Heterogeneous Multiprocessor System Design with ESPAM: Integration of Hardware IP Cores

MASTER'S THESIS

by

Ying Tao
ytao@liacs.nl

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

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

Copyright ©2006 by Ying Tao, Leiden, The Netherlands. All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without permission from the author.

Printed in the Netherlands

Contents

Acknowledgments **v**

1 Introduction **1**
1.1 ESPAM Introduction 2
1.2 Motivation 5
1.3 Related Work 7
1.4 Research Contributions 8
1.5 Thesis Organization 9

2 Heterogeneous Multiprocessor System Generation **11**
2.1 IP Core Integration in ESPAM– Basic Concept and Structure . . . 11
2.2 Structure of the IP core wrapper 15
2.2.1 Main structure and communication interface 15
2.2.2 Read Block 18
2.2.3 Execute Block 22
2.2.4 Control Block 23
2.2.5 Write Block 25
2.3 Generation of the IP Core Wrapper in the ESPAM 27
2.3.1 Information extraction from ESPAM models 27
2.3.2 IP Core Wrapper Files Generation in Xilinx Platform Studio Format 34
2.3.3 Integration Into XPS Project 38
2.4 Visitor Pattern 42

3 Case Studies **45**
3.1 Discrete Cosine Transform (DCT) 45
3.2 Sobel Edge Detection 49
3.3 Discrete Wavelet Transform (DWT) 52
3.4 A Genetic Algorithm 54
3.5 Conclusions 57

4 Getting Started: Tutorial for Heterogeneous System Design with COMPAAN/ESPAM tool chain 59

4.1 XPS Project Generation 59
4.1.1 KPN Specification Generation Using the COMPAAN tool . 60
4.1.2 Heterogeneous Embedded System Generation Using the ESPAM tool 62
4.1.3 Importing Project to XPS 67
4.2 Custom Modifications 67
4.2.1 Hardware Modifications 67
4.2.2 Software Modifications 70
4.3 XPS Project Execution and Results 74

5 Summary and Conclusions 77

Appendix 80

A Initial Matlab code for DWT application 81

B MHS file for Sobel Heterogeneous Embedded System project 83

C The main code of the software program in the host processor 89

Bibliography 91

Acknowledgments

I would like to give my special thanks to Todor Stefanov and Hristo Nikolov for their guidance and support throughout my Master thesis project. Not only much knowledge but also enlightenment are given by them during this period.

Also, I would like to acknowledge all the members in my group who helped and supported me during my thesis.

Ying Tao
Leiden, The Netherlands
December 05, 2006

Chapter 1

Introduction

Embedded systems nowadays need to support ever-increasing functionality and high flexibility, which calls for the emergence of multiprocessor systems, in place of single processor systems, for the embedded System-on-Chip platforms. The multiprocessor Systems-on-Chips (MPSoCs) are much more efficient than uniprocessor systems because they can exploit parallelism between tasks.

Because of the complexity and parallel nature of MPSoCs, the multiprocessor system brings us several problems to consider:

- How to partition an application into concurrent processes to match the parallel nature of MPSoCs. That means in case of sequential languages, such as C or Matlab, we need to convert sequential specifications into parallel specifications.
- How to map these application processes to the multiprocessor platform efficiently and systematically.
- How to specify the platform at a high level of abstraction instead of the current register transfer level (RTL). The RTL level is clearly too low for complex multiprocessor platform design.
- How to make MPSoCs flexible and re-usable, so that they can easily be modified in response to bugs, user requirements or adapted to different applications.

Bearing these aspects in mind, the Embedded System-Level Platform Synthe-

sis and Application Mapping (ESPAM) tool [1] [2] is being developed to convert systematically and automatically a System-level specification to a RTL-level specification. Previous work on ESPAM has already implemented this automatic conversion for homogeneous multiprocessor systems, which consist of only programmable processors as processing components. The processing components are connected to communicate with each other by communication components such as FIFO component, crossbar component, and bus component. This thesis focuses on further improving ESPAM by achieving automatic conversion for heterogeneous multiprocessor systems which supports both programmable processors and fixed hardware modules (dedicated hardware IP cores) as processing components.

This chapter is organized as follows. In Section 1.1, an introduction to ESPAM and its previous work is given. In Section 1.2 the motivation for this thesis project is explained. Section 1.3 presents other related work and section 1.5 describes the organization of this thesis.

1.1 ESPAM Introduction

With the applications demanding higher and higher processing performance on the embedded systems, multiprocessor systems have to be used to exploit parallelism. This requires the mapping of applications onto multiprocessor systems. However, traditionally the mapping is done manually and depends very much on the expertise of the hardware designer, who has to possess an accurate knowledge of both the underlying hardware systems and the applications.

Thus, ESPAM is developed to allow system designers to specify a system and its related applications at a higher level of abstraction called System-Level to save much design effort and time. ESPAM can convert this specification into a RTL-Level specification which can be used as input for current commercial synthesis tools. Figure 1.1 shows the system design flow of ESPAM.

The system design flow shows that two levels of specifications are involved with our ESPAM tool: *System-Level* specification and *RTL-Level* specification. The input of ESPAM is *System-Level* specification, which consists of three parts:

- *Platform Specification*: It specifies the topology of a platform using generic parameterized system components. Two types of components, processing components and communication components, are involved.

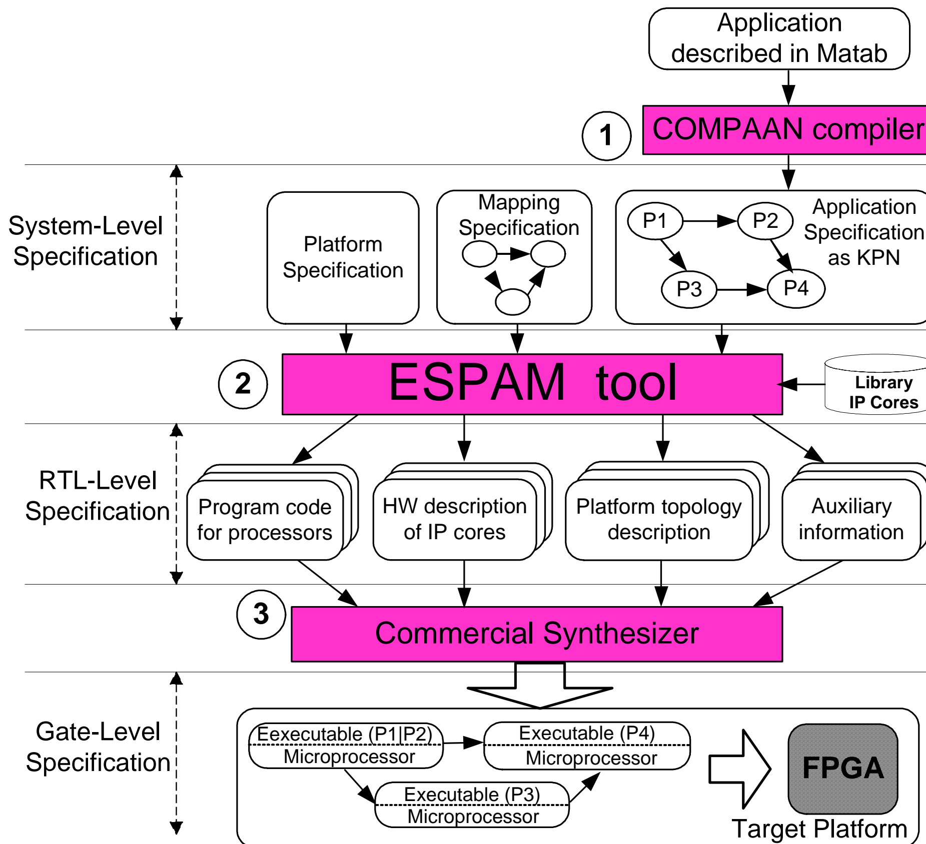

Figure 1.1: System design flow

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

- *Mapping Specification*: It specifies the corresponding relations between all processes and FIFO channels of *Application Specification* and all components of *Platform Specification*.

For mapping applications onto multiprocessor systems, the parallelism available in an application must be revealed and exploited first. Because most of the applications are typically specified as sequential programs using a high-level programming language such as C/C++ or Matlab, an abstract concurrent model is needed to reveal the implicit concurrency of the applications. In our design flow shown in Figure 1.1, COMPAAN [3] [4] [5] [6] is used to convert a sequential application into Kahn Process Network (KPN) [7] specifications. The applications

COMPAAN can handle are parameterized static affine nested loop programs, which can be described using a subset of the Matlab language.

The KPN specification represents an application in terms of distributed control and distributed memory, which can lead to efficient implementations on FPGAs for stream oriented applications. The KPN model of computation assumes a network of concurrent autonomous processes that communicate in a point-to-point fashion over unbounded FIFO channels (buffers), using a *blocking-read* synchronization primitive. Each process in the network is specified as a sequential program that executes concurrently with other processes.

After getting the KPN *Application Specification*, users can specify their *Platform Specification* and *Mapping Specification*. ESPAM takes these three specifications as input and finds out the correspinding relations between the KPN processes and target platform components according to the *Mapping Specification*. Then ESPAM generates the target physical platform by instantiating the platform components which are given in the *Platform Specification* as well as the components which function as an interface [8] for communication between the generated embedded system and outside host processors. Finally, program code files for each processor in the target platform are generated by the ESPAM tool.

The specification generated by the ESPAM tool is at RTL level. It includes *Platform topology description* which describes the platform topology in great detail; *Hardware descriptions of IP cores* which contains all used predefined IP cores from *Library IP Cores* as well as reconfigurable IP cores which are generated according to the KPN specification of an application; *Program codes for processors* which are program source code files in C for each processor component according to the behavior of the corresponding process in the KPN; *Auxiliary information* which contains supply files for giving tight control of the overall specifications. Also the generated specification includes an interface which enables the communication between the target embedded system and the host processor via several memories.

These generated descriptions by ESPAM can be accepted by a commercial synthesizer as input for conversion into a *Gate-Level* specification. This *Gate-Level* specification is the final detailed implementation of the system which conforms to the user's highly abstract specifications.

Thus, the previous work on ESPAM, together with the COMPAAN compiler, allows a fully automated system design flow that maps sequential applications written in Matlab onto homogeneous multiprocessor platforms with manually speci-

fied mapping between processes and processors. Both *one-to-one* and *many-to-one* mappings are supported. *Many-to-one* mapping means more than one process in the *Application Specification* can be mapped onto one processor in the *Platform Specification*. However, one channel in the *Application Specification* is still mapped onto one FIFO channel in the *Platform Specification*. This many-to-one mapping is useful when the rescource on the Xilinx field programmable gate arrays (FPGAs) board are not enough for one-to-one mapping. The generated system on an FPGA chip is able to communicate with outside host processors by exchanging data with off-chip ZBT SSRAM memories through an interface.

1.2 Motivation

In the previous work on ESPAM, generated embedded systems are still homogeneous multiprocessor embedded systems in which, the processing components are only programmable processors. In many cases, a homogeneous system can no longer meet the application requirements because for different types of processes different types of processing components are needed for efficiency. It is a common knowledge that a dedicated hardware IP core can work more efficiently than a general processor which has to compile and execute software programs for the same function. Thus, we find it necessary to extend our ESPAM tool with support for automatic generation of heterogeneous embedded systems where both processor components and dedicated hardware IP cores are used as processing components. The configurable MicroBlaze embedded soft processor core [9] [10], which is used in our ESPAM generated projects, can integrate customized user Intellectual Property (IP) cores [11]. This integration can result in a dramatic acceleration in execution time due to algorithms being executed in parallel in hardware and not sequentially in software.

For proving the correctness and feasibility of this idea in ESPAM, in [12] ESPAM was used to generate a homogeneous embedded system for an MJPEG application with MicroBlaze processors. One processor component was replaced manually by a dedicated hardware module, which contains the hardware IP core for implementing a certain function, to test the efficiency of this replacement. The hardware module should have FIFO interfaces to communicate with other processing components. Since the hardware modules which are generated by the LAURA tool [13] meet this requirement and the generation is quicker and less error-prone compared to manual hardware module generation, LAURA was used to generate the hardware module for executing the most computationally inten-

sive process of the application. After replacing the corresponding processor with this hardware module, the total time performance of the application execution was greatly improved. By analyzing the execution time of seperate processing components, the reason lies in that the total execution time performance of an application depends mostly on the most computationally intensive process, which is also the most time-consuming one. Once the execution time of this process is decreased by implementing it with a hardware module, the total time performance is also improved.

From the experiment results in [12], we can see the advantage and necessity to implement automatic generation in ESPAM of heterogeneous systems which integrate dedicated hardware IP cores. But instead of integrating the hardware modules generated by the LAURA tool, we generate our own hardware modules because we want some properties which the LAURA generated hardware module networks do not currently support:

- First, we want the hardware modules to be clearly structured and better modularized. A hardware module generated by our ESPAM tool contains a hardware IP core and a wrapper around the IP core. Several predefined and parameterized components with a clearly defined interface compose a hardware module. For generating a hardware module, we only have to instantiate these components stored in a library and set the corresponding parameters.

- Second, as a result of making the hardware modules more structured and modularized, every component/part of the hardware modules becomes more independent and loosely coupled. Therefore, we can debug and optimize each component separately. This brings much convenience for efficient and effective optimization, which makes the performance of the generated systems better. The small hardware overhead that the modularization initially brings is canceled, in our case, by the hardware optimizations (possible due to the modularization) in the individual components. See the experimental section.

- Third, the systems LAURA generates are networks of only hardware modules. The LAURA networks have never been designed to support integration with multiple programmable processors. To integrate certain hardware IP cores into our ESPAM generated systems using LAURA generated hardware modules, we need to manually add some additional hardware to reset, in a specific way, the LAURA generated hardware module network [12]. To

avoid this inconvenience and make our ESPAM being more flexible by supporting both programmable processors and hardware IP cores, we need to extend the ESPAM to support hardware IP core integration.

By analyzing the structure and models ESPAM uses, we find it possible to get enough information for the dedicated hardware IP core integration from the current models. We will explain how to extract information from and integrate with our current ESPAM in detail in Chapter 2. With the extracted information, a wrapper can be generated around a hardware IP core which implements the functionality of a process in a hardware language. The wrapper thus makes the hardware IP core be able to communicate with other processing components via FIFO interface. The wrapper together with the hardware IP core is regarded as a hardware module. The IP core is not generated by ESPAM. It should be in the library.

Thus, the main task of this thesis project is to support automatic heterogeneous embedded systems generation according to the platform and mapping specifications which are given by the users. The generated hardware modules, which consist of hardware IP cores and wrappers, are in the format which can be synthesized by Xilinx Platform Studio(XPS) [9]. These hardware modules can be integrated into our ESPAM through communicating and cooperating with other processing components for execution of applications. Also for supporting reuse of design components, we decide to generate the necessary files for hardware modules in a parameterized form.

1.3 Related Work

Mapping applications to embedded system platforms systematically and automatically has been widely studied in the research community. The closest work to our work is the LAURA tool which has been developed at the Leiden Embedded Research Center (LERC). It generates embedded systems composed of hardware modules which have similar structure to our ESPAM generated hardware modules. Comparing to the LAURA tool, our ESPAM makes the parameterized nature of KPN clearer because every component of our generated hardware modules are well parameterized and the global parameters can be loaded from a Parameter Bus from the outside host processor conveniently. Also, our ESPAM makes the automatically generated hardware modules for XPS projects more modularized and structured than the ones generated by LAURA. Each hardware module is

composed of several parameterized components which are loosely coupled. The hardware modules are faster and more area efficient. Lastly the LAURA tool maps the KPN for an application only to hardware IP cores while our ESPAM can support both programmable processor cores and IP cores, which brings much more flexibility in the system implementation.

Another similar work is presented in [14]. It introduces a design flow for the generation of application-specific multiprocessor architectures. In this work the architecture generation is based on instantiation of generic multiprocessor architecture templates and on the automatic generation of communication coprocessors. This is similar to our ESPAM design flow because for our multiprocessor system and IP wrapper components generation, instantiation of generic parameterized architecture components is also used. However, the design flow in [14] is not as fully automated as our ESPAM tool. Many steps still need to be done manually. This makes a full implementation of a system with this design flow slower than that of our ESPAM tool.

In [15] a top-down design methodology with various abstraction levels called C-HEAP is introduced. It starts with a high-level executable specification and converges towards a silicon implementation. The methodology proposes a heterogeneous multi-processor architecture template based on distributed shared memory and presents an efficient and transparent protocol for communication and (re)configuration. C-HEAP is similar to our ESPAM design flow because for both methodologies several abstraction levels are traversed throughout the design process. One major difference is that our ESPAM platform model uses distributed memory while C-HEAP uses shared memory.

Companies such as Xilinx, Altera, etc. provide approaches and design tools attempting to facilitate efficient implementations of single or multiprocessor systems on an FPGA. However, the required input specifications are still very detailed. In contrast, our design methodology raises the design focus to a higher system level of abstraction that reduces the design time significantly.

1.4 Research Contributions

In this thesis, we present extensions to the ESPAM tool that implement our methodology to map KPNs generated by the COMPAAN tool onto a heterogeneous embedded system which consists of multiple processors and hardware modules. The main contributions are:

- Further improvement of ESPAM, in comparison to [12], to allow mapping of a Kahn Process Network specification onto heterogeneous embedded system which consists of multiple processors and hardware IP cores in a systematic and automated way;
- Generation of very clearly structured and modularized hardware IP core wrappers which leads to easy and efficient IP core integration in ESPAM;
- Comparison between LAURA and ESPAM in generating homogeneous systems consisting of only hardware modules.
- Validation of the presented approach with stream-oriented application experiments.
- Full implementation and integration of the presented approach in the ESPAM software is available on the CVS repository of the LERC group of LIACS.

1.5 Thesis Organization

The remaining part of this thesis is organized as follows.

Chapter 2 gives detailed description of the approach to generate heterogeneous embedded systems. The general structure of the generated IP core wrappers is presented. Also the information extraction from the related ESPAM models is depicted as well as the visitor pattern which is used to generate the hardware module part of the final system.

In Chapter 3 we present case studies that we conducted in order to validate and evaluate our generation approach presented in Chapter 2 on real-life applications. The results obtained from the experiments performed in these case studies are analyzed and compared to hardware systems generated by LAURA.

In Chapter 4 we give a detailed tutorial showing how to build a heterogeneous multiprocessor embedded system using our COMPAAN/ESPAM tool chain and the commercial synthesis tool Xilinx Platform Studio.

In the final chapter we give conclusions and limitations of the current work as well as some ideas for improvements in future work.

Chapter 2

Heterogeneous Multiprocessor System Generation

In Chapter 1 we discussed our motivation to make ESPAM support automatic heterogeneous embedded system generation. In this Chapter we elaborate in more details the approach and techniques for the generation. With these techniques, ESPAM supports soft/hard processor cores and dedicated hardware modules as processing platform components. The hardware modules can communicate with each other or MicroBlaze processor components via FIFOs. In Section 2.1 the basic concept for the IP core integration in ESPAM is introduced. In Section 2.2 the internal structure of the wrappers for hardware IP cores is given and explained in detail. In Section 2.3 we discuss how to extract relevant information from the ESPAM models and how to generate the wrappers in XPS format. Finally, in Section 2.4 we present the concept of visitor pattern and how we use it to implement the generation of the IP core wrappers.

2.1 IP Core Integration in ESPAM– Basic Concept and Structure

We already explained that the application is specified as a KPN which consists of concurrent processes and communication channels. For flexibility, we want to be able to map these processes to heterogeneous processing components. For clarity, we first show a simple example to illustrate the mapping procedure. In

```
%parameter N 2 10;

for i = 1 : 1 : 1,
   [ a(i) ] = Init;
end

for i = 2 : 1: N,
   [a(i)] = Compute (a(i-1));
end

for i =N : 1 : N,
   [] = Pass(a(i));
end
```

Figure 2.1: Simple example in Matlab

```
void P2::main() {
 for (int i = 2 ; i <= N ; i += 1 ) {
   if (i-2 == 0) {
     in_0 = read(IP1);
   }
   if (i-3 >= 0) {
     in_0 = read(IP2);
   }                                   READ

   out_0 = Compute(in_0) ;             EXECUTE

   if (-i+N-1 >= 0) {
     write( OP1, out_0);
   }
   if (i-N == 0) {
     write( OP2, out_0);               WRITE
   }
 } // for i
}
```

Figure 2.2: Code for Process P2

Figure 2.1 a simple MATLAB program is shown. The corresponding KPN of this very simple MATLAB code is shown in the upper part of Figure 2.3. It contains three processes and is mapped onto a heterogeneous system. Processes *P1* and *P2* are mapped onto MicroBlaze processors and the middle process *P3* is mapped onto a hardware module. The KPN unbounded FIFO channels *Ch1*, *Ch2*, and *Ch3* are mapped onto the bounded hardware FIFOs *FIFO1*, *FIFO2*, and *FIFO3*, respectively.

In the lower part of Figure 2.3 we can see that each generated hardware module contains mainly four blocks: a *Read Block*, an *Execute Block*, a *Write Block* and a *Control Block*. Hardware modules communicate with each other or processors through FIFO interfaces. We divide the hardware module into these four blocks for the following reasons:

Firstly, the hardware modules are regarded as components of the systems generated by ESPAM. They should have a general structure as required for component based systems generated by ESPAM. To make it general enough, we should bear in mind the features of ESPAM based systems. Such as components should be semantically independent of one another and impose no restrictions on the other components they cope with. Each component has its own control. Components in ESPAM communicate with each other using point-to point channels through which no transfer of control information takes place. From this, we conclude that the hardware modules should at least have a control block and blocks for commu-

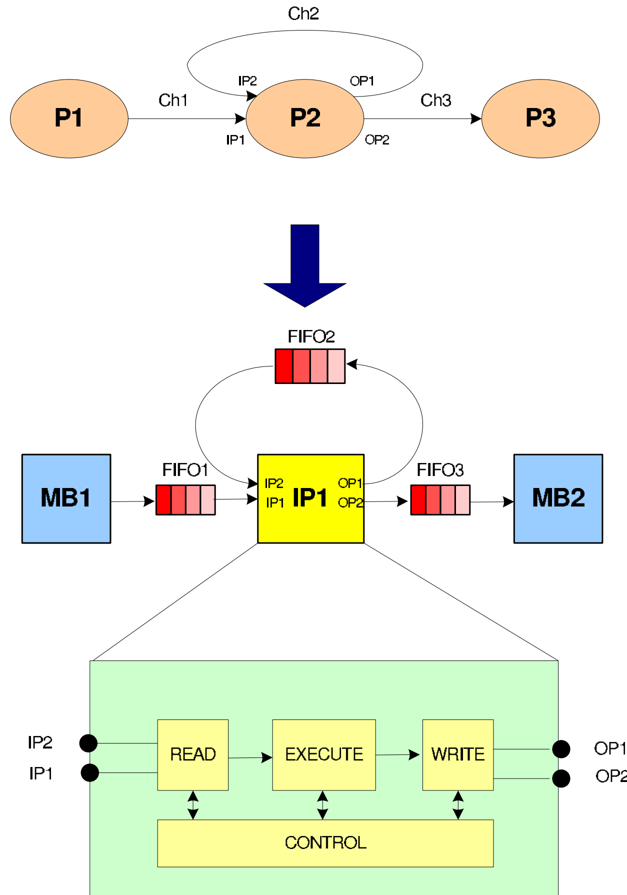

Figure 2.3: An example of mapping KPN onto Heterogeneous System

nication and execution.

Secondly, the sequential program for a process of a KPN produced by COMPAAN always follows a particular sequence of events. Let us consider the P2 process as it is specified by the sequential code given in Figure 2.2. First, the data is read from the input ports. Second, the actual computation takes place to perform the corresponding function *Compute* from the original Matlab program. Last the data is written to output ports. The three events are enclosed by a for-loop, indicating that the sequence of events needs to be repeated for a given number of times. This kind of process is suitable to operate in a stream based fashion, an operation model which is very applicable to multi-media and digital signal processing applications.

Thus it is reasonable to construct the hardware modules with four blocks re-

lated to communication with other components and their own control: *Read Block, Write Block, Execute Block* and *Control Block*. The *Read Block* is responsible to read data from other components according to the *application specification*. Similarly, as the name implies, the *Write Block* is responsible to write data into other components according to the specific applicatoin. The *Execute Block* contains the VHDL file(s) for function execution (the IP core). The *Control Block* contains a control unit which controls and synchronizes the execution of the function. It also contains a parameter loading component which loads global parameters.

Our hardware module structure is the same as that of the hardware virtual processors [13] generated by the LAURA tool because we find this structure matches the KPN process behaviour well. However, our implementation of the hardware blocks in this structure is different in order to make it more modularized.

Figure 2.4 shows an example of hardware module for the process *P2* in Figure 2.3. The code for this process is shown in Figure 2.2. We first analyze the code to help understanding the corresponding hardware module structure:

The function in process *P2* has one input argument *in_0* and one output argument *out_0*. The process *P2* has two input ports *IP1*, *IP2* and two output ports *OP1*, *OP2*. The I/O arguments take data from one of the I/O ports according to the control expressions. For example, when the control expression *i-2==0* is true, the input argument *in_0* reads data from the input port *IP1*. The iterator *i* is the loop iterator for the whole process. It goes from 2 to *N*. Here *N* is a global parameter of the application which can be set statically at design time or dynamically at run-time.

In the *Read Block* of Figure 2.4, a multiplexer is used to select data from input ports *IP1* or *IP2* for the input argument *in_0*. The control expressions for reading (*i-2 == 0* and *i-3 >= 0*) are evaluated in the component *EVAL_LOGIC_RD* with the current iterator value taken from the counter component which counts from 2 to *N*. The generated control signals are sent to the multiplexer to tell it which input port to read data from. Similarly, in the *Write Block* there is a demultiplexer to propagate data from the output argument *out_0* to the output ports *OP1* or *OP2* according to the control signals generated by *EVAL_LOGIC_WR*. The *EVAL_LOGIC_WR* component evaluates the control expressions for writing. The *Write Block* has its own counter so that reading and writing can be independent of each other. The *Execute Block* contains the IP core which implements the functionality of the function *Compute*. The *Control Block* communicates with the other three blocks to synchronize these blocks. Also, this block is used to set the global parameter *N* in process *P2*.

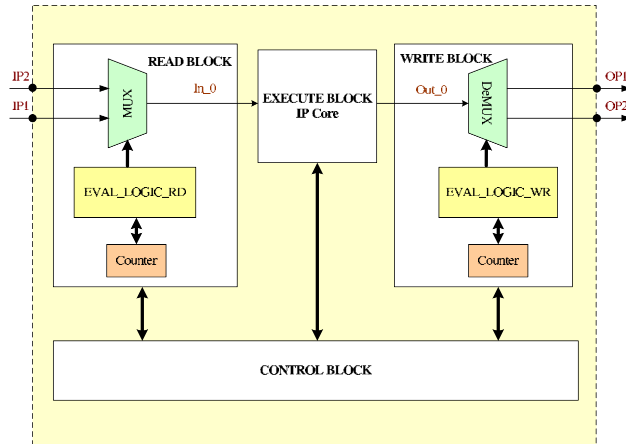

Figure 2.4: Hardware Module for P2

2.2 Structure of the IP core wrapper

We already explained the basic concept and main structure of the hardware modules. The next step is to figure out how to make the wrapper conforming to this structure. Since the main task is to generate wrappers around IP cores, there are two issues under consideration. The first one is how to organize the structure of the wrapper clearly. The second one is how to construct the communication interface of the wrapper so that it can conform to ESPAM's previous work. In this section, the wrapper structure generated by ESPAM is explained in detail as well as the communication interface.

2.2.1 Main structure and communication interface

The hardware modules we want to generate automatically are hardware components which can execute certain functionality and can communicate with other processing components in embedded systems. The IP core for implementing the functionality has to be taken from a library. To make sure the hardware modules can cooperate with other processing components, we generate a wrapper around the IP core according to the *application specification, mapping specification* and *platform specification*. Also, regarding communication with other components, we need to make the communication interface able to communicate using FIFO interface because ESPAM mainly supports data communication through FIFO channels. Below, we give more information about the structure of the wrapper we generate and the FIFO interface.

Since a hardware module has four blocks, our generated wrapper should also have the same structure – *Read Block*, *Execute Block*, *Write Block* and *Control Block*. In the previous section, the analysis of process *P2* code and its corresponding hardware module are given. Here, we give a more detailed and general description of a wrapper for a hardware IP core. Figure 2.5 shows the main structure of our wrapper for a hardware IP core. The ports left-most and right-most correspond to I/O ports of an KPN process/node (*IP1*,*IP2*,*OP1*, and *OP2* for process *P2*). The inputs and outputs to the *Execution Unit* are corresponding to the Input/Output Arguments of the function for the KPN process/node (*in_0* and *out_0* for process *P2*). Each I/O Argument can be connected to one or more I/O ports as shown in the figure and chooses the input/output from a certain I/O port according to the control conditions at a certain time. Thus, for each input argument of the function, one multiplexer is needed to select from input data ports. It is possible that an KPN process has no input ports or no output ports if it is a source node or a sink node.

The *Read Block* is the left most part in Figure 2.5 which is composed of several *multiplexers (RD_MUX)*, a *reading logic component (EVAL_LOGIC_RD)* and a *counter component (GEN_COUNTER)*. The *Write Block* is the right most part which is composed of one *demultiplexer (WR_DEMUX)*, one *writing logic component (EVAL_LOGIC_WR)* and a *counter component (GEN_COUNTER)*. The middle upper component *Execution Unit* belongs to the *Execute Block*. The middle lower components *Control Unit* and *Parameters* component constitute the *Control Block*. We explain the components in these four blocks separately in Subsections 2.2.2, 2.2.3, 2.2.4 and 2.2.5.

In ESPAM, we support FIFO based connection between a hardware module and a MicroBlaze soft processor. This is because all the communication channels/edges in the KPN are mapped onto hardware FIFO buffer components in the Platform Model. From Figure 2.5 we see that for every port (see port 1,2,3,4,5,6) of a hardware module, a bus interface which consists of ports corresponding to FIFO interface ports is created. This interface conforms to a common FIFO communication interface which we introduce below.

Figure 2.6 shows the ports of the common FIFO communication interface. The left side ports are used by a *write master* to write data into a FIFO buffer. The right side ports are used by a *read master* to read data from a FIFO buffer. A *write master* provides *W_Clk*, *W_Data* and *W_Write* signals to a FIFO. *W_Clk* is the master clock signal to asynchronously control the master writes to the FIFO. *W_Data* is the data bus to write data to the FIFO. *W_Write* is the signal that controls the write enable signal of the FIFO. The FIFO buffer gives back a *W_Full* signal

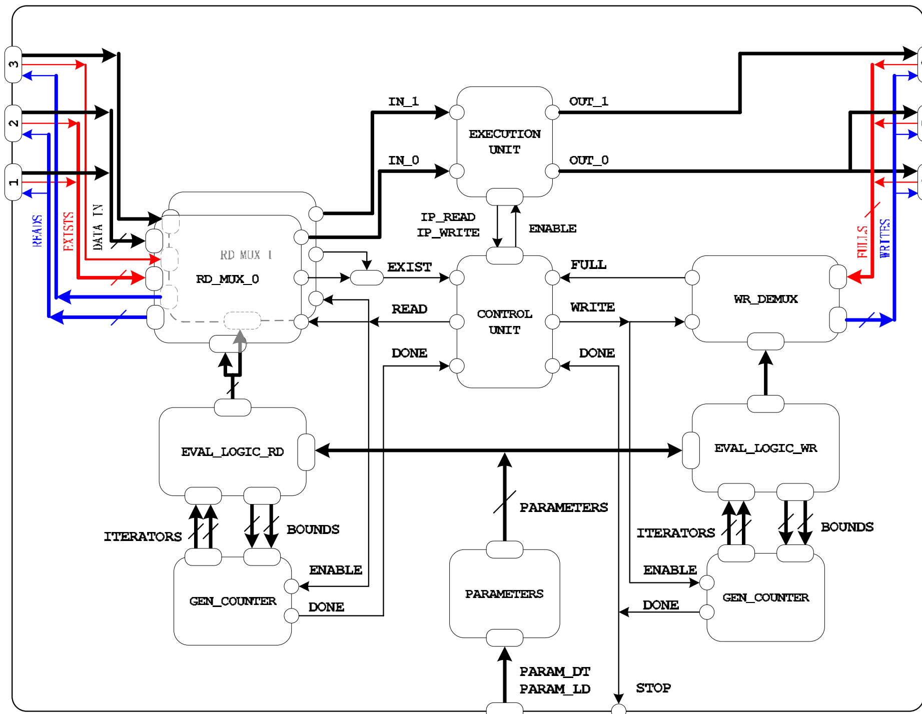

Figure 2.5: Main Structure of Harware IP Core Wrapper

as a feedback for the *master* to indicate whether the FIFO is full.

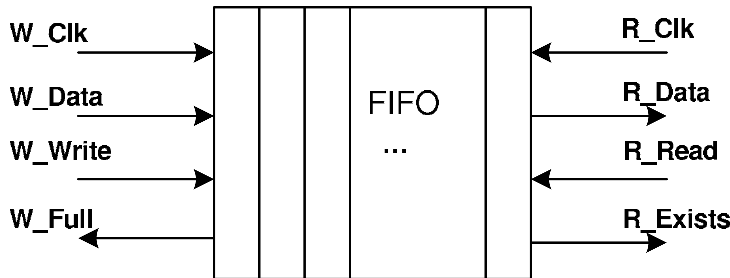

Figure 2.6: FIFO Communication Interface

In the right side of Figure 2.6, a *read master* provides *R_Clk* and *R_Read* signals to the FIFO. *R_Clk* is the slave read clock to asynchronously read the FIFO. *R_Read* is the signal that controls the read enable signal of the FIFO.*S_Data* is the data bus to read data from the FIFO. *R_Exists* is the signal indicating when the FIFO contains valid data to read.

2.2.2 Read Block

The *Read Block* fetches data from communication channels and assigns all input arguments of the *Execute Block* with corresponding data. Since there are normally more input ports than input arguments, the *Read Block* has to select from which port to fetch the data using the control information derived from the KPN.

As Figure 2.5 shows, the *Read Block* of our hardware IP core wrapper implementation consists of reading multiplexers, a reading logic component and a counter component. We explain the function and implementation of these components one by one in detail in the following subsections.

Reading Multiplexer

As the name implies, a *reading multiplexer* is used to select data from one of its inputs and deliver the data to its output. Figure 2.7 shows the interface definition of the *reading multiplexer* in VHDL:

```
entity read_mux is
generic(
   N_PORTS : natural := 1;
 PORT_WIDTH : natural := 32
);
port(
 IN_PORTS : in std_logic_vector(N_PORTS*PORT_WIDTH-1 downto 0);
 EXISTS   : in std_logic_vector(N_PORTS-1 downto 0);
 READS    : out std_logic_vector(N_PORTS-1 downto 0);

 OUT_PORT : out std_logic_vector(PORT_WIDTH-1 downto 0);
 EXIST    : out std_logic;
 READ     : in std_logic;

 CONTROL  : in std_logic_vector(N_PORTS-1 downto 0)
);
end read_mux;
```

Figure 2.7: Reading Multiplexer Interface

In the figure we can see that input interface of the multiplexer consists of ports *IN_PORTS*, *EXISTS* and *READS*. They are used for input data, existence signal of input data and read signal to the input source separately. Similarly for the output interface, *OUT_PORT* is for output data; *EXIST* is for existence signal of output data; *READ* is reading signal from the *Control Unit*. The *CONTROL* port is connected with control signals to decide from which input source will the data be delivered to the output. Only one input source is selected at a specific time for

output. The multiplexer entity has two parameters: *N_PORTS* and *PORT_WIDTH*. These parameters are set according to the specific multiplexer instance in the top level VHDL file for the hardware IP. The value for *N_PORTS* is the number of input ports for the multiplexer. *PORT_WIDTH* is the width of the ports.

In Figure 2.8, the implementation of the reading multiplexer is shown. Lines 3-5 determine whether to read from a certain input by checking the corresponding bit of the *CONTROL* and the *READ* signal from the *Control Unit*. The process in lines 7-16 assigns the *OUT_PORT* data with the correct input data and set the *EXIST* signal with the correct value which is the same as the input port exist value. What need to be mentioned is that in line 10 the *EXIST* value is set to be 1 by default. That is because it is possible that sometimes not every input argument needs data input, but we still want the *Control Unit* to be told that all the data for the execution exists to enable the function execution.

```
1   architecture RTL of read_mux is
    begin
       DEMUX_GEN : for i in 0 to N_PORTS-1 generate
        READS(i) <= CONTROL(i) and READ;
5      end generate;

       MUX_PRCSS :  process(EXISTS, CONTROL, IN_PORTS)
       begin
        OUT_PORT <= (others=>'0');
10        EXIST   <= '1';
         for i in 0 to N_PORTS-1 loop
            if( CONTROL(i) = '1'  ) then
               OUT_PORT <= IN_PORTS((i+1)*PORT_WIDTH-1 downto (i)*PORT_WIDTH);
           EXIST <= EXISTS(i);
15          end if;
         end loop;
       end process;
    end RTL;
```

Figure 2.8: Reading Multiplexer Implementation

Reading Logic Component

The *reading multiplexers* in Figure 2.5 are controlled by the *reading logic component* which is denoted as *EVAL_LOGIC_RD* in the figure. Its interface definition in VHDL is shown in Figure 2.9.

This component mainly evaluates logic conditions for input selection and then sends the corresponding *CONTROL* signal to the *reading multiplexers*. That means every logic condition for a certain input port of the process is evaluated and used to determine corresponding bits of the *CONTROL* output signal in Figure 2.9.

```
entity EVAL_LOGIC_RD is
  generic (
      N_IN_PORTS : natural := 1;
      N_CNTRS    : natural := 1;
    KWANT      : natural := 32;
     CNTR_WIDTH : t_counter_width := ( 0=>10, 1=>10, 2=>9, others=>10 );
     N_PAR      : natural;
      PAR_WIDTH  : natural
  );
  port (
      RST  : in std_logic;
      CLK  : in std_logic;

      PARAMETERS : in std_logic_vector(N_PAR*PAR_WIDTH-1 downto 0);

      LOWER_BND_OUT : out std_logic_vector(N_CNTRS*KWANT-1 downto 0);
      UPPER_BND_OUT : out std_logic_vector(N_CNTRS*KWANT-1 downto 0);
      ITERATORS  : in std_logic_vector(N_CNTRS*KWANT-1 downto 0);
      REG_CNTRS  : in std_logic_vector(N_CNTRS*KWANT-1 downto 0);

    CONTROL : out std_logic_vector(N_IN_PORTS-1 downto 0)
  );
end EVAL_LOGIC_RD;
```

Figure 2.9: Reading Logic Component Interface

We evaluate the control predicates as they appear in the original program of the processes of the KPN. By doing this the control is parameterized by the original parameters of the application because of the class of nested-loop program that COMPAAN accepts. For the generation of this component, we need to get the control information of every input port which will be explained later in Section 2.3.1. Also for improving the efficiency of the control information evaluation, if there are several same control expressions, we only keep one declaration for the expression in the VHDL file for this component.

Besides, this component also gets information of the loop iterators and determines their corresponding upper and lower bounds with the global parameter values taken from the *PARAMETERS* port in Figure 2.9. Then these bounds information is sent as output *LOWER_BND_OUT* and *UPPER_BND_OUT* in Figure 2.9 to the *counter component* for reference. For synchronization, this *reading logic component* also gets the loop iterator values *ITERATORS* and *REG_CNTRS* from the *counter component* for logic evaluation. Till now we only use the data in *REG_CNTRS* because it represents the regular incremented values of the counters' output. *ITERATORS* changes value as the value is incremented inside the counter before going through the register of the counter. This value is useful for computing the lower bound expressions of iterators.

The parameters for this component are defined inside the *generic* block from the second line in Figure 2.9. Their values are set in the top-level file of the

hardware module. *N_IN_PORTS* is the number of input ports of a hardware module. *N_CNTRS* is the number of counters. *KWANT* defines the width of data. *CNTR_WIDTH* lists the counter widths of the counters. *N_PAR* and *PAR_WIDTH* are the number and width of the run time loaded global parameters.

If the corresponding process of a hardware module has no input port, namely a source node, there will be no *Read Block* in the implementation of this hardware module. Also no input arguments for the corresponding *Execution Unit*.

Counter Component

The last component in our *Read Block* is the *counter component*. It consists of several counters. Each counter corresponds to a for-loop in the process implemented by a hardware module. In Figure 2.10 the input port *ENABLE* is used to enable the counting of the counters. *LOWER_BND_IN* and *UPPER_BND_IN* are ports for lower and upper bounds for the counters. All the counter lower/upper bounds are combined together to form a long vector for transfer. The corresponding upper and lower bound expressions are sent from the *EVAL_LOGIC_RD* component to the counters. The meaning of *ITERATOR* and *REG_CNTRS* was explained in the subsection for *EVAL_LOGIC_RD* component. Output port *DONE* is to indicate whether all the counters are done with counting. The first parameter *N_CNTRS* is used to indicate the number of the counters. *CNTR_WIDTH* is the widths of counters. *KWANT* is the width of input data.

```
entity gen_counter is
  generic (
      N_CNTRS    : natural := 1;
     KWANT      : natural := 32;
     CNTR_WIDTH : t_counter_width := ( 0=>10, 1=>10, 2=>9, others=>10 )
  );
  port (
      RST  : in std_logic;
      CLK  : in std_logic;

    ENABLE : in std_logic;

      LOWER_BND_IN : in std_logic_vector(N_CNTRS*KWANT-1 downto 0);
      UPPER_BND_IN : in std_logic_vector(N_CNTRS*KWANT-1 downto 0);
      ITERATORS    : out std_logic_vector(N_CNTRS*KWANT-1 downto 0);
    REG_CNTRS    : out std_logic_vector(N_CNTRS*KWANT-1 downto 0);
    DONE         : out std_logic
  );
end gen_counter;
```

Figure 2.10: Counter Component Interface

When the *ENABLE* signal is set to 1 by the *Control Unit*, the counter corresponding to the inner most loop will be enabled and begin to count from its corresponding lower bound. When it reaches its upper bound, it enables the counter for outer loop and continues counting from the lower bound as long as the *ENABLE* value is 1. This procedure will go on similarly till the counter for the outer most loop reaches its upper bound. Then the *DONE* signal of the whole *counter component* is set to 1 and reports to the *Control Unit*.

2.2.3 Execute Block

The *Execute Block* is the computational part of our hardware module. It can be seen as a sub-wrapper of the IP core which implements the functionality of the process associated to the hardware module. From Figure 2.5 we can see that the *Execute Block* consists of only one component–*Execution Unit*.

Execution Unit

As a sub-wrapper for the IP core, the *Execution Unit* contains a IP core as its component and defines some ports as an interface for data communication and execution control between the IP core and the other parts of the hardware module.

In Figure 2.11 the interface definition in VHDL of *Execution Unit* is given. The ports *IN_PORTS* and *OUT_PORTS* are for reading data from the input source and for writing output data. The input/output data of several ports are combined as a long vector for these two ports. The execution of the function component is enabled if the signal coming from *ENABLE* port in Figure 2.11 is set to 1. Thus the execution of our *Execution Unit* can be enabled or disabled by our *Control Unit*. Ports *IP_WRITE* and *IP_READ* are output ports which send signals to the *Control Unit* to indicate whether reading or writing can be performed. The parameters *N_INPORTS* and *N_OUTPORTS* are numbers of input arguments and output arguments of the corresponding function. The parameter *IP_RESET* is to indicate whether the reset for this component is active-high or active-low. The meaning of the parameter *KWANT* is the same as explained in the *EVAL_LOGIC_RD* component part.

```
entity EXECUTION_UNIT is
  generic(
    N_INPORTS : natural := 1;
    N_OUTPORTS : natural := 1;
  IP_RESET   : natural := 1;
  KWANT     : natural := 32
  );
  port (
    RST       : in std_logic;
    CLK       : in std_logic;

    IN_PORTS  : in std_logic_vector(N_INPORTS*KWANT-1 downto 0);
    OUT_PORTS : out std_logic_vector(N_OUTPORTS*KWANT-1 downto 0);

    ENABLE    : in std_logic;
  IP_WRITE  : out std_logic;
    IP_READ   : out std_logic
  );
end EXECUTION_UNIT;
```

Figure 2.11: Execution Unit

2.2.4 Control Block

The *Control Block* of hardware IP wrapper contains two components: a *Control Unit* and a *Parameters* component. These two components synchronize and control the reading, execution and writing of the whole hardware module. We introduce these two components in detail in the following subsections.

Control Unit

The most important part for our hardware module is the *Control Unit*. It serves as central control for the whole hardware module for internal synchronization. For every hardware module this *Control Unit* is the same because it mainly enables or disables the execution, reading and writing depending on its input signals. So, this component should be general enough so that it can generate correct control signals for any value combination of its input signals.

From Figure 2.5 we can see that the *Control Unit* sends signals through *READ*, *WRITE* and *ENABLE_EX* ports to enable the reading, writing and execution separately. Port *EXIST* is connected with a signal from reading multiplexers to indicate whether the data is available for reading. Port *FULL* is connected with a signal to indicate whether any of the FIFO buffers at the writing side is full. The two *DONE* ports which connect with signals from the *Read Block* and *Write Block* are to indicate whether reading/writing is done. The two input signals from *Execution Unit* are *IP_READ* and *IP_WRITE*. They are used to indicate whether *Execution*

Unit is ready to read or write at the moment.

In Figure 2.12 the implementation and logic of the *Control Unit* are shown. In line 15 the logic for enabling read is shown. When the reading data is available, all the FIFOs are not full, reading is not done and *RESET* is not enabled, *READ* can be set to 1 to enable the reading. Line 16 is the logic for enabling write. Writing enabling depends on the pipeline stages of the specific function implementation and the *ENABLE_EX* value. So in our *Control Unit* VHDL file there is also a process for calculating the pipeline delay in lines 1-13. The function execution enabling signal value depends on all the other input signals.

```
1   Pipe_Fill: process( CLK, RST )
    begin
      if( RST = '1' ) then
          delay_pipe <= (others => '0');
5     elsif( rising_edge(CLK) ) then

   if( sl_execute = '1' ) then
       delay_pipe(0) <= sl_read;
       delay_pipe(N_STAGES downto 1) <= delay_pipe(N_STAGES-1 downto 0);
10  end if;

      end if;
    end process Pipe_Fill;

15  sl_read  <= EXIST and not( FULL ) and not( DONE_RD ) and not( RST );
    sl_write <= delay_pipe(N_STAGES-1) and sl_execute;
    sl_execute <= sl_read when (DONE_RD='0' and BLOCKING > 0)
                  else FULL nor DONE_WR;

20  WRITE     <= sl_write and IP_WRITE;
    READ      <= sl_read  and IP_READ;
    ENABLE_EX <= sl_execute;
```

Figure 2.12: Control Unit

Parameters Component

The *Parameters* component in the *Execute block* is used to load run-time global parameters for the hardware module and transfer the parameters to the *reading* or *writing logic component*. These parameters can be set at run-time to change the iterator bounds and control expressions. They are defined in the original Matlab code as in the first line of Figure 2.1. Figure 2.13 is the VHDL interface definition for this component.

The port *PARAM_DT* is to transfer one parameter value from the outside host processor through a parameter bus. The port *PARAM_LD* connects with a signal which enables the loading of the parameter values. The port *PARAMETERS* is

```
entity PARAMETERS is
  generic (
       PAR_WIDTH : natural;
      N_PAR    : natural;
     PAR_VALUES : t_par_values
  );
  port (
     RST  : in std_logic;
     CLK  : in std_logic;

     PARAM_DT     : in std_logic_vector(PAR_WIDTH-1 downto 0);
     PARAM_LD     : in std_logic;

     PARAMETERS : out std_logic_vector(N_PAR*PAR_WIDTH-1 downto 0)
  );
end PARAMETERS;
```

Figure 2.13: Parameters Component

the output port for transferring all the parameter values to the em Read Block and *Write Block*. *PAR_VALUES* in Figure 2.13 are for the default parameter values when no parameter values are loaded from the host processor. *PAR_WIDTH* and *N_PAR* are parameter width and number of parameters, respectively.

The parameter values are loaded one by one through the port *PARAM_DT*. A temporary buffer is used to store all the parameter values. The first parameter value is loaded to the right most part (least significant part) of the temporary buffer. If there are more than one parameter, the loaded values will be shifted left so that the left parameter value can be loaded to the right most part of the temporary buffer. At the end, the values in the temporary buffer are transferred to the *PARAMETERS* port as output. *PARAM_LD* is to enable the parameter loading of this component. Since the *PARAM_LD* may be enabled more than one clock cycle and the data shifting depend on its signal, we have to make sure the data only shift once at each rising edge of the signal connected with *PARAM_LD* . Otherwise the data in the temporary buffer will be shifted left more than it should be. In this component, rising edge detection is used to achieve this.

2.2.5 Write Block

The *Write Block* writes back the results from the execution to the network communication channels. Similar to the *Read Block*, several output ports may share the same *Execute Block* output argument and the *Write Block* has to select the output ports to receive the corresponding data. The *Write Block* of our hardware IP core wrapper is very similar to the *Read Block*. From Figure 2.5 we can see

that it consists of a writing demultiplexer *WR_DEMUX*, a writing logic component *EVAL_LOGIC_WR* and a counter component *GEN_COUNTER*.

Writing Demultiplexer

The writing demultiplexer component is a component which functions as demultiplexer. Because the connections between the hardware module output arguments and the output ports of the hardware module can be determined, this component only has to enable the writing for each output port according to the control signals from the logic component. In Figure 2.14 the interface definition of this component is given.

```
entity write_demux is
generic(
   N_PORTS : natural := 1
);
port(
 WRITES  : out std_logic_vector(N_PORTS-1 downto 0);
 WRITE   : in std_logic;

 FULLS   : in std_logic_vector(N_PORTS-1 downto 0);
 FULL    : out std_logic;

 CONTROL : in std_logic_vector(N_PORTS-1 downto 0)
);
end write_demux;
```

Figure 2.14: Writing Demultiplexer

WRITES port connects with the signals for enabling writing for each output port of the hardware module. *WRITE* port connects with the signal from the *Control Unit* to enable writing. *FULLS* port connects with the full signals of every FIFO at the output side. *FULL* port tells the *Control Unit* whether any of the FIFO is full. The parameter *N_PORTS* is the number of output ports of the hardware module.

If any of the FIFO buffers at the writing side is full, the demultiplexer component can detect it from port *FULLS* and set the *FULL* signal to 1 for the *Control Unit*. If the writing is enabled through the *WRITE* port by the *Control Unit*, the demultiplexer component will enable the writing of corresponding output ports according to the *CONTROL* signal sent by the *writing logic component*.

Writing Logic Component

The *writing logic component* has the same function as the *reading logic component* except that it gets the information from the writing part. It mainly evaluates logic conditions for very output port of the hardware module and set the corresponding control signal for the *writing demultiplexer*. For evaluating these logic conditions, the current counter values are taken from the *counter component* as input for this component. The *counter component* can get the upperbound and lowerbound expressions for the iterators from the logic component.

Counter Component

The *counter component* at the *writing part* is exactly the same as that of the *reading part*. It is for synchronizing the writing of the hardware module. Detailed information about this component is in Section 2.2.2.

2.3 Generation of the IP Core Wrapper in the ESPAM

To make our ESPAM support heterogeneous system generation, we have to check how to get all the information needed from the existing ESPAM models and how to make the communication interface of the hardware modules conforming to ESPAM processor components. In this section we explain in detail the generation of the IP core wrappers.

2.3.1 Information extraction from ESPAM models

As explained in Chapter 1, for the dedicated hardware IP core integration in ESPAM, we do not generate the hardware IP cores for implementing specific functions. The IP cores can be taken from the library of ESPAM or other resources. What our ESPAM tool is supposed to generate is a wrapper which makes the hardware IP core be able to communicate and synchronize with other processing components of the target system. To achieve this, our wrapper should provide FIFO communication interfaces and contain information of the connection between the

hardware module and other processing components. Also the logic of reading and writing should be presented in the wrapper. These connection information and logic are dependent on the application, the platform and the mapping. Thus they can be taken from our existing ESPAM models' instances.

To be more concrete, for the wrapper generation, we need parameter information, iteration information for the function execution, ports to communicate with other processing components, arguments of the function and the correspondence between the arguments and the ports. Also the *control expressions* for each port which specify when can the port read or write through the corresponding argument for the function execution should be obtained.

Now let us first have an overview of the models of ESPAM to help us understand the information extraction from them. Figure 2.15 below shows the information structure and the models of our ESPAM.

On the top of Figure 2.15, the four input file types are shown. User specifications of mapping and platform are given in *.map* and *.pla* files. The Kahn Process Network specification of the application which can be taken form COMPAAN tool is presented in *.kpn* file. And for *.sch* file we use the original Matlab file for scheduling.

Again from Figure 2.15 we can see that the Kahn Process Network specification is represented in *Approximated Dependence Graph Model*(ADG Model) [16] of ESPAM. Every ADG is a graph composed of *ADG Nodes* and *ADG Edges*. The *AD Graph* of an application has exactly the same structure and topology of the KPN of the application. That means ADG nodes and KPN processes, ADG edges and KPN channels are of one-to-one correspondence relations.

Each *ADG Node* corresponds to one process in KPN and thus has an *ADG Function* relating to it. For every *ADG Node* there is also a node domain which is presented as *Linearly Bounded Set*(LBS) [16]. The values and names of *global parameters*, the name and upper/lower bounds of *iterators/indexes* for loops can all be obtained from the node's LBS domain. Each *ADG Edge* is corresponding to one communication channel in KPN. The nodes have their own *Input/Output Ports* to connect to edges for node communication. The *Input/Output Ports* also have port's LBS domain from which the *control information* for each port can be obtained. Also the *Input/Output Ports* reflects the node's communication with other nodes, which, in our case, hardware module's communication with other processing components.

To clarify the information extraction stated above, we consider the process *P2*,

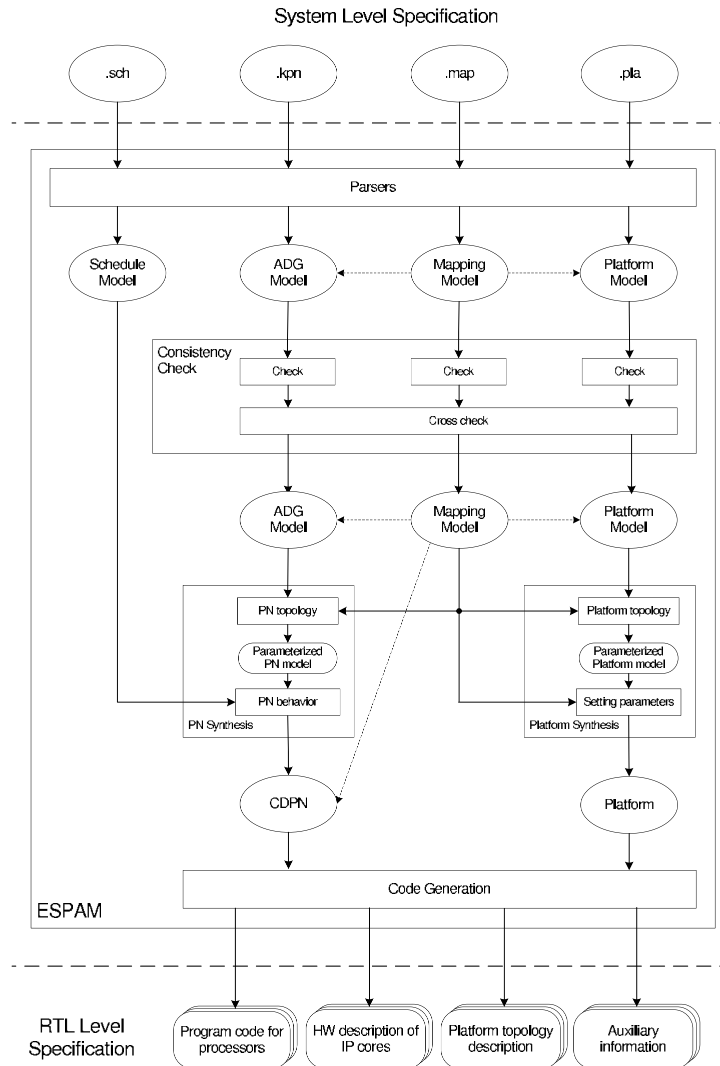

Figure 2.15: ESPAM Structure and Models

which executes the function *Compute()*, in the example given in Figure 2.1. The sequential code for process *P2* is shown in Figure 2.16 and corresponding ADG for process *P2* is shown next to it in Figure 2.17.

We can see that the process *P2* in KPN corresponds to the node *ND_2* in the ADG. This node name is assigned by ESPAM systematically. The information we need for wrapper generation includes global parameter information, function execution loop information and reading/writing control information. These information is stored in the node or port LBS domain, which contains a vector of Polytopes as its *linear bound*.

```
void P2::main() {
 for (int i = 2 ; i <= N ; i += 1 ) {
   if (i-2 == 0) {
     in_0 = read(IP1);
   }                                    READ
   if (i-3 >= 0) {
     in_0 = read(IP2);
   }

   out_0 = Compute(in_0) ;             EXECUTE

   if (-i+N-1 >= 0) {
     write( OP1, out_0);
   }                                    WRITE
   if (i-N == 0) {
     write( OP2, out_0);
   }
 } // for i
}
```

Figure 2.16: Process P2

```
1  <adg name="ex" levelUpNode="null">
     <parameter name="N" lb="2" ub="10" value="2" />
     ...
       <node name="ND_2" levelUpNode="ex">
5      <inport name="ND_2IP_1" node="ND_2" edge="ED_1" >
         <invariable name="a_1(1)" dataType="" />
         <bindvariable name="in_0" dataType="" />
         <domain type="LBS" >
           <linearbound index="i" staticControl="" dynamicControl="" parameter="N" >
10           <constraint matrix="[0, 1, 0, -2;
                                  1, 0, 0, 1]" />
             <context matrix="[1, -1, 10;
                               1, 1, -2]" />
           </linearbound>
15         <filterset index="" staticControl="" dynamicControl="" parameter="" >
             <constraint matrix="[]" />
           </filterset>
         </domain>
       </inport>
20     <inport name="ND_2IP_2" node="ND_2" edge="ED_2" >
         <invariable name="a_2(i-1)" dataType="" />
         <bindvariable name="in_0" dataType="" />
         <domain type="LBS" >
           <linearbound index="i" staticControl="" dynamicControl="" parameter="N" >
25           <constraint matrix="[1, 1, 0, -3;
                                  1, 0, 0, 1]" />
             <context matrix="[1, -1, 10;
                               1, 1, -2]" />
           </linearbound>
30         <filterset index="" staticControl="" dynamicControl="" parameter="" >
             <constraint matrix="[]" />
           </filterset>
         </domain>
       </inport>
35     <outport name="ND_2OP_1" node="ND_2" edge="ED_2" >
         <outvariable name="a_2(i)" dataType="" />
         <bindvariable name="out_0" dataType="" />
         <domain type="LBS" >
           <linearbound index="j" staticControl="" dynamicControl="" parameter="N" >
40           <constraint matrix="[1, -1, 1, -1;
                                  1, 0, 0, 1]" />
             <context matrix="[1, -1, 10;
                               1, 1, -2]" />
           </linearbound>
45         <filterset index="" staticControl="" dynamicControl="" parameter="" >
             <constraint matrix="[]" />
           </filterset>
         </domain>
       </outport>
50     <outport name="ND_2OP_1_d1" node="ND_2" edge="ED_3" >
         <outvariable name="a_2(i)" dataType="" />
         <bindvariable name="out_0" dataType="" />
         <domain type="LBS" >
           <linearbound index="i" staticControl="" dynamicControl="" parameter="N" >
55           <constraint matrix="[0, 1, -1, 0;
                                  1, 0, 0, 1]" />
             <context matrix="[1, -1, 10;
                               1, 1, -2]" />
           </linearbound>
60         <filterset index="" staticControl="" dynamicControl="" parameter="" >
             <constraint matrix="[]" />
           </filterset>
         </domain>
       </outport>
65      <function name="Compute" >
         <inargument name="in_0" dataType="" />
         <outargument name="out_0" dataType="" />
       </function>
        <domain type="LBS" >
70       <linearbound index="i" staticControl="" dynamicControl="" parameter="N" >
           <constraint matrix="[1, 1, 0, -2;
                                1, -1, 1, 0]" />
           <context matrix="[1, -1, 10;
                             1, 1, -2]" />
75       </linearbound>
         <filterset index="" staticControl="" dynamicControl="" parameter="" >
           <constraint matrix="[]" />
         </filterset>
       </domain>
80   </node>
   ...
   </adg>
```

Figure 2.17: ADG for Process P2

- Global run-time parameter information: From the Polytopes we can get the vector of *parameters* which in Figure 2.1 is *N* with its value 1, lowerbound 1 and upperbound 100.

- Function execution loop information: From the Polytopes the information of *iterators* can be obtained. In this case, iterator *i*, with its lower bound expression 2 as well as upperbound expression N.

- Reading/writing control information: The reading/writing control information specifies when should the I/O argument read from/write to a specific I/O port. It can be obtained from the LBS domain of the corresponding port. In the reading part of Figure 2.16, we can see that when the control expression *i-2 ==0* is true, input argument *in_0* will read data from input port *IP1*. If the control expression *i-3 >=0* is true, this input argument will read data from the other input port *IP2*.

From Figure 2.17 we can see all information we want to extract in the XML format for ADG. In line 2 there is information for the global parameter. The information for the two input ports *ND_2IP_1*, *ND_2IP_2* and two output ports *ND_2OP_1*, *ND_2OP_1_d1* begin in lines 5, 20, 35 and 50 separately. For each port, the *context matrix* contains bound information for the parameters; the *constraint matrix* stores information for reading/writing control expressions. For the ADG node, the *constraint matrix* in lines 71-72 contains the information of function execution loop bounds. In lines 66-67, we can see that there are one input argument *in_0* and one output argument *out_0* for the ADG function named *Compute*. Each I/O argument is connected to one or several ADG I/O Ports. To find out the correspondence, we can just check the corresponding name of the *Binding Variables* of an *ADG Port*. In line 7 the bind variable name for the input port *ND_2IP_1* is *in_0*. That means this input port corresponds to input argument *in_0*.

From the illustration and analysis above we can see that most of the information we need to put in the wrapper, such as communication with other processing components, loops of functional execution and the logic of reading and writing, is already stored in our ESPAM*ADG Model*.

Now let us think about the wrapper structure shown in Figure 2.5 again. For the *reading multiplexer* and *writing demultiplexer* we need to find out the correspondence between the I/O arguments and I/O ports. This information can be decided by checking the corresponding ESPAM ADG model instance information – the name of the bind variable of the I/O port. The I/O argument which has the same name as the bind variable is the argument which is connected with the

I/O port. For clarity and easy organizing, we order the ports according to the order of their related I/O arguments of the function. And till now our ESPAM only supports the situation that one I/O port be connected with only one function I/O argument. So the the corresponding I/O ports for the function arguments do not have intersection.

Also, in the Figure 2.5, *GEN_COUNTER* components need function execution loop information, which can be obtained from the ADG node Polytopes. *Parameters* component needs the global run-time parameter information from the ADG node Polytopes. *EVAL_LOGIC_RD* and *EVAL_LOGIC_WR* need reading/writing control information which can be obtained from the LBS domain of the I/O ports. The I/O argument names that *Execution Unit* needs can be directly obtained from the ADG node.

The parameter settings for the wrapper components are in the top-level file of a hardware module. Most values can also be known from the ADG model instance. For the interface of *reading multiplexer* in Figure 2.7, the number of related input ports *N_PORTS* for a input argument can be obtained by calculating the number of ADG input ports which have the bind variable name same as the ADG input argument name. For the interface of *EVAL_LOGIC_RD* component in Figure 2.9, *N_IN_PORTS* is the number of the ADG input ports; *N_CNTRS* is the number of iterators; *CNTR_WIDTH* contain the numbers of bits needed to present the counters' upperbounds; *N_PAR* is the number of global run-time parameters. For the *Execution Unit* component in Figure 2.11, *N_INPORTS* is the number of input arguments for the ADG function; *N_OUTPORTS* is the number of ADG function output arguments. All the other parameters which have the same name have the same meanings as these ones. The parameters we do not explain here are set values by system designers according to their needs.

In Figure 2.15 there are other two very important models: *Mapping Model* [1] and *Platform Model* [1]. The *Platform Model* is related to generating the part for our hardware IP core wrappers in XPS Microprocessor Hardware Specification (MHS) [17] files. And the *Mapping Model* acts like a bridge between *Platform Model* and *Application Model* [1]. Because the so-called *Application Model* uses KPN for *Application Specification*, it is corresponding to *ADG Model*. Thus the originally completely unrelated platform components and ADG nodes become associated through mappings.

The *Platform Model* of ESPAM is a set of generic parameterized components. The components are grouped into:

- *Processing Components :* It is used to run processes. Currently our ESPAM support programmable processor and dedicated hardware modules.

- *Memory Components :* One type of memory components specify the processors' local program and data memories. The other is to specify data communication storage (buffer) between processors and is thus called "Communication Memory".

- *Communication Components :* These components are a point-to-point network, a crossbar switch, and a shared bus. They specify the network topology of a multiprocessor platform.

- *Auxiliary Components :* Controller components for specifying an interface between processing, memory and communication components. Or Link components for connecting any two components in the platform model.

Using the *Platform Model* described above the users can specify many platform instances easily by connecting processing, memory and communication components using link components and setting parameter values of the components. How to specify a platform in the ESPAM defined XML format will be explained in a detailed tutorial in Chapter 4. The only restriction is the resource limitation of the physical platform onto which the multiprocessor systems are implemented. Also a *Consistency Check* is run on each platform model instance to find impossible and/or meaningless connections between platform components as well as parameter values that are out of range.

For the heterogeneous system generation, when a Platform Visitor visits a hardware module component during traversing all the components of a platform (the concept about Visitor will be introduced in Section 2.4), the information of its Input/Output Ports can be obtained from the Platform Model instance and being printed with other relating information in the MHS file of XPS project.

The *Mapping Model* in Figure 2.15 contains Mappings for binding the application and platform models together. A *Mapping Specification* gives the relation between all channels and processes in the *Application Specification*(in KPN) and all components in the *Platform Specification*. It is important and convenient to get the ADG node information corresponding to a hardware component through the Mapping Model, and vice versa. Currently in our ESPAM, one ADG Node is mapped onto one hardware component. This conserves the communication topology from the initial KPN and ensures that the task level parallelism described at the KPN level is propagated.

From Figure 2.15 we can see that *ADG Model* is later converted into CompaanDyn Process Network (CDPN) [16]. A CDPN consists of *CD processes* which communicate with each other through *CD Channels* and *CD Gates*. Sometimes, according to the mapping specification, each *CD process* is corresponding to several *ADG Nodes* which are mapped onto one processing platform component; each *CD Channel* corresponding to several *ADG Edges*; each *CD Gate* corresponding to several *ADG Ports*. But currently for a hardware component, its corresponding ADG node is related to only one CD process. Thus, the structure and topology of the ADG and CDPN for the hardware IP part are the same. It is sufficient to extract information only from the *ADG Model* as we stated earlier this section.

2.3.2 IP Core Wrapper Files Generation in Xilinx Platform Studio Format

For integrating hardware IP cores in an XPS project, Microprocessor Peripheral Definition (MPD) file [17] and Peripheral Analyze Order (PAO) file [17] are also needed besides the VHDL files we introduced in the previous sections. These two files are located under the *data* subdirectory of the hardware module folder in the project suite in Figure 2.21. XPS accesses these two files during synthesis for the hardware IP modules.

Before we introduce these two files, we first have to see how to make our general FIFO interface in Figure 2.6 match the XPS existing FIFO interface. The FIFO interface used by our MicroBlaze processors is mainly the FSL_V20 bus interface. FSL_V20 Fast Simplex Link (FSL) Bus [11] [18] is a uni-directional point-to-point communication channel bus used to perform fast communication between any two design elements on the FPGA when implementing an interface to the FSL bus. It can be used for fast transfer of data words between master and slave implementing the FSL interface. The block diagram in Figure 2.18 shows the signals for the FSL bus.

The correspondence of the general FIFO interface and the FSL_V20 interface is quite straight forward. All the general FIFO interface signals correspond to the similar FSL_V20 interface signals with *FSL_* in the front. The only two signals in FSL_V20 interface which are not presented in the general FIFO interface are the control signals *FSL_M_Control* and *FSL_S_Control*. In our case we do not set value for these two signals.

Since the directions of the signals in this block diagram are for FIFO com-

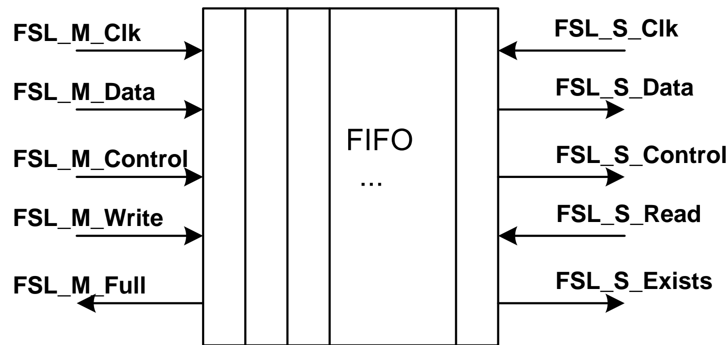

Figure 2.18: Fast Simplex Link (FSL) Bus Block Diagram

ponents, the signals of our corresponding IP core component should have the opposite directions for the data transfer. The signals on the left side are for a master processor component to write data to a FIFO. In the case for hardware IP cores as masters, *FSL_M_Clk* is the master clock signal as an output of the IP to asynchronously control the master writes to the FIFO. *FSL_M_Data* is the output data bus to write tokens to the FIFO. *FSL_M_Control* is a single bit control signal that is transmitted out from the IP together with the data at each clock edge. *FSL_M_Write* is the output signal that controls the write enable signal of the FIFO. *FSL_M_Full* is the input signal from the FIFO indicating when the FIFO is full.

The signals on the right side are for a slave processor component to read data from the FIFO. In the case for hardware IP cores as slaves, *FSL_S_Clk* is the output slave read clock to asynchronously read the FIFO. *FSL_S_Data* is the input data bus to read tokens from the FIFO. *FSL_S_Control* is the input signal that indicates the control bit associated with the data at the read end of the FIFO. *FSL_S_Read* is the output signal that controls the read enable signal of the FIFO. *FSL_S_Exists* is the input signal indicating when the FIFO contains valid data for the IP core to read.

The declarations of these bus interface signals for every I/O port of our hardware IP core are in Microprocessor Peripheral Definition (MPD) file [17] of an XPS project. An example is given to illustrate the interface signals in Section 2.3.2.

Microprocessor Peripheral Definition (MPD) file

An MPD file is used to define the interface of a peripheral. Ports, bus interfaces, parameters and default values for parameters of our hardware IP modules can be listed in the MPD file. The syntax of an MPD file is case insensitive and the MPD parameter/signal names must be HDL (VHDL, Verilog) compliant. The format for component definition and assignment commands are very similar to that of MHS files. *BEGIN peripheral_name* and *END* are used to begin and end a peripheral definition. Assignment commands for a peripheral between *BEGIN* and *END* are in the format *command name = value*. For our hardware IP modules we mainly set the default values of the parameters and declare the relations of ports and bus interfaces in this file.

A bus interface is a grouping of interface ports which are related. It provides a high level of abstraction for component connectivity of a common interface. Components can use a bus interface the same as if it were a single port. In our ESPAM, as explained previously, we define FSL bus interfaces with related ports. In an MPD file the definition of bus interfaces is in the format *BUS_INTERFACE BUS=bus_label, BUS_STD=bus_std, BUS_TYPE=bus_type*. The *bus_label* is a string which serves as the name of the bus. The *bus_std* which is DCR, LMB, OPB, PLB, or TRANSPARENT, is the bus standard of a bus interface. The *bus_type* is MASTER, MASTER_SLAVE, SLAVE, or UNDEF. This format is the most basic format and is used in our current ESPAM for hardware IP modules.

Figure 2.19 is part of a simple MPD file for a hardware IP module named *HWN_1*. The lines beginning with *BUS_INTERFACE* declares one slave FSL, one master FSL and one undefined bus interfaces with labels *In_1*, *Out_1* and *PAR_BUS*. They are corresponding to the input and output port of the hardware module and the bus for parameter loading. The line beginning with *PARAMETER* set the default value for parameter *RESET_HIGH* to 1 and declares its data type as natural. The PORT declaration *PORT ND_2IP_1_Rd = FSL_S_Read, DIR = O, BUS = In_1* shows that one port name of the hardware module is *ND_2IP_1_Rd*. This is an output port belongs to bus *In_1*. The corresponding signal name in the FSL bus interface for this port is *FSL_S_Read*. We can see that for master or slave FSL bus interfaces there are five signals. All these five signals' relation to the corresponding ports of the hardware module is listed in the MPD file. In the figure all the ports beginning with *ND_2IP_1* are corresponding to the signals for the input port of the ADG node for the hardware module. They are combined together to form the *In_1* slave FSL bus interface and can be treated as if it were a single port.

```
BEGIN HWN_1

## Peripheral Options
...

## Bus Interfaces
BUS_INTERFACE BUS = In_1, BUS_TYPE = SLAVE, BUS_STD = FSL
BUS_INTERFACE BUS = Out_1, BUS_TYPE = MASTER, BUS_STD = FSL
BUS_INTERFACE BUS = PAR_BUS, BUS_STD = TRANSPARENT, BUS_TYPE = UNDEF

## Generics for VHDL or Parameters for Verilog
PARAMETER RESET_HIGH = 1, DT = NATURAL
...

## Ports
PORT RST = "", DIR = I, SIGIS = RST
PORT CLK = "", DIR = I, SIGIS = CLK

PORT ND_2IP_1_Rd    = FSL_S_Read,    DIR = O, BUS = In_1
PORT ND_2IP_1_Din   = FSL_S_Data,    DIR = I, VEC = [(KWANT-1):0],
                      ENDIAN = LITTLE, BUS = In_1
PORT ND_2IP_1_Exist = FSL_S_Exists,  DIR = I, BUS = In_1
PORT ND_2IP_1_CLK   = FSL_S_Clk,     DIR = O, SIGIS = CLK, BUS = In_1
PORT ND_2IP_1_CTRL  = FSL_S_Control, DIR = I, BUS = In_1

PORT ND_2OP_1_Wr    = FSL_M_Write,   DIR = O, BUS = Out_1
PORT ND_2OP_1_Dout  = FSL_M_Data,    DIR = O, VEC = [(KWANT-1):0],
                      ENDIAN = LITTLE, BUS = Out_1
PORT ND_2OP_1_Full  = FSL_M_Full,    DIR = I, BUS = Out_1
PORT ND_2OP_1_CLK   = FSL_M_Clk,     DIR = O, SIGIS = CLK, BUS = Out_1
PORT ND_2OP_1_CTRL  = FSL_M_Control, DIR = O, BUS = Out_1


PORT PARAM_DT    = "PARAM_DATA", DIR = I, VEC = [(PAR_WIDTH-1):0], BUS = PAR_BUS
PORT PARAM_LD    = "PARAM_LOAD", DIR = I, BUS = PAR_BUS

PORT STOP        = "", DIR = O

END
```

Figure 2.19: MPD file example

PAO (Peripheral Analyze Order) file

Another important file in the *data* subdirectory of our hardware module folder is the PAO file. It contains a list of HDL files that are needed for synthesis, and defines the analyze order for compilation.

The format for the statements of the PAO file is *tooltarget libraryname filename hdllang*. The *tooltarget* can have lib, simlib or synlib as its value. For current ESPAM we use lib which mean that the file can be used for both synthesis and simulation. The *libraryname* specifies the library that contains the file. We use the IP as the library name for all files of the IP. The *filename* specifies the name of the file and optionally can have a file extension. The *hdllang* specifies the

language of the file which can be verilog or vhdl.

```
lib HWN_1_v1_00_a hw_node_pack vhdl
lib HWN_1_v1_00_a controller vhdl
lib HWN_1_v1_00_a counter vhdl
lib HWN_1_v1_00_a function vhdl
lib HWN_1_v1_00_a execution_unit vhdl
lib HWN_1_v1_00_a eval_logic_rd vhdl
lib HWN_1_v1_00_a eval_logic_wr vhdl
lib HWN_1_v1_00_a parameters vhdl
lib HWN_1_v1_00_a read_mux vhdl
lib HWN_1_v1_00_a write_demux vhdl
lib HWN_1_v1_00_a HWN_1 vhdl
```

Figure 2.20: PAO file example

In Figure 2.20 an example of PAO file of our hardware IP module is given. We decide the compilation order of the files by their dependency. From the figure we can see that the file *hw_node_pack.vhd* is listed as the first. Because this file contains type and function definitions that other files refer to. Also the file for the function implementation should be compiled before the execution_unit.vhd because it is a component for the execution_unit file. The top-level file for the hardware IP module should be listed at last because it contains all the other components and thus has dependency on other files for compilation. Because the compilation dependency is the same with different IP modules, we can generate the PAO files similar to the one in Figure 2.20.

2.3.3 Integration Into XPS Project

We already explained how to get the information we need for IP core wrapper generation from ESPAM's models. The next step to consider is how to integrate the new generated part into XPS project generated by ESPAM. Since ESPAM generates project files for XPS to synthesize, we first have to get an overview of XPS project files that ESPAM generates. Below we first introduce some information about XPS.

XPS is used to develop Xilinx Embedded Development Kit (EDK) - based system designs and provides a common fully integrated hardware/software development environment that supports the complete range of Xilinxs processor solutions. 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. In ESPAM, mainly the configurable MicroBlaze embedded soft processor cores are used. That is because 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. This feature is very advantageous for constructing our multiprocessor embedded systems.

Figure 2.21 shows the project suite which is generated by ESPAM which comprise an XPS project.

```
<PROJECT_ROOT>
|--- system.xmp
|--- system.mhs
|--- system.mss
|--- code/: Program codes
|-------- MemoryMap.h
|-------- aux_func.h
|-------- P_1/: program code for processor P_1
|------------- P_1.cpp
|------------- default_link_script
|-------- P_2
|-------- ...
|--- etc/: Optional files for implementation tools
|--- data/: UCF files
|--- pcores/: Customized hardware modules for the EDK project
|---------- fifo_if_ctrl_v1_00_a/
|---------- clock_cycle_counter_v1_00_a/
|---------- ...
|---------- HWN_1_v1_00_a: directory for an ESPAM generated pcore
|--------------- data
|--------------- devl
|--------------- hdl
|---------- HWN_2_v1_00_a
|---------- ...
```

Figure 2.21: Project directory structure

The top three files: *system.xmp*, *system.mhs* and *system.mss* store most of the project information. They are the corresponding Microprocessor Project (XMP) file [9], Microprocessor Hardware Specification (MHS) file [9] [17] and Microprocessor Software Specification (MSS) file [9] [17] for an XPS project.

An XMP file stores the top-level information of the project. It points the location of the MHS, MSS file, and the C/C++ program codes that need to be compiled into an executable file for a processor. It also includes the FPGA architecture family and the device type for which the XPS hardware tool flow needs to run as well as the software settings for this project.

An MHS file defines all hardware components in a platform. It includes the information of *Bus architecture, Peripherals, Processor, Connectivity* and *Address space*. Each component definition starts with *BEGIN peripheral_name* and ends with *END*. Between these are assignment commands with the format *command name = value*. There are in total three assignment commands: *BUS_INTERFACE,*

PARAMETER and *PORT*.

An MSS file contains directives for customizing libraries, drivers, and file systems. An MSS file has a dependency on a MHS file. For starting a definition for a driver, processor, or file system, keyword *BEGIN* is used. And keyword *END* ends the definition block. An MSS file has a simple *name = value* format for most statements and the *Parameter* keyword is required before every such *NAME, VALUE* pairs. If the parameter is within a *BEGIN-END* block, it is a local assignment, otherwise it is a global (system level) assignment.

In Figure 2.21, these three files are followed by the *code* directory. The software program code files for processors are stored here as well as file *aux func.h* and *MemoryMap.h*. 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. Then directory *etc* contains the files *bitgen.ut, bitgen_spartan3.ut, fast_runtime.opt and download.cmd* which store options for the XPS tool. Below in the *data* directory, the User Constraint File (UCF) [9] file, which contains constrains such as FPGA pin locations, timing, FPGA resource specification and I/O standards, is stored. ESPAM generate several UCF files for different FPGA boards, XPS use the UCF file named *system.ucf* for platform synthesis.

The *pcores* directory contains predefined hardware IP modules as well as hardware modules generated by ESPAM. In the *hdl* subdirectory several vhdl files of an IP core and its wrapper are included. In the *data* subdirectory the compiling order of the VHDL files in *hdl* subdirectory and the communication ports and interface of the hardware module are described seperately in Peripheral Analyze Order (PAO) file and Microprocessor Peripheral Definition (MPD) file. More information of these files are introduced in Section 2.3.2.

To integrate the hardware IP core wrapper generation part into the previous homogeneous system generating ESPAM, we only need to consider the hardware related files introduced above. They are the MHS file and the files in the *pcores* directory. Other files can stay unchanged because our hardware IP core wrapper generation part does not bring any effect to them.

For the *pcores* directory, we have to generate VHDL files, PAO file and MPD file for each hardware module component which is specified in the *platform specification*. A corresponding subdirectory for the hardware module should be generated to store all these files, such as the *HWN_1_v1_00_a* directory in Figure 2.21. The VHDL, PAO, MPD files can be generated by using the information extracted

from the *ADG Model*, *Mapping Model* and *Platform Model* as explained before in Section 2.3.1.

For the MHS file, for each hardware module component in the *platform specification*, a *BEGIN-END* block is generated with assignment commands in between. With these assignment commands the parameters of the hardware module can be set; the interfaces and ports of the hardware module can be connected with the FIFO or wire according to the specific application. The corresponding FIFO name for a bus interface can be decided by getting the link information between the FIFO and the bus interface from the *Platform Model* instance of the system. Other information between the *BEGIN-END* block can be generated independent of the specific hardware module. We show a sample below to help understanding.

Figure 2.22 below is the hardware module part definition of a MHS file. The lines which begin with *PARAMETER* keyword are used to set values for different parameters of the hardware module. The lines which begin with *BUS_INTERFACE* are used to declare the connection relations of different bus interfaces. In the figure, *In_1* and *Out_1* are bus interfaces which has grouped signals for FIFO communication. According to the name, *In_1* is connected to the reading/slave end of a FIFO; *Out_1* is connected to the writing/master end of a FIFO. The bus interface *PAR_BUS* are connected with signals related to parameter loading. The last few lines which begin with *PORT* are used to declare the connection relations of other signals of this hardware module, such as clock or reset signals. The *STOP* port in the figure is used to indicate the completion status of the execution of the whole hardware module.

```
BEGIN HWN_1
  PARAMETER INSTANCE = HWN_1_ip
  PARAMETER HW_VER = 1.00.a
  PARAMETER RESET_HIGH = 0
  PARAMETER PAR_WIDTH = 16
  PARAMETER KWANT = 32

  BUS_INTERFACE In_1 = FIFO_MB_1_Out_1
  BUS_INTERFACE Out_1 = FIFO_HWN_1_Out_1
  BUS_INTERFACE PAR_BUS = PARBUS
  PORT CLK = sys_clk_s
  PORT RST = net_design_rst
  PORT STOP = net_fin_signal_IP_1
END
```

Figure 2.22: Sample Hardware module Part in MHS file

For integrating the hardware IP core wrapper generation part into our ESPAM, the hardware module parts of MHS file are automatically generated and added to the MHS file. Each hardware module has its own part similar to Figure 2.22 in MHS file. Also in the MHS file, the port information for parameter register should

be added to the part for Host Interface component *zbt_main*. And the value setting for parameter *PAR_WIDTH*, finishing signals for hardware modules, bus interface *PAR_BUS* for parameter bus should all be added to the part for the *host_design_ctrl* component. These two components in MHS are for the communication between the host processors, off-chip ZBT SSRAM memories and our Function Design. They are introduced in detail in [8]. Since the load of parameters and transfer of finishing signals are related to the communication between host, ZBT and our function design, the corresponding parts of these two components in MHS are changed.

For the integration, we not only have to know the whole project suite and modify the MHS file, but also have to make our hardware modules communicate with other processing components well. Since in ESPAM FIFOs are used for this communication, we also need to study the FIFO communicating interface and make the signals for I/O ports of our IP core wrapper conform to it.

The Microblaze core provides 8 input and 8 output interfaces to Fast Simplex Link (FSL) buses. The FSL buses are uni-directional non-arbitrated dedicated communication channels. In ESPAM Microblaze processor components communicate with *fsl_v20* FIFO components through their FSL interfaces by default. If the number of Input or Output ports of the process for a MicroBlaze is more than 8, additional Local Memory BUS (LMB) interfaces of the MicroBlaze are used to communicate with other processing components through FIFO controllers. Currently the FIFO controller can translate between FSL and LMB protocols.

For making the communication interface of our hardware module conform to the previous homogeneous ESPAM, our ESPAM supports FSL_V20 Fast Simplex Link (FSL) Bus for the hardware modules' I/O interfaces. Through this FSL interface the hardware module components can communicate with the *fsl_v20* FIFO components. More details about the FSL bus interface are explained in Section 2.2.1.

2.4 Visitor Pattern

Till now, all the hardware IP core wrapper related files that are generated by our ESPAM tool are introduced. To generate all these files, we mainly use the *Visitor Pattern* [19] which is a commonly used design pattern. In this section we introduce this *visitor pattern* and how it is applied in our ESPAM.

A *visitor* represents an operation to be performed on the elements of an object structure. The elements provide interfaces for a visitor to access their internal state. Compared to *Iterator Pattern* [19], *Visitor Pattern* can traverse objects of different classes while a iterator can only traverse objects of the same class. Our ESPAM uses the *Visitor Pattern* for code and file generation because our platform and application instances contain objects of different classes and a certain operation will be applied on many of these objects. Also for our IP core wrapper generation part we only get information from already existing model instances without adding any extra information. This fact also matches the function of a visitor because a visitor normally only access elements' internal state and information without changing it.

Adding a new operation for the whole object structure can be done by simply adding a new visitor. But if there is a new element class being created, in every visitor a new function to operate on this new class needs to be added. So the *Visitor Pattern* 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. Every concrete element implements an *Accept* operation that takes a visitor as an argument. The concrete visitor which visits this concrete element can thus call the function related to this element class among all its functions.

For our hardware IP core wrapper files, we simply add a new visitor which traverses the platform instance and generate IP core wrapper files when it discovers hardware IP components among the platform elements. This new visitor inherits from an abstract *PlatformVisitor* class because it also traverses platform instances. And this *Platform Visitor* implements the most basic interface class *Visitor* in our ESPAM.

Chapter 3

Case Studies

In this Chapter we present four case studies. We analyze the results obtained from the experiments performed in the case studies and give some comments and explanation.

These four case studies are about applications of Discrete Cosine Transform (DCT), Sobel Edge Detection, Discrete Wavelet Transform (DWT) and one genetic algorithm. To validate and verify our ESPAM heterogeneous system generation and the advantage of heterogeneous systems, we generate one homogeneous system and one heterogeneous system for the DCT application. Then we can check whether the heterogeneous system works correctly and compare the performance data of these two systems. Another heterogeneous system for the Sobel application is presented in Chapter 4. After this, we want to compare the performance of ESPAM generated systems and LAURA generated ones. Since LAURA generated systems consist of only hardware modules, we also generate homogeneous systems with all hardware modules for the other three applications. When we get the ESPAM and LAURA generated homogeneous systems for Sobel, DWT and the genetic algorithm applications, we compare the performance data and analyze them.

3.1 Discrete Cosine Transform (DCT)

In this case study we design a heterogeneous embedded system for a Discrete Cosine Transform application. It is a very simple application to validate the correct-

ness of our heterogeneous embedded system generation by the proposed COMPAAN/ESPAM tool chain.

The initial Matlab code of this application is shown in Figure 3.1. The first two lines declare global parameters *N* and *M* which stand for the number of blocks operated on and the block size. The data ranges for them are 1-10 and 10-100. In the experiments we use 8×8 data blocks. Lines 4-8 contain a loop to read data from the blocks to array *a*. The next loop applies DCT on the data and stores it in array *c*. The last loop stores the data into *dct_out*. Three function calls are involved: *ReadDataA()*, *DCT()* and *Sink()*.

```
1     %parameter N 1 10;
      %parameter M 10 100;

      for i = 1:1:N,
5       for j = 1:1:M,
          [  a(i,j) ] = ReadDataA();
        end
      end

10    for i = 1:1:N,
        for j = 1:1:M,

          [ c(i,j) ] = DCT( a(i,j) );

15        end
        end

      for i = 1:1:N,
        for j = 1:1:M,
20
          [ dct_out(i,j) ] = Sink( c(i,j) );

        end
      end
```

Figure 3.1: Initial Matlab code for DCT application

We first convert this initial Matlab code into a KPN using our COMPAAN tool. Each function call corresponds to one process/node in the KPN. The result KPN is shown in Figure 3.2. From the figure we see that the KPN consists of three nodes corresponding to three functions. Two channels are in the KPN for communications between the nodes.

After getting the KPN of the application, we conduct two experiments to validate our heterogeneous system generation and compare the system performance of homogeneous and heterogeneous systems for this application. In the first experiment, we map the DCT application onto a homogeneous system platform with three MicroBlazes. In the second experiment, we map the DCT process onto hardware IP modules while keeping other two processes the same as in the first

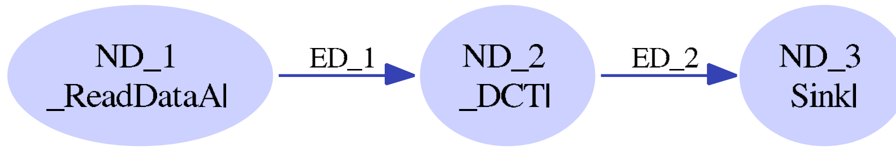

Figure 3.2: KPN for DCT application

experiment.

Since the input for our ESPAM tool also requires *Platform Specification* and *Mapping Specification* files, we need to write our *.pla* and *.map* files for the first experiment according to the experiment purpose. They are given in Figure 3.3 and Figure 3.4.

In this *Platform Specification*, each MicroBlaze processor uses one of the custom memory controllers to connect to one bank of ZBT SSRAM on the target FPGA platform. The MicroBlaze *MB_1* uses the memory controller *ZBT_CTRL_1* to read the initial data blocks from the first ZBT SSRAM bank while *MB_3* uses *ZBT_CTRL_3* to write the result data blocks to the fifth ZBT SSRAM bank. From Figure 3.3 we can see that the *data memory size* and *program memory size* for each processor are also set in the *Platform Specification*.

The *Mapping Specification* is quite straightforward. It specifies which process is mapped onto which processing component. From Figure 3.4 and the KPN for the DCT application, we can see that process *ReadDataA()* is mapped onto processor *MB_1*; process *DCT()* is mapped onto *MB_2*; and process *Sink()* is mapped onto *MB_3*.

For the second experiment, we only make a small modification which maps the *DCT()* process onto a hardware IP module. Thus for *Platform Specification*, we only need to replace the declaration part for *MB_2* with the part we show in Figure 3.5 as well as deleting the corresponding *ZBT_CTRL_2* and *mb_opb_2* parts. For *Mapping Specification* we replace the part for process *DCT()/ND_2* with the part in Figure 3.6.

In these two experiments, we use one 8×8 data block to test the two embedded systems. We get the clock cycle numbers for both systems by getting the number of the counter which is attached to *MB_3*. The result clock cycle numbers of these two systems are shown in Table 3.1. The result clock cycles may not reflect the time merit of hardware modules completely because the MicroBlaze processors may be too slow to catch the hardware IP module's execution time. But even

```
<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="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"/>
   </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 3.3: *Platform Specification* of Homogeneous System for DCT application

with this condition, the time performance of the heterogeneous system with hardware modules is more than 30 times better than the homogeneous system with all MicroBlaze processors.

	Homo Sys	Hetero Sys
Clock Cycles	148214	4334

Table 3.1: DCT Experiment

The DCT case study constructs a very simple heterogeneous system with two

```
<mapping name="myMapping">

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

   <processor name="MB_2">
       <process name="ND_2" />
   </processor>

   <processor name="MB_3">
       <process name="ND_3" />
   </processor>

</mapping>
```

Figure 3.4: *Mapping Specification* of Homogeneous System for DCT application

```
   <processor name="HWN" type="CompaanHWNode">
   </processor>
```

Figure 3.5: Modification of *Platform Specification* for Heterogeneous System

```
   <processor name="HWN">
       <process name="ND_2" />
   </processor>
```

Figure 3.6: Modification of *Mapping Specification* for Heterogeneous System

MicroBlaze processors and one hardware module. In the tutorial of the next chapter, a more complicated heterogeneous system construction with the Sobel Edge Detection application will be presented.

3.2 Sobel Edge Detection

In this case study we consider a more complicated application, a Sobel Edge Detection application, to evaluate our design flow. The Sobel application performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial gradient that corresponds to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input greyscale image.

The initial Matlab code is shown in Figure 3.7. The first two lines declare two parameters *N* and *M* whose ranges are both from 50 to 1000. They stand for the width and height of the input grayscale image. Lines 4-9 specify the types of the data which are used in the code. This is necessary for this application because the default type will be *char* which is shorter than the type *int* needed

```
1     %parameter N 50 1000;
      %parameter M 50 1000;

      %typedef a int;
5     %typedef image int;
      %typedef Jx int;
      %typedef Jy int;
      %typedef Sbl int;
      %typedef Sink int;
10
      for j = 1:1:M,
         for i = 1:1:N,
              [ a(j,i)     ] = _Read_m();
              [ image(j,i) ] = Copy(a(j,i));
15
         end
      end


20    for j = 2:1:M-1,
        for i = 2:1:N-1,

           [ Jx(j,i) ] = Jc(  image(j-1,i-1),image(j,i-1),image(j+1,i-1),
                              image(j-1,i+1),image(j,i+1),image(j+1,i+1) );
25
         end
      end


30    for j = 2:1:M-1,
        for i = 2:1:N-1,

           [ Jy(j,i) ] = Jc(  image(j-1,i-1),image(j-1,i),image(j-1,i+1),
                              image(j+1,i-1),image(j+1,i), image(j+1,i+1) );
35
        end
      end


40    for j = 2:1:M-1,
        for i = 2:1:N-1,

           [ Sbl(j,i) ] = Sb( Jx(j,i), Jy(j,i) );

45      end
      end

      for j = 2:1:M-1,
        for i = 2:1:N-1,
50
          [ Sink(j,i) ] = _Write_m( Sbl(j,i) );

        end
      end
```

Figure 3.7: Initial Matlab code for Sobel Edge Detection application

here. Lines 11-17 contain a loop for data initialization. We add an extra line 14 to copy the data from array a to make the control for the reading and writing

being separated. Because we use MicroBlaze to implement initialization from the memory and the controls for other process are very simple, we need this *Copy* process to be implemented as an IP module to check whether our ESPAM can generate more complicated controls correctly. The next two loops in lines 30-37 calculate the horizontal and vertical derivative approximations respectively. Lines 40-46 contain the loop for calculating the final gradients and store them in array *Sbl*. The last loop writes the data in array *Sbl* into *Sink*.

First, we need to convert the initial Matlab code which is shown in Figure 3.7 into a KPN specification. For this step we use the COMPAAN tool which can automatically transform the Matlab code into a KPN which reveals the task-level parallelism in the sobel application. The KPN of the sobel application which is generated by COMPAAN is shown in Figure 3.8. From the KPN we can see that there are in total 6 processes and 16 channels corresponding to the Matlab code. The function corresponding to each process can be seen in the figure.

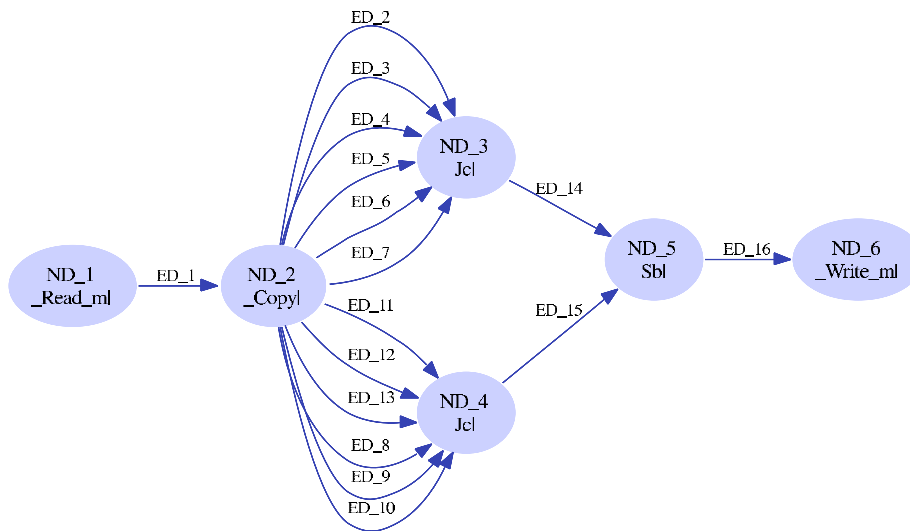

Figure 3.8: KPN for Sobel Edge Detection application

After getting the KPN of the application, we conduct two experiments with ESPAM to verify the generated homogeneous and heterogeneous systems for this application. In the first experiment, we map the first and last node of the KPN onto two MicroBlazes for reading from and writing to the off-chip ZBT memory. All the other nodes in between are mapped onto hardware IP modules. We use one 150×275 pixel image as the input of the embedded system. The system works

and gives the correct result. This verifies our heterogeneous system generation by ESPAM.

In the second experiment we use ESPAM to map the Sobel application onto a homogeneous system platform with all hardware IP modules. Each node is mapped onto one hardware module. How to write the *Platform Specification* and *Mapping Specification* for a certain design will be shown in the tutorial in the next chapter with an example of the first experiment here. For this experiment, we also construct the same embedded system with all hardware components generated by the LAURA tool to compare the performance of our ESPAM tool with it. The experiments are done with Xilinx ISE Foundation based on FPGA device Virtex2 xc2v6000. The result is shown in Table 3.2.

LAURA				
Slices	LUT	FFs	Freq (MHz)	Clock Cycles
2126(6%)	2772(4%)	2115(3%)	59.347	494937
ESPAM				
Slices	LUT	FFS	Freq (MHz)	Clock Cycles
1641(4%)	2090(3%)	1763(2%)	51.099	245447

Table 3.2: Sobel Experiment with 150×275 Image Size

In the tables, *Slices* indicates the number of slices used on the FPGA. The percentage number shows the percentage of the used slices to all the slices available on the FPGA. *LUT* indicates the number of look-up tables for the constructed embedded system. *FFs* means the number of flip-flops. *Freq* and *Clock Cycles* indicate system frequency and the number of clock cycles for the execution of the Sobel application on the constructed system. From the tables we can see that for the Sobel application, the homogeneous embedded system generated by ESPAM uses less resource on the FPGA board. Considering the frequency and the clock cycle numbers, the system time performance of the ESPAM generated system is around twice better than that of LAURA generated system.

3.3 Discrete Wavelet Transform (DWT)

In this case study, a quite complex application is used for designing a heterogeneous embedded system. The application is called Discrete Wavelet Transform (DWT) which is performed on one image. The initial Matlab code can be referred to Appendix A. The KPN generated by COMPAAN for this application is shown in Figure 3.9.

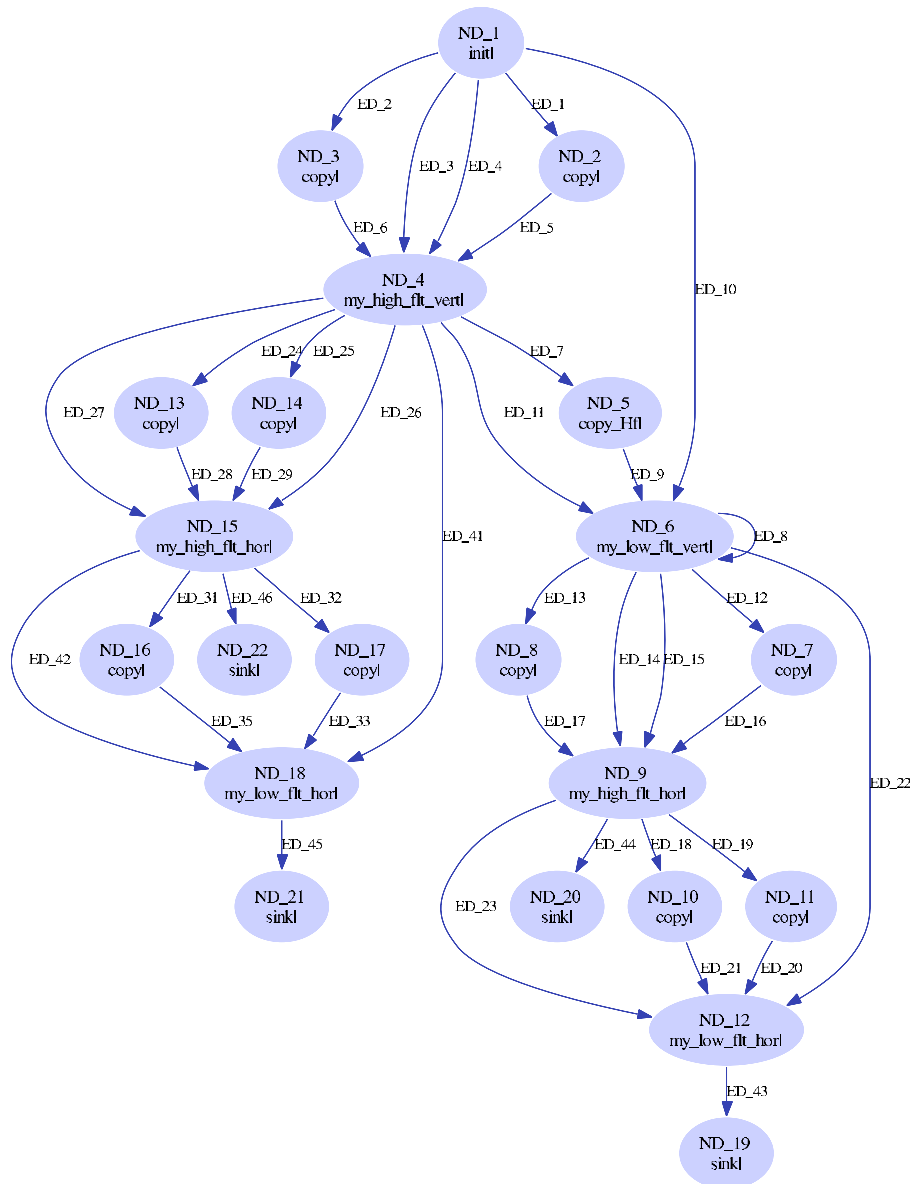

Figure 3.9: KPN for DWT application

With our ESPAM tool, we conduct two experiments for this DWT application. In the first experiment, we map the *init()/ND_1* process onto one Microblaze and the four *Sink* processes onto another MicroBlaze. All the other processes in between are mapped onto hardware IP modules. We use one 128×128 pixel image as the input of the embedded system. The system works and gives the correct result. This verifies our heterogeneous system generation by ESPAM.

In the second experiment we map the DWT application onto a homogeneous system generated by ESPAM with all hardware modules. Also we construct the same embedded system with all hardware components generated by the LAURA tool to compare the performance of our ESPAM tool with LAURA. The result is shown in Table 3.3 and Table 3.4. They list the performance of the experiments with different input image sizes. For different image sizes, only the clock cycle numbers are different.

LAURA				
Slices	LUT	FFs	Freq (MHz)	Clock Cycles
5826(17%)	7380(10%)	5562(8%)	54.975	65487
ESPAM				
Slices	LUT	FFS	Freq (MHz)	Clock Cycles
4666(13%)	5616(8%)	4343(6%)	51.395	16597

Table 3.3: DWT Experiment with 128×128 Image Size

	LAURA	ESPAM
Clock Cycles	262095	66005

Table 3.4: DWT Experiment with 256×256 Image Size

From the tables we can see that for the DWT application, the homogeneous embedded system generated by ESPAM uses less resource on the FPGA board. Although the frequency of the ESPAM generated system is a bit lower, the total execution time is still much lower than that of the LAURA generated system. Also, regardless of the input image size, the time performance of the ESPAM generated system is four times better than that of the system generated by LAURA.

3.4 A Genetic Algorithm

In this case study, we further evaluate the correctness and efficiency of the heterogeneous system generated by ESPAM. The application is a genetic algorithm which is applied on a input stream containing values 0 and 1. The initial Matlab code is shown in Figure 3.10. The first line declares parameter N which is the size of the input stream ranging from 6 to 10000. Lines 3-7 define the types of the data used in the code. Lines 9-11 contain a loop to read the stream data into array *stream*. Lines 13-17 contain a loop to propagate the data to two outputs. The *zero2()* function in Line 20 assigns zero values for array s and y. The *XNOR_SUM* function in Line 25 calculates the output value using the third input depending on the XNOR value of the first and second input. The *SQUARE_SUM* function

```
1    %parameter N 6 10000;

     %typedef stream int;
     %typedef b int;
5    %typedef s int;
     %typedef y int;
     %typedef sink int;

     for k=1:1:N,
10     [ stream(k) ] = ReadStream();
     end

     for k=1:1:N-1,
       for i=1:1:N,
15       [ b(k,i), stream(i) ] = ReadMatrixB( stream(i) );
       end
     end

     for k=1:1:N-1,
20     [ s(k), y(k) ] = zero2();
     end

     for j=1:1:N-1,
       for i=1:1:N-j,
25        [ s(j), b(j,i), b(j,i+j) ] = XNOR_SUM( b(j,i),  b(j,i+j), s(j) );
       end
     end

     for k=1:1:N-1,
30     [ y(k+1) ] = SQUARE_SUM( s(k), y(k) ); %% produces y
     end

     for k=N:1:N, %% y(1)
       [ sink(k)] = Output( y(k) );
35   end
```

Figure 3.10: Initial Matlab code for one Genetic Algorithm application

calculates the sum of the second input and the square of the first input. The last function *Output* writes the result data into array *Sink*.

The KPN generated by COMPAAN for this application is given in Figure 3.11. Again, for ESPAM, we map all the processes onto hardware IP modules. And for comparison we also conduct the same experiment with the LAURA tool. The experiment results are shown in Table 3.5 and Table 3.6. These experiments are conducted with input data streams with size 20, 40, and 60 items.

LAURA				
Slices	LUT	FFs	Freq (MHz)	Clock Cycles
1957(5%)	2620(3%)	1713(2%)	65.075	720
ESPAM				
Slices	LUT	FFS	Freq(MHz)	Clock Cycles
1614(4%)	2282(3%)	1275(1%)	50.888	745

Table 3.5: Genetic Algorithm Experiment with input stream size 20

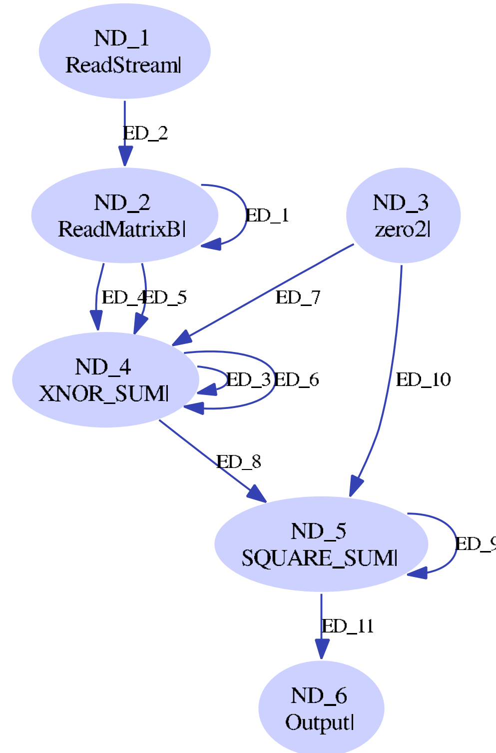

Figure 3.11: KPN for one Genetic Algorithm application

	LAURA	ESPAM
Clock Cycles (40)	3020	3084
Clock Cycles (60)	6920	7025

Table 3.6: Genetic Algorithm Experiment with input stream size 40 and 60

We can see from the tables that our ESPAM generated system still uses less programmable resource on the FPGA. The system time performance of the ESPAM generated system is a bit worse than the LAURA generated one. This is not consistent with the previous experiment results. One possible reason for the inconsistency is that for previous experiments ESPAM and LAURA generated systems use the same IP cores; while for this experiment we used different IP cores in ESPAM because the ones used by LAURA do not work with ESPAM.

3.5 Conclusions

In this chapter we conducted several experiments and from the experimental results, we can see that by mapping the most computational intensive processes onto hardware modules the system performance can be greatly improved.

By comparing systems generated by ESPAM and LAURA, we conclude that:

- ESPAM generated systems use less hardware resources on the FPGA. The reason is that our ESPAM generated systems with hardware modules are more clearly structured and modularized, thereby redundant hardware is removed.
- The system time performances of the ESPAM generated systems are better than the similar ones generated by LAURA. The reason is that the *Control Unit* component in ESPAM hardware modules is simpler.
- The clock frequencies of the ESPAM generated systems are a bit lower than the ones used in the systems generated by LAURA. One reason is that the logic evaluation component in ESPAM hardware modules is not optimized. A possible optimization is to make this component pipelined as in LAURA.

Chapter 4

Getting Started: Tutorial for Heterogeneous System Design with COMPAAN/ESPAM tool chain

In this chapter, we give a detailed tutorial to show how to design a heterogeneous embedded system with processors and hardware modules using the COMPAAN/ESPAM tool chain as well as how to make modifications to make the generated design suitable for the XPS synthesis tool. In this tutorial, the application of *Sobel Edge Detection* which is explained in Chapter 3 is chosen as an example to show the steps.

This chapter is further organized as follows. In Section 4.1 we describe how to generate an XPS project using our COMPAAN/ESPAM tool chain and import the generated files to XPS. In Section 4.2 the manual modifications needed to make to the imported project are introduced. The last section introduces how to use XPS to generate the final bitstream file and how to use a software program in an outside host processor to download the final bitstream file onto the target FPGA board to get the result data.

4.1 XPS Project Generation

In this section we elaborate how to generate files for XPS using COMPAAN and ESPAM for the Sobel edge detection application. Also the steps for importing the

generated project suite to XPS will be described.

4.1.1 KPN Specification Generation Using the COMPAAN tool

The first step for our heterogeneous embedded system generation is to use the COMPAAN tool to transform the initial Matlab code of a certain application into KPN specification. Thus the initial Matlab code is the top-level entry for our COMPAAN/ESPAM tool chain. In figure 3.7 the initial sequential Matlab code for the Sobel edge detection application is given. From the figure we can see that there are six function calls: *_Read_m()*, *Copy()*, two *Jc()*, *Sb()* and *_Write_m*. For each function call there is a process/node generated for the result KPN.

We use three commands in Figure 4.1 to generate the KPN specification for our Sobel edge detection application.

```
1) matparser --input Sobel.m --output Sobel.sac --compile --verbose -r
2) dgparser --input Sobel.sac --output Sobel --xml -r
3) panda --input Sobel.rdg -c Sobel.m --xml -ls --lms -RP -r
```

Figure 4.1: Commands of the COMPAAN tool

The first command uses the MATPARSER tool [20] to transform the initial Matlab code into a Single Assignment Code (SAC), which resembles the Dependence Graph (DG) of the initial Matlab code. The usage of the options of this command is explained below:

- --**input**: This option is used to specify the input file. It should be followed by a filename which points to the file where the initial Matlab code is stored.
- --**output**: This option is used to specify the output file. It should be followed by a filename which points to the file where the SAC results, for example, will be written.
- --**compile**: This option tells MATPARSER to convert the Matlab code into a SAC.
- --**verbose**: This option tells MATPARSER to produce information messages showing the progress and status of the conversion.
- -**r**: This option applies a set of optimizations on a solution tree which describes data-dependencies. The optimizations include removing redundant

if/else statements, removing redundant index statements, and removing redundant sub-graphs.

The second command uses the DGPARSER tool. It converts the SAC into a Polyhedral Reduced Dependence Graph (PRDG) data structure, which is a compact mathematical representation of the DG in terms of polyhedra. The options for the DGPARSER tool are:

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

The third command uses the PANDA tool to convert the PRDG into a KPN process network [6] [21]. The options for this command are:

- --**input**: This option specifies the input PRDG file generated by DGPARSER.
- -**c**: This option describes a valid global schedule as a Matlab program for all the nodes in the PRDG. We can use any valid schedule specified as a Matlab program. Here we use the original Matlab code.
- --**xml**: This option tells PANDA to generate the corresponding KPN in XML format.
- -**ls** -**lms**: These options tell PANDA to select communication linearization model, since the communication is not always in order. For more details see [6] [21].
- -**RP**: This option makes sure that the number of data tokens which a producer process sends is the same as the number of tokens a consumer process needs. For more details see [6] [21]. An alternative option is *–R2* which is used in the case that there exist mathematical divisions in the reading/writing conditions of the nodes mapped onto hardware IP modules. The resulting conditions in the processes will have module 2 expressions in place of

divisions because division is not efficient in hardware implementation. The DWT case study in Section 3.3 need to use *–R2* option.

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

After executing the three commands described above, we can get the KPN specification in XML format. The KPN of the Sobel edge detection application which is generated by COMPAAN is shown in Figure 3.8 in the previous chapter. There are six processes/nodes in the KPN corresponding to the six function calls in the initial Matlab Code.

4.1.2 Heterogeneous Embedded System Generation Using the ESPAM tool

From the system design flow chart in Figure 1.1 we can see that besides the *Application Specification* in KPN generated by COMPAAN, we also need *Platform Specification* and *Mapping Specification* as inputs for our ESPAM tool. These two specifications are shown in Figure 4.2 and Figure 4.3.

Because we only want to map the data initialization and output process onto MicroBlaze processors to communicate with the host processor, we map process *_Read_m()*(ND_1) and process *_Write_m()*(ND_6) onto MicroBlaze processors *MB_1* and *MB_2*. All the other processes are mapped onto hardware IP modules in lines 7-12 in Figure 4.3. Although in the *Mapping Specification* only one *HWN*(hardware node) is specified, all the processes declared in between (*Copy()*(ND_2), *Jc()*(ND_3), *Jc()*(ND_4) and *Sb()*(ND_5)) are generated as hardware modules separately.

In the *Platform Specificatoin* in Figure 4.2 we can see that each MicroBlaze processor connects to one bank of ZBT SSRAM on the target FPGA platform through one custom memory controller. The MicroBlaze *MB_1* connects to the memory controller *ZBT_CTRL_1* through the link *mb_opb_1* to read the initial image data from the first ZBT SSRAM bank. The MicroBlaze *MB_2* connects to the memory controller *ZBT_CTRL_2* through the link *mb_opb_2* to write the resulting data to the second ZBT SSRAM bank. We also set the data memory size and program memory size for these two MicroBlaze processors to 8129 bytes each. In the *Platform Specification* we do not set any properties currently.

```
<platform name="myPlatform">

  <processor name="MB_1" type="MB" data_memory="8129" program_memory="8129">
    <port name="OPB_1" type="OPBPort"/>
  </processor>

  <processor name="HWN" type="CompaanHWNode">
  </processor>

  <processor name="MB_2" type="MB" data_memory="8129" program_memory="8129">
    <port name="OPB_2" 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"/>
  </link>

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

</platform>
```

Figure 4.2: *Platform Specification* for Heterpgeneous Embedded System of Sobel application

When the *Application Specification*, *Platform Specification* and *Mapping Specification* are all ready, we can start to run our ESPAM tool to automatically generate all of the necessary XPS project files for the Sobel edge detection application. The command of our ESPAM tool is shown in Figure 4.4. The options are explained below:

- --**platform**: This option specifies the *Platform Specification* file.
- --**kpn**: This option specifies the *Application Specification* file as KPN specification generated by COMPAAN.
- --**mapping**: This option specifies the *Mapping Specification* file.
- --**scheduler**: This option specifies a file which is used to describe a valid global schedule among the processes in the *Application Specification*.

```
1    <mapping name="myMapping">

        <processor name="MB_1">
            <process name="ND_1" />
5       </processor>

        <processor name="HWN">
            <process name="ND_2" />
            <process name="ND_3" />
10          <process name="ND_4" />
            <process name="ND_5" />
        </processor>

        <processor name="MB_2">
15          <process name="ND_6" />
        </processor>

     </mapping>
```

Figure 4.3: *Mapping Specification* for Heterpgeneous Embedded System of Sobel application

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

Figure 4.4: Command of the ESPAM tool

- --**xps**: This option is used to tell our ESPAM tool to generate all necessary files of an XPS project.
- --**libxps**: This option specifies the library which stores predefined components or files that are common for all projects. Such as some common custom IP cores, component files which are the same for all ESPAM generated IP core wrappers, the UCF files and some optional files for XPS implementation tools. Our ESPAM tool can copy these files from the library specified by this option during project generation. The $<libXPS>$ specifies the path to this library which is currently the CVS repository path *.../espam/src/espam/libXPS*.
- --**debugger**: This option is used to tell our ESPAM tool to generate components used for debugging.

After we run the command in Figure 4.4, the XPS project for this Sobel edge detection heterogeneous embedded system is generated. The project directory structure is shown in Figure 4.5.

The *system.xmp*, *system.mhs* and *system.mss* files are the corresponding XMP, MHS and MSS files which have been introduced in Section 2.3.3. The MHS file

```
<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_3/
|----------------------- P_3.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/
|------------- HWN_1_v1_00_a
|------------------- data
|------------------- devl
|------------------- hdl
|------------- HWN_2_v1_00_a
|------------------- data
|------------------- devl
|------------------- hdl
|------------- HWN_3_v1_00_a
|------------------- data
|------------------- devl
|------------------- hdl
|------------- HWN_4_v1_00_a
|------------------- data
|------------------- devl
|------------------- hdl
```

Figure 4.5: XPS project directory structure for the Sobel edge detection embedded system

for this system can be found in Appendix B. The *loader.exe* file is a program used to download and run the bitstream file. Directory *etc* contains *bitgen.ut* [22], *bitgen spartan3.ut*, *fast runtime.opt* [22] and *download.cmd* which store options

for the Xilinx implementation tools. The *data* directory contains several UCF files for different FPGA devices. These files are used to specify implementation constraints such as timing, FPGA pin locations, FPGA resource specification and IO standards. In our case, we use the system ADMXRCII.ucf UCF file for the physical implementation in the selected FPGA device.

Directory *code* contains the program code files for each processor in the platform. Each processor has a corresponding subdirectory in which the program source code file is stored. In Figure 4.5 the subdirectories *P_1* and *P_3* are for the two MicroBlaze processors separately. In the top level of the *code* directory, there are two files named *aux func.h* and *MemoryMap.h*. They are common for program codes of all processors. The *aux func.h* file declares read and write primitives as well as empty wrappers of all function calls in the initial Matlab code. The *MemoryMap.h* file specifies physical addresses of the components in the platform.

What need to be mentioned is that our ESPAM tool does not deal with the implementations of the function calls in the initial Matlab code. It only generates empty wrappers for these functions without implementation. To implement the application in XPS, these empty wrappers have to be replaced with corresponding software function source code or hardware function files. In this tutorial we map the *Copy()*, the two *Jc()* and the *Sb()* processes onto hardware IP modules. Thus we need to replace the generated empty wrapper VHDL files with VHDL files containing implementations. For the *_Read_m()* and *_Write_m()* processes which are mapped onto MicroBlaze processors the corresponding empty wrappers are replaced by corresponding software code. These will be explained in detail in Section 4.2.

The *pcores* directory contains all predefined hardware IP modules and the hardware IP modules 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 hardware IP modules working as an interface for the embedded system with the outside host processor and memory banks, the detailed information can be referred to [8]. The *fifo_if_ctrl_v1_00_a* is the LMB FIFO controller which is explained in [23].

The *clock_cycle_counter_v1_00_a* is the hardware IP module for debugging. The *myCLKRST_v1_00_a* is the hardware module which is used to generate the system clock and reset. The *cb_wrapper_v1_00_a*, *LMB_VB_CTRL_v1_00_a* and *VB_Wrapper_v1_00_a* are the hardware modules for the crossbar communication component. These components are not used in our current projects.

The four subdirectories *HWN_1_v1_00_a*, *HWN_2_v1_00_a*, *HWN_3_v1_00_a* and *HWN_4_v1_00_a* are corresponding to the four hardware modules for processes *Copy()*(ND_2), *Jc()*(ND_3/4), *Sb()*(ND_5). In each of these four directories there are again three subdirectories *data*, *devl* and *hdl*. The *data* directory contains the MPD and PAO files which we explained in Section 2.3.2. The *devl* directory currently contains no files. The *hdl* directory contains the VHDL files, which are listed in the PAO file, for the hardware module.

4.1.3 Importing Project to XPS

After the files for an XPS project is generated by our ESPAM, we have to import his project into XPS. To startXPS, we use the start menu of Windows: **start->Xilinx Platform Studio 7.1i->Xilinx Platform Studio**. In the XPS tool, select the menu option: **File->Open Project**. Then in the new dialog box, select the XMP file — *system.xmp* of our XPS project by double clicking on it. By doing these the project is loaded to XPS automatically. In our case we use XPS version 7.1. Because the files generated by ESPAM for our XPS project is based on XPS version 6.3, XPS will automatically ask whether we want it to be upgraded to version 7.1. We should just click *YES* to let the project be upgraded. We can get a visual 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**.

4.2 Custom Modifications

After we import the project to XPS, we still need to make some modifications manually on both hardware and software.

4.2.1 Hardware Modifications

In this section, we first describe the hardware modifications in detail. We will make changes to the hardware and system settings, the MHS file and the hardware modules.

Hardware Setting Modification

We modify the UCF file name first. In the *data* directory of our XPS project, there are several UCF files. When we import the project to XPS, it automatically recognizes and uses the UCF file named *system.ucf*. Since the UCF file we need to use for our project is *system_ADMXRCII.ucf* and there is already a file named *system.ucf* in the *data* directory, we have to first rename the original *system.ucf* to *system_old.ucf* and change the name of *system_ADMXRCII.ucf* file to *system.ucf*.

Another thing we need to modify is the *fast_runtime.opt* file. Although XPS will automatically upgrade our project files from version 6.3 to version 7.1, the fast_runtime.opt file which is stored in the *etc* directory will not be upgraded according to the corresponding version 7.1 format. 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 can only be set to *std*, *med* and *high*. Therefore, we need to manually change this number 5 to *std* as shown in Figure 4.6. The modified part is in bold font.

```
...
...
Program par
-w;                 # Overwrite existing placed and routed ncd
-ol  std;             # Overall effort level
<inputdir><design>_map.ncd;  # Input mapped NCD file
<design>.ncd;                # Output placed and routed NCD
<inputdir><design>.pcf;      # Input physical constraints file
END Program par
...
...
```

Figure 4.6: Modified fast_runtime.opt file

Also we need to set our target FPGA board. In the *System* tab of XPS, there is *Project Options* where the option *Device* resides in. Double click the *Device* option, then in the new dialog box poped up we set the target device to : *Architecture: virtex2, Device Size: xc2v6000, Package: ff1152, Grade: -5*. At last we click the *OK* button and XPS set this target device for our project.

MHS File Modification

The generated MHS file for this Sobel project is shown in Appendix B. We need to adjust the size of the FIFOs in the MHS file. By default, our ESPAM tool allocates 2048 bytes (512×32) for each FIFO. 512 is the data depth of a FIFO and 32 is the data width of a FIFO. Lines 357 and 358 of Appendix B show the

example of FIFO size setting in the MHS file. However, from the initial Matlab code, we find out that 2048 bytes are not enough for all the FIFOs. Since the off-chip memory is big enough for this application, we enlarge all the FIFOs to size 4096 bytes (1024×32). An example for the modification of the size of FIFO *FIFO_MB_1_Out_1* is shown in Figure 4.7. Sizes of other FIFOs can be modified in the same way.

```
...
...
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 = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END
...
...
```

Figure 4.7: Modified FIFO size example

Hardware module Modification

Since our ESPAM only generated wrappers for hardware modules, we need to add the corresponding VHDL function implementation files for each hardware module. In our project, we replace the empty function VHDL files with the corresponding VHDL files implementing the desired functions. For the hardware module *HWN_1_v1_00_a*, the *Copy.vhd* file which simply copies the input to the outputs should be added as replacement for the corresponding *Copy()* process. For the hardware module *HWN_2_v1_00_a* and *HWN_3_v1_00_a*, the *Jc.vhd* file which calculates the derivative approximations should be added for the corresponding *Jc()* processes. For the last hardware module *HWN_4_v1_00_a*, the *Sb.vhd* file which implements the calculation of the final gradients should be added for the corresponding *Sb()* process.

Because the function files are added manually and the corresponding function wrapper in the *execution_unit.vhd* is generated according to the ESPAM application model instance, the interface definition of the entities in these VHDL files can be different. We have to check the definitions of the entities *Copy*, *Jc* and *Sb*. Then compare them with the corresponding component declaration for these entities. If they are different, we should modify the part in the *execution_unit.vhd* to

conform to the entity definition in the added function VHDL files. In this project, the *execution_unit.vhd* for the hardware module *HWN_1_v1_00_a* does not need to be modified. All the others need to be modified. Since for all the other hardware modules, there is one more output port *RDY* in the entity definition than in the component declaration in *execution_unit.vhd*, we simply add this port to the component declaration. An example for *HWN_4_v1_00_a* is given in Figure 4.8.

```
...
...
  architecture RTL of EXECUTION_UNIT is

    component Sb is
port (
      RST   : in std_logic;
      CLK   : in std_logic;

        in_0  : in std_logic_vector(KWANT - 1 downto 0);
        in_1  : in std_logic_vector(KWANT - 1 downto 0);

        out_0 : out std_logic_vector(KWANT -1 downto 0);

        RDY   : out std_logic;
        EN    : in std_logic
      );
    end component;
...
...
```

Figure 4.8: Modified *execution_unit.vhd* example

The last thing we need to modify for the hardware modules is to modify the corresponding pipeline stage number of the function in the top-level VHDL file for the hardware module. Because currently our ESPAM has not supported pipeline stage setting in the application specification, we have to set the stage numbers manually in the corresponding line in the top-level files. The top-level files for the hardware modules in this project are *HWN_1.vhd*, *HWN_2.vhd*, *HWN_3.vhd* and *HWN_4.vhd* in the corresponding hardware module directories. In these files there is a part for parameter setting for the hardware module. An example of the corresponding modification for the hardware module *HWN_4_v1_00_a* is given in Figure 4.9. The pipeline stage numbers for the *Copy*, *Jc* and *Sb* functions are one, two and two separately.

4.2.2 Software Modifications

After the hardware modifications, we have to make some modifications to the software part of the project.

```
...
...
  -- Setting the parameters of the HW Node
  constant c_IN_PORTS     : natural := 2; -- # of input ports of a HW node
  constant c_OUT_PORTS    : natural := 1; -- # of output ports of a HW node
  constant c_IN_FUNC_VAR  : natural := 2; -- # of input ports of a HW IP
  constant c_OUT_FUNC_VAR : natural := 1; -- # of output ports of a HW IP
  constant N_PAR          : natural := 2; -- # of global parameters
  constant c_par_values   : t_par_values := (0=>4, 1=>63, others=>0 );
                          -- each number represents
                             the default value of a parameter
  constant c_COUNTERS     : natural := 2; -- # of iterators
  constant c_cntr_widths  : t_counter_width := ( 0=>11, 1=>11, others=>10 );
  constant c_STAGES       : natural := 2;
                          -- # of stages of the pipeline or delay
  constant c_BLOCKING     : natural := 1;
                          -- block (or not) the pipeline
                             if there is no input data
  constant c_IP_RESET     : natural := 1; -- active level of the
                                             HW IP reset signal
...
...
```

Figure 4.9: Modified *HWN_4.vhd*

We already explained that our ESPAM only generates empty wrappers for function calls, we first need to import all the implementations of the function calls manually. The function calls for the two MicroBlaze processors for this project are *_Read_m* and *_Write_m* for data initialization and output. We can first copy the implementation source code files for these two function calls *Sobel_func.h*, *Video_in.h* and *Video_in.cpp* to the *code* directory. Then we manually import these files for the corresponding MicroBlaze processor.

There is an **Application** tab in XPS where two software projects for the two processors can be found. In each software project, there are *Sources* option and *Headers* option. Double click the *Sources* option, then in the new dialog box we can import the implementation program code files for each processor. Double click the *Headers* option, then in the new dialog box we can import the head files for each processor. Since both processors use the same header and implementation files, for both *Proj_MB_1* and *Proj_MB_2* we import *Video_in.cpp* in *Sources* option; *Sobel_func.h* and *Video_in.h* in *Headers* option. After importing normally we also set the stack size for each processor project by double clicking the *Compiler Options* option in the **Application** tab to fill in the stack size. But for this Sobel project the default stack size is big enough, we do not need to set the stack size manually.

The next 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 4.10.

The bold lines in the code highlight the modification which we need to do manually. In Line 6 we include *Sobel_func.h* header files. In lines 21-22, we replace the empty wrapper with the actual function call. The program code of the other processors can be modified in the same way.

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

    int main ()

10    int clk_num;
      *clk_cntr = 0;

      // Input Arguments

15    // Output Arguments
      tCH_1 out_0ND_1;

      for( int j =  ceil1(1); j <=  floor1(M ); j += 1 )
        for( int i =  ceil1(1); i <=  floor1(N ); i += 1 )
20
            //_Read_m(out_0ND_1) ;
            out_0ND_1 = _Read_m() ;

            writeFSL(ND_1_OG_1_CH_1, &out_0ND_1,
25                   (sizeof(tCH_1)+(sizeof(tCH_1)%4)+3)/4);
         // for i
       // for j

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

Figure 4.10: Modified program code for processor *P_1*

We also need to modify the *aux_func.h* file which is generated by our ESPAM. The modified *aux_func.h* file is shown in Figure 4.11. The bold lines in the code highlight the modification that we need to do manually. In lines 25-26 we change the parameter numbers into 450 and 275 which are the size of the input image width and height. Also because we have already replaced the empty wrappers with the actual function calls in program code of each processor, we need to comment the inline empty wrapper declarations which are generated by ESPAM in lines 28-33.

The last thing is to modify the *MemoryMap.h* file. The modified file is shown in Figure 4.12. In lines 28-29 we add the physical address assignment for *ZBT_MEMORY_MB1* and *ZBT_MEMORY_MB2*. Because in the file *Video_in.cpp* we need these two addresses to read data from and write result to. The complete modified project can be found in the CVS repository:
docs/students/YingTao/experiment/Sobel_ip_int.zip

```
1    #ifndef __AUX_FUNC_H__
     #define __AUX_FUNC_H__

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

     typedef int tCH_1;
     typedef int tCH_2;
     typedef int tCH_3;
10   typedef int tCH_4;
     typedef int tCH_5;
     typedef int tCH_6;
     typedef int tCH_7;
     typedef int tCH_8;
15   typedef int tCH_9;
     typedef int tCH_10;
     typedef int tCH_11;
     typedef int tCH_12;
     typedef int tCH_13;
20   typedef int tCH_14;
     typedef int tCH_15;
     typedef int tCH_16;

     // Parameters
25   #define N  450
     #define M  275

     /*inline
     void  _Read_m( tCH_1 *out_0 )
30

     inline
     void  _Write_m( tCH_16 in_0, char *out_0 )

35   */

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

Figure 4.11: Modified *aux_func.h* file

```
1    #ifndef __MEMORYMAP_H_
     #define __MEMORYMAP_H_

     #define PCTRL_BRAM1_MB_1 0x00000000 //read from PCTRL_BRAM1_MB_1 address for MB_1
5    #define PCTRL_BRAM1_MB_1 0x00000000 //write to PCTRL_BRAM1_MB_1 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 ZBT_CTRL_1 0xf0000000 //read from ZBT_CTRL_1 address for MB_1
10   #define ZBT_CTRL_1 0xf0000000 //write to ZBT_CTRL_1 address for MB_1

     //MB_1 FIFOs
     #define ND_1_OG_1_CH_1 0 //write to CDChannelCH_1 address for MB_1

15   #define PCTRL_BRAM1_MB_2 0x00000000 //read from PCTRL_BRAM1_MB_2 address for MB_2
     #define PCTRL_BRAM1_MB_2 0x00000000 //write to PCTRL_BRAM1_MB_2 address for MB_2
     #define DCTRL_BRAM1_MB_2 0x00000000 //read from DCTRL_BRAM1_MB_2 address for MB_2
     #define DCTRL_BRAM1_MB_2 0x00000000 //write to DCTRL_BRAM1_MB_2 address for MB_2

20   #define ZBT_CTRL_2 0xf0000000 //read from ZBT_CTRL_2 address for MB_2
     #define ZBT_CTRL_2 0xf0000000 //write to ZBT_CTRL_2 address for MB_2

     //MB_2 FIFOs
     #define ND_6_IG_1_CH_16 0 //read from CDChannelCH_16 address for MB_2
25
     #define clk_cntr (volatile int *)0xf8000000
     #define FIN_SIGNAL (volatile long *)0xf9000000
     #define ZBT_MEMORY_MB1 (volatile long *)0xf0000000
     #define ZBT_MEMORY_MB2 (volatile long *)0xf0000000
30   #define ZBT_MEM_CLK (volatile long *)0xf00ffffc

     #endif
```

Figure 4.12: Modified *MemoryMap.h* file

4.3 XPS Project Execution and Results

Once we finish with importing our project to XPS and all of the modifications for our project, we are ready to use XPS to generate the final bitstream file. The bitstream file is used to configure the FPGA chip to implement the Sobel edge detection application. We use the following commands to generate the bitstresm step by step. All these commands can be found in the menu option **Tools** in XPS tool.

- **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.
- **Compile Program Source**: This command invokes the cross compiler *mc-gcc*. This compiler generates several ELF executable files, one for each processor in the system, by compiling the program code of each processor. If *LibGen* has not been executed, this command first invokes *LibGen*.
- **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.
- **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 invoke them one by one. At the end of this stage, we can get the resulting bitstream file *download.bit* which is located in the *implementation* directory of our project. This bitstream file contains the entire FPGA configuration information regarding both the software and the hardware part of the heterogenous embedded system.

In order to download the final bitstream file onto the target FPGA board to get the resulting data, we need to use a software program in an outside host processor to communicate with our target ADM-XRC-II board. This software program

is a Microsoft Visual C++ 6.0 project which uses the ADM-XRC application-programming interface (API) to take care of open, close and device I/O control calls to the driver of the ADM-XRC-II board. The main code of the software program is shown in Appendix C.

From the appendix, we can see that in Line 35 the initial image data is put into the first off-chip memory bank. Lines 36-40 initialize the other five memory banks with zero. Lines 49-51 load the first parameter by putting the first parameter value to address for the parameter register on the FPGA board and enable the command for loading parameter for a while. The second parameter is loaded in the same way in Lines 54-56. Then in lines 63-69, our Sobel heterogeneous embedded system reads the initial image data from the off-chip memory, executes the Sobel edge detection application for the initial image data and writes the resulting image data into the second off-chip memory bank corresponding to the second MicroBlaze processor. In line 79 the outside host processor reads back the resulting image data from the off-chip memory. The last few lines 90-100 write the image data into a raw file and convert it into a JPEG file.

Therefore, in order to download the final bitstream file onto the target FPGA board and get the resulting data, we just need to copy the XPS generated final bitstream file to the directory of this software program. Then we compile and run this software program with Microsoft Visual C++ 6.0 to get the resulting image data in the outside host processor. This software program can be found in the CVS repository:
docs/students/YingTao/experiment/PentiumProgram_Sobel.zip

Chapter 5

Summary and Conclusions

In this thesis, we first introduced our tool ESPAM and the motivation for us to make ESPAM support heterogeneous multiprocessor system generation. Later more details about the structure of the IP core wrapper and the implementation ideas are explained. After that case study of several XPS projects and detailed tutorial are given.

Our system design methodology allows efficient and effective mapping of a class of multimedia and signal processing applications onto heterogeneous multiprocessor platforms in a systematic and automated way. By using our ESPAM tool, designers can easily design heterogeneous multiprocessor embedded systems for various applications and get the XPS project implementation. This thesis mainly integrate hardware IP cores into ESPAM generated systems. Evaluation of the result system performances and comparison with other similar work are done in this thesis. Better time performance can be met with this heterogeneous architecture.

Essential to the hardware IP core part is the structure of the IP wrappers. We make them conform to the behaviour of Kahn Process Network processes. The internal components of a wrapper are clearly structured and modularized, which makes the components loosely coupled for easy separate optimization. Since these predefined and parameterized internal components are with clearly defined interface, we only have to instantiate these components by setting the component parameters for wrapper generations.

Future improvements for our ESPAM tool can be made in three aspects.

Firstly, we can make the IP core wrapper part more general, automatic and

optimized.

- Currently we only support the iterator step as +1. This can be made parameterized later.
- We have to take the IP cores manually from a library and add them. A later work can let the platform specification part contain the specification for IP cores of certain functions. So the IP cores can be copied from ESPAM library if it exists instead of being added manually.
- For the control information evaluation for reading and writing, we can consider to further optimize the control expressions or use another way to evaluate the expressions to make sure any expression evaluation will always be faster than the execution of the function.
- The hardware modules in ESPAM take scalar values as data inputs. Further work can make complex data types also supported by dividing them at reading and restoring them in temporary memories for execution.

Secondly, we can make the already existing ESPAM model instances support more general cases.

- The input ports of an ADG node may be related to more than one or none input argument of the ADG function. However, our hardware modules do not support this situation yet because the application input from COMPAAN does not deal with this. Further work can be done to support this situation.
- ESPAM models support FIFO channels as communication channels. Communication components other than FIFOs, such as Reordering Channels, should be able to be mapped as platform channels to deal with out-of-order communication [24].

The last improvement can be made to make the ESPAM design flow more automated.

- Currently the application specification input can be generated by COMPAAN. However, we still have to specify manually the mapping specification and the platform specification. In the future we can integrate an automatic mapping part in our ESPAM tool. This part can give one or several

optimal mappings which map the applications onto specific platforms. The mappings can be decided by considering the device resource and the application processes.

- If we use COMPAAN generated KPN as the application specification input of ESPAM, we have to calculate the FIFO sizes and modify them manually if it is necessary. We can also use the *Process Network Generator* tool (PNgen) generated networks as the application specification input of ESPAM. Then we can get the FIFO sizes measured in tokens and calculate the exact sizes using the token size. We can further automate this aspect in ESPAM.

Appendix A

Initial Matlab code for DWT application

```
1   %parameter Nrow 10 256; %% The first time Nrow=Height/2 So, the max image height is 512
    %parameter Ncol 10 256; %% In the algorithm we use Ncol*2 and Ncol

    %typedef image int;
5   %typedef tmpLine int;
    %typedef Hf int;
    %typedef oldHf int;
    %typedef Lf int;
    %typedef tmp int;
10  %typedef buffLow int;
    %typedef buffHigh int;
    %typedef LL int;
    %typedef LH int;
    %typedef HL int;
15  %typedef HH int;
    %typedef result int;

     for i=0:1:2*Nrow -1,
       for j=0:1:2*Ncol-1,
20       [image(i,j)]=init();
       end
     end

     for i=0:1:Nrow-1,
25     %% DWT by columns at pixel level (3 elements per column with subsampling)
       for j=0:1:2*Ncol -1,

          if i>=Nrow-1,
            [ tmpLine ] = copy( image(2*i,j) );
30        else
            [ tmpLine ] = copy( image(2*i+2,j) );
          end

          %% compute high frequency coeff.
35        // High Pass Filter
          [ Hf(j) ] = my_high_flt_vert( image(2*i,j),image(2*i+1,j),tmpLine );

          if i<=0,
            [ oldHf(j) ] = copy_Hf( Hf(j) );
40        end

          %% compute low frequency coeff.
          // Low Pass Filter
          [ Lf(j), oldHf(j) ] = my_low_flt_vert( oldHf(j), image(2*i,j), Hf(j) );
45     end
```

```
      %% variables Lf and Hf represent 1 line of Low and High pass filtered image resectively
      %% DWT by rows at pixel level (Low Pass Filter with subsampling)
      ---------------------------------
50    for j=0:1:Ncol-1,

         if j>=Ncol-1, %% 2*j==Ncol-2
           [ tmp ] = copy( Lf(2*j) );
         else
55         [ tmp ] = copy( Lf(2*j+2) );
         end

         // High Pass Filter
         [ buffLow(2*j+1), HL(i,j) ] = my_high_flt_hor( Lf(2*j), Lf(2*j+1), tmp );
60
         if j<=0,
           [ tmp ] = copy( buffLow(2*j+1) ); // or HL[c][i][j]; // one reg is enough
         else
           [ tmp ] = copy( buffLow(2*j-1) ); // or HL[c][i][j-1];
65       end

         // Low Pass Filter
         [ buffLow(2*j), LL(i,j) ] = my_low_flt_hor( tmp, Lf(2*j),  buffLow(2*j+1) );

70    end

      %% DWT rows row at pixel level (High Pass Filter with subsampling)
      ---------------------------------
      for j=0:1:Ncol-1,
75
         if j>=Ncol-1,
           [ tmp ] = copy( Hf(2*j) );
         else
           [ tmp ] = copy( Hf(2*j+2) );
80       end

         // High Pass Filter
         [ buffHigh(2*j+1), HH(i,j) ] = my_high_flt_hor( Hf(2*j), Hf(2*j+1), tmp );

85       if j<=0,
           [ tmp ] = copy( buffHigh(2*j+1) ); // or HH[c][i][j];
         else
           [ tmp ] = copy( buffHigh(2*j-1) ); // or HH[c][i][j-1];
         end
90
         // Low Pass Filter
         [ buffHigh(2*j), LH(i,j) ] = my_low_flt_hor( tmp, Hf(2*j),  buffHigh(2*j+1) );

       end
95   end

     %% The Sink

     for i=0:1:Nrow-1,
100     for j=0:1:Ncol-1,
          [ result(i,j) ]           = sink( LL(i,j) );
          [ result(i,Ncol+j) ]      = sink( HL(i,j) );
          [ result(Nrow+i,j) ]      = sink( LH(i,j) );
          [ result(Nrow+i,Ncol+j) ] = sink( HH(i,j) );
105     end
     end
```

Appendix B

MHS file for Sobel Heterogeneous Embedded System project

```
1   ## File automatically generated by ESPAM
2
3
4    PARAMETER VERSION = 2.1.0
5    PORT lclk = lclk , DIR = IN
6    PORT mclk = mclk , DIR = IN
7    PORT ramclki = ramclki , VEC = [1:0] , DIR = IN
8    PORT ramclko = ramclko , VEC = [1:0] , DIR = OUT
9    PORT lreseto_l = lreseto_l , DIR = IN
10   PORT lwrite = lwrite , DIR = IN
11   PORT lads_l = lads_l , DIR = IN
12   PORT lblast_l = lblast_l , DIR = IN
13   PORT lbterm_l = lbterm_l , DIR = INOUT
14   PORT ld = ld , VEC = [31:0] , DIR = INOUT
15   PORT la = la , VEC = [23:2] , DIR = IN
16   PORT lreadyi_l = lreadyi_l , DIR = OUT
17   PORT lbe_l = lbe_l , VEC = [3:0] , DIR = IN
18   PORT fholda = fholda , DIR = IN
19   PORT ra0 = ra0 , VEC = [19:0] , DIR = OUT
20   PORT rd0 = rd0 , VEC = [31:0] , DIR = INOUT
21   PORT rc0 = rc0 , VEC = [8:0] , DIR = OUT
22   PORT ra1 = ra1 , VEC = [19:0] , DIR = OUT
23   PORT rd1 = rd1 , VEC = [31:0] , DIR = INOUT
24   PORT rc1 = rc1 , VEC = [8:0] , DIR = OUT
25   PORT ra2 = ra2 , VEC = [19:0] , DIR = OUT
26   PORT rd2 = rd2 , VEC = [31:0] , DIR = INOUT
27   PORT rc2 = rc2 , VEC = [8:0] , DIR = OUT
28   PORT ra3 = ra3 , VEC = [19:0] , DIR = OUT
29   PORT rd3 = rd3 , VEC = [31:0] , DIR = INOUT
30   PORT rc3 = rc3 , VEC = [8:0] , DIR = OUT
31   PORT ra4 = ra4 , VEC = [19:0] , DIR = OUT
32   PORT rd4 = rd4 , VEC = [31:0] , DIR = INOUT
33   PORT rc4 = rc4 , VEC = [8:0] , DIR = OUT
34   PORT ra5 = ra5 , VEC = [19:0] , DIR = OUT
35   PORT rd5 = rd5 , VEC = [31:0] , DIR = INOUT
36   PORT rc5 = rc5 , VEC = [8:0] , DIR = OUT
37
38  BEGIN lmb_v10
39   PARAMETER INSTANCE = PBUS_MB_1
40   PARAMETER HW_VER = 1.00.a
41   PARAMETER C_EXT_RESET_HIGH = 0
42   PORT SYS_Rst = net_design_rst
43   PORT LMB_Clk = sys_clk_s
44  END
45
46  BEGIN lmb_v10
47   PARAMETER INSTANCE = DBUS_MB_1
48   PARAMETER HW_VER = 1.00.a
49   PARAMETER C_EXT_RESET_HIGH = 0
50   PORT SYS_Rst = net_design_rst
51   PORT LMB_Clk = sys_clk_s
52  END
53
54  BEGIN opb_v20
55   PARAMETER INSTANCE = mb_opb_1
56   PARAMETER HW_VER = 1.10.c
57   PARAMETER C_EXT_RESET_HIGH = 0
58   PORT SYS_Rst = net_design_rst
59   PORT OPB_Clk = sys_clk_s
60  END
61
62  BEGIN fin_ctrl
63   PARAMETER INSTANCE = fin_ctrl_P1
64   PARAMETER HW_VER = 1.00.a
65   PARAMETER C_BASEADDR = 0xf9000000
66   PARAMETER C_HIGHADDR = 0xf900000f
67   PARAMETER C_AB = 8
68   BUS_INTERFACE SLMB = DBUS_MB_1
69   PORT S1_FinOut = net_fin_signal_P1
70  END
71
72  BEGIN clock_cycle_counter
73   PARAMETER INSTANCE = clock_cycle_counter_P1
74   PARAMETER HW_VER = 1.00.a
75   PARAMETER C_BASEADDR = 0xf8000000
76   PARAMETER C_HIGHADDR = 0xf8000003
77   BUS_INTERFACE SLMB = DBUS_MB_1
78   PORT LMB_Clk = sys_clk_s
79  END
80
81  BEGIN microblaze
82   PARAMETER INSTANCE = MB_1
83   PARAMETER HW_VER = 4.00.a
84   PARAMETER C_NUMBER_OF_PC_BRK = 1
85   PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
86   PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
87   BUS_INTERFACE MFSL0 = FIFO_MB_1_Out_1
88   BUS_INTERFACE DLMB = DBUS_MB_1
89   BUS_INTERFACE ILMB = PBUS_MB_1
90   BUS_INTERFACE DOPB = mb_opb_1
91   PARAMETER C_FSL_LINKS = 1
92   PORT CLK = sys_clk_s
93  END
94
95  BEGIN lmb_v10
96   PARAMETER INSTANCE = PBUS_MB_2
```

```
97   PARAMETER HW_VER = 1.00.a
98   PARAMETER C_EXT_RESET_HIGH = 0
99   PORT SYS_Rst = net_design_rst
100  PORT LMB_Clk = sys_clk_s
101 END
102
103 BEGIN lmb_v10
104  PARAMETER INSTANCE = DBUS_MB_2
105  PARAMETER HW_VER = 1.00.a
106  PARAMETER C_EXT_RESET_HIGH = 0
107  PORT SYS_Rst = net_design_rst
108  PORT LMB_Clk = sys_clk_s
109 END
110
111 BEGIN opb_v20
112  PARAMETER INSTANCE = mb_opb_2
113  PARAMETER HW_VER = 1.10.c
114  PARAMETER C_EXT_RESET_HIGH = 0
115  PORT SYS_Rst = net_design_rst
116  PORT OPB_Clk = sys_clk_s
117 END
118
119 BEGIN fin_ctrl
120  PARAMETER INSTANCE = fin_ctrl_P2
121  PARAMETER HW_VER = 1.00.a
122  PARAMETER C_BASEADDR = 0xf9000000
123  PARAMETER C_HIGHADDR = 0xf900000f
124  PARAMETER C_AB = 8
125  BUS_INTERFACE SLMB = DBUS_MB_2
126  PORT Sl_FinOut = net_fin_signal_P2
127 END
128
129 BEGIN clock_cycle_counter
130  PARAMETER INSTANCE = clock_cycle_counter_P2
131  PARAMETER HW_VER = 1.00.a
132  PARAMETER C_BASEADDR = 0xf8000000
133  PARAMETER C_HIGHADDR = 0xf8000003
134  BUS_INTERFACE SLMB = DBUS_MB_2
135  PORT LMB_Clk = sys_clk_s
136 END
137
138 BEGIN microblaze
139  PARAMETER INSTANCE = MB_2
140  PARAMETER HW_VER = 4.00.a
141  PARAMETER C_NUMBER_OF_PC_BRK = 1
142  PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
143  PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
144  BUS_INTERFACE SFSL0 = FIFO_HWN_4_Out_15
145  BUS_INTERFACE DLMB = DBUS_MB_2
146  BUS_INTERFACE ILMB = PBUS_MB_2
147  BUS_INTERFACE DOPB = mb_opb_2
148  PARAMETER C_FSL_LINKS = 1
149  PORT CLK = sys_clk_s
150 END
151
152 BEGIN zbt_main
153  PARAMETER INSTANCE = host_zbt_main
154  PARAMETER HW_VER = 1.00.a
155  BUS_INTERFACE HOST_BUFF_0_PORT = buff_rd_0
156  BUS_INTERFACE HOST_BUFF_1_PORT = buff_rd_1
157  BUS_INTERFACE HOST_BUFF_2_PORT = buff_rd_2
158  BUS_INTERFACE HOST_BUFF_3_PORT = buff_rd_3
159  BUS_INTERFACE HOST_BUFF_4_PORT = buff_rd_4
160  BUS_INTERFACE HOST_BUFF_5_PORT = buff_rd_5
161  BUS_INTERFACE HOST_MUX_PORT = mux_to_host
162  PORT lclk = lclk
163  PORT mclk = mclk
164  PORT ramclko = ramclko
165  PORT ramclki = ramclki
166  PORT lreseto_l = lreseto_l
167  PORT lwrite = lwrite
168  PORT lads_l = lads_l
169  PORT lblast_l = lblast_l
170  PORT lbterm_l = lbterm_l
171  PORT ld = ld
172  PORT la = la
173  PORT lreadyi_l = lreadyi_l
174  PORT lbe_l = lbe_l
175  PORT fholda = fholda
176  PORT CLK_out = sys_clk_s
177  PORT RST_out = sys_rst_s
178  PORT COMMAND_REG = net_command
179  PORT DESIGN_STAT_REG = net_design_status
180  PORT PARAMETER_REG = net_parameter
181 END
182
183 BEGIN host_design_ctrl
184  PARAMETER INSTANCE = host_design_controller
185  PARAMETER HW_VER = 1.00.a
186  PARAMETER N_FIN = 6
187  PARAMETER PAR_WIDTH  = 16
188  PORT RST = sys_rst_s
189  PORT COMMAND_REG = net_command
190  PORT STATUS_REG = net_design_status
191  PORT PARAMETER_REG = net_parameter
192  PORT RST_OUT = net_design_rst
193  PORT FIN_REG_0 = net_fin_signal_P1
194  PORT FIN_REG_1 = net_fin_signal_P2
195  PORT FIN_REG_2 = net_fin_signal_IP_1
196  PORT FIN_REG_3 = net_fin_signal_IP_2
197  PORT FIN_REG_4 = net_fin_signal_IP_3
198  PORT FIN_REG_5 = net_fin_signal_IP_4
199  BUS_INTERFACE PAR_BUS = PARBUS
200 END
201
202 BEGIN mux
203  PARAMETER INSTANCE = multiplexer
204  PARAMETER HW_VER = 1.00.a
205  PARAMETER N_MUX = 2
206  BUS_INTERFACE MUX_BUFF_PORT = buff_to_mux
207  BUS_INTERFACE MUX_DESIGN_0_PORT = mux_design_0
208  BUS_INTERFACE MUX_DESIGN_1_PORT = mux_design_1
209  BUS_INTERFACE MUX_HOST_PORT = mux_to_host
210  PORT ra0 = ra0
211  PORT ra1 = ra1
212  PORT ra2 = ra2
213  PORT ra3 = ra3
214  PORT ra4 = ra4
215  PORT ra5 = ra5
216  PORT rc0 = rc0
217  PORT rc1 = rc1
218  PORT rc2 = rc2
219  PORT rc3 = rc3
220  PORT rc4 = rc4
221  PORT rc5 = rc5
222  PORT RST = sys_rst_s
223  PORT CNTRL = net_command
224 END
225
226 BEGIN buffers
227  PARAMETER INSTANCE = buff
228  PARAMETER HW_VER = 1.00.a
229  BUS_INTERFACE BUFF_MUX_PORT = buff_to_mux
230  BUS_INTERFACE BUFF_RD_0_PORT = buff_rd_0
231  BUS_INTERFACE BUFF_RD_1_PORT = buff_rd_1
232  BUS_INTERFACE BUFF_RD_2_PORT = buff_rd_2
233  BUS_INTERFACE BUFF_RD_3_PORT = buff_rd_3
234  BUS_INTERFACE BUFF_RD_4_PORT = buff_rd_4
235  BUS_INTERFACE BUFF_RD_5_PORT = buff_rd_5
236  PORT rd0 = rd0
237  PORT rd1 = rd1
238  PORT rd2 = rd2
239  PORT rd3 = rd3
240  PORT rd4 = rd4
241  PORT rd5 = rd5
242 END
243
244 BEGIN opb_zbt_controller
245  PARAMETER INSTANCE = ZBT_CTRL_1
246  PARAMETER HW_VER = 1.00.a
247  PARAMETER C_BASEADDR = 0xf0000000
248  PARAMETER C_HIGHADDR = 0xf00fffff
249  PARAMETER C_EXTERNAL_DLL = 1
250  PARAMETER C_ZBT_ADDR_SIZE = 20
251  BUS_INTERFACE SOPB = mb_opb_1
252  BUS_INTERFACE DESIGN_BUFF_PORT = buff_rd_0
253  BUS_INTERFACE DESIGN_MUX_PORT = mux_design_0
254 END
255
256 BEGIN opb_zbt_controller
257  PARAMETER INSTANCE = ZBT_CTRL_2
258  PARAMETER HW_VER = 1.00.a
259  PARAMETER C_BASEADDR = 0xf0000000
260  PARAMETER C_HIGHADDR = 0xf00fffff
261  PARAMETER C_EXTERNAL_DLL = 1
262  PARAMETER C_ZBT_ADDR_SIZE = 20
```

```
263   BUS_INTERFACE SOPB = mb_opb_2
264   BUS_INTERFACE DESIGN_BUFF_PORT = buff_rd_1
265   BUS_INTERFACE DESIGN_MUX_PORT = mux_design_1
266 END
267
268 BEGIN HWN_1
269   PARAMETER INSTANCE = HWN_1_ip
270   PARAMETER HW_VER = 1.00.a
271   PARAMETER RESET_HIGH = 0
272   PARAMETER PAR_WIDTH = 16
273   PARAMETER KWANT = 32
274
275   BUS_INTERFACE In_1 = FIFO_MB_1_Out_1
276   BUS_INTERFACE Out_1 = FIFO_HWN_1_Out_1
277   BUS_INTERFACE Out_2 = FIFO_HWN_1_Out_2
278   BUS_INTERFACE Out_3 = FIFO_HWN_1_Out_3
279   BUS_INTERFACE Out_4 = FIFO_HWN_1_Out_4
280   BUS_INTERFACE Out_5 = FIFO_HWN_1_Out_5
281   BUS_INTERFACE Out_6 = FIFO_HWN_1_Out_6
282   BUS_INTERFACE Out_7 = FIFO_HWN_1_Out_7
283   BUS_INTERFACE Out_8 = FIFO_HWN_1_Out_8
284   BUS_INTERFACE Out_9 = FIFO_HWN_1_Out_9
285   BUS_INTERFACE Out_10 = FIFO_HWN_1_Out_10
286   BUS_INTERFACE Out_11 = FIFO_HWN_1_Out_11
287   BUS_INTERFACE Out_12 = FIFO_HWN_1_Out_12
288   BUS_INTERFACE PAR_BUS = PARBUS
289   PORT CLK = sys_clk_s
290   PORT RST = net_design_rst
291   PORT STOP = net_fin_signal_IP_1
292 END
293
294 BEGIN HWN_2
295   PARAMETER INSTANCE = HWN_2_ip
296   PARAMETER HW_VER = 1.00.a
297   PARAMETER RESET_HIGH = 0
298   PARAMETER PAR_WIDTH = 16
299   PARAMETER KWANT = 32
300
301   BUS_INTERFACE In_2 = FIFO_HWN_1_Out_1
302   BUS_INTERFACE In_3 = FIFO_HWN_1_Out_2
303   BUS_INTERFACE In_4 = FIFO_HWN_1_Out_3
304   BUS_INTERFACE In_5 = FIFO_HWN_1_Out_4
305   BUS_INTERFACE In_6 = FIFO_HWN_1_Out_5
306   BUS_INTERFACE In_7 = FIFO_HWN_1_Out_6
307   BUS_INTERFACE Out_13 = FIFO_HWN_2_Out_13
308   BUS_INTERFACE PAR_BUS = PARBUS
309   PORT CLK = sys_clk_s
310   PORT RST = net_design_rst
311   PORT STOP = net_fin_signal_IP_2
312 END
313
314 BEGIN HWN_3
315   PARAMETER INSTANCE = HWN_3_ip
316   PARAMETER HW_VER = 1.00.a
317   PARAMETER RESET_HIGH = 0
318   PARAMETER PAR_WIDTH = 16
319   PARAMETER KWANT = 32
320
321   BUS_INTERFACE In_8 = FIFO_HWN_1_Out_7
322   BUS_INTERFACE In_9 = FIFO_HWN_1_Out_8
323   BUS_INTERFACE In_10 = FIFO_HWN_1_Out_9
324   BUS_INTERFACE In_11 = FIFO_HWN_1_Out_10
325   BUS_INTERFACE In_12 = FIFO_HWN_1_Out_11
326   BUS_INTERFACE In_13 = FIFO_HWN_1_Out_12
327   BUS_INTERFACE Out_14 = FIFO_HWN_3_Out_14
328   BUS_INTERFACE PAR_BUS = PARBUS
329   PORT CLK = sys_clk_s
330   PORT RST = net_design_rst
331   PORT STOP = net_fin_signal_IP_3
332 END
333
334 BEGIN HWN_4
335   PARAMETER INSTANCE = HWN_4_ip
336   PARAMETER HW_VER = 1.00.a
337   PARAMETER RESET_HIGH = 0
338   PARAMETER PAR_WIDTH = 16
339   PARAMETER KWANT = 32
340
341   BUS_INTERFACE In_14 = FIFO_HWN_2_Out_13
342   BUS_INTERFACE In_15 = FIFO_HWN_3_Out_14
343   BUS_INTERFACE Out_15 = FIFO_HWN_4_Out_15
344   BUS_INTERFACE PAR_BUS = PARBUS
345   PORT CLK = sys_clk_s
346   PORT RST = net_design_rst
347   PORT STOP = net_fin_signal_IP_4
348 END
349
350 BEGIN fsl_v20
351  PARAMETER HW_VER = 2.00.a
352  PARAMETER INSTANCE = FIFO_MB_1_Out_1
353  PARAMETER C_EXT_RESET_HIGH = 0
354  PARAMETER C_ASYNC_CLKS = 0
355  PARAMETER C_IMPL_STYLE = 1
356  PARAMETER C_USE_CONTROL = 0
357  PARAMETER C_FSL_DWIDTH = 32
358  PARAMETER C_FSL_DEPTH = 1024
359  PORT FSL_Clk = sys_clk_s
360  PORT SYS_Rst = net_design_rst
361 END
362
363 BEGIN fsl_v20
364  PARAMETER HW_VER = 2.00.a
365  PARAMETER INSTANCE = FIFO_HWN_1_Out_1
366  PARAMETER C_EXT_RESET_HIGH = 0
367  PARAMETER C_ASYNC_CLKS = 0
368  PARAMETER C_IMPL_STYLE = 1
369  PARAMETER C_USE_CONTROL = 0
370  PARAMETER C_FSL_DWIDTH = 32
371  PARAMETER C_FSL_DEPTH = 1024
372  PORT FSL_Clk = sys_clk_s
373  PORT SYS_Rst = net_design_rst
374 END
375
376 BEGIN fsl_v20
377  PARAMETER HW_VER = 2.00.a
378  PARAMETER INSTANCE = FIFO_HWN_1_Out_2
379  PARAMETER C_EXT_RESET_HIGH = 0
380  PARAMETER C_ASYNC_CLKS = 0
381  PARAMETER C_IMPL_STYLE = 1
382  PARAMETER C_USE_CONTROL = 0
383  PARAMETER C_FSL_DWIDTH = 32
384  PARAMETER C_FSL_DEPTH = 1024
385  PORT FSL_Clk = sys_clk_s
386  PORT SYS_Rst = net_design_rst
387 END
388
389 BEGIN fsl_v20
390  PARAMETER HW_VER = 2.00.a
391  PARAMETER INSTANCE = FIFO_HWN_1_Out_3
392  PARAMETER C_EXT_RESET_HIGH = 0
393  PARAMETER C_ASYNC_CLKS = 0
394  PARAMETER C_IMPL_STYLE = 1
395  PARAMETER C_USE_CONTROL = 0
396  PARAMETER C_FSL_DWIDTH = 32
397  PARAMETER C_FSL_DEPTH = 1024
398  PORT FSL_Clk = sys_clk_s
399  PORT SYS_Rst = net_design_rst
400 END
401
402 BEGIN fsl_v20
403  PARAMETER HW_VER = 2.00.a
404  PARAMETER INSTANCE = FIFO_HWN_1_Out_4
405  PARAMETER C_EXT_RESET_HIGH = 0
406  PARAMETER C_ASYNC_CLKS = 0
407  PARAMETER C_IMPL_STYLE = 1
408  PARAMETER C_USE_CONTROL = 0
409  PARAMETER C_FSL_DWIDTH = 32
410  PARAMETER C_FSL_DEPTH = 1024
411  PORT FSL_Clk = sys_clk_s
412  PORT SYS_Rst = net_design_rst
413 END
414
415 BEGIN fsl_v20
416  PARAMETER HW_VER = 2.00.a
417  PARAMETER INSTANCE = FIFO_HWN_1_Out_5
418  PARAMETER C_EXT_RESET_HIGH = 0
419  PARAMETER C_ASYNC_CLKS = 0
420  PARAMETER C_IMPL_STYLE = 1
421  PARAMETER C_USE_CONTROL = 0
422  PARAMETER C_FSL_DWIDTH = 32
423  PARAMETER C_FSL_DEPTH = 1024
424  PORT FSL_Clk = sys_clk_s
425  PORT SYS_Rst = net_design_rst
426 END
427
428 BEGIN fsl_v20
```

```
429  PARAMETER HW_VER = 2.00.a
430  PARAMETER INSTANCE = FIFO_HWN_1_Out_6
431  PARAMETER C_EXT_RESET_HIGH = 0
432  PARAMETER C_ASYNC_CLKS = 0
433  PARAMETER C_IMPL_STYLE = 1
434  PARAMETER C_USE_CONTROL = 0
435  PARAMETER C_FSL_DWIDTH = 32
436  PARAMETER C_FSL_DEPTH = 1024
437  PORT FSL_Clk = sys_clk_s
438  PORT SYS_Rst = net_design_rst
439 END
440
441 BEGIN fsl_v20
442  PARAMETER HW_VER = 2.00.a
443  PARAMETER INSTANCE = FIFO_HWN_1_Out_7
444  PARAMETER C_EXT_RESET_HIGH = 0
445  PARAMETER C_ASYNC_CLKS = 0
446  PARAMETER C_IMPL_STYLE = 1
447  PARAMETER C_USE_CONTROL = 0
448  PARAMETER C_FSL_DWIDTH = 32
449  PARAMETER C_FSL_DEPTH = 1024
450  PORT FSL_Clk = sys_clk_s
451  PORT SYS_Rst = net_design_rst
452 END
453
454 BEGIN fsl_v20
455  PARAMETER HW_VER = 2.00.a
456  PARAMETER INSTANCE = FIFO_HWN_1_Out_8
457  PARAMETER C_EXT_RESET_HIGH = 0
458  PARAMETER C_ASYNC_CLKS = 0
459  PARAMETER C_IMPL_STYLE = 1
460  PARAMETER C_USE_CONTROL = 0
461  PARAMETER C_FSL_DWIDTH = 32
462  PARAMETER C_FSL_DEPTH = 1024
463  PORT FSL_Clk = sys_clk_s
464  PORT SYS_Rst = net_design_rst
465 END
466
467 BEGIN fsl_v20
468  PARAMETER HW_VER = 2.00.a
469  PARAMETER INSTANCE = FIFO_HWN_1_Out_9
470  PARAMETER C_EXT_RESET_HIGH = 0
471  PARAMETER C_ASYNC_CLKS = 0
472  PARAMETER C_IMPL_STYLE = 1
473  PARAMETER C_USE_CONTROL = 0
474  PARAMETER C_FSL_DWIDTH = 32
475  PARAMETER C_FSL_DEPTH = 1024
476  PORT FSL_Clk = sys_clk_s
477  PORT SYS_Rst = net_design_rst
478 END
479
480 BEGIN fsl_v20
481  PARAMETER HW_VER = 2.00.a
482  PARAMETER INSTANCE = FIFO_HWN_1_Out_10
483  PARAMETER C_EXT_RESET_HIGH = 0
484  PARAMETER C_ASYNC_CLKS = 0
485  PARAMETER C_IMPL_STYLE = 1
486  PARAMETER C_USE_CONTROL = 0
487  PARAMETER C_FSL_DWIDTH = 32
488  PARAMETER C_FSL_DEPTH = 1024
489  PORT FSL_Clk = sys_clk_s
490  PORT SYS_Rst = net_design_rst
491 END
492
493 BEGIN fsl_v20
494  PARAMETER HW_VER = 2.00.a
495  PARAMETER INSTANCE = FIFO_HWN_1_Out_11
496  PARAMETER C_EXT_RESET_HIGH = 0
497  PARAMETER C_ASYNC_CLKS = 0
498  PARAMETER C_IMPL_STYLE = 1
499  PARAMETER C_USE_CONTROL = 0
500  PARAMETER C_FSL_DWIDTH = 32
501  PARAMETER C_FSL_DEPTH = 1024
502  PORT FSL_Clk = sys_clk_s
503  PORT SYS_Rst = net_design_rst
504 END
505
506 BEGIN fsl_v20
507  PARAMETER HW_VER = 2.00.a
508  PARAMETER INSTANCE = FIFO_HWN_1_Out_12
509  PARAMETER C_EXT_RESET_HIGH = 0
510  PARAMETER C_ASYNC_CLKS = 0
511  PARAMETER C_IMPL_STYLE = 1
512  PARAMETER C_USE_CONTROL = 0
513  PARAMETER C_FSL_DWIDTH = 32
514  PARAMETER C_FSL_DEPTH = 1024
515  PORT FSL_Clk = sys_clk_s
516  PORT SYS_Rst = net_design_rst
517 END
518
519 BEGIN fsl_v20
520  PARAMETER HW_VER = 2.00.a
521  PARAMETER INSTANCE = FIFO_HWN_2_Out_13
522  PARAMETER C_EXT_RESET_HIGH = 0
523  PARAMETER C_ASYNC_CLKS = 0
524  PARAMETER C_IMPL_STYLE = 1
525  PARAMETER C_USE_CONTROL = 0
526  PARAMETER C_FSL_DWIDTH = 32
527  PARAMETER C_FSL_DEPTH = 1024
528  PORT FSL_Clk = sys_clk_s
529  PORT SYS_Rst = net_design_rst
530 END
531
532 BEGIN fsl_v20
533  PARAMETER HW_VER = 2.00.a
534  PARAMETER INSTANCE = FIFO_HWN_3_Out_14
535  PARAMETER C_EXT_RESET_HIGH = 0
536  PARAMETER C_ASYNC_CLKS = 0
537  PARAMETER C_IMPL_STYLE = 1
538  PARAMETER C_USE_CONTROL = 0
539  PARAMETER C_FSL_DWIDTH = 32
540  PARAMETER C_FSL_DEPTH = 1024
541  PORT FSL_Clk = sys_clk_s
542  PORT SYS_Rst = net_design_rst
543 END
544
545 BEGIN fsl_v20
546  PARAMETER HW_VER = 2.00.a
547  PARAMETER INSTANCE = FIFO_HWN_4_Out_15
548  PARAMETER C_EXT_RESET_HIGH = 0
549  PARAMETER C_ASYNC_CLKS = 0
550  PARAMETER C_IMPL_STYLE = 1
551  PARAMETER C_USE_CONTROL = 0
552  PARAMETER C_FSL_DWIDTH = 32
553  PARAMETER C_FSL_DEPTH = 1024
554  PORT FSL_Clk = sys_clk_s
555  PORT SYS_Rst = net_design_rst
556 END
557
558 BEGIN bram_block
559  PARAMETER INSTANCE = BRAM1_MB_1
560  PARAMETER HW_VER = 1.00.a
561  BUS_INTERFACE PORTA = BUS_DCTRL_BRAM1_MB_1
562  BUS_INTERFACE PORTB = BUS_PCTRL_BRAM1_MB_1
563 END
564
565 BEGIN lmb_bram_if_cntlr
566  PARAMETER INSTANCE = DCTRL_BRAM1_MB_1
567  PARAMETER HW_VER = 1.00.b
568  PARAMETER C_MASK = 0xff000000
569  PARAMETER C_BASEADDR = 0x00000000
570  PARAMETER C_HIGHADDR = 0x00003fff
571  BUS_INTERFACE SLMB = DBUS_MB_1
572  BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM1_MB_1
573 END
574
575 BEGIN lmb_bram_if_cntlr
576  PARAMETER INSTANCE = PCTRL_BRAM1_MB_1
577  PARAMETER HW_VER = 1.00.b
578  PARAMETER C_MASK = 0xff000000
579  PARAMETER C_BASEADDR = 0x00000000
580  PARAMETER C_HIGHADDR = 0x00003fff
581  BUS_INTERFACE SLMB = PBUS_MB_1
582  BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM1_MB_1
583 END
584
585 BEGIN bram_block
586  PARAMETER INSTANCE = BRAM1_MB_2
587  PARAMETER HW_VER = 1.00.a
588  BUS_INTERFACE PORTA = BUS_DCTRL_BRAM1_MB_2
589  BUS_INTERFACE PORTB = BUS_PCTRL_BRAM1_MB_2
590 END
591
592 BEGIN lmb_bram_if_cntlr
593  PARAMETER INSTANCE = DCTRL_BRAM1_MB_2
594  PARAMETER HW_VER = 1.00.b
```

```
595   PARAMETER C_MASK = 0xff000000
596   PARAMETER C_BASEADDR = 0x00000000
597   PARAMETER C_HIGHADDR = 0x00003fff
598   BUS_INTERFACE SLMB = DBUS_MB_2
599   BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM1_MB_2
600  END
601
602  BEGIN lmb_bram_if_cntlr
603   PARAMETER INSTANCE = PCTRL_BRAM1_MB_2
604   PARAMETER HW_VER = 1.00.b
605   PARAMETER C_MASK = 0xff000000
606   PARAMETER C_BASEADDR = 0x00000000
607   PARAMETER C_HIGHADDR = 0x00003fff
608   BUS_INTERFACE SLMB = PBUS_MB_2
609   BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM1_MB_2
610  END
```

Appendix C

The main code of the software program in the host processor

```
void FPGA::MJPEG()

{

    UINT bank_6 = 5*bankSize;
    UINT bank_5 = 4*bankSize;
    UINT bank_4 = 3*bankSize;
    UINT bank_3 = 2*bankSize;
    UINT bank_2 = 1*bankSize;
    UINT bank_1 = 0;

    int par1=450;
    int par2=275;

    // Initialization
    system("convert car_gray.jpg -interlace partition RGB:car");
    fh1 = mropen("car.B");

    for (int n=0; n<6*bankSize; n++) {
        rambuf[n] = 0;
    }

    for ( n=0; n<450*275; n++) {
        rambuf[n] = (DWORD)bgetc(fh1);
    }

    mclose(fh1);


    printf("change to initialise memory mode\n");

    fpgaSpace[COMMAND_REG] = cmd_Initialize; // initialise memory mode + access to banks to host
    fpgaSpace[COMMAND_REG];

    status = writeSSRAM(rambuf , 0, 450*275, dma);
    status = writeSSRAM(rambuf+bankSize , bankSize, bankSize, dma);
    status = writeSSRAM(rambuf+2*bankSize , 2*bankSize, bankSize, dma);
    status = writeSSRAM(rambuf+3*bankSize , 3*bankSize, bankSize, dma);
    status = writeSSRAM(rambuf+4*bankSize , 4*bankSize, bankSize, dma);
    status = writeSSRAM(rambuf+5*bankSize , 5*bankSize, bankSize, dma);

    if (status != ADMXRC2_SUCCESS) {
        printf("exiting\n");
        exit(0);
    }


//--- Load first parameter ---
    fpgaSpace[PARAM_REG]   = par1;
```

```
50       fpgaSpace[COMMAND_REG] = cmd_LoadPar; // set the strobe
51       fpgaSpace[COMMAND_REG] = 0x0;         // clear the strobe
52
53   //--- Load second parameter ---
54       fpgaSpace[PARAM_REG]   = par2;
55       fpgaSpace[COMMAND_REG] = cmd_LoadPar; // set the strobe
56       fpgaSpace[COMMAND_REG] = 0x0;         // clear the strobe
57
58   //------------------------------------------------
59
60
61       printf("\nchange to execute mode\n\n");
62
63       fpgaSpace[COMMAND_REG] = cmd_Execute; // execute mode + access to banks to design
64       fpgaSpace[COMMAND_REG];
65
66       while(1){
67           temp = fpgaSpace[STATUS_REG];
68           if (temp == stat_Finished) break;
69       }
70
71
72       // read the packet from Bank5 of the FPGA board
73
74       printf("\nchange to read memory mode\n\n");
75       fpgaSpace[COMMAND_REG] = cmd_Read; // read memory mode + access to banks to host
76       fpgaSpace[COMMAND_REG];
77
78
79       status = readSSRAM(rambuf + bankSize,  bank_2, 448*273, dma);
80
81
82       if (status != ADMXRC2_SUCCESS) {
83         printf("Error: failed to read SSRAM\n");
84         exit(1);
85       }
86
87
88       // Store the jpeg image
89
90       fh4 = mwopen("car_sobel.raw");
91
92       for (int k = 0; k < 448*273; k++) {
93
94         bputc(rambuf[bankSize + k],fh4);
95
96       }
97
98       mclose(fh4);
99
100      system("convert -depth 8 -size 448x273 gray:car_sobel.raw car_sobel.jpg");
101
102      return;
103
104  }
```

Bibliography

[1] Hristo Nikolov, Todor Stefanov, and Ed F. Deprettere. Multi-processor System Design with ESPAM. In *4th IEEE/ACM/IFIP Int. Conf. on HW/SW Codesign and System Synthesis (CODES-ISSS'06)*, Seoul, Korea, October 22-25 2006.

[2] Hristo Nikolov, Todor Stefanov, and Ed F. Deprettere. Efficient Automated Synthesis, Programming, and Implementation of Multi-processor Platforms on FPGA Chips. In *16th Int. Conference on Field Programmable Logic and Applications (FPL'06)*, Seoul, Korea, October 22-25 2006.

[3] 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)*, Madrid, Spain, August 28-30 2006.

[4] 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.

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

[6] 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.

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

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

[9] Platform studio user guide: Xilinx, inc. http://www.xilinx.com/ise/embedded/edk7_1docs/ps_ug.pdf.

[10] Microblaze software reference guide: Xilinx, inc. http://www.xilinx.com/ise/embedded/edk7_1docs/mb_ref_guide.pdf.

[11] Connecting customized ip to the microblaze soft processor uing the fast simplex link(fsl) channel: Xilinx, inc. http://www.xilinx.com/bvdocs/appnotes/xapp529.pdf.

[12] Wei Zhong. Embedded system-level platform synthesis and application mapping for heterogeneous and hierarchical multiprocessor systems. Technical report, Leiden University, The Netherlands, May 2006. Internal Report 06-05.

[13] 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.

[14] Damien Lyonnard, Sungjoo Yoo, Amer Baghdadi, and Ahmed A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *in Proc. 38th Design Automation Conference (DAC2001)*, Las Vegas, USA, June 18-22 2001.

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

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

[17] Platform specification format reference manual: Xilinx, inc. http://www.xilinx.com/ise/embedded/edk7_1docs/psf_rm.pdf.

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

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

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

[21] 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.

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

[23] 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.

[24] A. Turjan, B. Kienhuis, and E. Deprettere. Realizations of the extended linearization model in the compaan tool chain. In *Proc. of the 2nd Int. Workshop on Systems, Architectures, Modeling, and Simulation, (SAMOS 2002)*, Samos, Greece, July 2002.