3
 BUS_INTERFACE ILMB = PBUS_MB_3
 BUS_INTERFACE DOPB = mb_opb_3
 PARAMETER C_FSL_LINKS = 3
 PORT CLK = sys_clk_s
END

BEGIN lmb_v10
 PARAMETER INSTANCE = PBUS_MB_4
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END

BEGIN lmb_v10
 PARAMETER INSTANCE = DBUS_MB_4
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END

BEGIN opb_v20
 PARAMETER INSTANCE = mb_opb_4
 PARAMETER HW_VER = 1.10.c
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT OPB_Clk = sys_clk_s
END

BEGIN fin_ctrl
 PARAMETER INSTANCE = fin_ctrl_P4
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf9000000
 PARAMETER C_HIGHADDR = 0xf900000f
 PARAMETER C_AB = 8
 BUS_INTERFACE SLMB = DBUS_MB_4
 PORT Sl_FinOut = net_fin_signal_P4
END

BEGIN clock_cycle_counter
 PARAMETER INSTANCE = clock_cycle_counter_P4
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf8000000
 PARAMETER C_HIGHADDR = 0xf8000003
 BUS_INTERFACE SLMB = DBUS_MB_4
 PORT LMB_Clk = sys_clk_s
END

BEGIN microblaze
 PARAMETER INSTANCE = MB_4
 PARAMETER HW_VER = 4.00.a
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 BUS_INTERFACE MFSL0 = FIFO_MB_4_Out_1
 BUS_INTERFACE SFSL0 = FIFO_MB_3_Out_1
 BUS_INTERFACE SFSL1 = FIFO_MB_1_Out_4
 BUS_INTERFACE SFSL2 = FIFO_MB_1_Out_5
 BUS_INTERFACE SFSL3 = FIFO_MB_1_Out_6
 BUS_INTERFACE SFSL4 = FIFO_MB_1_Out_7
 BUS_INTERFACE DLMB = DBUS_MB_4
 BUS_INTERFACE ILMB = PBUS_MB_4
 BUS_INTERFACE DOPB = mb_opb_4
 PARAMETER C_FSL_LINKS = 5
 PORT CLK = sys_clk_s
END

BEGIN lmb_v10
 PARAMETER INSTANCE = PBUS_MB_5
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END

BEGIN lmb_v10
 PARAMETER INSTANCE = DBUS_MB_5
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT LMB_Clk = sys_clk_s
END

BEGIN opb_v20
 PARAMETER INSTANCE = mb_opb_5
 PARAMETER HW_VER = 1.10.c
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = net_design_rst
 PORT OPB_Clk = sys_clk_s
END
```

```
BEGIN fin_ctrl
 PARAMETER INSTANCE = fin_ctrl_P5
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf9000000
 PARAMETER C_HIGHADDR = 0xf900000f
 PARAMETER C_AB = 8
 BUS_INTERFACE SLMB = DBUS_MB_5
 PORT Sl_FinOut = net_fin_signal_P5
END

BEGIN clock_cycle_counter
 PARAMETER INSTANCE = clock_cycle_counter_P5
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf8000000
 PARAMETER C_HIGHADDR = 0xf8000003
 BUS_INTERFACE SLMB = DBUS_MB_5
 PORT LMB_Clk = sys_clk_s
END

BEGIN microblaze
 PARAMETER INSTANCE = MB_5
 PARAMETER HW_VER = 4.00.a
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 BUS_INTERFACE SFSL0 = FIFO_MB_1_Out_8
 BUS_INTERFACE SFSL1 = FIFO_MB_1_Out_9
 BUS_INTERFACE SFSL2 = FIFO_MB_1_Out_10
 BUS_INTERFACE SFSL3 = FIFO_MB_4_Out_1
 BUS_INTERFACE DLMB = DBUS_MB_5
 BUS_INTERFACE ILMB = PBUS_MB_5
 BUS_INTERFACE DOPB = mb_opb_5
 PARAMETER C_FSL_LINKS = 4
 PORT CLK = sys_clk_s
END

BEGIN zbt_main
 PARAMETER INSTANCE = host_zbt_main
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE HOST_BUFF_0_PORT = buff_rd_0
 BUS_INTERFACE HOST_BUFF_1_PORT = buff_rd_1
 BUS_INTERFACE HOST_BUFF_2_PORT = buff_rd_2
 BUS_INTERFACE HOST_BUFF_3_PORT = buff_rd_3
 BUS_INTERFACE HOST_BUFF_4_PORT = buff_rd_4
 BUS_INTERFACE HOST_BUFF_5_PORT = buff_rd_5
 BUS_INTERFACE HOST_MUX_PORT = mux_to_host
 PORT lclk = lclk
 PORT mclk = mclk
 PORT ramclko = ramclko
 PORT ramclki = ramclki
 PORT lreseto_l = lreseto_l
 PORT lwrite = lwrite
 PORT lads_l = lads_l
 PORT lblast_l = lblast_l
 PORT lbterm_l = lbterm_l
 PORT ld = ld
 PORT la = la
 PORT lreadyi_l = lreadyi_l
 PORT lbe_l = lbe_l
 PORT fholda = fholda
 PORT CLK_out = sys_clk_s
 PORT RST_out = sys_rst_s
 PORT COMMAND_REG = net_command
 PORT DESIGN_STAT_REG = net_design_status
END

BEGIN host_design_ctrl
 PARAMETER INSTANCE = host_design_controller
 PARAMETER HW_VER = 1.00.a
 PARAMETER N_FIN = 5
 PORT RST = sys_rst_s
 PORT COMMAND_REG = net_command
 PORT STATUS_REG = net_design_status
 PORT RST_OUT = net_design_rst
 PORT FIN_REG_0 = net_fin_signal_P1
 PORT FIN_REG_1 = net_fin_signal_P2
 PORT FIN_REG_2 = net_fin_signal_P3
 PORT FIN_REG_3 = net_fin_signal_P4
 PORT FIN_REG_4 = net_fin_signal_P5
END

BEGIN mux
 PARAMETER INSTANCE = multiplexer
 PARAMETER HW_VER = 1.00.a
 PARAMETER N_MUX = 5
 BUS_INTERFACE MUX_BUFF_PORT = buff_to_mux
 BUS_INTERFACE MUX_DESIGN_0_PORT = mux_design_0
 BUS_INTERFACE MUX_DESIGN_1_PORT = mux_design_1
 BUS_INTERFACE MUX_DESIGN_2_PORT = mux_design_2
 BUS_INTERFACE MUX_DESIGN_3_PORT = mux_design_3
 BUS_INTERFACE MUX_DESIGN_4_PORT = mux_design_4
 BUS_INTERFACE MUX_HOST_PORT = mux_to_host
 PORT ra0 = ra0
 PORT ra1 = ra1
 PORT ra2 = ra2
 PORT ra3 = ra3
 PORT ra4 = ra4
 PORT ra5 = ra5
 PORT rc0 = rc0
 PORT rc1 = rc1
 PORT rc2 = rc2
 PORT rc3 = rc3
 PORT rc4 = rc4
 PORT rc5 = rc5
 PORT RST = sys_rst_s
 PORT CNTRL = net_command
END

BEGIN buffers
 PARAMETER INSTANCE = buff
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE BUFF_MUX_PORT = buff_to_mux
 BUS_INTERFACE BUFF_RD_0_PORT = buff_rd_0
 BUS_INTERFACE BUFF_RD_1_PORT = buff_rd_1
 BUS_INTERFACE BUFF_RD_2_PORT = buff_rd_2
 BUS_INTERFACE BUFF_RD_3_PORT = buff_rd_3
 BUS_INTERFACE BUFF_RD_4_PORT = buff_rd_4
 BUS_INTERFACE BUFF_RD_5_PORT = buff_rd_5
 PORT rd0 = rd0
 PORT rd1 = rd1
 PORT rd2 = rd2
 PORT rd3 = rd3
 PORT rd4 = rd4
 PORT rd5 = rd5
END

BEGIN opb_zbt_controller
 PARAMETER INSTANCE = ZBT_CTRL_1
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf0000000
 PARAMETER C_HIGHADDR = 0xf00fffff
 PARAMETER C_EXTERNAL_DLL = 1
 PARAMETER C_ZBT_ADDR_SIZE = 20
 BUS_INTERFACE SOPB = mb_opb_1
 BUS_INTERFACE DESIGN_BUFF_PORT = buff_rd_0
 BUS_INTERFACE DESIGN_MUX_PORT = mux_design_0
END

BEGIN opb_zbt_controller
 PARAMETER INSTANCE = ZBT_CTRL_2
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf0000000
 PARAMETER C_HIGHADDR = 0xf00fffff
 PARAMETER C_EXTERNAL_DLL = 1
 PARAMETER C_ZBT_ADDR_SIZE = 20
 BUS_INTERFACE SOPB = mb_opb_2
 BUS_INTERFACE DESIGN_BUFF_PORT = buff_rd_1
 BUS_INTERFACE DESIGN_MUX_PORT = mux_design_1
END

BEGIN opb_zbt_controller
 PARAMETER INSTANCE = ZBT_CTRL_3
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf0000000
 PARAMETER C_HIGHADDR = 0xf00fffff
 PARAMETER C_EXTERNAL_DLL = 1
 PARAMETER C_ZBT_ADDR_SIZE = 20
 BUS_INTERFACE SOPB = mb_opb_3
 BUS_INTERFACE DESIGN_BUFF_PORT = buff_rd_2
 BUS_INTERFACE DESIGN_MUX_PORT = mux_design_2
END

BEGIN opb_zbt_controller
 PARAMETER INSTANCE = ZBT_CTRL_4
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf0000000
 PARAMETER C_HIGHADDR = 0xf00fffff
 PARAMETER C_EXTERNAL_DLL = 1
 PARAMETER C_ZBT_ADDR_SIZE = 20
 BUS_INTERFACE SOPB = mb_opb_4
 BUS_INTERFACE DESIGN_BUFF_PORT = buff_rd_3
 BUS_INTERFACE DESIGN_MUX_PORT = mux_design_3
END

BEGIN opb_zbt_controller
 PARAMETER INSTANCE = ZBT_CTRL_5
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf0000000
 PARAMETER C_HIGHADDR = 0xf00fffff
 PARAMETER C_EXTERNAL_DLL = 1
 PARAMETER C_ZBT_ADDR_SIZE = 20
 BUS_INTERFACE SOPB = mb_opb_5
 BUS_INTERFACE DESIGN_BUFF_PORT = buff_rd_4
 BUS_INTERFACE DESIGN_MUX_PORT = mux_design_4
END
```

```
BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_1
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_2_Out_1
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_2
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_3
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_3_Out_1
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_4
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_5
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_6
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_7
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_8
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_9
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_10
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_4_Out_1
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 512
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fifo_if_ctrl
 PARAMETER INSTANCE = CTRL_MB_1_FIFOs
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xc0800000
 PARAMETER C_HIGHADDR = 0xc080000f
 PARAMETER C_AB = 8
 PARAMETER C_FIFO_WRITE = 2
 PARAMETER C_FIFO_READ = 0
 BUS_INTERFACE FIFO_WRITE_1 = FIFO_MB_1_Out_9
 BUS_INTERFACE FIFO_WRITE_2 = FIFO_MB_1_Out_10
 BUS_INTERFACE SLMB = DBUS_MB_1
END

BEGIN bram_block
 PARAMETER INSTANCE = BRAM1_MB_1
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = BUS_DCTRL_BRAM1_MB_1
 BUS_INTERFACE PORTB = BUS_PCTRL_BRAM1_MB_1
END
```

```
BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = DCTRL_BRAM1_MB_1
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x0000ffff
 BUS_INTERFACE SLMB = DBUS_MB_1
 BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM1_MB_1
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = PCTRL_BRAM1_MB_1
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x0000ffff
 BUS_INTERFACE SLMB = PBUS_MB_1
 BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM1_MB_1
END

BEGIN bram_block
 PARAMETER INSTANCE = BRAM2_MB_1
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = BUS_DCTRL_BRAM2_MB_1
 BUS_INTERFACE PORTB = BUS_PCTRL_BRAM2_MB_1
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = DCTRL_BRAM2_MB_1
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00010000
 PARAMETER C_HIGHADDR = 0x00017fff
 BUS_INTERFACE SLMB = DBUS_MB_1
 BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM2_MB_1
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = PCTRL_BRAM2_MB_1
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00010000
 PARAMETER C_HIGHADDR = 0x00017fff
 BUS_INTERFACE SLMB = PBUS_MB_1
 BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM2_MB_1
END

BEGIN bram_block
 PARAMETER INSTANCE = BRAM1_MB_2
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = BUS_DCTRL_BRAM1_MB_2
 BUS_INTERFACE PORTB = BUS_PCTRL_BRAM1_MB_2
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = DCTRL_BRAM1_MB_2
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
 BUS_INTERFACE SLMB = DBUS_MB_2
 BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM1_MB_2
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = PCTRL_BRAM1_MB_2
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
 BUS_INTERFACE SLMB = PBUS_MB_2
 BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM1_MB_2
END

BEGIN bram_block
 PARAMETER INSTANCE = BRAM1_MB_3
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = BUS_DCTRL_BRAM1_MB_3
 BUS_INTERFACE PORTB = BUS_PCTRL_BRAM1_MB_3
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = DCTRL_BRAM1_MB_3
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00003fff
 BUS_INTERFACE SLMB = DBUS_MB_3
 BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM1_MB_3
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = PCTRL_BRAM1_MB_3
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00003fff
 BUS_INTERFACE SLMB = PBUS_MB_3
 BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM1_MB_3
END

BEGIN bram_block
 PARAMETER INSTANCE = BRAM1_MB_4
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = BUS_DCTRL_BRAM1_MB_4
 BUS_INTERFACE PORTB = BUS_PCTRL_BRAM1_MB_4
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = DCTRL_BRAM1_MB_4
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
 BUS_INTERFACE SLMB = DBUS_MB_4
 BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM1_MB_4
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = PCTRL_BRAM1_MB_4
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
 BUS_INTERFACE SLMB = PBUS_MB_4
 BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM1_MB_4
END

BEGIN bram_block
 PARAMETER INSTANCE = BRAM1_MB_5
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = BUS_DCTRL_BRAM1_MB_5
 BUS_INTERFACE PORTB = BUS_PCTRL_BRAM1_MB_5
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = DCTRL_BRAM1_MB_5
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
 BUS_INTERFACE SLMB = DBUS_MB_5
 BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM1_MB_5
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = PCTRL_BRAM1_MB_5
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
 BUS_INTERFACE SLMB = PBUS_MB_5
 BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM1_MB_5
END
```

Appendix B

MSS File for M-JPEG Encoder Five Processors Homogeneous Embedded System

```
PARAMETER VERSION = 2.2.0

BEGIN OS
PARAMETER OS_NAME = standalone
PARAMETER OS_VER = 1.00.a
PARAMETER PROC_INSTANCE = MB_1
END

BEGIN PROCESSOR
PARAMETER DRIVER_NAME = cpu
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = MB_1
PARAMETER COMPILER = mb-gcc
PARAMETER ARCHIVER = mb-ar
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = opbarb
PARAMETER DRIVER_VER = 1.02.a
PARAMETER HW_INSTANCE = mb_opb_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = fin_ctrl_P1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = clock_cycle_counter_P1
END

BEGIN OS
PARAMETER OS_NAME = standalone
PARAMETER OS_VER = 1.00.a
PARAMETER PROC_INSTANCE = MB_2
END

BEGIN PROCESSOR
PARAMETER DRIVER_NAME = cpu
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = MB_2
PARAMETER COMPILER = mb-gcc
PARAMETER ARCHIVER = mb-ar
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = opbarb
PARAMETER DRIVER_VER = 1.02.a
PARAMETER HW_INSTANCE = mb_opb_2
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = fin_ctrl_P2
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = clock_cycle_counter_P2
END

BEGIN OS
PARAMETER OS_NAME = standalone
PARAMETER OS_VER = 1.00.a
PARAMETER PROC_INSTANCE = MB_3
END

BEGIN PROCESSOR
PARAMETER DRIVER_NAME = cpu
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = MB_3
PARAMETER COMPILER = mb-gcc
PARAMETER ARCHIVER = mb-ar
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = opbarb
PARAMETER DRIVER_VER = 1.02.a
PARAMETER HW_INSTANCE = mb_opb_3
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = fin_ctrl_P3
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = clock_cycle_counter_P3
END

BEGIN OS
PARAMETER OS_NAME = standalone
PARAMETER OS_VER = 1.00.a
PARAMETER PROC_INSTANCE = MB_4
END

BEGIN PROCESSOR
PARAMETER DRIVER_NAME = cpu
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = MB_4
PARAMETER COMPILER = mb-gcc
PARAMETER ARCHIVER = mb-ar
END
```

```
BEGIN DRIVER
PARAMETER DRIVER_NAME = opbarb
PARAMETER DRIVER_VER = 1.02.a
PARAMETER HW_INSTANCE = mb_opb_4
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = fin_ctrl_P4
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = clock_cycle_counter_P4
END

BEGIN OS
PARAMETER OS_NAME = standalone
PARAMETER OS_VER = 1.00.a
PARAMETER PROC_INSTANCE = MB_5
END

BEGIN PROCESSOR
PARAMETER DRIVER_NAME = cpu
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = MB_5
PARAMETER COMPILER = mb-gcc
PARAMETER ARCHIVER = mb-ar
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = opbarb
PARAMETER DRIVER_VER = 1.02.a
PARAMETER HW_INSTANCE = mb_opb_5
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = fin_ctrl_P5
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = clock_cycle_counter_P5
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = host_zbt_main
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = host_design_controller
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = multiplexer
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = buff
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = ZBT_CTRL_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = ZBT_CTRL_2
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = ZBT_CTRL_3
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = ZBT_CTRL_4
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = ZBT_CTRL_5
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_2_Out_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_2
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_3
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_3_Out_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_4
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_5
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_6
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_7
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_8
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_9
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_1_Out_10
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = FIFO_MB_4_Out_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = generic
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = CTRL_MB_1_FIFOs
END
```

```

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = DCTRL_BRAM1_MB_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = PCTRL_BRAM1_MB_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = DCTRL_BRAM2_MB_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = PCTRL_BRAM2_MB_1
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = DCTRL_BRAM1_MB_2
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = PCTRL_BRAM1_MB_2
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = DCTRL_BRAM1_MB_3
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = PCTRL_BRAM1_MB_3
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = DCTRL_BRAM1_MB_4
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = PCTRL_BRAM1_MB_4
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = DCTRL_BRAM1_MB_5
END

BEGIN DRIVER
PARAMETER DRIVER_NAME = bram
PARAMETER DRIVER_VER = 1.00.a
PARAMETER HW_INSTANCE = PCTRL_BRAM1_MB_5
END
```

Bibliography

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

[2] 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.

[3] 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.

[4] 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.

[5] Kai Huang and Ji Gu. *Automatic Platform Synthesis and Application Mapping for Multiprocessor Systems On-Chip*. master thesis, Leiden Embedded Research Center, LIACS, Leiden University, 2005.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] S. Vassiliadis, S. Wong, G.N. Gaydadjiev, K Bertels, G.K. Kuzmanov, and E. Moscu Panainte. The Molen Polymorphic Processor. *IEEE Transactions on Computers*, November 2004.

[11] S. Murat Bicer, Frank Pihofer, Graham Bardouleau, and Jeffrey Smith. Next Generation Architecture for Heterogeneous Embedded Systems. In *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, USA, June 2003.

[12] Xuejian Luan, Jing Ying, and Minghui Wu. A Heterogeneous Evolutional Architecture for Embedded Software. In *Proceedings of the Fifth International Conference on Computer and Information Technology (CIT'05)*, Shanghai, China, September 21-23 2005.

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

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

[15] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System Design using Kahn Process Networks: The C ompaan/Laura Approach. In *Proc. Int. Conference Design, Automation and Test in Europe (D ATE'04)*, pages 340–345, Paris, France, February 16-20 2004.

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

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

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

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

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

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

[22] Arun N. Netravali and Barry G. Haskell. *Digital Pictures: Representation, Compression and Standards (Applications of Communications Theory)*. Plenum Press, 1988.

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

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

[25] Platform specification format reference manual: Xilinx, inc.
http://www.xilinx.com/ise/embedded/psf_rm.pdf.

[26] Alpha Data. *ADM-XRC-II, PCI Mezzanine Card, User Guide, Version 1.5*. Alpha Data Parallel Systems Ltd, 2002.

[27] Wei Zhong. Communication between field programmable gate array chip and host processor (pentium) with memory access : Research project report. Technical report, LIACS, the Netherlands, 2005.

[28] 64-bit on-chip peripheral bus, architectural specifications, version 2.0.
http://www.xilinx.com/ise/embedded/edk_docs.htm.

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

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

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

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

[33] 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.

[34] 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.

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