RAPID GENERATION OF HARDWARE FUNCTIONALITY IN HETEROGENEOUS PLATFORMS

Darren Reilly[1], Roger Woods[1], John McAllister[1] and Richard Walke[2]

[1]School of Electrical and Electronic Engineering, Queen's University Belfast, UK
E-mail: {darren.reilly, r.woods, jp.mcallister}@ee.qub.ac.uk

[2]Real Time Embedded Systems, QinetiQ Ltd., Great Malvern, WORCS, UK
E-mail: walke@signal.qinetiq.com

ABSTRACT

One of the key problems in complex digital system design is the rapid generation of efficient hardware functionality. The paper introduces an architecture template for targeting FPGA implementations as part of a dataflow based design flow for heterogeneous platforms, thereby allowing a designer to perform system level optimizations for consistent FPGA performance. The architecture provides scalable capabilities in both communications and processing allowing the core to be scaled to the problem size. Matrix multiplication is used to demonstrate the capabilities of this methodology giving speeds ranging from 121.4MHz to 188.3MHz without optimization.

1. INTRODUCTION

System requirements are becoming very complex due to silicon technology improvements and user demands and so there is a growing need to move to a higher level of abstraction to model system functionality. Implementing systems on heterogeneous platforms is a difficult and complex task. Issues to be addressed include effective system partitioning and communications both of which can have implications on buffering and communication needs.

A major collaborative project involving QinetiQ, BAE Systems and Queen's University Belfast [1] is looking at system level design flows for FPGA/microprocessor based systems. FPGAs are particularly suited to computationally intensive algorithms such as matrix multiplication and QR decomposition because of their ability to realize high levels of parallelism Dataflow has been chosen as the model of computation (MOC) to capture system functionality as it has the capability of describing the functionality of a system without the need for detailed implementation or timing detail. It provides a suitable starting point at which these details can be added through the design flow. A tool which has been gaining popularity as a means of performing multi-processor implementations from a dataflow description is GEDAE [2]. GEDAE's main strength lies in partitioning and generating schedules for multiprocessor platforms but there is no systematic flow to go from GEDAE into programmable hardware. Providing this functionality rapidly and efficiently is a complex and time consuming task.

This paper examines a technique for the rapid generation of efficient hardware functionality for FPGA as part of a system level design flow. An architecture template is proposed that allows high level characteristics to be captured but which also allow algorithms to be efficiently implemented. For this reason a counter based controller has been used which allows recursive algorithms in the form of a nested loop program (NLP) to be directly mapped to hardware. Compaan [7] uses this technique and can be used in conjunction with this architecture to provide transformations of loops to vary memory usage in a core.

Architecture templates [8] have been demonstrated before but they are not suitable for truly high-level exploration as they operate synchronously and it is very difficult to utilize these types of structures as synchronous scheduling of a complete system on heterogeneous platform is very complex. Fixed architecture templates have been developed including the Imagine Processor [9] which is targeted mainly at image processing and the picoArray [10] which is mainly targeted towards 3G cellular base stations. However, these templates have largely pre-fixed architectures whereas the work here tends to develop the circuit architecture based on the computational requirements of the algorithm under consideration thereby providing the best match between the hardware realization and system functionality.

The paper is structured as follows. In Section 2, we introduce the basic concepts of dataflow for modeling

systems and its implementation in hardware. In Section 3, we look at an example NLP to derive the requirements for an architecture template which can be used in a system level flow. Section 4 shows how matrix multiplication can be mapped to the template in various ways and Section 5 provides results for the various implementations in terms of speed and area.

2. DATAFLOW AND HARDWARE MODEL

Dataflow has been generating considerable interest due to its capabilities in modeling systems at a high level. It allows the user to capture functionality and provides a suitable starting point for a system level flow. Dataflow works on tokens of data which can be single scalars or multidimensional arrays of data. The dataflow model consists of actors and infinite queues by which actors communicate. The actor fires or processes data whenever its prerequisite conditions have been met, e.g. when all inputs have at least one token of data. Once fired, the tokens on the inputs are consumed and tokens produced on the outputs. An example dataflow graph (Fig. 1) shows the state of the graph before and after firing.

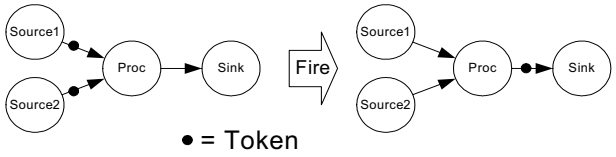

Fig. 1: Dataflow Graph before and after firing

In the example shown only 1 token resides on any input or output. However, in large and complex systems, multiple tokens may reside in the queues especially when increasing overall system performance.

Hardware implementation of dataflow graphs is illustrated by the generic hardware model in Fig. 2. As can be seen, it places restrictions on the graph such as finite queue sizes. The generic hardware model can be used for all hardware actors in a graph. The purpose of the controller is to schedule the data in the FIFOs onto the processor and to also check for firing conditions.

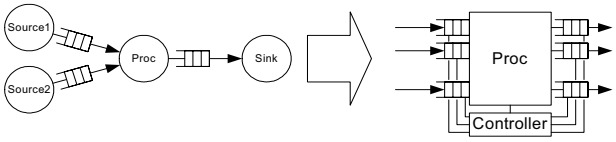

Fig. 2: Dataflow Model to Hardware Model

FIFOs normally operate on the basis that when the active element is read from the FIFO, it is then disregarded. However this is not sufficient for our hardware model. It is necessary to provide this 'normal' functionality when handling tokens but tokens can contain multiple elements of data. Therefore a mechanism is needed to allow multiple and random access to elements within the active token in the FIFO before disregarding the complete token when it is no longer required. This requires extra functionality within the FIFO so that elements within the active token can be accessed as dictated by the algorithm and that the token is only disregarded when the processor has finished firing. Fig. 3 illustrates how this can be achieved. This represents some of the detailed challenges that needs addressed when mapping dataflow to hardware.

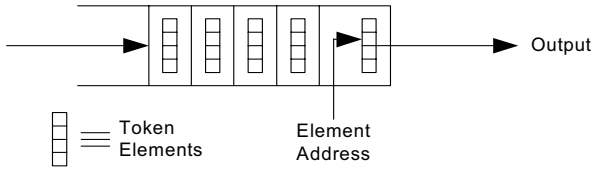

Fig. 3: Token FIFO with Addressable Elements

3. ARCHITECTURE TEMPLATE DERIVATION

Many DSP algorithms are recursive and can be described in a (nested) *for* loop structure, such as QR decomposition and many matrix operations such as matrix multiplication. This structure can also be implemented in hardware using counters and suitable control hardware. In the nested loop example of Fig. 4, the program has been rearranged to separate the control and processing with a processor outline identified.

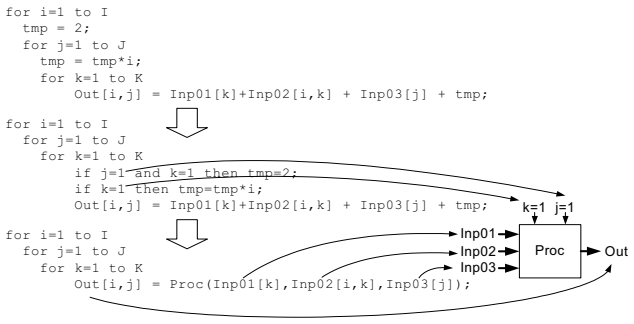

Fig. 4: Example NLP

To generate the control circuit for this, the *for* loops are mapped to cascaded counters which generate the same sequences as the loops in the program. The upper and lower counter bounds are the same as those of the *for* loop program. This can be seen in Fig. 5.

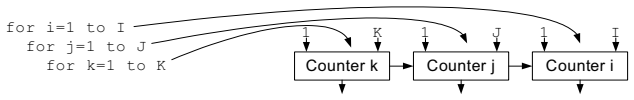

Fig. 5: Example Control Generation

From Fig. 4, it can be seen that control signals are required when *k=1* and *j=1*. The addresses for input and

output memories are generated from the *i*, *j* and *k* counters. Latencies are inherent in processing and so control signal delays need to be generated correctly to allow correct scheduling of data. The control structure (Fig. 6) simplifies the mapping of loops to the counters. The processor control signals are generated as needed, and the addresses generated from the relevant counters. Each generated signal passes through a parameterisable delay so that they can be easily balanced against the operator and memory read latencies.

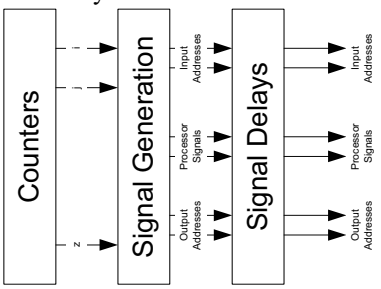

Fig. 6: Controller Structure

Given the dataflow model system description, interfaces need to be well defined and flexible. This is done by treating data at the token rather than element level and using a produce/consume mechanism to indicate token state. As flexibility is also required for data transfer between actors so that performance can be increased if necessary, multiple element transfer is supported. By providing this interface, actors in the dataflow graph can be mapped to processors generated using this architecture template. Higher performance levels may be achieved by increasing the number of processors and communication bandwidth.

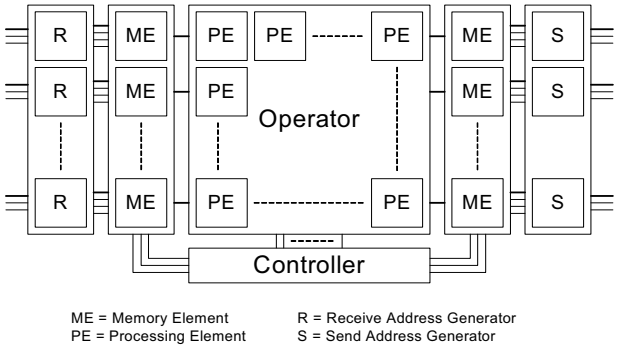

Fig. 7: Proposed Architecture Template

The proposed architecture template (Fig. 7) provides scaling in both communications and hardware allowing it to be configured, scaled and refined for efficient use of resources. System-level characteristics such as an asynchronous interface are captured which also can be scaled to the problem size (typically a simple change in controller counter boundaries which can be made at run-time). The matrix multiplication algorithm is mapped to the template architecture in the next section.

4. MATRIX MULTIPLICATION EXAMPLE

The NLP form of the matrix multiplication algorithm is given in Fig. 8. This also shows how the program is simplified by extracting the processing details before further manipulations take place. This simplifies the mapping process to the controller counters and extracts control signals required for the processor.

```
for i=1 to 8
  for j=1 to 8
    for k=1 to 8
      if k=1 then C[i,j] = 0;
      C[i,j] = C[i,j] + A[i,k]*B[k,j];
```

⇨

```
for i=1 to 8
  for j=1 to 8
    for k=1 to 8
      C[i,j] = Proc(i,j,k);
```

Fig. 8: Matrix Multiplication Algorithm

From Fig. 8, the code relative to the processor can be identified and the processor derived as shown in Fig. 9. It can also be seen that a control signal for the mux is required which happens when *k*=0. The *for* loops in the algorithm can be emulated by counters with the same bounds providing the basis for the controller in Fig. 10.

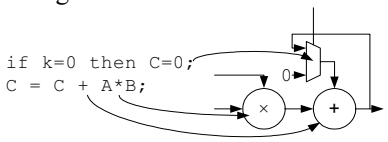

Fig. 9: Processor Derivation

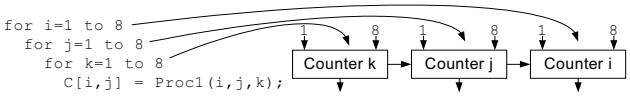

Fig. 10: Mapping to counters for 1 processor

Whilst this shows the mapping for one processor, it is possible to manipulate the loops in the program to extract extra levels of parallelism (Fig. 11). It is also indicated how the loops are mapped to the counters which can then be optimized by removing some of the excess counters, given that some of them have fixed values and others realize the same behavior. The addresses for memories are generated from the counter values. The mux control signal is generated using a comparator when $k = 0$.

5. RESULTS

A number of matrix multiplication implementations were carried out including a small (8x8x8), large (4x512x4) and an irregular (13x37x16) implementation. The latter is used to illustrate the flexibility of this approach. Table 1 below shows the circuit speeds and resource usage for Xilinx Virtex II XCV6000, when mapping to a single processor. The circuit was synthesized using Synplify Pro

7.2.1 and implemented using Xilinx ISE 6.1. Where no BlockRAMs were used as the memories required were small, LUTs were used configured as 16x1 RAMs to implement the memories.

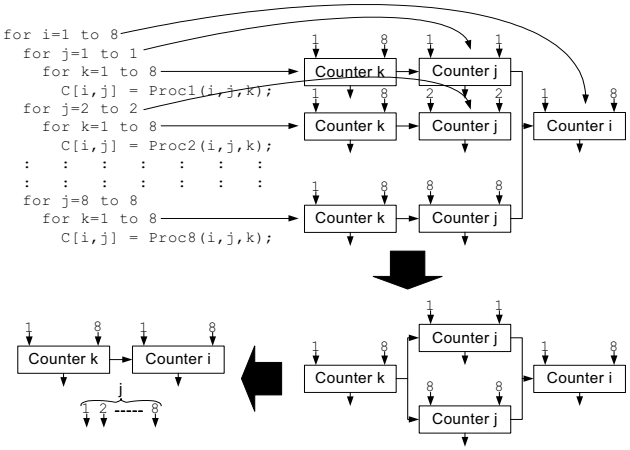

Fig. 11: Mapping to counters for 8 processors

M	N	P	Clk (MHz)	Matrix Mults/s	LUTs	Block RAMs
8	8	8	188.3	367,773.44	331	0
13	37	16	135.1	17,550.01	547	3
4	512	4	153.9	18,786.62	355	9

Table 1: Performance Results with 1 processor

M	N	P	Clk (MHz)	Matrix Mults/s	LUTs	Block RAMs
8	8	8	165.0	2,578,125.00	1473	0
13	37	16	121.4	126,195.43	877	3
4	512	4	139.3	13,035.16	1522	9

Table 2: Performance Results with 8 processors

Table 2 show the speed and resource usage when the circuit was implemented with 8 processors. The largest LUT usage (1522) represents 2.25% of the FPGA area. It can be seen from the results that the clock speeds drop with larger matrices. For (13, 37, 16), the clock frequency and resource usage is worst due to the extra resources needed for the irregular conditions. Numbers rounded to a power of 2 will provide much more efficient results, so it would be possible to pad out the matrices with zeroes which whilst wasting clock cycles, will result in a higher throughput due to the higher clock speed.

Many implementation of matrix multiplication exists [12], [13], [14] but they are all synchronous and the communications required for system integration is not presented. This requires extra area so a direct comparison is not relevant although clock speeds are of the same order. Considering that no optimizations have been carried out, the results indicate that the use of a template provides a controlled and efficient implementation of functionality for a dataflow graph.

6. CONCLUSIONS

An architecture template to develop cores from dataflow descriptions has been presented. It provides a controlled way of dictating the architecture and therefore a predetermined performance in terms of area and speed. Currently a limitation exists in the architecture that only allow the processors to access one token at the one and same time. The work will be expanded to address this as it will have major advantages in load balancing a network. The flow will also be further enhanced by linking with tools such as IRIS [11] and Compaan [7] to allow rapid implementation on FPGA.

7. REFERENCES

[1] J. McAllister, et al. "Design Technologies for DSP Algorithm Implementation on Heterogeneous Architectures", *Proc. SPIE Advanced Signal Processing Algorithms, Archs, and Impl. XIII*, vol. 5205, pp. 585-596 August 2003.

[2] W. I. Lundgren, "Out of the Graphical Box – from DSP Function to Implementation", *IEE FPGA Developers Forum*, London, UK, 21-22 October 2003.

[3] Handel-C: Available at http://www.celoxica.com.

[4] SystemC: Available at http://wwwsystemc.org.

[5] Synopsys CoCentric SystemC Compiler: Available at http://www.synopsys.com.

[6] Catapult C: Available at http://www.mentor.com.

[7] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere, "Compaan: Deriving Process Networks from from Matlab for Embedded Signal Processing Architectures", *In Proc. CODES*, San Diego, CA, USA, May 3-5 2000.

[8] Benkrid K., Crookes D. "From application descriptions to hardware in Seconds: A logic-based approach to bridging the Gap", *IEEE Trans. VLSI*, vol. 12, no. 4, pp. 420-436, Apr. 2004.

[9] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, Brucek Khailany, "The Imagine Stream Processor", *Proc. IEEE Int'l Conf. on Computer Design*, pp. 282-288, September 2002.

[10] picoArray: http://www.picochip.com.

[11] D. Trainor, R. F. Woods and J. V. McCanny, "Architectural Synthesis of Digital Signal Processing Algorithms using IRIS", *Journal of VLSI Signal Processing*, vol. 16, no. 1 pp. 41-56, May 1997.

[12] Prassana Kumar V. K., Tsai Y.: "On Synthesising Optimal Family of Linear Systolic Arrays for Matrix Multiplication", *IEEE Trans. Comput*., vol. 40, no. 6, pp. 770-774, June 1991.

[13] I. V. Ramakrishnan an P.J. Varman, "Modular Matrix Multiplication on a Linear Array", *IEEE Trans. Comput.*, vol. C-35, no. 11, pp. 952-958, 1986.

[14] A. Amira, F. Bansaali, "An FPGA Based Parameterisable System for Matrix Product Implementation", *IEEE Workshop SIPS*, San Diego, California, pp. 75-79, October 2002.