EFFICIENT AUTOMATED SYNTHESIS, PROGRAMING, AND IMPLEMENTATION OF MULTI-PROCESSOR PLATFORMS ON FPGA CHIPS

Hristo Nikolov *Todor Stefanov* *Ed Deprettere*

Leiden Embedded Research Center
Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands
email: {nikolov, stefanov, edd}@liacs.nl

ABSTRACT

Emerging embedded System-on-Chip (SoC) platforms are increasingly becoming multiprocessor architectures. The advances in the FPGA chip technology make the implementation of such architectures in a single chip feasible and very appealing. Although the FPGA chip technology is well developed by companies such as Xilinx and Altera, the concepts and the necessary tool support for building multiprocessor systems on a single FPGA chip are still not mature enough. As a consequence, system designers experience significant difficulties in 1) designing multiprocessor systems on FPGAs in a short amount of time and 2) programming such systems in order to satisfy the performance needs of applications executed on them.

In this paper we present our concept for multiprocessor system design, programming, and implementation that addresses and solves the above two problems in a particular way. We have implemented the concept in a tool called ESPAM *which is briefly introduced as well. Also, we present some results obtained by applying our concept and* ESPAM *tool to automatically generate multiprocessor systems that execute a real-life application, namely a Motion-JPEG encoder.*

1. INTRODUCTION

The complexity of embedded multimedia and signal processing applications has reached a point where the performance requirements of these applications can no longer be supported by embedded system platforms based on a single processor. Therefore, modern embedded system platforms have to be multiprocessor architectures. Fortunately, the FPGA chip technology available today allows to build such multiprocessor architectures in a single chip. As examples we can mention the VirtexII-Pro and StratixII families of chips developed by the two leading FPGA companies Xilinx and Altera, respectively. In the recent years a lot of attention has been paid by these companies to integrate enormous amount of hardware resources in these chips. However, not sufficient attention has been paid to the development of system-level concepts, methodologies, and tools that utilize these resources for efficient design, programming, and implementation of high-performance multiprocessor systems. For example, the most developed tools for processor-based systems on FPGAs are the Embedded Development Kit (EDK) [1] for Xilinx chips and the System On a Programmable Chip (SOPC) builder [2] integrated in the Quartus II software for Altera chips. One can hardly find any multiprocessor design concept and methodology behind these state of the art tools. Moreover, the tools mainly support multiprocessor system design based on a shared bus and memory communication between processors that limits significantly the performance that can be achieved. Below we clarify our statements by discussing the weak points of the Xilinx' EDK. Similar weak points can be found in the SOPC builder of Altera.

The Xilinx' Embedded Development Kit (EDK) consists of a library of IP cores and some tools that aim at helping the designers in building single or multiprocessor systems based on MicroBlaze and/or PowerPC microprocessor. However, the EDK tools require input system specifications which are so detailed that designing a single processor system is still error-prone and time consuming, let alone alternative multiprocessor systems. Moreover, the EDK IP library supports very limited communication structures for connecting processors, e.g, PowerPCs can be connected only via slow PLB buses and shared memories, MicroBlazes can be connected only via FSL links, PowerPCs and MicroBlazes can not be connected easily and efficiently to form a heterogeneous system. These structures do not allow to build high-performance multiprocessor systems. The design process even gets significantly difficult for a designer when it comes to the programming of a multiprocessor system. The EDK tools does not support at all the partitioning of an application into concurrent tasks in order to allow efficient application mapping onto several processors. This is the most important and difficult procedure for efficient programming that currently is performed by hand. The EDK software environment provides only a compiler for each processor type that can be used after the partitioning. Also, for the programming, there is no support concerning the logical inter-processor data communication and synchronization even for systems with shared bus architectures. The hardware bus arbitration guarantees only that a processor can access the resources connected to the bus but can not schedule the accesses of different processors which is application dependent.

1.1. Paper Contributions

In this paper we address the problems stated above by presenting our concept for multiprocessor system design, programming and implementation on FPGAs. We have devised, developed, and implemented a general approach to connect and synchronize programmable processors of arbitrary types via several communication components which is the main contribution of the paper. We have carefully identified and developed a set of computation and communication components which are used to build a multiprocessor system. Currently, our communication components are several types of shared buses, a crossbar switch, and a point-to-point network. Regardless of the type of the communication component, our concept uses an inter-processor communication and synchronization mechanism that involves distributed communication memories with First-In-First-Out (FIFO) organization. Since no shared memories are used, the synchronization in the platform is implemented in a very simple and efficient way by blocking read/write operations on empty/full FIFO buffers in the communication memory.

We have also developed a methodology how to program multiprocessor systems (or how to map applications onto multiprocessor systems) in a systematic and automated way which is another important contribution of this paper. In our approach we use the Kahn Process Network (KPN) model of computation to specify an application as a composition of concurrent tasks. We have chosen the KPN model because of its simple communication and synchronization mechanism. In this paper we show that carefully exploiting and efficiently implementing the simple communication and synchronization mechanism of a KPN allows an efficient, systematic, and automated programming of multiprocessor systems.

We have develop a tool called ESPAM (Embedded System-level Platform synthesis and Application Mapping) that implements our concept and methodology. More specifically, ESPAM allows a system designer to specify a multiprocessor system at a high level of abstraction in a short amount of time, say a few minutes. Then ESPAM refines this specification to a real implementation in a systematic and automated way. This reduces the design time from months to hours which we consider as another contribution of our work.

1.2. Related Work

A recent work describing an exploration framework for building efficient FPGA multiprocessor systems for dataflow and stream oriented applications is presented in [3, 4]. This framework explores architectures and allocates application tasks to maximize throughput. The architecture topologies are limited to a network of $MicroBlaze$ processors interconnected using buses (the slow On-chip Peripheral Bus – OPB) and direct FSL links. This work is related to our work in the sense that we also target FPGA multiprocessor systems for dataflow and stream oriented applications using $MicroBlaze$ processors. However, we have developed a different concept of how to connect processors into a homogeneous or heterogeneous multiprocessor system. Our concept relies on communication controllers and memories for communication and synchronization between processors that allow to connect not only $MicroBlaze$ processors using buses and FSL links but also $MicroBlaze$ and/or $PowerPC$ processors connected in a point-to-point network or via a crossbar. In addition to that our concept is fully implemented, i.e., our ESPAM tool generates automatically a synthesizable (RTL) specification of a multiprocessor system along with the program code executed on each processor. In the work [3, 4] mentioned above, the authors do not discuss if they generate automatically RTL-synthesizable multiprocessor systems and how the systems are programmed.

FPGA companies such as Xilinx, Altera, etc. provide approaches and design tools attempting to facilitate efficient implementations of single or multiprocessor systems on an FPGA. Recent survey of multiprocessor solutions with FPGAs [5] shows that these state-of-the-art approaches and tools support only processor-coprocessor systems and shared memory bus-based multiprocessor systems. In addition to that, our concept for multiprocessor system design allows an automated implementation and programming of multiprocessor systems which is general enough to support different communication components (not only shared bus). Moreover, our design methodology raises the design focus to a higher system level of abstraction that reduces the design time significantly.

2. SYNTHESIS OF MULTIPROCESSOR PLATFORMS

In order to support systematic and automated synthesis of multiprocessor platforms we have carefully identified and developed a set of computation and communication components. In this section we give a detailed description of our concept for building a multiprocessor platform using our components. The components are grouped into:

- *Processing Components*: Currently, we use the Xilinx VirtexII-Pro FPGA technology and thus we support only $MicroBlaze$ and $PowerPC$ programmable processors.
- *Memory Components*: We use the dual-port memory blocks embedded in the VirtexII-Pro chips to implement processors' local program and data memories as well as data communication storages (buffers) between processors. Further on we will call the data communication storages ”Communication Memories”.
- *Communication Controller*: The communication controller component implements an interface between processing, memory, and communication components.
- *Communication Components*: We have developed the following components: a point-to-point network, a crossbar switch, and a shared bus with several arbitration schemes. These components determine the communication network topology of a multiprocessor platform.

Using the components described above a system designer can construct many alternative platform instances easily, simply by connecting processing, memory, and communication components. We have developed a general approach to connect and synchronize programmable processors of arbitrary types via a communication component. Our approach is explained below using the example instance of a multiprocessor platform depicted in Figure 1. It contains several processors

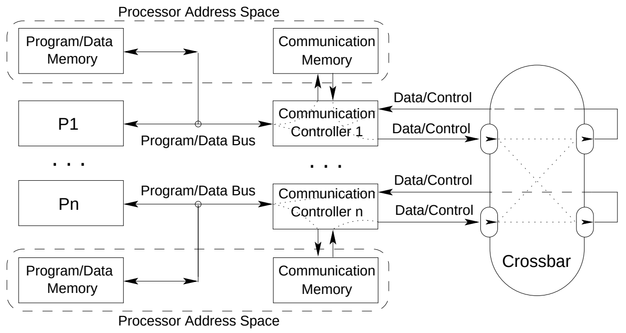


Fig. 1. Example of a Multiprocessor Platform.

connected to a communication component (in this example - a crossbar switch CB) using communication memories (CM) and communication controllers (CC). The processors transfer data between each other through the CMs. A communication controller connects a communication memory to the data bus of the processor it belongs to and to a communication component. Since any programmable processor has a data bus, processors of different types can easily be connected into a heterogeneous multiprocessor platform by using our CCs. Each CC implements the processor's local bus-based access protocol to the CM for write operations and the access to the communication component (CB) for read operations. In our approach each processor writes only to its local communication memory and uses the communication component only to read data from all other communication memories. Thus, memory contention is avoided. The CMs are organized as one or more FIFO buffers. We have chosen such organization because the inter-processor synchronization in the platform can be implemented in a very simple and efficient way by blocking read/write operations on empty/full FIFO buffers located in the communication memory.

2.1. Processing Components

Our processing components include the $MicroBlaze$ (MB) softcore processor and the $PowerPC$ (PPC) hardcore processor. These are the two processors supported by Xilinx. The VirtexII-Pro FPGAs can have up to 4 $PowerPCs$ integrated in the chip and thus our multiprocessor systems can contain up to 4 $PowerPC$ processors. The number of the $MicroBlaze$ processors is limmited only by the programmable resources available on the chip and it depends on the size of the targeted FPGA.

2.2. Memory Components

We implement the program, data, and communication memories of a processor by using the dedicated dual-port memory blocks (BRAM) of the VirtexII-Pro FPGA. Logically, a communication memory is organized as one or more FIFO buffers. A FIFO buffer is seen by a processor as two memory locations in its address space. A processor uses the first location to read/write data from/to the FIFO buffer, thereby realizing inter-processor data transfer. The second location is used to read the status of the FIFO. The status indicates whether a FIFO is full (data cannot be written) or whether a FIFO is empty (data is not available). This information is used for the inter-processor synchronization. The FIFO behavior is implemented by the Communication Controller described bellow.

2.3. Communication Controller

The structure of the Communication Controller (CC) is shown in Figure 2. It consists of two blocks, namely Interface Unit and FIFOs Unit. The Interface Unit contains an address decoder, fifos control logic, and logic to generate read requests to the communication component.

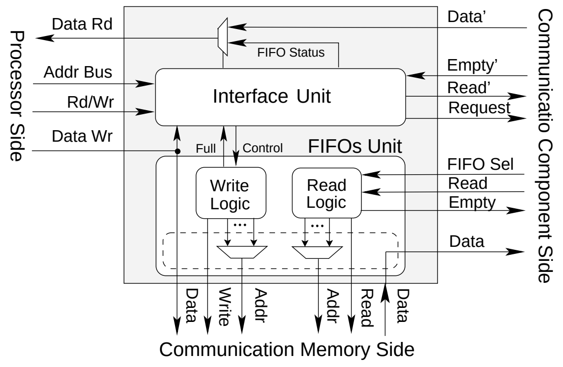


Fig. 2. Communication Controller.

When a processor has to write data to its local Communication Memory (CM), first it checks if there is room in the corresponding FIFO by reading its status. If the FIFO is full, the processor blocks. Otherwise, it sends the data to the CC. The Interface Unit decodes the FIFO address sent by the processor along with the data and generates control signals (select FIFO, write data, or read status) to the write logic of the FIFOs Unit. The latter implements the FIFO behavior. For each FIFO buffer the FIFOs Unit contains read and write counters that indicate the read and write positions in the buffer. By observing the values of the counters the FIFO empty/full status is generated. Since the FIFO buffers are implemented with dual-port BRAMs, the values of the corresponding counters are used as read and write addresses for these BRAMs. The FIFOs Unit also includes an interface logic (BRAM interface) that realizes the access to the Communication Memory (CM) connected to the CC. When a FIFO is accessed, the BRAM interface logic generates the read/write strobes and propagate the data bus and the corresponding read/write FIFO addresses to the BRAM memory (see the bottom part of Figure 2). Note that since we use dual-port BRAMs and having separate read and write logic, a FIFO can be accessed for read and write operations simultaneously by different processors. No more than two FIFOs in a CM can be accessed at a time – one for read operation and one for write operation. We do not consider this as a limitation of our implementation because our processors can not access more than one data memory locations at a time due to the processors' sequential execution of input and/or output data instructions. We implemented the CC as a generic component and by using parameters we specify how many FIFO buffers are realized by the CC and what is the size of each FIFO buffer. In this way we obtain an optimal BRAM utilization because several FIFO buffers (if they are small enough) are placed within one BRAM block. If the total buffers size is larger than the size of a BRAM, the CM is implemented with several BRAMs.

Recall that a processor can access FIFOs located in other processors' CMs via a communication component for read operations only. First, the processor checks if there is any data in the FIFO the processor wants to read. When a processor checks for data, the Interface Unit sends a request to the communication component for granting a connection to the CM in which the FIFO is located. A connection is granted only if a communication line is available and there is data in the FIFO. If a connection is not granted, the processor blocks until a connection is granted. When a connection is granted the CC connects the data bus of the communication component (the upper part of the 'communication component side' in Figure 2) to the data bus of the processor and the processor has a direct access to the CM in which the FIFO is located. The data (one or many 32-bit words) is transferred without any additional delay (reading a 32-bit word from a BRAM requires only 2 clock cycles). After the data is read the connection has to be released. This allows other processors to access the same CM. When data is read from a FIFO of a CM, the signals to the read logic of the FIFOs Unit (FIFO Sel and Read) are generated by the communication component (the bottom part of the 'communication component side' in Figure 2) as a response of a request from another CC. The BRAM interface logic generates the address and the read strobe to the BRAM and propagate the read data bus to the communication component interface.

The described blocking mechanism when accessing the CMs has to be done in the processors. The blocking can be realized in hardware (usually processors have dedicated embedded hardware to stall the processor) or in software by executing empty loops. We use the latter approach because it is more general. Different processors are stalled in hardware in different ways and therefore our CC has to be aware of many possibilities. This will result in a more complex and less generic controller. Realizing the blocking mechanism in software makes the controller more generic, thereby simplifying the integration of different types of processors into a multiprocessor system (not only $MicroBlaze$ and $PowerPC$).

2.4. Crossbar Communication Component

In this section we present the implementation of our Crossbar communication component. Our general approach to connect processors that communicate data through communication memories (CM) with FIFO organization allows the crossbar structure to be very simple. This results in a smaller crossbar with reduced number of communication and routing resources and thus reducing the design area and power consumption. The structure of our crossbar component is depicted in Figure 3. It consists of two main parts, crossbar switch (CBS) and crossbar controller (CBC).

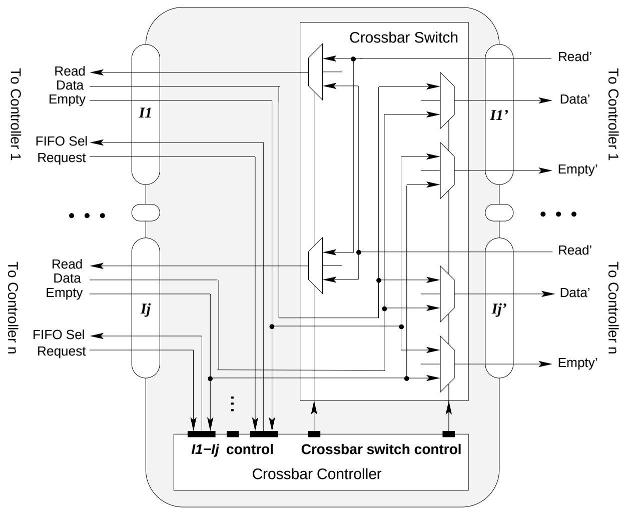


Fig. 3. Crossbar Component.

The CBS implements uni-directional connections between communication memories and processors – recall that a processor uses a communication component only to read data. We use multiplexers for the CBS implementation instead of building tri-state busses. This is because the latest FPGA technology relies on multiplexers to make on-chip buses, rather than on-chip tri-states. Each processor and its local CM are connected to a communication controller (CC) as shown in Figure 1. The CC is connected to the crossbar using one I and one I' interface depicted in Figure 3. For example, processor P1 and its local communication memory CM1 are connected to CC1 which is connected to $I1$ and $I1'$. Processor P1 can read data from all CMs in the system through interface $I1'$. Interface $I1$ connects CM1 to the crossbar and through $I1$ every processor can access this CM1. Due to the uni-directional communications and the FIFO organization of CMs, the number of signals and busses that has to be switched by our crossbar are reduced a lot. Since the addresses for accessing CMs are generated locally by the CCs, address busses are not switched through the crossbar. The crossbar switches only 32-bit data buses in one direction only and two control signals per bus. These control signals are the Read strobe and the Empty status flag for a FIFO.

There are two additional control busses within each I interfaces, namely 'Request' and 'FIFO Sel' as shown in the left part of Figure 3. The 'Request' buses are used by the CCs to generate requests for granting a connection as described earlier and also for releasing a connection after the data is read. The requests are processed by the crossbar controller (CBC) using Round-Robin policy. If a request is for granting a connection, the CBC checks its request table whether the required connection is available at the moment. The request table contains information about the status (available or not available) of all connections. The table is updated each time a connection is granted or released. If the requested connection is not available at the moment, the CBC suspends the current request and proceeds with the next one. If the requested connection is available, the CBC puts the address of the FIFO to be accessed on the FIFO Sel bus of the proper crossbar interface I. Then, the controller checks the FIFO status by reading the Empty signal of the same interface I. We call this stage accepting a request. If there is data in the FIFO, the CBC grants the connection by switching properly the CBS, updates its request table, and proceeds with the next request. If there is no data available the CBS suspends the current request and proceeds with the next one. When the request is for releasing a connection, the CBC just updates the request table and proceeds with the next request.

2.5. Shared Bus Communication Component

We have developed a shared bus communication component with several arbitration policies, namely Round-Robin, Fixed Priority, and Time Division Multiple Access (TDMA). In general, in a shared bus multiprocessor architecture a bus arbiter grants the bus to only one processor at a certain point of time. To implement such shared bus arbiter we have modified our Round-Robin crossbar controller (CBC): 1) the request table format has been limited to allow the granting of only one request for connection at a time; 2) Fixed Priority and TDMA policies for processing multiple requests were added. To implement the bus we have to connect and switch multiple wires using multiplexers or tri-state buffers. In general, using multiplexers there is no difference in the implementation of the switch logic of a crossbar compared to the switch logic of a bus. Therefore, for the switch logic of our bus we use the CBS of our crossbar component. By developing a shared bus component we show that our approach to connect and synchronize programmable processors of arbitrary types via a communication component is general enough and applies not only to a crossbar component.

2.6. Point-to-Point Network

In this section we describe how we implement a point-to-point communication in our platforms. Point-to-point means that processors in a multiprocessor platform have direct connections between each other. The number of direct connections and the topology of the point-to-point network depend on the applications to be mapped. Since there is no communication component such as a crossbar or a bus, there are no requests for granting connections thus no additional communication delay is introduced in the platform. Because of this, the highest possible communication performance can be achieved in a multiprocessor platform. We use the example in Figure 4 to explain how we build point-to-point multiprocessor platforms with $MicroBlaze$ and $PowerPC$ processors. The example platform contains two $MicroBlaze$ (MB1 and MB2) and one $PowerPC$ (PPC) processors. In compliance with

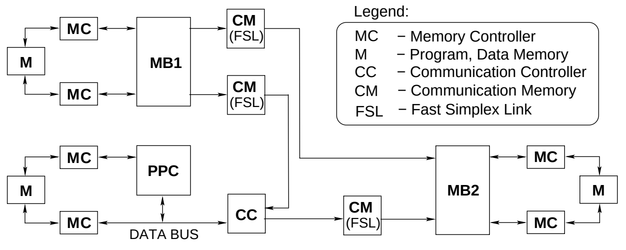


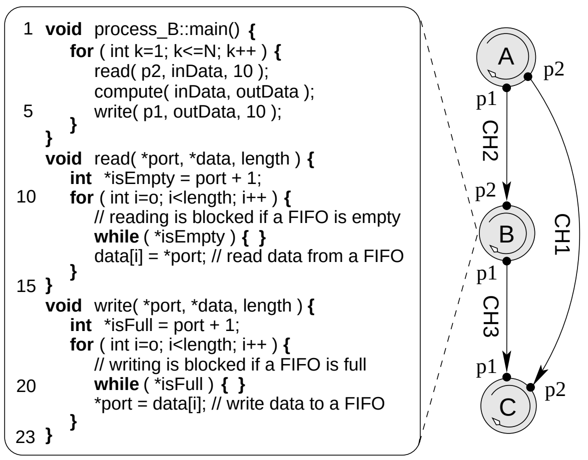
Fig. 4. Point-to-Point Architecture.

our concept for building multiprocessor platforms described earlier the processors communicate data through CMs with FIFO organization. In case of point-to-point connections each CM is nothing more than a single FIFO component. Thus, we implement the CMs by using a FIFO IP core called Fast Simplex Link (FSL) provided by Xilinx. The CMs (FSLs) are connected directly to the read and write FSL ports of the $Microblaze$ processors MB1 and MB2 – see Figure 4. Each $MicroBlaze$ processor has 8 embedded dedicated very fast read/write FSL interface ports. The $PowerPC$ processor does not have FSL ports. Therefore, to connect one or more FSLs to a $PowerPC$ processor we use a very simplified version of our communication controller (CC) described in Section 2.3. The simplified CC only translates the processor data bus signals to FSL input/output signals. It is parameterized and it supports up to 128 FSLs for read and write operations. A simplified CC is used with $MicroBlaze$ processors as well. If there are more than 8 FSLs to be connected to a $MicroBlaze$ processor, then 8 of them are connected directly to FSL ports and the rest are connected to the $MicroBlaze$ data bus through a CC.

3. PROGRAMING OF MULTIPROCESSOR PLATFORMS

In our methodology for programing a multiprocessor platform the main step is the partitioning of an application into concurrent tasks where the inter-task communication and synchronization is *explicitly* specified in each task. Such partitioning can be done by hand or automatically (see next section) and it allows each task or group of tasks to be compiled separately by a standard compiler in order to generate an executable code for each processor in the platform. The partitioning of an application into concurrent tasks requires the use of a parallel model of computation (MoC) in order to specify functionally the application. We use the Kahn Process Network [6] MoC for application specification. In general, a Kahn Process Network (KPN) is a network of concurrent autonomous processes that communicate data in a point-to-point fashion over unbounded FIFO channels, using a blocking-read on an empty FIFO as synchronization mechanism. Since in our platform implementation the FIFO channels are bounded, we use a blocking-write on full FIFO as well. A simple example of a KPN is shown in the right part of Figure 5. Three processes A, B, and C are connected through FIFO channels. Each process in the network performs a sequential computation concurrently with the other processes. In the left part of the same figure we show simple example of code executed by process B. The process reads data from its input channel via port *p2* (line 3). If data is not available the process blocks on reading until data arrives. Then it performs a computation on the data (line 4), and writes the result to its output channel via port *p1* (line 5). Lines 3 to 5 are repeated several times. Lines 8 to 23 show the blocking read/write synchronization primitives. As we described in Section 2.3 we implement them by executing empty loops (lines 12 and 20).

We motivate our choice of using KPNs by observing that on the one hand a KPN specifies an application as a composition of concurrent processes where the computation, control, and memory are distributed. On the other hand, the multiprocessor platforms we consider have components that run in parallel, i.e., the computation, control, and memory are distributed over the components. Thus, the KPN parallel processing model matches our concept of building multiprocessor platforms



```
void  process_B::main() {
    for ( int k=1; k<=N; k++ ) {
        read( p2, inData, 10 );
        compute( inData, outData );
        write( p1, outData, 10 );
    }
}
void  read( *port, *data, length ) {
    int  *isEmpty = port + 1;
    for ( int i=o; i<length; i++ ) {
        // reading is blocked if a FIFO is empty
        while ( *isEmpty ) {  }
        data[i] = *port; // read data from a FIFO
    }
}
void  write( *port, *data, length ) {
    int  *isFull = port + 1;
    for ( int i=o; i<length; i++ ) {
        // writing is blocked if a FIFO is full
        while ( *isFull ) {  }
        *port = data[i]; // write data to a FIFO
    }
}
```

Fig. 5. Kahn Process Network Example.

very well and the mapping of KPN specifications onto the latter can be done in a systematic and automated way. This is achieved by exploiting the following characteristic of a KPN: 1) The control is completely distributed to the individual processes and there is no global scheduler present. As a consequence, mapping a KPN over a number of processors is a simple task; 2) The exchange of data is distributed over FIFO channels. There is no notion of a global memory that has to be accessed by multiple processes (processors). Therefore, resource contention does not occur; 3) The synchronization between the processes is by blocking read/write on FIFO channels. Such synchronization is implemented very easily and efficiently using software primitives in our multiprocessor platforms.

4. AUTOMATED MULTIPROCESSOR DESIGN

We have implemented our concept for multiprocessor platform synthesis, programing, and implementation in a tool called ESPAM. In this section we give a brief overview of our design methodology using ESPAM that is depicted as a design flow in Figure 6. ESPAM requires as input three specifications: 1) *Platform Specification* describing the topology of a platform using our components presented in Section 2; 2) *Application Specification* describing an application as a Kahn Process Network (KPN), i.e., network of concurrent processes communicating via FIFO channels; 3) *Mapping Specification* describing the relation between all processes and FIFO channels in *Application Specification* and all components in *Platform Specification*. We describe the input specifications using XML format. In our case describing a multiprocessor platform is a very simple task that can be performed in a few minutes because ESPAM requires a high-level platform specification which does not contain any details about the physical interfaces of the components. The platform specification contains only a simple topology description, i.e., processing components connected directly to each other or via a communication component. Similarly, writing the mapping specification takes a few minutes as well. The only time-consuming task is to specify an application as a KPN. However, for applications specified as parameterized static affine nested loop programs in Matlab or C the generation of KPNs is automated by the COMPAAN tool [7, 8].

Following the platform specification, ESPAM constructs a platform instance and runs a consistency check on that instance. The platform instance is an abstract model of a multiprocessor platform because at this stage no information about the target physical platform is taken into account. The model defines only the key components of the platform and their attributes. Then, ESPAM refines the abstract platform model to an elaborate (detailed) parameterized RTL model which is ready for an implementation on a target physical platform. We call this refinement process platform synthesis. Finally, ESPAM creates program code for each processor in the multiprocessor system in accordance with the application and mapping specifications.

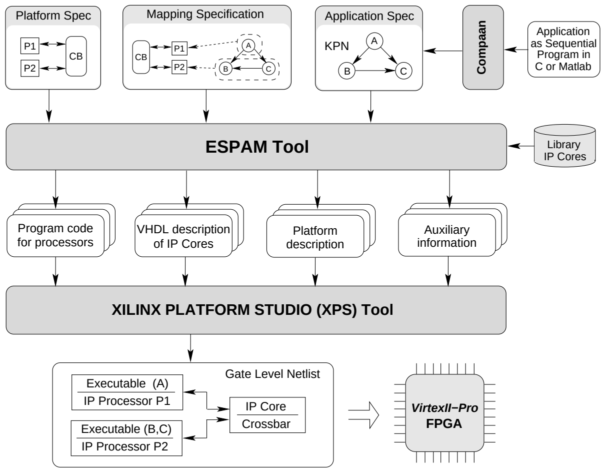


Fig. 6. ESPAM System Design Flow.

As an output ESPAM generates files that can be imported directly as a project in the Xilinx Platform Studio (XPS) [1]. The files suit of this project consists of four parts: 1) *Platform Description* defining in great detail the processors network (multiprocessor platform) in Hardware Specification File (MHS) and Software Specification File (MSS); 2) *Hardware descriptions of IP cores* containing VHDL files of IP cores used in 1). Some of them are predefined library components (processors, memories, etc.) taken from *Library IP Cores*, see Figure 6. The others are custom IP cores generated by ESPAM. They are needed as a glue/interface logic between components in the platform; 3) *Program code for processors* — to execute the application on the synthesized multiprocessor platform, ESPAM generates program source code files for each processor in the platform; 4) *Auxiliary information* containing the XPS project file and files that define precise timing requirements and prioritizing signal constraints, e.g., the User Constraints File (UCF). With the four parts described above, the XPS tool can implement the multiprocessor system at the gate level and it can generate the VirtexII-Pro bit-stream file used to configure the FPGA chip.

5. EXPERIMENTS AND RESULTS

In this section, we present some of the results we have obtained by implementing and executing a Motion JPEG (M-JPEG) encoder application onto several multiprocessor platforms using our design flow depicted in Figure 6. We use the ADM-XPL FPGA board manufactured by Alpha Data Parallel Systems Ltd [9]. The board contains one VirtexII-Pro FPGA (xc2vp20-FF896-6) which has 2 $PowerPC$ microprocessors integrated in it. We implemented multiprocessor systems with 2 $MicroBlaze$ and 2 $PowerPC$ processors connected in a point-to-point (P2P) network, through a crossbar, and through a shared bus. As described in Section 4, the inputs to our system design flow are the *Application*, *Platform*, and *Mapping Specifications*. We started with the M-JPEG application given as a sequential C program and derived the *Application Specification* (KPN) using the COMPAAN tool. For each platform, we wrote the *Platform* and *Mapping Specifications* by hand in approximately 10 minutes. This is a very simple task because our specifications are at a high level of abstraction (not RTL level). Having all three input specifications, our ESPAM tool generated the output (a multiprocessor system at RTL level) within three minutes. Then we imported the generated output files to the XPS tool for physical implementation, i.e., mapping, place, and route onto our prototyping FPGA. It took the XPS tool about an hour for the physical implementation. All tools run on a Pentium IV machine at 1.8GHz with 1GB of RAM.

5.1. Synthesis Results

In Table 1 we present the overall resource utilization of the multiprocessor systems we consider in our experiments. We also present the utilization results for the communication controllers (CC), a 4-port crossbar component (CB), and a 4-port shared bus component (BUS) respectively. The FPGA resources are grouped into slices that contain 4-Input Look-Up tables and Flip-Flops. The first three rows in the table show that the multiprocessor systems utilize around 40% of the slices in the FPGA. Also, the last three rows show that our communication component (CB or BUS) together with the CCs in each system utilize very small portion of the FPGA slices – around 5%. These numbers clearly indicate that our concept to connect processors through communication components and communication memories is very efficient in terms of slice utilization. The last column in Table 1 shows that the multiprocessor systems utilize almost 100% of the on-chip memory which is only 176KB – 88 BRAM blocks of 2KB each. This high utilization is not related to inefficiency in our concept to connect processors via communication memories because for each M-JPEG system we use a maximum of 9 BRAM blocks to implement FIFO buffers, distributed over 4 communication memories. The high BRAM utilization is due to the fact that the M-JPEG is relatively complex application. Almost all BRAM blocks are used for the program and data memory of the 4 microprocessors in our platforms (on average 38KB per processor).

Table 1. Resource Utilization (xc2vp20-FF896-6).

	#Slices	#4-Input LUT	#Flip-Flops	#BRAMs
4 Proc. Shared Bus	3640 (39%)	4722 (25%)	2354 (12%)	85 (99%)
4 Proc. Crossbar	3653 (39%)	4748 (25%)	2357 (12%)	85 (99%)
4 Proc P2P System	3263 (35%)	3929 (21%)	2405 (12%)	88 (100%)
4 CCs	288 (2%)	468 (2%)	116 (1%)	—
4 Port CB	397 (3%)	587 (3%)	56 (1%)	—
4 Port Bus	366 (3%)	541 (2%)	47 (1%)	—

Based on our experience we conclude that the main limitation of how large multiprocessor system can be build on a single FPGA chip still remains the amount of the on-chip memory. Fortunately, the main FPGA chip vendor Xilinx started to realize this limitation. Therefore, in every new Xilinx FPGA chip the amount of the on-chip memory increases. For example, the largest Virtex4-FX FPGA contains 552 BRAMs. Using the FPGA on-chip memory instead of external memories is crucial for our high-performance multiprocessor systems because external memories are slower than the on-chip BRAMs. In addition to that, the $MicroBlaze$ processor can be connected to an external memory only through the slow on-chip peripheral bus (OPB) which will reduce the performance even more if external memory is used.

5.2. Performance Results

The performance numbers presented in this section are collected by running real M-JPEG multiprocessor system implementations on our FPGA board. For all multiprocessor systems we have HW/SW demos. These systems were generated by our ESPAM tool. For each multiprocessor system we measured the exact number of clock cycles needed to process an image with size 128 by 128 pixels. These numbers, depicted in Figure 7, are taken from simple hardwired timers and counters automatically integrated by ESPAM in each system. The most left bar shows the performance of the M-JPEG application ran on a single processor as sequential program. We use this performance number for comparison with our multiprocessor systems. The achieved speedup by the shared bus multiprocessor system (the second bar) is only 1.42x as the theoretical maximum is 4x. This clearly shows that a shared bus architecture is not an efficient architecture for building high-performance multiprocessor systems. We achieved performance speedup of 2.60x for the system with a crossbar component and 3.75x for the system with point-to-point (P2P) connections. The performance difference between this system and the system with the crossbar component is due to the fast FSL links used in the point-to-point communications. The performance speedup of a P2P system implemented without FSL links is almost the same (2.75x) as the speedup of the system with the crossbar component – see the third and the fourth bar in Figure 7.

Based on the performance numbers presented above we conclude that: 1) the M-JPEG application mapped onto 4 processors with FSL point-to-point connections gives a speedup closer to the theoretical maximum (4x) compared to a single processor system; 2) for the M-JPEG application mapped onto 4 processors, communication through a crossbar does not sacrifice the performance compared to point-to-point communication without fast FSL links.

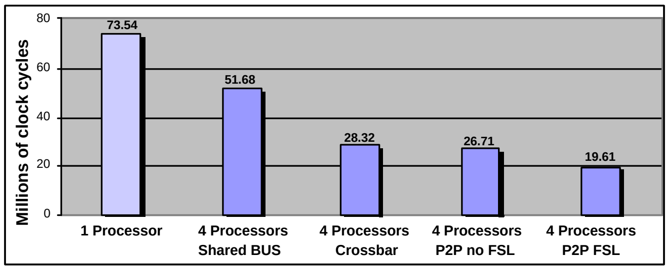


Fig. 7. Performance Results.

6. CONCLUSION

In this paper we presented our general approach (implemented in the ESPAM tool) for automated design, programming and implementation of multiprocessor systems on FPGAs. While the state-of-the-art development tools supports shared bus architectures only, our approach is general enough to be applied on multiprocessor systems with different communication topologies. Moreover, it allows these systems to be programmed in automated way which significantly reduces the design time.

The results presented in this paper show that our approach of connecting processors through communication controllers and communication memories is efficient in terms of slice utilization and performance speedup. The amount of the on-chip memory is the main limiting factor of how large multiprocessor system can be build on a single FPGA chip. For an M-JPEG encoder application mapped onto 2 $MicroBlaze$ and 2 $PowerPC$ processors the communication logic utilizes only 5% of the FPGA resources. We achieved speedup of 3.75x as the theoretical maximum is 4x. The implementation utilizes all available on-chip memory.

7. REFERENCES

[1] "Xilinx, Inc. Xilinx Platform Studio and the Embedded Development Kit, EDK version 8.1i edition." www.xilinx.com/ise/embedded_design_prod/platform_studio.htm.

[2] "Altera, Inc. Quartus II Handbook Volume 4: SOPC Builder, Dec 2005." www.altera.com/literature/quartus2/lit-qts-sopc.jsp.

[3] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An Automated Exploration Framework for FPGA-based Soft Multiprocessor Systems," in *Proc. IEEE-ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, New Jersey, USA, Sept. 19-21 2005.

[4] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer, "An FPGA-based Soft Multiprocessor System for IPv4 Packet Forwarding," in *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finland, Aug. 24-26 2005.

[5] "Multiprocessor Solutions with FPGAs," White paper, FPGA and Programmable Logic Journal, 2005, www.fpgajournal.com/whitepapers_2005/altera_20050224.htm.

[6] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

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

[8] A. Turjan, B. Kienhuis, and E. 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, Sept. 23-25 2004.

[9] "www.alpha-data.com/adm-xpl.html," Alpha Data Parallel Systems, Ltd.