The FPGAworld Conference addresses all aspects of digital and hardware/software system engineering on FPGA technology. It is a discussion and network forum for researchers and engineers working on industrial and research projects, state-of-the-art investigations, development and applications. The proceedings contain the academic presentations a separate report contain also the industrial presentations; for more information see (www.fpgaworld.com/conference).

ISBN 978-91-976844-0-8

SPONSORS


2nd FPGAworld CONFERENCE, Lennart Lindh, Vincent John Mooney III (external), MRTC report ISSN 1404-3041 ISRN MDH-MRTC-188/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, September, 2005

Copyright and Reprint Permission for personal or classroom use are allowed with credit to FPGAworld.com. For commercial or other for-profit/for-commercial-advantage uses, prior permission must be obtained from FPGAworld.com.

Additional copies of 2008 or prior Proceedings may be found at www.FPGAworld.com or at Jönköpings University library (www.jth.hj.se), ISBN 978-91-976844-0-8
2008 PROGRAM COMMITTEE

General Chair
Lennart Lindh, FPGAworld, Sweden

Publicity Chair
David Källberg, FPGAworld, Sweden

Academic Programme Chair
Vincent J. Mooney III, Georgia Institute of Technology, USA

Academic Publicity Chair
Santiago de Pablo, University of Valladolid, Spain

Academic Programme Committee Members
Ketil Roed, Bergen University College, Norway
Lennart Lindh, Jönköping University, Sweden
Pramote Kuacharoen, National Institute of Development Administration, Thailand
Mohammed Yakoob Siyal, Nanyang Technological University, USA
Fumin Zhang, Georgia Institute of Technology, USA
Santiago de Pablo, University of Valladolid, Spain
General Chairman’s Message

The FPGAworld program committee welcomes you to the 5th FPGAworld conference. This year’s conference is held in Electrum-Kista, Stockholm. We hope that the conferences provide you with much more then you expected. This year it is the fourth time we have academic reviewed papers; this is an important step to incorporate the academic community into the conference program. Due to the high quality, 5 out of the 17 papers submitted this year were presented.

We will try to balance academic and industrial presentations (Stockholm), exhibits and tutorials to provide a unique chance for our attendants to obtain knowledge from different views. This year we have the strongest program in FPGAworld’s history.

The FPGAworld 2008 conference is bigger than the FPGAworld 2007 conference. Totally we are close to 300 participants (Stockholm and Lund).

All are welcome to submit industrial/academic papers, exhibits and tutorials to the conference, both from student, academic and industrial backgrounds. Together we can make the FPGAworld conference exceed even above our best expectations!

Please check out the website (http://fpgaworld.com/conference/) for more information about FPGAworld 2008. In addition, you may contact David Källberg (david@fpgaworld.com) for more information.

We would like to thank all of the authors for submitting their papers and hope that the attendees enjoyed the FPGAworld conference 2008 and you welcome to next year’s conference in 2009 in September.

Lennart Lindh

General Program Chair
Contents

Sessions B1-B4: Methodologies for FPGA design............................................

§ A Java-Based System for FPGA Programming...........................................

§ Automated Design Approach for On-Chip Multiprocessor Systems

§ ASM++ charts: an Intuitive Circuit Representation
  Ranging from Low Level RTL to SoC Design.............................................

Sessions B5-B7: Custom Hardware Modules for FPGAs .........................

§ Space-Efficient FPGA-Implementations of FFTs
  in High-Speed Applications.................................................................

§ The ABB NoC - a Deflective Routing 2x2 Mesh NoC
  targeted for Xilinx FPGAs.................................................................
A Java-Based System for FPGA Programming

Jacob A. Bower, James Huggett, Oliver Pell and Michael J. Flynn
Maxeler Technologies
{Jacob, jhuggett, oliver, flynn}@maxeler.com

Abstract

Photon is a Java-based tool for programming FPGAs. Our objective is to bridge the gap between the ever increasing sizes of FPGAs and the tools used to program them. Photon’s primary goal is to allow rapid development of FPGA hardware. In this paper we present Photon by discussing both Photon’s abstract programming model which separates computation and data I/O, and by giving an overview of the compiler’s internal operation, including a flexible plug-and-play optimization system. We show that designs created with Photon always lead to deeply-pipelined hardware implementations, and present a case study showing how a floating-point convolution filter design can be created and automatically optimized. Our final design runs at 250MHz on a Xilinx Virtex5 FPGA and has a data processing rate of 1 gigabyte per second.

1. Introduction

Traditional HDLs such as VHDL or Verilog incur major development overheads when implementing circuits, particularly for FPGA which would support fast design cycles compared to ASIC development. While tools such as C-to-gates compilers can help, often existing software cannot be automatically transformed into high-performance FPGA designs without major re-factoring.

In order to bridge the FPGA programming gap we propose a tool called Photon. Our goal with Photon is to simplify programming FPGAs with high-performance data-centric designs.

Currently the main features of Photon can be summarized as follows:

- Development of designs using a high-level approach combining Java and an integrated expression parser.
- Designs can include an arbitrary mix of fixed and floating point arithmetic with varied precision.
- Plug-and-play optimizations enabling design tuning without disturbing algorithmic code.
- VHDL generation to enable optimizations via conventional synthesis tools.
- Automation and management of bitstream generation for FPGAs, such as invoking FPGA vendor synthesis tools and simulators.

The remainder of this paper is divided as up as follows: In Section 2, we compare Photon and other tools for creating FPGA designs. In Section 3 we describe Photon’s programming model which ensures designs often lead to high-performing FPGA implementations. In Sections 4 and 5 we give an overview of how Photon works internally and present a case study. Finally, in Section 6 we summarize our work and present our conclusions on Photon so far.

2. Comparisons to Other Work

In Table 1 we compare tools for creating FPGA designs using the following metrics:

- Design input – Programming language used to create designs.
- High level optimizations – Automatic inference and optimizing computation hardware, simplification of arithmetic expressions etc.
- Low level optimizations – Boolean expression minimisation, state-machine optimizations, eliminating unused hardware etc.
- Floating-point support – Whether the tool has intrinsic support for floating-point and IEEE compliance.
- Meta-programmability – Ability to statically meta-program with weaker features being conditional compilation and variable bit-widths and stronger features such as higher-order design generation.

VHDL and Verilog use a traditional combination of structural constructs and RTL to specify designs. These tools typically require a high development effort. Such conventional tools typically have no direct support for floating
point arithmetic and therefore require external IP. Meta-
programmability e.g. generics in VHDL are fairly inflexi-
ble [1]. The advantage of VHDL and Verilog is that they
give the developer control over every aspect of the micro-
architecture, providing the highest potential for an optimal
design. Additionally, synthesis technology is relatively ma-
ture and the low-level optimizations can be very effective.
Other tools often produce VHDL or Verilog to leverage the
low-level optimizers present in the conventional synthesis
tool-chain.

Impulse-C [2] and Handel-C [3] are examples of C-to-
gates tools aiming to enable hardware designs using lan-
guages resembling C. The advantage of this approach is
existing software code can form a basis for generating hard-
ware with features such as ROMs, RAMs and floating-
point units automatically inferred. However, software code
will typically require modifications to support a particular
C-to-gates compiler’s programming model. For example
explicitly specifying parallelism, guiding resource mapping,
and eliminating features such as recursive function
calls. The disadvantage of C-to-gates compilers is that the
level of modification or guidance required of a developer
may be large as in general it is not possible to infer a high-
performance FPGA design from a C program. This arises
as C programs in general are designed without parallelism
in mind and are highly sequential in nature. Also, meta-
programmability is often limited to the C pre-processor as
there is no other way to distinguish between static and dy-
namic program control in C.

PamDC [4], JHDL [5], YAHDL [1] and ASC [6] are
examples of Domain Specific Embedded Languages [7]
(DSELs) in which regular software code is used to imple-
ment circuit designs. With this approach all functionality
to produce hardware is encapsulated in software libraries
with no need for a special compiler. These systems are a
purely meta-programmed approach to generating hardware
with the result of executing a program being a net-list or
HDL for synthesis. Of these systems, PamDC, JHDL and
YAHDL all provide similar functions for creating hardware
structurally in C++, Java and Ruby respectively. YAHDL
and PamDC both take advantage of operator overloading
to keep designs concise, whereas JHDL designs are often
more verbose. YAHDL also provides functions for au-
tomating build processes and integrating with existing IP
and external IP generating tools. ASC is a system built
on top of PamDC and uses operator overloading to spec-
ify arithmetic computation cores with floating-point opera-
tions.

Photon is also implemented as a DSEL in Java. Photon’s
underlying hardware generation and build system is based
on YAHDL rewritten in Java to improve robustness. Un-
like JHDL, Photon minimizes verbosity by using an inte-
grated expression parser which can be invoked from regular
Java code. Photon also provides a pluggable optimization
system unlike the other DSELs presented, which generate
hardware in a purely syntax directed fashion.

### 3. Photon Programming Model

Our goal with Photon is to find a way to bridge the grow-
ing FPGA size versus programming gap when accelerat-
ing software applications. In this section we discuss the
programming model employed by Photon which provides
high-performance FPGA designs.

FPGA designs with the highest performance are gen-
erally those which implement deep, hazard free pipelines.
However, in general software code written without par-
allelism in mind tends to have loops with dependen-
cies which cannot directly be translated into hazard free
pipelines. As such, software algorithm implementations
often need to be re-factored to be amenable to a high-
performance FPGA implementation. Photon’s program-
mapping model is built around making it easy to implement
suitably re-factored algorithms.

When developing our programming model for Photon,
we observe that dense computation often involves a sin-
gle arithmetic kernel nested in one or more long running
loops. Typically, dense computation arises from repeat-

<table>
<thead>
<tr>
<th>Design input</th>
<th>Photon</th>
<th>Impulse-C</th>
<th>Handel-C</th>
<th>Verilog</th>
<th>VHDL</th>
<th>PamDC</th>
<th>JHDL</th>
<th>YAHDL</th>
<th>ASC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point</td>
<td>Java</td>
<td>C</td>
<td>C</td>
<td>Verilog</td>
<td>VHDL</td>
<td>C++</td>
<td>Java</td>
<td>Ruby</td>
<td>C++</td>
</tr>
<tr>
<td>High-level optimizations</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Low-level optimizations</td>
<td>Via VHDL</td>
<td>Via HDL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Meta-programmability</td>
<td>Strong</td>
<td>Weak</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>Strong</td>
<td>Strong</td>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>Build automation</td>
<td>Yes</td>
<td>Yes</td>
<td>Limited</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1. Comparison of tools for creating FPGA designs from software code.
ing the main kernel because the loop passes over a very large set of data, or because a small set of data is being iterated over repeatedly. Examples of these two cases include convolution, in which a DSP operation is performed on a large set of data [8], and Monte-Carlo simulations repeatedly running random-walks in financial computations [9].

In the Photon programming model we implement applications following this loop-nested kernel pattern by dividing the FPGA implementation into two separate design problems:

1. Creating an arithmetic data-path for the computation kernel.
2. Orchestrating the data I/O for this kernel.

Thus, we turn organizing data I/O for the kernel into a problem that can be tackled separately from the data-path compiler. Thus this leaves us with an arithmetic kernel which does not contain any loop structures and hence can be implemented as a loop-dependency free pipeline.

In Photon we assume the Data I/O problem is solved by Photon-external logic. Based on this assumption, Photon designs are implemented as directed acyclic graphs (DAGs) of computation. The acyclic nature of these graphs ensures a design can always be compiled to a loop-dependency free pipeline.

Within a Photon DAG there are broadly five classes of node:

- I/O nodes – Through which data flows into and out of the kernel under the control of external logic.
- Value nodes – Nodes which produce a constant value during computation. Values may be hard-coded or set via an I/O side-channel when computation is not running.
- Computation nodes – Operations including: arithmetic (+, ÷ . . .), bit-wise (&, or, . . .), type-casts etc.
- Control nodes – Flow-control and stateful elements, e.g.: muxes, counters, accumulators etc.
- Stream shifts – Pseudo operations used to infer buffering for simulating accessing to data ahead or behind the current in-flow of data.

To illustrate Photon’s usage and graph elements, consider the pseudo-code program in Listing 1. This program implements a simple 1D averaging filter passing over data in an array din with output to array dout. The data I/O for this example is trivial: data in the array din should be passed linearly into a kernel implementing the average filter which outputs linearly into an array dout.

```python
def average_filter(din, dout, N):
    for i in range(N):
        if i > 0 and i < N-1:
            dout[i] = (din[i] + din[i-1] + din[i+1]) / 3
        else:
            dout[i] = din[i]
```

Listing 1. Pseudo-code 1D averaging filter.

Figure 1 shows a Photon DAG implementing the averaging kernel from Listing 1. Exploring this graph from top-down: data flows into the graph through the din input node, from here data either goes into logic implementing an averaging computation or to a mux. The mux selects whether the current input data point should skip the averaging operation and go straight to the output as should be the case at the edges of the input data. The mux is controlled by logic which specifies whether we are at the edges of the stream. The edge of the stream is detected using a combination of predicate operators (<, >, &) and a counter which increases once for each item of data which enters the stream. The constant input N − 1 to the < comparator can be implemented as a simple constant value, meaning the size of data which can be processed is fixed at compilation time. On the other hand, the constant input can be implemented as a more advanced value-node that can be modified via a side-channel before computation begins, thus allowing data-streams of any size to be processed. The logic that performs the averaging computation contains a number of arithmetic operators, a constant and two stream-shifts. The stream-shift operators cause data to be buffered such that it arrives at the addition operator one data-point behind (−1) or one data-point ahead (+1) of the unshifted
data which comes directly from \( \text{din} \).

To implement our 1D averaging Photon DAG in hardware, the design undergoes processing to arrive at a hardware implementation. Figure 2 illustrates the result of Photon processing our original DAG. In this processed DAG, buffering implements the stream-shifting operators and ensures data input streams to DAG nodes are aligned. Clock-enable logic has also been added for data alignment purposes.

With this newly processed DAG, data arriving at \( \text{din} \) produces a result at \( \text{dout} \) after a fixed latency. This is achieved by ensuring that data inputs to all nodes are aligned with respect to each other. For example the mux before \( \text{dout} \) has three inputs: the select logic, \( \text{din} \) and the averaging logic. Without the buffering and clock-enable logic, data from \( \text{din} \) would arrive at the left input to the mux before the averaging logic has computed a result. To compensate, buffering is inserted on the left input to balance out the delay through the averaging logic. For the mux-select input a clock-enable is used to make sure the counter is started at the correct time.

After Photon processes a DAG by inserting buffering and clock-enable logic, the DAG can be turned into a structural hardware design. This process involves mapping all the nodes in the graph to pre-made fully-pipelined implementations of the represented operations and connecting the nodes together. As the design is composed of a series of fully-pipelined cores, the overall core is inherently also fully-pipelined. This means Photon cores typically offer a high degree of parallelism with good potential for achieving a high clock-speed in an FPGA implementation.

**Listing 2.** Photon floating-point add/mul design.

```java
class AddMul extends PhotonDesign {
    AddMul(BuildManager bm) {
        super(bm, "AddMulExample");
        Var a = input("a", hwFloat(8, 24));
        Var b = input("b", hwFloat(8, 24));
        Var c = input("c", hwFloat(8, 24));
        Var d = output("d", hwFloat(8, 24));
        d.connect( mul(add(a, b), c);
    }
}
```

4. Implementation of Photon

In this section we given an overview of Photon’s concrete implementation. Of particular interest in Photon is the mechanism by which designs are specified as Java programs which is covered first in Section 4.1. We then discuss Photon’s compilation and hardware generation process in Section 4.2.

4.1. Design Input

Photon is effectively a Java software library and as such, Photon designs are created by writing Java programs. Executing a program using the Photon library results in either the execution of simulation software for testing a design or an FPGA configuration programming file being generated. When using thePhoton library a new design is created by extending the \texttt{PhotonDesign} class which acts as the main library entry point. This class contains methods which wrap around the creation and inter-connection of standard Photon nodes forming a DAG in memory which Photon later uses to produce hardware. New nodes for custom hardware units, e.g. a fused multiply-accumulate unit, can also be created by users of Photon.

Listing 2 shows an example Photon program. When executed this program creates a hardware design which takes three floating-point numbers \( a, b \) and \( c \) as inputs, adds \( a \) and \( b \) together and multiplies the result by \( c \) to produce a single floating-point output \( d \). Method calls in the code specify a DAG which has six nodes: three inputs, an output, a multiplier and an add. These nodes are created by calls to the input, output, mul and add methods respectively. The input and output methods take a string parameter to specify names of I/Os for use by external logic and for performing data I/O. Another parameter specifies the I/O type. For the example in this paper, we use IEEE single precision floating-point numbers. The floating point type is declared using a call to \texttt{hwFloat} which makes a
// Create 1/0s
input("din", hwFloat(8, 24));
output("dout", hwFloat(8, 24));

// Average Computation
eval("prev_din = streamShift(-1, din)");
eval("next_din = streamShift(1, din)");
eval("avg = (prev_din + din + next_din) / 3");

// 8-bit counter
eval("count = simpleCounter(8, 255)");

// Select logic with N hard-coded to 10
eval("sel = (count > 1) & (count < 10)");

// Max connected to output
eval("dout = sel ? avg : din");

Listing 3. 1D averaging design implemented using Photon expressions.

Floating point type object with an 8 bit exponent and a 24 bit mantissa following the IEEE specification. We can also create floating-point numbers with other precisions, fixed-point and/or integer types. Types used at I/Os propagate through the DAG and hence define the types of operator nodes. Casting functions can be used to convert and constrain types further within the design.

One drawback of using Java method calls to create a DAG is verbosity, making it hard to read the code or relate lines back to the original specification. To resolve the function-call verbosity the Photon library provides a mechanism for expressing computation using a simple expression-based language. Statements in this integrated expression parser can be written using a regular Java strings passed to an eval method. The eval method uses the statements in the string to call the appropriate methods to extend the DAG.

To demonstrate our eval expressions, Listing 3 shows how our 1D averaging example from Figure 1 is implemented in Photon using eval calls.

4.2. Compilation and Hardware Generation

In addition to using Java for design specification, Photon also implements the compilation and hardware generation process entirely in Java. Photon’s design management features cover optimization of Photon designs, generation of VHDL code, and calling external programs such as synthesis, simulation, IP generation, and place-and-route.

After a Photon design is fully specified, Photon turns the specified DAG into a design which can be implemented in hardware. Photon achieves this primarily by executing a number of “graph-passes”. A graph-pass is piece of Java code which visits every node in the DAG in topological order. Typically, Photon passes transform the graph by adding and/or deleting nodes, for example to implement optimizations. Photon has a default set of optimization passes which are used for all designs but users may also develop their own, for example to detect application specific combinations of nodes in a graph and mutate them to improve the hardware implementation.

Of the default graph passes the most important are those which allow the data stream inputs to nodes, inserting buffering or clock-enable logic as illustrated in the difference between Figure 1 and Figure 2. We refer to this process as ‘scheduling’ the graph.

We perform scheduling using two graph-passes. The first scheduling pass traverses the graph passively, collecting data about the latency (pipeline-depth) of each node in the graph. We then determine an offset in our core pipeline at which each node should be placed in order to ensure that data for all its inputs arrives in synchrony. After all the offsets in a schedule are generated a second pass applies these offsets by inserting buffering to align node inputs.

Sub-optimal offsets cause unnecessary extra buffering to be inserted into the graph, wasting precious BlockRAM and shift-register resources. To combat this inefficiency we calculate a schedule for the offsets using Integer Linear Programming (ILP). Our ILP formulation ensures all nodes are aligned such that their input data arrives at the same time while minimising the total number of bits used in buffering. Thus, Photon’s scheduled designs always have optimal buffering.

After all other graph-passes, a final graph-pass produces a hardware design. By this stage in compilation every node in the DAG has a direct mapping to an existing piece of parameterisable IP. Thus, this final pass creates a hardware design, by instantiating one IP component per node in the graph. Hardware is created in Photon using further Java classes to describe structural designs either directly using Java or by including external pre-written HDL, or running external processes to generate IP, e.g. CoreGen for floating-point units. After a design is fully described, external synthesis or simulation tools are invoked by Java to produce the required output for this execution. The system used to implement this low-level structural design and tool automation is based on the model described in [1].

5. Case Study

As a case study we consider a simple 2D convolution filter. This kind of design is common in many digital image processing applications.

The filter we implement is shown in Figure 3. The filter is separable, using the equivalent of two 1D 5 point convo-
5.1. Optimizations

The compilation of the convolution case study illustrates two of the optimization graph-passes during the Photon compilation process.

The Photon implementation of the filter makes use of several large stream-shifts on the input data. These shifts are necessary as each output data-point requires the 9 surrounding points to compute the convolved value. These stream-shifts result in a large number of buffers being added to the Photon design. Photon helps reduce this buffering using a graph-pass that combines the multiple delay buffers into a single long chain of buffers. This ensures each data item is only stored once, reducing buffering requirements.

Photon is able to use the precise value of the filter coefficient constants to optimize the floating-point multipliers. Specifically, some of the coefficients are a power of two, which can be highly optimized. To implement this, Photon includes another graph-pass which identifies floating-point multiplications by a power of two and replaces them with a dedicated node representing a dedicated hardware floating-point multiply-by-two IP core. This IP core uses a small number of LUTs to implement the multiplication rather than a DSP as in the conventional multipliers.

5.2. Implementation Results

For synthesis we target our filter design to a Xilinx Virtex-5 LX110T FPGA, with a clock frequency of 250MHz. At this speed, with data arriving and exiting the circuit once per cycle, we achieve a sustained computation rate of 1GB/s.

Table 2 shows the area impact of the Photon optimization graph-passes on the filter hardware. The multiplier power of two substitution pass reduces the number of DSP blocks used from 10 to 6, and the delay merging pass reduce BRAM usage from 18 RAMB36s to 4. The number of LUTs required for the original and optimized designs are similar.

<table>
<thead>
<tr>
<th></th>
<th>LUT/FF Pairs</th>
<th>DSPs</th>
<th>RAMB36s</th>
</tr>
</thead>
<tbody>
<tr>
<td>No opts.</td>
<td>6192</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>BRAM opts.</td>
<td>6297</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Mult. opts.</td>
<td>5851</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>All opts.</td>
<td>5925</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2. Resource usage for 2D convolution filter on a Virtex 5 with various optimizations.

6. Conclusion

In this paper we introduce Photon, a Java-based FPGA programming tool. We describe the programming model for Photon in which data I/O is separated from computation allowing designs to implicitly be easy to pipeline and hence perform well in an FPGA. We give an overview of Photon’s implementation as a library directed by user created Java programs. Finally, we present a case study demonstrating that Photon’s pluggable optimization system can be used to improve the resource utilisation of designs. Our current and future work with Photon includes developing a system for making it easier to create the data I/O logic external to Photon designs, and creating more advanced optimization passes.

References

Automated Design Approach for On-Chip Multiprocessor Systems

P. Mahr, H. Ishebabi, B. Andres, C. Loerchner, M. Metzner and C. Bobda
Department of Computer Science, University of Potsdam, Germany
{pmahr,ishebabi,andres,lorchner,metzner,bobda}@cs.uni-potsdam.de

Abstract

This paper presents a design approach for adaptive multiprocessor systems-on-chip on FPGAs. The goal in this particular design approach is to ease the implementation of an adaptive multiprocessor system by creating components, like processing nodes or memories, from a parallel program. Therefore message-passing, a paradigm for parallel programming on multiprocessor systems, is used. The analysis and simulation of the parallel application provides data for the formulation of constraints of the multiprocessor system. These constraints are used to solve an optimization problem with Integer Linear Programming: the creation of a suitable abstract multiprocessor hardware architecture and the mapping of tasks onto processors. The abstract architecture is then mapped onto a concrete architecture of components, like a specific Power-PC or soft-core processor, and is further processed using a vendor tool-chain for the generation of a configuration file for an FPGA.

1. Introduction

As apparent in current developments the reduction of transistor size and the exploitation of instruction-level parallelization can no longer be continued to enhance the performance of processors [1]. Instead, multi-core processors are a common way of enhancing performance by exploiting parallelism of applications. However, designing and implementing multiple processors on a single chip leads to new problems, which are absent in the design of single-core processors. For example, an optimal communication infrastructure between the processors needs to be found. Also, software developers have to parallelize their applications, so that the performance of the application is increased through multiple processors. In the case of multiprocessor systems-on-chip (MPSoCs), which combine embedded heterogeneous or homogeneous processing nodes, memory systems, interconnection networks and peripheral components, even more problems arise. Partly because of the variety of technologies available and partly because of their sophisticated functionality [2] [3]. To reduce design time, high level design approaches can be employed. In [4], [5], [6] and [7] design methodologies and corresponding tool-support are described.

In principle two communication paradigms for parallel computing with multiprocessor systems exist, the communication through shared memory (SMP), i.e. cache or memory on a bus-based system, and the passing of messages (MPI) through a communication network. SMP architectures, like the Sun Niagara processor [8] or the IBM Cell BE processor [9], are the common multiprocessors today. MPI is typically used in computer clusters, where physically distributed processors communicate through a network.

This paper presents a survey about our developments in the area of adaptive MPSoC design with FPGAs (Field-Programmable Gate Arrays) as a flexible platform for Chip-Multi-Processors. In section 2 an overview of the proposed design approach for MPSoCs is given. Therefore the steps for architectural synthesis, starting with the analysis and simulation of a parallel program and ending with the generation of a bitfile for the configuration of an FPGA are described in general. In the following section 3 an on-chip message passing software library for communication between tasks of a parallel program, and a benchmark for the purpose of evaluation are presented. Section 4 summarizes the formulation of architecture constraints for the design space exploration with Integer Linear Programming. These constraints are formulated out of the results of the analysis and simulation of a parallel program. The following section 5 gives an overview about the creation of MPSoCs using abstract components. Finally, this paper is concluded in section 6 and a brief overview about future work is given in section 7.

2. System design using architectural synthesis

To get an efficient multiprocessor system-on-chip from a parallel program several approaches are possible. In figure 1 our proposed synthesis flow using an analytical approach is shown. The architectural synthesis flow starts with parallel applications that are modeled as a directed graph, where
the nodes represent tasks and the edges represent communication channels [10].

Figure 1. Architectural Synthesis Flow

In the first step of the design flow, information on data traffic and on task precedence is extracted from functional simulations of the parallel program. Information on the number of cycles of a task when executed on a specific processor is determined from cycle accurate simulations. This information is used to formulate an instance of an Integer Linear Programming (ILP) problem.

In the following step, called Abstract Component creation, a combinatorial optimization is done by solving an ILP problem. Additionally to the information gathered in the first step, platform constraints, e.g. area and speed of the target platform, are needed as well. As a result of this step an abstract system description including the (abstract) hard- and software parts is generated.

The third step is called Component mapping. The abstract system description, which consists of abstract processors, memories, communication components or hardware accelerators and software tasks linked onto abstract processors, is mapped on a concrete architecture of components like PPC405, MicroBlaze or on-chip BRAMs. If needed, an operating systems can be generated with scripts and makefiles and can be mapped onto a processor as well. This step can be done using the PinHaT software (Platform-independent Hardware generation Tool) [11].

In the final step a bitfile is generated from the concrete components using the platform vendor tool-chain performing logic synthesis and place & route. However, simulations for the validation of the system on higher and lower levels is needed as well and should be performed during the proposed steps.

3. On-Chip Message Passing

In this section the on-chip communication between processing nodes using a subset of the MPI standard is described [12]. Therefore a small-sized MPI library was developed (see figure 2), which is similar to the approaches described in [13], [14] and [15].

Figure 2. SoC-MPI Library

The library consists of two layers. A Network independent layer (NInL) and a network dependent layer (NDeL), for the separation of the hardware dependent part from the hardware independent part of the library. The advantage of this separation is the easy migration of the library to other platforms. The NInL provides MPI functions, like MPI_Send, MPI_Receive, MPI_BSend or MPI_BCast. These functions are used to perform the communication between processes in the program. The NDeL is an accumulation of network dependent functions for different network topologies. In this layer the ranks and addresses for concrete networks are determined and the cutting and sending of messages depending on the chosen network is carried out. Currently the length of a message is limited to 64 Bytes, due to the limited on-chip memory of FPGAs. Longer messages are therefore cutted into several smaller messages and are send in series. The parameters of the MPI functions, like count, comm or dest (destination), are also used as signals and parameters for the hardware components of the network topology. That is, the parameters are used to build the header and the data packets for the communication over a network. The MPI parameters are directly used for the control, data and address signals.

The following MPI functions are currently supported:
Init, Finalize, Initialized, Wtime, Wtick, Send, Recv, SSend, BSend, RSend, SendRecv, Barrier, Gather, BCast, Comm_Size, Comm_Rank.

Figure 3. Configuration of processing nodes

In figure 3 several processing nodes are connected together via a star network. Additionally node 0 and 1 are directly connected together via FSL (Fast Simplex Link) [16]. Each processing node has only a subset of the SoC-MPI Library with the dependent functions for the network topology.

3.1. Benchmarks

The MPI library is evaluated using Intel MPI Benchmarks 3.1, which is the successor of the well known package PMB (Pallas MPI Benchmarks) [17]. The MPI implementation was benchmarked on a Xilinx ML-403 evaluation platform [18], which includes a Virtex 4 FPGA running at 100MHz. Three MicroBlaze soft-core processors [19] were connected together via a star network. All programs were stored in the on-chip memories.

In Figure 4 the results of the five micro benchmarks are shown. Due to the limited on-chip memory not all benchmarks could be performed completely. Furthermore, a small decay between 128 and 256 Bytes message size exists, because the maximum MPI message length is currently limited to 251 KBytes and a message larger than that must be split into several messages. Further increase of the message size would lead to a bandwidth closer to the maximum possible bandwidth, which is limited through the MicroBlaze and was measured with approximately 14 MBytes/s.

4. Abstract component creation using Integer Linear Programming

In this flow, Integer Linear Programming (ILP) is used for automated design space exploration with the goal of locating the best possible abstract architecture for a given parallel application under given constraints. The simultaneous optimization problem is to map parallel tasks to a set of processing elements and to generate a suitable communication architecture, which meets the constraints of the target platform, and minimizes the overall computation time of the parallel program. The input for this step is obtained by using a profiling tool for the mpich2 package. In the following two subsections area and time constraints of processors and communication infrastructure are described in separate.

4.1. Processors - sharing constraint, area constraint and costs

A few assumptions about processors and tasks need to be made, because it is possible to map several tasks on a processor: (1) a task scheduler exists, so that scheduling is not involved in the optimization problem. (2) Task mapping is static. (3) The instruction sequence of a task is stored on the local program memory of the processor, e.g. instruction cache, and hence the size of the local program memory limits the number of tasks, which can be mapped onto a processor. (4) Finally the cost of switching tasks in terms of processor cycles does not vary from task to task.

Let \( I_i \in I_0, ..., I_n \) be a task, \( J_j \in J_0, ..., J_m \) a processor and \( x_{ij} = 0, 1 \) a binary decision variable, whereas \( x_{ij} = 1 \) means that task \( I_i \) is mapped onto processor \( J_j \).

\[
\sum_{j=0}^{m} x_{ij} = 1, \forall I_i
\]  

A constraint for task mapping (equation 2), called address space constraint, and the cost of task switching (equation 3) can be formulated, where \( s_{ij} \) is the size of a task \( I_i \) of a processor \( J_j \) with the program memory size \( s_j \) and \( t_j \) is the cost (time) of task switching.
\[
\sum_{i=0}^{n} x_{ij} \cdot s_{ij} \leq s_j, \forall I_i
\] (2)

\[
T_{SWITCH} = \sum_{j=0}^{m} \sum_{i=0}^{n} x_{ij} \cdot t_j
\] (3)

For the calculation of the area of the processors \( A_{PE} \) the area of of a single processor \( a_j \) is needed. Because \( x_{ij} \) only shows if a task \( I_i \) is mapped onto a processor \( J_j \) and does not show the number of processors in the system or the number of instantiations of a processor, an auxiliary variable \( v_j = 0, 1 \) is needed. For each instance of a processor \( J_j \) there is a corresponding virtual processor \( v_j \) and for all tasks mapped to a certain processor there is only one task which is mapped to the corresponding virtual processor. This leads to the following constraint (equation 4) so that the area of the processors can be calculated with equation 5.

\[
v_j \leq \sum_{i=0}^{n} x_{ij}, \forall J_j
\] (4)

\[
A_{PE} \geq \sum_{j=0}^{m} v_j \cdot a_j
\] (5)

4.2. Communication networks - Network capacity and area constraint

Several assumptions have to be made before constraints about the communication network can be formulated. The communication of two tasks mapped onto the same processor is done via intra-processor communication, which have a negligible communication latency and overhead, compared to memory access latency. All processors can use any of the available communication networks and can use more than one network. A communication network has arbitration costs resulting from simultaneous access on the network. It is assumed, that tasks are not prioritized and an upper bound on arbitration time for each network can be computed for each network topology depending on the number of processors. Finally, it is not predictable when two or more tasks will attempt to access the network. Though a certain probability can be assumed.

\( \lambda_{i1,i2} \) is an auxiliary 0-1 decision variable that is 1, if two communicating tasks are mapped on different processors. The sum of \( x_{i1,j1} \) and \( x_{i2,j2} \) equals two if the tasks are on different processors as seen in equation 6.

\[
\lambda_{i1,i2} = \frac{x_{i1,j1} + x_{i2,j2}}{2}
\] (6)

A communication topology \( C_k \in C_1, \ldots, C_n \) may have a maximum capacity of processors attached to it. This constraint is described in equation 7. \( y_k \) is a binary decision variable with value 1 if a communication topology \( C_k \) is used for communication between two tasks \( I_{i1} \) and \( I_{i2} \), and 0 otherwise. \( I_{i1} \prec I_{i2} \) is a precedence operator. \( I_{i1} \) is preceded by \( I_{i2} \) which means, that a data transfer is performed from \( I_{i1} \) to \( I_{i2} \). The maximum number of processes, which can use a topology \( C_k \) is described by \( M_k \).

\[
y_k + \sum_{I_{i1},I_{i2}|I_{i1} \prec I_{i2}} \lambda_{i1,i2} \leq M_k, \forall C_k
\] (7)

The total area cost of the communication network (resources for routing) can be calculated with equation 8, where \( A_k \) is the area cost of topology \( C_k \).

\[
A_{NET} \geq \sum_{j=0}^{m} A_k \cdot y_k
\] (8)

The cost of the topology in terms of computation time is calculated in 10, whereas \( z_{k_{i1,i2}} \) is a binary decision variable, which is 1 if a network will be used by two tasks. Otherwise the variable \( z_{k_{i1,i2}} \) is 0. \( D_{i1,i2} \) is the amount of data to be transferred between the two communicating tasks and \( p_k \) is the probability that network arbitration will be involved when a task wants to communicate. The upper bound arbitration time is \( \tau_k \).

\[
z_{k_{i1,i2}} \geq \lambda_{i1,i2} + y_k - 1
\] (9)

\[
T_{NET} = \sum_{I_{i1},I_{i2}|I_{i1} \prec I_{i2}} \left( \sum_{k=0}^{K} (D_{i1,i2} + \tau_k \cdot p_k)z_{k_{i1,i2}} \right)
\] (10)

Finally the total area cost \( A \) is calculated form the area of the processing elements \( A_{PE} \) (equation 5) and the area for the routing resources \( A_{NET} \) (equation 8).

\[
A \geq A_{PE} + A_{NET}
\] (11)

The cost of computation time can be calculated with equation 12, whereas \( T_{ij} \) is the time requirement to process a task \( I_i \) on a processor \( J_j \). The objective, in this case, is to minimize computation time of a (terminating) parallel program. However for non-terminating programs, like signal processing programs, the objectives are different.

\[
T = \min \left( \sum_{j=0}^{m} \sum_{i=0}^{n} x_{ij} \cdot T_{ij} + T_{NET} + T_{SWITCH} \right)
\] (12)
5. Component Mapping

In this section the mapping of abstract components onto concrete ones is described. This component-based approach is similar to the one described by Cesário et al. [20], where a high-level component-based methodology and design environment for application-specific MPSoC architectures is presented. For this task, a component-based design environment called PinHaT for the generation and configuration of the system infrastructure was developed. This environment offers a vendor independent framework with which users can specify the requirements of their system and automatically produce the necessary system configuration and startup code. The Tool was developed as a Java application.

Generally PinHaT follows a two-step approach. In the first step an abstract specification of the system using abstract components like CPUs, Memories or Hardware Accelerators are described. In the following step these abstract components will be refined and mapped to concrete components, e.g. a specific CPU (PPC405, MicrocBlaze) or a Hardware-Divider. Also the software tasks are mapped onto the concrete components. The structure of PinHaT is shown in figure 5. A detailed overview about the PinHaT tool is given in [11].

![Diagram of PinHaT](image)

**Figure 5. Structure of PinHaT**

The component mapping is divided into the generation and the configuration of the system infrastructure, where hardware is generated and software is configured. In this flow, the input to PinHaT is obtained by high level synthesis, as described in section 4.

5.1. Generation of the System Infrastructure

The generation of the system infrastructure, that is the mapping of an abstract system description onto a concrete hardware description, is done by PinHaT. PinHaT uses XML in conjunction with a document type definition file (DTD) as an input for the abstract system description. An input consists of valid modules, which are CPU, Memory, Communication Modules, Periphery and Hardware Accelerator. The hardware mapping onto concrete components is divided into three phases.

In the first phases an internal XML tree is built by parsing the input file. For all nodes of this tree an adequate class is instantiated. These classes can be easily added to the framework to extend the IP-Core base. In a subsequent step, another parser creates the platform specific hardware information file from the gathered information. In the second phase individual mappers for all components and target platforms, are created, followed by the last phase, where a mapper creates the platform dependent hardware description files. These dependent hardware description files are then passed to the vendor’s tool chain, e.g. Xilinx EDK or Altera Quartus II.

5.2. Configuration of the System Infrastructure - SW Mapping

In the case of software, a task is mapped onto a concrete processor. This is in contrast to the mapping of abstract components, e.g. processors or memories, to concrete ones during the generation of the system infrastructure.

For the mapping step, parameters of the software must be specified for each processor. The parameters include information about the application or the operating system, like source-code, libraries or the os-type. With these information scripts and Makefiles for building the standalone applications and the operating systems are created. While standalone applications only need compiling and linking of the application, building the operating system is more difficult. Depending of the operating system, different steps, like configuration of the file-system or the kernel parameters, are necessary. The result of the task mapping is an executable file for each processor in the system.

6. Conclusion

In this paper a concept for the design automation of multiprocessor systems on FPGAs was presented. A small-sized MPI library was implemented to use message passing for the communication between tasks of a parallel program. Simulation and analysis of the parallel program is carried
out to gather information about task precedence, task interaction and data traffic between tasks. These information is needed to formulate constraints, for the solving of an Integer Linear Programming problem. As a result an abstract system description, consisting of hardware components, like processing nodes or hardware accelerators, and linked tasks is created. With the help of the PinHaT software the components of the abstract system description can be mapped onto concrete components, like specific processors. Finally the configuration file for an FPGA can be created using the vendor tool-chain.

7. Future Work

Currently the PinHaT software is being extended into an easy to use software solution including the architectural synthesis of a parallel program. Also new abstract and corresponding concrete components, like new network topologies or processing nodes like the OpenRISC processor will be included to enhance flexibility of the architectural synthesis flow. Furthermore concepts of adaptivity of multiprocessor systems are to be analysed and a detailed evaluation of on-chip message-passing in the face of communication overhead and latency needs to be carried out.

References

Abstract

This article presents a methodology to describe digital circuits from register transfer level to system level. When designing systems it encapsulates the functionality of several modules and also encapsulates the connections between those modules. To achieve these results, the possibilities of Algorithmic State Machines (ASM charts) have been extended to develop a compiler. Using this approach, a System-on-a-Chip (SoC) design becomes a set of linked boxes where several special boxes encapsulate the connections between modules. The compiler processes all required boxes and files, and then generates the corresponding HDL code, valid for simulation and synthesis. A small SoC example is shown.

1. Introduction

System-on-a-Chip (SoC) designs integrate processor cores, memories and custom logic joined into complete systems. The increased complexity requires more effort and more efficient tools, but also an accurate knowledge on how to connect new computational modules to new peripheral devices using even new communication protocols and standards.

A hierarchical approach may encapsulate on black boxes the functionality of several modules. This technique effectively reduces the number of components, but system integration becomes more and more difficult as new components are added every day.

Thus, the key to a short design time, enabling “product on demand”, is the use of a set of predesigned components which can be easily integrated through a set of also predesigned connections, in order to build a product.

Because of this reason, Xilinx and Altera have proposed their high end tools named Embedded Development Kit [1] and SoPC Builder [2], respectively, that allow the automatic generation of systems. Using these tools, designers may build complete SoC designs based on their processors and peripheral modules in few hours. At a lower scale, similar results may be found on the Hardware Highway (HwHw) web tool [3].

2. ASM++ charts

The Algorithmic State Machine (ASM) method for specifying digital designs was originally documented on 1973 by C.R. Clare [6], who worked at the Electronics Research Laboratory of Hewlett Packard Labs, based on previous developments made by T. Osborne at the University of California at Berkeley [6]. Since then it has been widely applied to assist designers in expressing algorithms and to support their conversion into hardware [7-10]. Many texts on digital logic design cover the ASM method in conjunction with other methods for specifying Finite State Machines (FSM) [11-12].

A FSM is a valid representation of the behavior of a digital circuit when the number of transitions and the complexity of operations is low. The example of fig. 1 shows a FSM for a 12x12 unsigned multiplier that computes ‘outP = inA * inB’ through twelve conditional additions. It is fired by a signal named ‘go’, it signals the answer using ‘done’, and indicates through ‘ready’ that new operands are welcome.
Figure 1. An example of FSM for a multiplier.

However, on these situations traditional ASM charts may be more accurate and consistent. As shown at fig. 2, they use three different boxes to fully describe the behavior of cycle driven RTL designs: a “state box” with rectangular shape defines the beginning of each clock cycle and may include unconditional operations that must be executed during (marked with ‘=’) or at the end (using the delay operator ‘←’) of that cycle; “decision boxes” –diamond ones– are used to test inputs or internal values to determine the execution flow; and finally “conditional output boxes” –with oval shape– indicate those operations that are executed during the same clock cycle, but only when previous conditions are valid. Additionally, an “ASM block” includes all operations and decisions that are or can be executed simultaneously during each clock cycle.

The advantages of FSM for an overall description of a module are evident, but the ASM representation allows more complex designs through conditions that are introduced incrementally and detailed operations located where designer specifies.

However, ASM notation has several drawbacks:

- They use the same box, rectangular ones, for new states and unconditional operations executed at those states. Because of this property, ASM diagrams are compact, but they are also more rigid and difficult to read.
- Sometimes it is difficult to differentiate the frontier between different states. The complexity of some states requires the use of dashed boxes (named ASM blocks) or even different colors for different states.

- Due to the double meaning of rectangular boxes, conditional operations must be represented using a different shape, the oval boxes. But, actually, all operations are conditional, because all of them are state dependent.
- Additionally, designers must use lateral annotations for state names, for reset signals or even for links between different parts of a design (see fig. 2).
- Finally, the width of signals and ports cannot be specified when using the current notation.

Proposed ASM++ notation [13-14] tries to solve all these problems and extend far beyond the possibilities of this methodology. The first and main change introduced by this new notation, as seen at fig. 3, is the use of a specific box for states --we propose oval boxes, very similar to those circles used in bubble diagrams-- thus now all operations may share the same box, a rectangle for synchronous assignments and a rectangle with bent sides for asynchronous assertions. Diamonds are kept for decision boxes because they are commonly recognized and accepted.

Figure 3 shows additional features of ASM++ charts, included to allow their automatic compilation to generate HDL code. In addition to an algorithmic part, a declarative section may describe the design name, its implementation parameters, the external interface, one or more internal signals. The synchronization signal and its reset sequence can be fully specified in a very intuitive way too. A box for ‘defaults’ has been added to easily describe the circuit behavior when any state leave any signal free. Furthermore, all boxes use standard VHDL or Verilog expressions, but never both of them; the ASM++ compiler usually detects the HDL and then generates valid HDL code using the same language.
3. Hierarchical design using ASM++ charts

As soon as a compiler generates the VHDL or Verilog code related to an ASM++ chart, the advanced features of modern HDL languages can be easily integrated on them. The requirements for hierarchical design have been included through the following elements:

– Each design begins with a ‘header’ box that specifies the design name and, optionally, its parameters or generics.

– Any design may use one or several pages on a MS Visio 2007 document\(^1\), saved using its VDX format. Each VDX document may include several designs identified through their header boxes.

– Any design may instantiate other designs, giving them an instance name. As soon as a lower level module is instantiated, a full set of signals named “instance_name.port_name” (see fig. 5) is created to ease the connections with other elements. Later on, any ‘dot’ will be replaced by an ‘underline’ because of HDL compatibility issues.

– When the description of an instantiated module is located on another file, a ‘RequireFile’ box must be used before the header box to allow a joint compilation. However, the ASM++ compiler identifies any previously compiled design to avoid useless efforts and invalid duplications.

– VHDL users may include libraries or packages using their ‘library’ and ‘use’ sentences, but also before any header box.

– Nowadays, compiler does not support reading external HDL files, in order to instantiate hand written modules. A prototype of them, as shown at fig. 4, can be used instead.

Using these features, an example with a slightly improved multiplier can be easily designed. First of all, a prototype of a small FIFO memory is declared, as shown at fig. 4, thus compiler may know how to instantiate and connect this module, described elsewhere on a Verilog file. Then three FIFO memories are instantiated to handle the input and output data flows, as shown at fig. 5, so several processors may feed and retrieve data from this processing element.

\(\text{Figure 4. A prototype of an external design.}\)

\(\text{Figure 5. An example of hierarchical design.}\)

The ASM++ chart of fig. 5 can be compared with its arranged compilation result, shown below. The advantages of this methodology on flexibility, clarity and time saving are evident. Not always a text based tool is faster and more productive than a graphical tool.

\[
\begin{align*}
\text{parameter width = 16; // 16x16 => 32}
\text{parameter depth = 6; // 64-level buffers}
\text{input clk, reset;}
\text{output readyA;}
\text{input pushA;}
\text{input [width-1:0] inA;}
\text{output readyB;}
\text{input pushB;}
\text{input [width-1:0] inB;}
\text{output readyP;}
\text{input popP;}
\text{output [2*width-1:0] outP;}
\text{wire activate;}
\text{wire fifoA_clk, fifoA_reset;}
\text{wire [width-1:0] fifoA_dataIn, fifoA_dataOut;}
\text{wire fifoA_push, fifoA_pop;}
\text{wire fifoA_empty, fifoA_full;}
\text{fifo # (}
\text{.width(width), .depth (depth)}
\text{) fifoA (}
\text{.clk (fifoA_clk), .reset (fifoA_reset),}
\text{.data_in (fifoA_dataIn), .data_out (fifoA_dataOut),}
\text{.push (fifoA_push), .pop (fifoA_pop),}
\text{.empty (fifoA_empty), .full (fifoA_full)}
\text{);}\n\text{wire fifoB_clk, fifoB_reset;}
\text{wire [width-1:0] fifoB_dataIn, fifoB_dataOut;}
\text{wire fifoB_push, fifoB_pop;}
\text{wire fifoB_empty, fifoB_full;}
\text{fifo # (}
\text{.width(width), .depth (depth)}
\text{) fifoB (}
\text{.clk (fifoB_clk), .reset (fifoB_reset),}
\text{.data_in (fifoB_dataIn), .data_out (fifoB_dataOut),}
\text{.push (fifoB_push), .pop (fifoB_pop),}
\text{.empty (fifoB_empty), .full (fifoB_full)}
\text{);}\n\end{align*}
\]
4. Encapsulating connections using pipes

Following this bottom-up methodology, the next step is using ASM++ charts to design full systems. As stated above, a chart can be used to instantiate several modules and connect them, with full, simple and easy access to all port signals.

However, system designers need to know how their available IP modules can or must be connected, in order to build a system. Probably, they need to read thoroughly several data sheets and try different combinations, to finally match their requirements. Nonetheless, when they become experts on those modules, newer and better IP modules are developed, so system designers must start again and again.

This paper presents an alternative to this situation, called “Easy-Reuse”. During the following explanations, please, refer to figures 6 to 9.

- First of all, a fully new concept must be introduced: an ASM++ chart may describe an entity/module that will be instantiated, like ‘multiplier’ at fig. 3, but additionally it may be used for a description that will be executed (see figs. 8 and 9). The former will just instantiate a reference to an outer description, meanwhile the later will generate one or more sentences inside the modules that call them. To differentiate those modules that will be executed, header boxes enclose one or more module names using ‘<’ and ‘>’ symbols. Later on, these descriptions will be processed each time an instance or a ‘pipe’ (described below) calls them.

- Furthermore, the ASM++ compiler has been enhanced with PHP-like variables [15]. They are immediately evaluated during compilation, but they are available only at compilation time, so no circuit structures will be directly inferred from them. Their names are preceded by a dollar sign (“$”), they may be assigned with no previous declaration and store integer values, strings or lists of freely indexed variables.

In order to differentiate several connections that may use the same descriptor, variables are used instead of parameters or generics. The corresponding field at a header box, when using it to start a connection description, is used to define default values for several variables (see fig. 8); these specifications would be changed by pipes on each instantiation (see fig. 6).

- Usual ASM boxes are connected in a sequence using arrows with sense; a new box called “pipe” can be placed out of the sequence and connect two instances through single lines, with no arrows.

- When compiler finishes the processing of the main sequence, it searches all pipes, looks for their linked instances, and executes the ASM charts related to those connections. Before each operation, it defines two automatic variables to identify the connecting instances. As said above, the pipe itself may define additional variables to personalize and differentiate each connection.

As soon as several pipes may describe connections to the same signal, a resolution function must be defined to handle their conflicts. A tristate function would be used, but HDL compilers use to refuse such connections if they suspect contentions; furthermore, modern FPGAs do not implement such resources any more because of their high consumption, thus these descriptions are actually replaced by gate-safe logic. Subsequently, a wired-OR, easier to understand than a wired-AND, has
been implemented when several sources define different values from different pipe instantiations or, in general, from different design threads.

- The last element required by ASM++ charts to manage automatic connections is conditional compilation. A diamond-like box, with double lines at each side, is used to tell the ASM++ compiler to follow one path and fully ignore the other one. Thus, different connections are created when, for example, a FIFO memory is accessed from a processor to write data, to read data or both.

Using these ideas, a SoC design may now encapsulate not only the functionality of several components, but also their connections.

Figure 6 describes a small SoC that implements a Harvard-like DSP processor (see [13]) connected to a program memory, a 32-level FIFO and a register. First of all, two C-like compiler directives are used to specify the HDL language and a definition used later; a VDX file that describes the DSP processor is also included before giving a name to the SoC design. Then, all required modules are instantiated and connected using pipes.

![Figure 6. A small SoC design using pipes.](image)

A small program memory has been designed for testing purposes, as shown at fig. 7: the upper chart describes a ROM memory with a short program that emulates the behavior of a Xilinx Block RAM, and the lower chart describes how this synchronous memory must be connected to the DSP. This figure illustrates the use of automatic variables (‘$ProgMem’ and ‘$DSPuva18’, whose values will be “mem_01” and “dsp_01”, respectively) and the difference between modules that can be instantiated or executed.

![Figure 7. Charts may describe connections.](image)

The pipe at figure 6 with text “RW250” describes the connection of a FIFO memory (see fig. 4) to a DSPuva18 processor [13], thus it executes the ASM++ chart shown at fig. 8. When executing this pipe, a ‘0’ value is firstly assigned to variables ‘$port’, ‘$write_side’ and ‘$read_side’, as stated by the header box; then these values are changed as specified by the pipe box (see the defined value of ‘RW250’); finally, the chart of figure 8 generates the HDL code that fully describes how “fifo_01” device is connected to “dsp_01” processor for reading and writing using port ‘250’ for data and port ‘251’ for control (getting the state through a read and forcing a reset through a write).
Two final ASM++ charts will be described at figure 9, but other required charts have not been included for shortness. The chart at left specifies how the instance named ‘SoC_iface’ at figure 6 must be executed, not instantiated, in order to generate two control inputs and to connect them to all modules. The diagram at right generates additional I/O signals and connects them to the register controlled by the DSP through its port ‘0’.

Several sentences of the HDL code generated by the ASM++ compiler when processing these diagrams are displayed following, revealing that ASM++ charts are fully capable of describing SoC designs using an intuitive, easy to use and consistent representation.

5. Conclusions

This article has presented a powerful and intuitive methodology for SoC design named Easy-Reuse. It is based on a suitable extension of traditional Algorithmic State Machines, named ASM++ charts, its compiler and a key idea: charts may describe entities or modules, but they also may describe connections between modules. The ASM++ compiler developed to process these charts in order to generate VHDL or Verilog code has been enhanced further to understand a new box called pipe that implements the required connections. The result is a self-documented diagram that fully describes the system for easy maintenance, supervision, simulation and synthesis.

6. Acknowledgments

The authors would like to acknowledge the financial support for these developments of eZono AG, Jena, Germany, ISEND SA, Valladolid, Spain, and the Spanish Government (MEC) under grant CICYT ENE2007-67417/ALT with FEDER funds.
References


[15] The PHP Group, on-line at http://www.php.net, last release has been PHP 5.2.6 at May 1st, 2008.
Space-Efficient FPGA-Implementations of FFTs in High-Speed Applications

Stefan Hochgürtel, Bernd Klein
Max Planck Institut für Radioastronomie, Bonn, Germany
{shochgue, bklein}@mpifr-bonn.mpg.de

Abstract

Known and novel techniques are described to implement a Fast Fourier Transform (FFT) in hardware, such that parallelized data can be processed. The usage of both - real and imaginary FFT-input - can help saving hardware. Based on the different techniques, flexible to use FFT-implementations have been developed by combining standard FFT-components (partly IP) and are compared, according to their hardware utilization. Finally, applicability has been demonstrated in practice by a FFT-implementation with 8192 channels as part of a FPGA-spectrometer with a total bandwidth of 1.5 GHz.

1. Introduction

A number of radio-astronomical telescopes are now in operation for observations within the mm and submm wavelength atmospheric windows. Each of these windows stretches over many 10s of GHz, and heterodyne receivers have now been developed to cover a large fraction thereof. The necessary spectrometers that allow a simultaneous coverage of such wide bandwidths (>1.0 GHz) at high spectral resolution (1 MHz or better) have only recently become feasible through the use of FPGAs that can handle multiple Gigabytes of data per second. Another great advantage of FPGA-based spectrometers over the conventional filter-banks or auto-correlators is that their production is cheaper and that they are more compact and consume less power. Their stability furthermore allows the parallel stacking to cover wider bandwidths than each individual ADC/FPGA module can currently cover.

A common way to implement an FPGA spectrometer is to feed the data stream of an Analog/Digital Converter (ADC) to an FPGA that computes a Fast Fourier Transform (FFT). An $N$-point FFT takes $N$ time-consecutive data samples and transforms them to a frequency spectrum with $N$ spectral channels. According to the Shannon sampling theorem [4], the ADC’s sample-rate $f_S$ determines the spectral bandwidth $f = f_S/2$. The sensitivity specifies the spectrometer’s ability to detect weak signals. It is in principle given by the ADC’s bit-resolution, but can be increased by an integration (averaging) in time of each spectral channel. Integration can also be performed in time-domain, which improves sensitivity by using each input-sample multiple times. Since this averaging may neutralize a periodic input-signal, integration in time-domain must be combined with a window function. This is called weighted-overlap-add (WOLA).

2. Related Work

Efficient FFT-cores are commercially available from RF Engines [3], who provide pipelined FFTs as well as combined FFTs (Section 4.1). Another source of pipelined FFTs in multiple variations is the Xilinx IP-Core-Generator, which is freely bundled with Xilinx-ISE [5]. Since Xilinx IP-cores are freely available, we use them for our pipelined FFTs.

A powerful commercially available spectrometer-card was developed by Acqiris [1]. It has been in operation as a spectrometer backend at the APEX telescope [2] since 2006. It is based on a Xilinx Virtex-II Pro 70 FPGA, fed by two ADCs that deliver 2 Gigasamples per second (Gs/s) at 8 bit resolution. Its bandwidth is thereby limited to 1.0 GHz. In collaboration with Acqiris, the ARGOS-project [7] has implemented a FFT-spectrometer on this board. Starting with 8 bit input-samples, the data width grows scaled to some degree, from 9 bits after preprocessing to 18 bits after a 32k-pt FFT. Unscaled growth would require at least $9 + \log(32768) = 24$ bits after the FFT, since it sums up 32768 values of 9 bits. Although scaled broadening saves hardware, available to implement FFTs with a larger number of channels, it reduces precision and therefore potentially reduces sensitivity. The preprocessing comprises windowing but no WOLA. At the output, 32 consecutive bits can be chosen from each 36 bits wide channel in integrated power-spectrum, in order to transmit them by PCI-interface.
3. The Algorithm

We use an ADC that samples data at rates up to 3.0 GHz and a Virtex-4 FPGA with a maximum clock-rate of 400 MHz [6]. So, input-data stream must be split up into some parallel data-streams with reduced speed to allow the FPGA to handle it. In this section, we will first recap how a complex FFT-input can be used for real only input-data, in order to show how both the real and imaginary part of a complex FFT-input can be used for real only input-data, in order to reduce the hardware requirements [8].

3.1. Split FFTs

A $M \cdot N$-pt FFT calculates $M \cdot N$ spectral channels $\hat{a}_k$ out of $M \cdot N$ input-samples $a_j$, where $M$ and $N$ are powers of two:

$$\hat{a}_k = \sum_{j=0}^{M \cdot N-1} e^{-2\pi i \frac{j \cdot k}{M \cdot N}} \cdot a_j$$  \hspace{1cm} (1)

The input can be split up into $N$ groups $n$ of $M$ consecutive samples $m$, so that $j = M \cdot n + m$. The same way the output is split into $M$ groups $p$ of $N$ consecutive channels $q$, with $k = N \cdot p + q$. Applying this to (1) results in a double-sum that can be simplified, since $e^{-2\pi i \cdot n \cdot p} = 1$, with $n, p \in \mathbb{N}$:

$$\hat{a}_{N \cdot p + q} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} e^{-2\pi i \frac{(m + n \cdot m) \cdot (N \cdot p + q)}{M \cdot N}} \cdot a_{M \cdot n + m}$$

$$= \sum_{m=0}^{M-1} e^{-2\pi i \frac{(m + n \cdot m) \cdot (N \cdot p + q)}{M \cdot N}} \cdot \sum_{n=0}^{N-1} e^{-2\pi i \frac{n^2}{N}} \cdot a_{M \cdot n + m}$$  \hspace{1cm} (2)

The inner sums can be computed as $M$ single $N$-pt FFTs, fed with undersampled data:

$$a_{M \cdot n + m} = \sum_{n=0}^{N-1} e^{-2\pi i \frac{n^2}{N}} \cdot a_{M \cdot n + m}$$  \hspace{1cm} (3)

Thereby $M$ spectra $a'_{m, q}$ are calculated, time-shifted to each other by the original sampling-rate, each one with a fraction $1/M$ of the original bandwidth. If their channels $a'_{m, q}$ are multiplied by the correct power of $e^{-2\pi i}$, a set of $N$ different $M$-pt FFTs remains to be computed. These complex factors are called twiddle-factors, since a multiplication with $e^{-2\pi i \cdot x}$, with $x \in \mathbb{R}$ equals a clockwise rotation (twiddle) by $x$ full turns. The correct twiddle here depends on time-shift (FFT-number $0 \leq m < M$) and channel $0 \leq q < N$:

$$a_{m, q}'' = e^{-2\pi i \frac{m^2}{N \cdot M}} \cdot a'_{m, q}$$  \hspace{1cm} (4)

$$\hat{a}_{N \cdot p + q} = \sum_{m=0}^{M-1} e^{-2\pi i \frac{m^2}{N \cdot M}} \cdot a_{m, q}''$$  \hspace{1cm} (5)

3.2. Use imaginary input

A complex FFT fed with real input-samples produces a spectrum whose second half is complex conjugated to the first half:

$$\hat{a}_{N \cdot k} = \sum_{j=0}^{N-1} e^{-2\pi i \frac{j \cdot k}{N}} \cdot a_j = \overline{\hat{a}_k}$$  \hspace{1cm} (6)

To prevent a waste of resources, real samples may alternately feed real and imaginary input of a half-sized FFT:

$$\tilde{a}_k = \sum_{n=0}^{N-1} e^{-2\pi i \frac{j \cdot k}{2}} \cdot (a_{2j} + i \cdot a_{2j+1})$$  \hspace{1cm} (7)

All intermediate channels $\tilde{a}$ form corresponding pairs $\hat{a}_k$ and $\overline{\hat{a}_{N-k}}$. By adding and subtracting them, two independent half-sized FFTs can be calculated:

$$a'_{0, k} = \frac{\hat{a}_{0, k} + \hat{a}_{N-k}}{2} = \sum_{n=0}^{N-1} e^{-2\pi i \frac{j \cdot k}{2}} \cdot a_{2j}$$

$$a'_{1, k} = \frac{\hat{a}_{0, k} - \hat{a}_{N-k}}{2} = \sum_{n=0}^{N-1} e^{-2\pi i \frac{j \cdot k}{2}} \cdot a_{2j+1}$$  \hspace{1cm} (8)

A twiddle-multiplication and an addition yield the final spectral channels:

$$a_k = a'_{0, k} + e^{-2\pi i \frac{N \cdot k}{M}} \cdot a'_{1, k}$$  \hspace{1cm} (9)

4. Implementation

Since ADC-data must be demultiplexed to reduce its rate to below a clock-rate realistic for the FPGA, we obtain $P$ parallel data-streams, each one containing undersampled data with offsets of one sample to the adjoining streams. If a $P$-pt FFT is insufficient for spectral resolution, input-samples have to be collected over $Q$ clock-cycles to calculate a $P \cdot Q$-pt FFT. The structure of a FFT demands $P$ and $Q$ to be powers of 2.
4.1. Combining Parallel and Pipelined FFTs

The way a FFT is split up (Section 3.1) suggests feeding each parallel data-stream into a pipelined $N$-pt FFT ($N = Q$) to calculate the inner sums $a_{m,q}$. After twiddle-multiplication, a parallel $M$-pt FFT can be used to calculate $M = P$ spectral channels in any clock cycle. Altogether this creates a combined FFT (Figure 1) that operates sequentially on parallel data-streams. This implementation is simple and hardware-efficient as well as flexible in adjusting its width $P$ and its length $Q$ to values, appropriate for any clock-rate and any spectrum-size.

Keeping in mind that a FFT may exceed the capabilities of a single FPGA, it is desirable to split it into independent parts. Since the pipelined FFTs are expected to be the largest part of a combined FFT, separating them from each other would have the highest impact on resource consumption. In a combined FFT data from all parallel streams has to be set against each other at the end. If the pipelined FFTs' instances are separated to different, parallel FPGAs, recombination of data from different FPGAs at high clock-rates would be required. Therefore independent data-streams at the end would simplify parallelization over several FPGAs.

This is achieved by a parallel $N$-pt FFT ($N = P$) at the beginning. Since each FFT in (3) needs undersampled input, the data has to be reordered first, dependent on the size of the complete FFT: the parallel FFT at the front (Figure 2) requests $M = Q$ subsequent samples at each input, skipping the next $M \cdot N - M$ samples, which are needed at its other inputs in parallel. This is done by a memory module that stores $M \cdot N$ samples, followed by a two-dimensional shift-register. Behind the parallel FFT, the data-streams become independent from each other. Each one is multiplied by twiddle-factors and the resulting $a_{m,q}'$ are fed into pipelined $M$-pt FFTs to calculate the $N$th part of the complete spectrum.

So we combined a reorder unit, a parallel FFT and multiple pipelined FFTs, such that data-streams become independent after parallel FFT and can therefore be split into multiple partitions at the end. In the following this technique is called splitting FFT.

4.2. Reduce FFT-width

The number of channels that are computed by a combined or splitting FFT can be increased in two ways with different effects on hardware utilization: increase its width $P$ or increase its length $Q$.

Doubling $P$ will double the number of pipelined FFTs and twiddle-multipiers and therefore the hardware utilized by them. The parallel FFT grows with $P$ as $O(P \cdot \log(Q))$.

Increasing the length $Q$ on the other hand, does not lead to changes in parallel FFT - it simply operates more cycles per FFT. Twiddle-multiplication needs no more logic resources, but memory use increases linearly with $Q$, due to the need to store more twiddles. Doubling the channels calculated from a pipelined FFT requires storing all previous channels again and adding another pipeline-stage. Thus the pipelined FFTs' memory use grows linear with $Q$, while their logic utilization grows logarithmically:

\[
\text{Logic}(P, Q) = O(P \cdot \log(P) \cdot \log(Q))
\]

\[
\text{Mem}(P, Q) = O(P \cdot Q)
\]

A combined or splitting FFT's use of FPGA-logic grows stronger with its width than with its length.

To optimize the resource usage, $P$ should be reduced, which could be achieved by less demultiplexing of input-

---

Figure 1. 8 pipelined 128-pt FFTs followed by a parallel 8-pt FFT form a combined 1024-pt FFT (8x128).

Figure 2. Reordering the input, allows parallel FFT first and splitting data-streams afterward. A parallel 8-pt FFT followed by 8 pipelined 128-pt FFTs form a splitting 1024-pt FFT (8x128).
data. This either requires higher clock-rates, which is not realistic in most cases, or lower sampling-rates and thereby reduced spectrometer bandwidth. Using the imaginary inputs of a combined or splitting FFT for one half of the real input-samples has the same effect: the width of a combined or splitting FFT can be reduced to $P/2$ (Figure 3). Instead of doubling the FFT-length $Q$, pairs of intermediate channels $\tilde{a}$ from the FFT-output are transformed to the final channels, as described in Section 3.2. This transformation is similar to one stage in a pipelined FFT, but slightly more complex. According to eq. (10), this leads to a more efficient use of the FPGA-logic. Finally note, that spectral resolution is preserved, although the number of computed channels is halved, because half of the channels are complex-conjugated to the others if the FFT input is real (eq. 6). The hardware that is saved through this trick can be used to increase $Q$, allowing a better spectral resolution.

4.3. Application

A combined FFT and a splitting FFT were implemented in synthesizable VHDL together with a module to transform intermediate channels, when imaginary inputs are used. To enable flexible use in different environments these implementations are generic in FFT-width, FFT-length, and input-datawidth. The pipelined FFTs come from the Xilinx Core-Generator.

These modules are embedded into a spectrometer-design (Figure 4) including some pre- and postprocessing around the FFT: in the time-domain, WOLA is performed with sets of up to 4 input-samples. The weighting is given by an arbitrary window function, which can be programmed from the host over Ethernet. This preprocessing unit increases the sensitivity and decreases leakage-effects. Behind the FFT, power-spectra are calculated by squaring each channel’s absolute value, followed by an integration (average) over an adjustable time-period. The design is unscaled to prevent loss of sensitivity during calculation: the datawidth after each unit is equal to the datawidth before plus the logarithm of the number of summed values: log(4) bits in 4xWOLA and log($P$, $Q$) in FFT and channel-transformation unit. Two extra bits are spent to compensate window-multiplication and to prevent overflow by twiddle-multiplication in the FFT. Squaring doubles datawidth and constant 64 bits are finally reserved for each channel of the integrated power spectrum. These are converted to 32 bit float and sent to a host-PC by Ethernet. Concurrently parameters can be received by Ethernet to adjust all modules in the design.

Currently this design runs on a custom-built board (Figure 5) with a Virtex-4-SX55 FPGA. Up to 3 Gs/s of data is delivered by an A/D-Converter and received by a Spartan-3-1000 FPGA that controls all units on the board and connects to the host.

5. Results

5.1. Hardware usage

To quantify the impact on resource consumption, the different FFT-types described in Section 4 are compared with respect to the following Virtex-4 primitives: Internal Memory in Block-RAM, DSP48-Slices used as multipliers, and Slices, each containing 2 Flip-Flops and 2 Look-Up-Tables (LUT). Four FFT-implementations were instantiated with comparable sizes, inserted into the spectrometer-design shown in Figure 4 and synthesized. We compared a combined 256-pt FFT (16x16), a splitting 256-pt FFT (16x16), a splitting 128-pt FFT (8x16), using its imaginary
inputs and a combined 128-pt FFT (8x16) using its imaginary inputs. A length of $Q = 16$ is chosen, since a spectrometer with a splitting 512-pt FFT (16x32) would have exceeded the size of the SX55 FPGA. All four variants are programmed onto the FPGA on the board shown in Figure 5.

Comparing combined FFT and splitting FFT (Figure 6), mainly two effects are observed that affect the hardware utilization: The number of consumed DSP-Slices only depends on datawidth in pipelined FFTs, parallel FFT and twiddle-multipliers. Since the number of DSP-Slices grows discontinuously with bitwidth, it is misleading to generalize what FFT-type performs better. Utilization of Block-RAM and Slices is also influenced by bitwidth, but the main reason for the difference is the initial reordering in a splitting FFT, which consumes memory, registers, and logic. To conclude, a splitting FFT needs slightly more hardware than a combined FFT, but it can be split more easily over multiple FPGAs.

As predicted in Section 4.2, the hardware consumption can be reduced significantly by using the FFT’s imaginary inputs: About 40% of registers and logic is saved and about 30% of the multipliers (Figure 6), whereas the use of Block-RAMs hardly changes. A major advantage is the reduction in Slice consumption, since their utilization is most critical in this application.

The reduced hardware consumption allows improvement of multiple properties of the spectrometer. Since bandwidth and sensitivity are already optimized well, the option taken here is to increase the spectral resolution by using a FFT with more channels. When almost all Slices are used, as with the 256-pt FFTs, the number of FlipFlops and LUTs must be observed to keep Slice utilization comparable, since unrelated logic is only mapped into one and the same Slice,
when almost no Slice is left completely unused. The hardware utilization of the 256-pt FFTs (16x16, not using their imaginary inputs) is closest to that of 2048-pt FFTs (8x256) that do use their imaginary inputs (Figure 6). As eq. (10) suggests, halving the FFT-width $P=16$ allows squaring its length $Q=16$, with a comparable amount of used logic.

Since these FFTs are chosen for comparability, their sizes do not define this applications’ maxima in any case. The biggest FFT, implemented in this application so far, is a combined 8192-pt FFT using its imaginary inputs. It has 32 times more independent channels, than the biggest FFT, not using its imaginary inputs: A combined 512-pt FFT. The spectrometer including the former one utilizes almost 100% of the 24576 Slices, 84% of the 49152 FlipFlops, 77% of the 49152 LookUpTables, 70% of the 512 DSP48-Slices, and 59% of the 320 BlockRAMs in a Virtex-4-SX55 FPGA.

In conclusion, the improvement when using imaginary inputs is either 40% of the chip-logic or a squared FFT-length $Q$: A 8x256-pt FFT replaced a 16x16-pt FFT and a 8x1024-pt FFT replaced a 16x32-pt FFT.

5.2. Spectrometer

The clock-rate Xilinx-ISE [5] achieves for a design dramatically degrades with a growing number of occupied Slices. Although Xilinx-ISE calculates a maximum clock-rate of $1333/8 = 166.7$ MHz, the spectrometer has been successfully overclocked to $1800/8 = 225$ MHz. Still the usual clock-speed is $1500/8 = 187.5$ MHz for reliable operation. A spectrometer with 8192 channels and 1.5 GHz bandwidth has now been implemented successfully, based on a combined 8192-pt FFT with imaginary inputs used. Input-data is preprocessed by 4xWOLA, weighted by a Flat-Top-Window and about 180,000 single spectra are integrated over 1 second.

6. Conclusion and Further Work

The test spectrum, shown in Figure 7, demonstrates, that the described technique and its FFT implementation work well in practice on our prototype spectrometer card.

Our spectrometer is fed by an ADC with up to 3 Gs/s, and thereby produces a spectral bandwidth of 1.5 GHz. This is significantly wider than other operational digital spectrometers currently produce. In our implementation, the 8-bit input data samples grow completely unscaled through the design to prevent any loss of precision and therefore sensitivity. Under this condition, the implementation of a combined 8192-pt FFT, using imaginary inputs, fits on a Virtex-4-SX55 FPGA, and can produce 32 times more channels than if only the real input was used. Time-domain data can be preprocessed by up to 4xWOLA and any window, programmed from a host-PC. The integration over adjustable time-periods leads to a final 64 bits channel size. Finally the channels are converted to 32 bit float-values and the data is transferred to the host-PC over Ethernet - a standard that guaranties compatibility over large time-periods.

Although the performance of our fully functional spectrometer already exceeds that of others, we plan to further improve both, its bandwidth and spectral resolution.

A currently announced ADC with 5 Gs/s, in combination with a Virtex-5 FPGA, could provide a significantly higher bandwidth. Both components shall be placed onto a slightly modified version of our existing, custom-built board. Since doubling the bandwidth requires double parallelization, the design is expected to reach the limits of a Virtex-5-SX95T, preventing an increase of the number of channels.

One way to increase the channel number would be to split the task between several FPGAs. We have explored this direction with another custom-build prototype board.

Our future work will concentrate on reducing the hardware consumption in a FPGA. A large hardware savings potential lies in reduced precision. We shall quantify how bitwidth affects sensitivity for data as well as twiddlefactors in different parts of the design.

References

The ABB NoC – a Deflective Routing 2x2 Mesh NoC targeted for Xilinx FPGAs

Johnny Öberg\textsuperscript{1}, Roger Mellander\textsuperscript{2}, Said Zahrai\textsuperscript{2}

\textsuperscript{1}Royal Institute of Technology (KTH), Dept. of Electronics, Computer, and Software Systems
johnny@imit.kth.se

\textsuperscript{2}ABB Corporate Research AB,
{roger.mellander, said.zahrai}@se.abb.com

Abstract

The ABB NoC is a 2 by 2 Mesh NoC targeted for Xilinx FPGAs. It implements a deflective routing policy and is used to connect four MicroBlaze processors, which implement an area- and timing-critical multiprocessor embedded real-time system. Each MicroBlaze is connected to the NoC via a Network-Interface (NI) that communicates through a Fast Simplex Link (FSL) interface together with Block RAM (BRAM) memories that implement a shared memory between the NoC and the resource.

Application programs use device drivers to communicate with the Network-Interface of the NoC. The NI sets up message transfer, with a maximum length of 2040 bytes, and sends flits with the size of 32 bit data plus 11 bit headers through the network.

The design has been implemented on Xilinx Virtex2P and Virtex4 FPGAs. The NoC design has a throughput of 200 Mbps, is about 2600 slices large and operates at 111 MHz in the Virtex2P technology and 125 MHz in the Virtex4 technology.

1. Introduction

An industrial control system is often divided into subsystems that interact and together provide the required functionality. Dividing a system into separately developed subsystems provides additional flexibility, in particular when standard components are used. As an example, one may consider a machine controller with several layers of intelligent component with powerful PLCs at the highest level and small embedded systems for sensing and actuation at the lowest level.

In the competitive world of industrial markets, reduced size, cost and power consumption are important goals or constraints in development projects. Integration of the closely related components often gives positive effects but might reduce the flexibility and scalability of the system. The use of FPGAs can be a possible path into highly integrated systems on a singe chip which, from an architectural point of view, can be considered as large systems consisting of separate subsystems.

Each of these subsystems might contain one or a number of IP-cores and CPUs.

An industrial control system used for control of machines and processes is usually composed of different nodes with variable computing capacity responsible for executing different tasks in the system. Communication between these nodes occurs through different types of communication links and protocols such as proprietary ones, Ethernet and field buses. Similarly, a SoC implementation of such a system using a number of processors on the same chip relies on suitable choice of communication interfaces, such as FIFOs, memory-mapped links or buses, each of which with its strengths and weaknesses.

Bus based platforms suffer from limited scalability and poor performance for large systems [1]. This has led to proposals for building regular packet switched networks on chip as suggested by Dally, Sgroi, and Kumar [1, 2, 3]. These Network-on-Chips (NoCs) are the network based communication solution for SoCs. They allow reuse of the communication infrastructure across many products thus reducing design-and-test effort as well as time-to-market. In this respect, NoC provides an attractive alternative where a number of units are connected in a standard way.

The aim of this paper is to present implementation and evaluation of a NoC used in a SoC in an industrial application. Ethernet connections (100 Mbps) and shared memories of the original design should be replaced with a NoC system that handles data transfer through a DMA technique. The resource software will typically send messages over the NoC at a regular interval and the maximum length of a message is 1500 bytes. For compatibility to earlier systems, the design should be implemented using Xilinx Virtex2P (or later) technology platform. To be able to obtain sufficient real-time performance, the processors need to run at least at 100 MHz, with the support of hardware accelerators, or as fast as possible, preferably at higher speed than 150 MHz, to be able to run without hardware accelerators. Thus, the application is both area- and timing-critical.
2. The ABB NoC

The ABB NoC is a 2 by 2 Manhattan style 2-D Mesh NoC. It is based on the Nostrum NoC Concept [4..9] developed by the Royal Institute of Technology, KTH, in Stockholm. It is used to connect four MicroBlaze [13] embedded processor systems together.

The NoC consists of four three-port switches, see Figure 1. The two ports A and B are connected to neighbouring switches and the remaining port R is connected to the Network-Interface of the resource. Each port supports full duplex transmissions, i.e., they are built using two links, one in each direction, to make it possible to transmit packets in both directions simultaneously.

Each MicroBlaze is connected to the NoC via a Network-Interface, consisting of a controller, two Fast Simplex Links (FSL-links) [12] and two Block RAM (BRAM) memories [11], situated on the Local Memory Bus (LMB-bus) [10] of the MicroBlaze processor. The FSL-link is used to setup the NoC-Frame transmission, while the two BRAM memories functions as a write and a read buffer, respectively. To connect the ABB NoC to the four MicroBlaze systems, eight BRAMs together with eight eight FSL links per resource. To be able to reuse the device drivers, the FSL-link is connected to the same port (FSL port 0) and the BRAMs have the same memory map on all resources. A Block Diagram of the connection is shown in Figure 2.

Switch Architecture

The Switch is implemented using a data path and a controller, as shown in Figure 3 below. The controller can be further split into a Decision control algorithm and a four state control FSM which determines the advance of the deflection algorithm and the signalling to/from the NI.

The input ports are connected to one input buffer each. From the input buffers, the valid bit, the UD, LR and HC counters are extracted and forwarded to the decision controller. The input buffers are connected to three 4-to-1 muxes, one mux for each output port A, B, and R. The decision controller investigates the priority of each message, decides which packet should go to which output and sets the select signal accordingly. A packet with both UD and LR counter zero will be output on port R. A packet with either UD or LR non-zero will be routed to port A or B. A positive value means that the packet should be sent downwards/rightwards in the NoC. A negative value means that the packet should be sent upwards/leftwards in the NoC. The UD and LR counters are updated accordingly.

If no packet should be output, an empty packet with the valid bit set to '0' is inserted at the output stream. If only one port has a valid packet, that packet immediately gets its desired port. If two packets are competing for a port, the one with highest priority selects first. The default priority order is A, B, and then R. In the case of port A and port B, the port with the oldest packet selects first, i.e., the one with the highest HC value is allowed to select an output port first. If port A and port B has the same HC, port A selects first.
The switch controller is a four clock cycle FSM. It sets five signals: the input and output buffers’ load enable signals, the write_R and read_R signals to communicate with the NI, and when to output the mux select signal. The deflection algorithm is implemented to calculate the switch decision in state S0 and S1, so a new Select value is output at state S3 every four clock cycles. If a packet should be written to the NI, the Write_R signal is set to 1 in state S2 and S3. If a packet has been read by the switch, the Read_R signal is set to 1 during state S2, and the Load_Enable signal to 1 during state S3.

The Network Interface (NI) forms the link between the switch and the resource. In the ABB NoC, the NI translates FSL messages from the MicroBlazes to the NoC and vice versa. It uses two BRAMs together with their BRAM controllers, to work as a read and write buffer for the NoC. The read and write buffers are implemented as a two-ported shared memory, with one end connected to the LMB Bus of the MicroBlaze and the other to the NI.

The translation is done by two FSMs that work independently of each other. One FSM translates FSL messages from the MicroBlaze to NoC packets (FSL2NoC) while the other stores incoming NoC packets in the receiver BRAM (NoC2FSL). The block diagram is shown in Figure 4.

The FSL2NoC FSM has five states and a delay counter. The NoC2FSL FSM has four states to match the four states of the switch. The two FSMs are shown in Figure 5 and 6.

The NoC2FSL FSM waits in the idle state until there is a valid data on the NoC interface (indicated by Write_R=’1’). Upon data valid, the FSM goes to the Setup Data state in the case that the incoming message was a data setup word. If not, it goes to the Write Data state. In the Setup Data state, the Write Buffer Address pointer for that sender is reset and an FSL message for that buffer is prepared and stored. In the Write Data state the received data is stored at the next free position in the BRAM. In case it was the last data word for that message, an interrupt signal is asserted, indicating that this write buffer has stored its last data. Both the Setup Data State and the Write Data State then goes to the Wait for Write_R state, where the FSM waits until the Write_R signal from the NoC Switch goes low. When that happens, the Write Buffer Address is decremented by one and the FSM then goes back to the idle state to wait for the next data word.

The FSL2NoC waits in the idle state until there is data on the FSL Slave Interface. It then reads the data and initiates a message transfer by sending a Setup Word to the intended target. Sending involves waiting in the Wait for Read_R state until the NoC Switch sends a Read_R acknowledging that it has read the data. It then goes to the delay state to wait twelve clock cycles. The delay is used to guarantee that packets arrive in order at the receiving end. It then proceeds to read another data from the input BRAM. When all packets have been sent, the sender goes back to the idle state.

The NI initiates the transfer, and sends one data word every 16th clock cycle until all data has been transferred. For a 2 by 2 switch, this means that all packets will arrive in order, since a deflected packet will have enough time to arrive at the destination before another packet will arrive. Thus, Quality-of- Service (QoS) is maintained. For a larger network, this property does not hold. Then a re-ordering mechanism has to be implemented.

3. Protocol Stack

The protocol stack for the ABB NoC is outlined in Figure 7. Application programs call device driver
functions that setup a communication message. The Network interface then sets up a communication link with the receiving node, and starts to transmit the data. Data is transmitted over the NoC, one flit at a time.

<table>
<thead>
<tr>
<th>Application Program</th>
<th>Device Drivers</th>
<th>Network protocol</th>
<th>Switch Protocol</th>
</tr>
</thead>
</table>

**Figure 7. ABB NoC Protocol Stack**

**Device Driver prototypes**

To access the NoC services, four C-language primitives have been developed that can be used as Device Drivers to the NoC. They are named:

- `int flush_buffer_to_noc(coord node, int buffer, int num_words)`
- `int get_noc_msg(void)`
- `int get_write_buffer(int val)`
- `int get_recv_buffer(int val)`

The data type coord is the absolute coordinate of the target node in the system, e.g., the value `{0,0}` correspond to node 0, `{0,1}` to node 1, etc. It is defined as:

```c
typedef struct {
    int row;
    int col;
} coord;
```

The MicroBlaze processor in the resource is using the Fast Simplex Link protocol [12] to communicate with the NI. In addition, the MicroBlaze is connected to two two-port Block RAMs (BRAMs) [11], where the other port is connected to the NI, that are used as a read and a write buffer, respectively. In order to avoid confusion, an explanation regarding the naming convention is in place. The BRAM that the NoC is reading, is written by the corresponding MicroBlaze resource. Thus, from the software point of view it is a Write Buffer, whereas from the NoC point of view it is a Read Buffer. The same goes for the NoC Write Buffer, which is viewed as a Read Buffer by the software.

The function `flush_buffer_to_noc()` takes the MicroBlaze write buffer number (0 or 1) and the length of the message in number of 32-bit words as arguments. It sets up an FSL message to the NoC and initiates the transfer by issuing a blocking write to the FSL.

The function `get_noc_msg()` performs a blocking read from the FSL port. It returns a value that the NoC NI has put the FSL-link. The layout of the received message is shown in Figure 8.

**Figure 8: Bit interpretation of NI-to-NoC data word**

The function `get_write_buffer()` takes the write buffer number (0 or 1) as a parameter and returns a pointer to the base address in the memory of the corresponding write buffer.

The function `get_recv_buffer()` takes the recv buffer number (0, 1, 2, or 3) as a parameter and returns a pointer to the base address in the memory of the corresponding receive buffer.

**Network protocol**

A write to the NI from the MicroBlaze processor is interpreted according to Figure 9 below. The NI takes the targets absolute position, counted from the zero reference – the Upper Left (UL) node in the upper left corner, and converts it to a relative address that can be used by the switch by subtracting its own position in the network from the target’s address. The Length of message is extracted and stored in a counter. The counter together with the message buffer number form the offset in address space used to retrieve data from the send buffer.

**Figure 9: Bit interpretation of NoC message setup command word**

The NI responds by sending a message frame of data, outlined in Figure 10 below. The message frame in the ABB NoC is composed of 32 bit words. It is composed of two setup words plus the number of data words.

**Figure 10: The NoC Message Frame**

The first data word that is transmitted is a setup message that initializes the receiving node with the number of data words to expect plus the first nine bits of the global clock. The second word is the remaining 32 bits of the global clock. The Global Clock is a 41 bit counter that counts microseconds since the last reset. It counts 12.75 days before it wraps around. It is padded to all messages that are sent in the NoC. It is used to keep track of when things are happen, in case something goes wrong. Since the global clock is added to all messages, a local resource can always retrieve it by sending a zero-length message to itself. After the global clock has been transmitted, the message itself is transmitted, MSWord first, LSWord last.

The NI uses the sender source ID to determine which write buffer that should be used to save the
incoming message. Write buffer 0 (ID 00) has the address 0x0000A000, Write buffer 1 (ID 01) 0x0000A800, Write buffer 2 (ID 10) has address 0x0000B000, and Write buffer 3 (ID 11) 0x0000B800. The maximum length of a message is 2040 bytes. The remaining eight bytes (two words) are used for the setup word and global clock word that is padded to every message.

**Switch protocol**

For every data word, or flit, from the MicroBlaze, the NI adds a header. The header is 11 bits long and data is 32 bits long. The header is interpreted by the switch to determine where the packet’s destination is and how it should be routed. The header is shown in Figure 11 below. The receiving NI uses only two fields from the header, the data type field and the message source field. The rest of the header is discarded upon arrival.

<table>
<thead>
<tr>
<th>Bit position</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Data type</td>
</tr>
<tr>
<td>9</td>
<td>Source ID</td>
</tr>
<tr>
<td>8</td>
<td>Hop Counter (HC)</td>
</tr>
<tr>
<td>7</td>
<td>Up/Down Counter (UD)</td>
</tr>
<tr>
<td>6</td>
<td>Left/Right Counter (LR)</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 11: NoC Header contents**

The data type field is two bits wide. The first bit is the data valid bit that is used by the switch to determine if the data is a valid packet or not. The NI uses the second bit in the data type field to determine the type of data. If it is a ‘1’, it is a message setup data field. If it is a ‘0’, it indicates a normal data field. During Word 0, the NI transmits a Setup Data. During the rest of the transmission, the NI transmits Normal Data (the second word of the global clock is handled as global data).

The second field is used by the NI. It is the message source field. It indicates which data source the message comes from and to which write buffer the NI should save data. This field is needed since multiple transmissions from different sources can occur simultaneously. To ensure that messages do not get scrambled in the case of multiple incoming simultaneous messages, each source gets its own input queue.

The Hop Counter (HC) field is used by the switch to determine the priority of the messages. It is incremented by one for every switch that the data has visited. Thus, the message gets a higher priority the older the message is. This mechanism ensures that data does not get stuck in the NoC in the case of a contention.

The Up/Down (UD) and the Left/Right (LR) counter fields are the relative address fields.

4. **Experiments and Results**

**Simulation, Validation and Prototyping**

In order to verify the functionality, two type of simulations were performed. One functional simulation on the NoC and its interfaces, and one structural on the whole design. An iterative bottom-up verification methodology was used. First the NoC was verified that it functioned as it should, then the Network-Interface was added and simulated.

In the second phase, the NoC was verified together with its resources in a structural VHDL simulation of the whole system. The simulation run the first 40 us of the system at start up. The GPIO signals were used to “print” out software status messages on the simulation window. This was necessary to verify that the device drivers worked properly together with the MicroBlazes.

After debugging, the system was working satisfactory within the given parameters. The system was downloaded to a prototype board to verify that it worked in accordance with the structural simulations.

**Synthesis**

From synthesis point of view, several experiments were conducted. The goal with the design was to make it as fast as possible and achieve at least 100 MHz speed on a Xilinx FPGA. The NoC itself was synthesized first for a Virtex2P with 1152 pins. It gave an area of 2606 Slices. The estimated frequency was however only 94.85 MHz after several rewritings of the code to make it run faster. Since the number of I/Os for the NoC itself is also more than an FPGA can accommodate, it was not possible to get a proper timing estimate after place & route.

Instead, a simple prototype system, composed of four MicroBlazes resources, where each resource was connected to its own GPIO interface, was designed and synthesized. The system was tested on a prototype board, running at 80 Mhz, and was found to work satisfactory. However, speed was still a problem. After some investigations, the cause of the problem turned out to be the pinning of the prototype board. The prototype board has fixed pinning for its I/Os, so if all resources should have at least two pins connected to the LEDs on the prototype board, it is impossible to achieve a higher speed since at least one of the resources would be placed at the other end at the chip compared to the location of the pin connected to the LED.

With a careful placement of I/Os, it was possible to meet the timing constraints and achieve 100 MHz speed. It was also found that this timing was the same as if the design had been synthesized without any placement constraints on the pinning at all (auto-placement of pins). The placement constraints were therefore removed.

<table>
<thead>
<tr>
<th>Table I: Area and Speed for 100 MHz constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
</tr>
<tr>
<td>NoC</td>
</tr>
<tr>
<td>2x2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>System</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
The system was then synthesised for the Virtex4 technology from Xilinx. The results of the synthesis are shown in Table 1. The NoC now shrunk slightly to 2575 slices, and the speed was improved significantly, with 24%.

To find out what the highest obtainable speed was, the system was synthesized with several different speed constraints. The results are summarised in Table 2. For a Virtex2P system, the maximum speed was 111 MHz and for a Virtex4 system 125 MHz.

<table>
<thead>
<tr>
<th>System</th>
<th>Virtex2P (vp2xc30)</th>
<th>Virtex4 (v4fx60)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>8.982 ns</td>
<td>7.996 ns</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>111.3 MHz</td>
<td>125.1 MHz</td>
</tr>
</tbody>
</table>

Since the NI emits a packet every 16\textsuperscript{th} clock cycle, and data is 32 bits wide, this means that the maximum communication speed is two bits\*clock rate, i.e., for a clock speed of 100 MHz, the transmission rate would be 200 Mbps. This is twice the speed of the Ethernet connections in the original design. For the 111 MHz and 125 MHz implementations, the communication speed is 222 Mbps and 250 Mbps, respectively.

### 5. Summary and Conclusion

The ABB NoC is a 2 by 2 Mesh NoC. It is using Deflective routing as a switch protocol and it is targeted for implementation on Xilinx FPGAs. It connects four MicroBlaze processors together that implement an area- and timing-critical Real-time Embedded System. The MicroBlaze is connected to the NoC via a two-way FSL interface together with two two-ported BRAMs that implements a shared memory functions as a read and write buffer respectively.

The NoC transmits messages that are maximum 2040 Bytes long, i.e., 510 words. The remaining eight bytes are reserved for the global clock. The global clock counts microseconds since the last reset. It is used to enable a possibility to trace messages in case something should go wrong.

Data words are transmitted every 16\textsuperscript{th} clock cycle. Every switch in the NoC has an FSM controller that switch a data word every 4\textsuperscript{th} clock cycle. The receiving Network-Interface is capable of receiving a message data in synchrony with the switch, i.e., every 4\textsuperscript{th} cycle. A transmitted data word is 32 bits long. Thus, the maximum communication speed is two bits\*clock rate, i.e., for a 100 MHz clock system the transmission rate is 200 Mbps.

The NoC occupies 2600 slices and runs at 111 MHz for the Virtex2P and at 125 MHz for the Virtex4 technology.

The NoC fulfills the basic needs of ABB, i.e., it provides a standardized interface with a predictable behaviour that a resource IP designer can take into account when designing the IP. The size of the NoC was a bit large, not in terms of area, but in terms of the memory needed to implement the buffers. The way the buffers were implemented limits the scalability of the system. For larger NoCs, a buffering scheme that occupies less memory is needed, perhaps the custom-based FIFO solution that [14] is advocating, together with a sorting unit to avoid re-ordering of packets. ABB is also interested to see how the NoC structure can be used to implement fault-tolerant designs. This will be done in the future.

### 6. References


Call for FPGAworld Conference 2009

Academic/Industrial Papers, Product Presentations, Exhibits and Tutorials

September 2009, Stockholm, Sweden
September 2009, Lund, Sweden (no academic)

The submissions should be in at least one of these areas

- DESIGN METHODS - models and practices
  - Project methodology.
  - Design methods as Hardware/software co-design.
  - Modeling of different abstraction.
  - IP component designs.
  - Interface design: supporting modularity.
  - Integration - models and practices.
  - Verification and validation.
  - Board layout and verification.
  - Etc.

- TOOLS
  - News
  - Design, modeling, implementation, verification and validation.
  - Instrumentation, monitoring, testing, debugging etc.
  - Synthesis, Compilers and languages.
  - Etc.

- HW/SW IP COMPONENTS
  - New IP components for platforms and applications,
  - Real-time operating systems, file systems, internet communications.
  - Etc.

- PLATFORM ARCHITECTURES
  - Single/multiprocessor Architecture.
  - Memory architectures.
  - Reconfigurable Architectures.
  - HW/SW Architecture.
  - Low power architectures.
  - Etc.

- APPLICATIONS
  - Case studies from users in industry, academic and students
  - HW/SW Component presentation.
  - Prototyping.
  - Etc.

- SURVEYS, TRENDS AND EDUCATION
  - History and surveys of reconfigurable logic.
  - Tutorials.
  - Student work and projects.
  - Etc.

www.fpgaworld.com/conference