

Architectural Exploration Using GEDAE for Heterogeneous Architectures

John McAllister, Kevin Colgan, Roger Woods
Queen's University Belfast,
Belfast, UK
BT9 5AH
+44 (0)28 274275
`{j.p.mcallister,r.woods}@ee.qub.ac.uk`

Richard Walke
QinetiQ Real Time Systems Laboratory (RTSL)
St. Andrews Road, Great Malvern
Worcs., UK, WR14 4XB
+44 (0)1684 895968
`r.walke@signal.qinetiq.com`

ABSTRACT

A hardware design and architectural exploration flow, with GEDAE as the entry point, is presented. SystemC bridges the gap between task level specification and RTL hardware implementation, allowing hardware components to be created in a one-to-one mapping between task and processor. Design constraints may require component selection from a range of implementation possibilities, varying primarily in throughput, hardware resource usage and latency. Architectural manipulation tools such as IRIS can be used to produce these using architectural synthesis. This paper discusses creation of a range of architectural solutions for the case of a Finite Impulse Response as required by many signal processing systems.

1. Introduction

To meet stringent real time constraints for heavily computationally intensive digital signal processing (DSP) systems, dedicated hardware processing architectures on Field Programmable Gate Array (FPGA) must be added to multiprocessor platforms. Identifying best implementation targets for tasks is part of the extended partitioning problem [1]. A difficulty here is generation of a sufficient hardware solution. Of course, the use of highly optimized and commercially available intellectual property (IP) components is a possibility. However, in the case where these are not available, or inconvenient to use, an implementation must be created.

Coupled with the superior processing power of hardware components is the disadvantage of increased design difficulty. Typically, the generation of highly optimized architectures requires design at a much more detailed level than in software design. This increased detail leads to long development times, inhibiting design productivity at the traditional RTL level of design.

This paper examines the problem of rapidly creating a range of efficient hardware component solutions for a task. This includes the use of emerging hardware design tools and techniques, and how custom techniques can be applied to the problem. To create a range of implementation options from which the appropriate instance is to be chosen, the use of architectural manipulation techniques, such as hardware sharing, are analyzed.

The remainder of this paper is organized as follows. Section 2 discusses requirements for a hardware design approach, and outlines some of the factors involved. It then evaluates the SystemC language in terms of suitability as a medium for an 'carrier' language for the methodology. In section 3, the essential

components for a simple approach are outlined. This approach uses GEDAE as the starting point. Section 4 uses the design flow to implement a simple circuit, a fixed point FIR filter. Section 5 discusses the use of architectural manipulation techniques for creating a range of implementation options from which the suitable option may be chosen, and finally section 6 looks at the use of algorithmic synthesis techniques to generate these solutions.

2. DSP Algorithm Synthesis

It is well known in the electronic design automation (EDA) industry [2] that the ideal solution, in terms of area and power efficiency and customizability, that application specific circuit design, is the ideal implementation method for high end DSP algorithms, such those used in Radar and Sonar.

The use of general purpose DSP processors, and combinations of RISC and DSP processor in multiprocessor platforms, has become a popular practice in DSP system implementation. However, as computational complexity increases for high end operations, they cannot maintain a sufficiently high throughput rate to be an efficient implementation platform. Networks of these processors suffer from communication overhead, which degrades performance. The implementations of these processors cannot be parameterized in the same way application specific hardware can. Integrating FPGA into these multiprocessor systems allows implementation of processors with computational complexities above that which RISC or DSP can achieve. It is also possible to scale this computational complexity with minimal performance (throughput) degradation.

There are two opposing extreme approaches to hardware component creation. 'Top down' approaches take an algorithm representation and create a hardware representation from this

representation. This is a fast method, however due to the fact that the designer does not have complete control over the hardware created (it is inferred), it may be inefficient. ‘Bottom up’ approaches start at a very low level of component specification, and construct architectures hierarchically from this low level. Since in this case the hardware architectures is not *created* but *described*, the efficiency of the synthesis results are much more in the hands of the designer. However, this approach is also time consuming and error prone due to the large amount of design detail. Ideally the approach taken should exploit the best features of both extreme approaches. It should describe, not create, the hardware architecture. This description should be at a sufficiently low level of component granularity so as to maintain efficiency in the resultant architecture, but high enough a level to make the design process as fast as possible. It is possible to take advantage of architectural manipulation and synthesis techniques to ensure that no overhead is associated with the hierarchical component composition process. This is a ‘meet-in-the-middle’ or hierarchical approach.

2.1 Requirements For A Hardware Design Approach

Figure 1 shows a ‘black box’ approach to the hardware design problem.

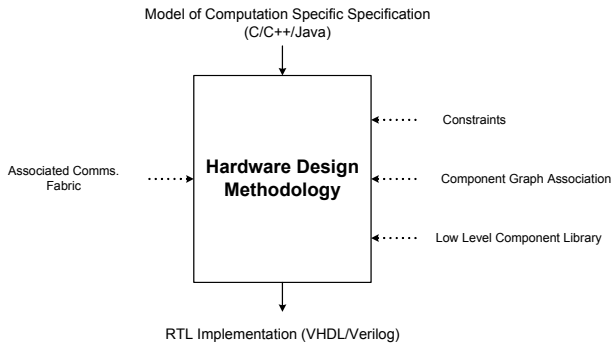


Figure 1: ‘Black Box’ Design Approach

The input will be a high level model of computation (MOC) specific specification for the component. This is usually in a high level language like C++ or Java. Common MOC include Synchronous Dataflow (SDF) [7] or Communicating Sequential Processes (CSP) [3]. The result of the design is an RTL level specification which may be synthesized for implementation. This is commonly in a hardware design language (HDL) such as VHDL or Verilog.

Factors effecting the implementation are shown on the left and right hand sides of the box in figure 1. For example, the constraints imposed on the global design process will certainly effect factors like how big the component is, or its throughput rate. The low level component library, of which the top level component is to be composed will certainly effect how efficient the implementation is. The type of graph in which the component is placed (static/dynamic/nondeterministic) may effect the type of concurrency employed in the processor. The concurrency may be spatial (parallel) or temporal (pipelined). Also the type of communication employed at the processor interface may effect buffering levels at the component interfaces.

One of the first, and traditionally most difficult aspects to overcome is immediately obvious from figure 1. The translation between high level specification language (C++/Java) and the much more detailed HDL (VHDL/Verilog) is traditionally an error prone process. Hence the efforts of the Open SystemC Initiative (OSCI) [4] to develop the SystemC language so that it may express designs in both behavioural and RTL level forms [5].

2.2 SystemC

The structure of SystemC 2.0 is shown in figure 2.

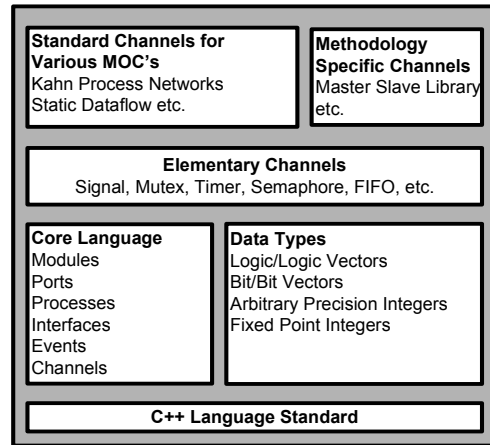


Figure 2: Structure of SystemC 2.0

As this illustration shows, the language structure has been carefully chosen to enable heterogeneous MOC based design. The lower language levels (core language and data types) are primarily to create a HDL from C++. However, the more interesting parts are the upper levels, incorporating standard channels for interprocess communication. The defining characteristic of a MOC is process interaction [6]. In dataflow [7], this is a blocking read, non-blocking write FIFO mechanism. Hence SystemC, in a single language, has the ability to express both the inputs and outputs of the methodology as shown in figure 1, providing the capability to traverse the entire design flow using a single language. It is conceivable that a GEDAE graph could be directly translated to a SystemC dataflow specification. Given the possibility of RTL hardware description in SystemC and direct synthesis using tools such as Synopsys CoCentric [8], it is evident that a framework for an iterative hardware design methodology, from domain specific specification to implementation is in place with SystemC.

3. Hardware Creation Approach

3.1 Hierarchical Component Creation

To create hardware components quickly and efficiently, a semi-custom hardware creation methodology is required [2]. This involves use of commercial IP cores where possible, and creation from a library of optimized, parameterised functional building blocks where possible in other cases. This leads to a ‘meet-in-the-middle’ construction technique, based on that in [2], as shown in figure 3.

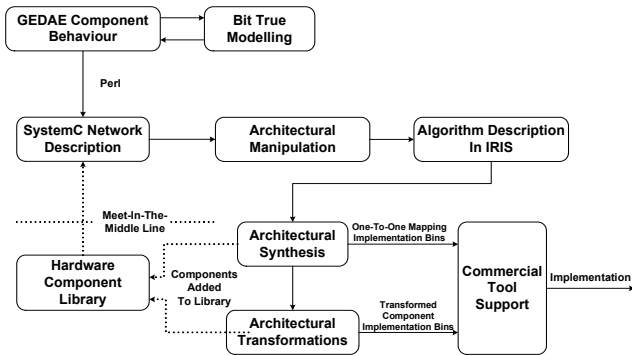


Figure 3: Meet-In-The-Middle Methodology

This meet-in-the-middle methodology makes use of existing efficiently optimized components (in the hardware component library). Initially this library should be populated by fundamental components (adders, multipliers etc.). Included with these is documentation on their properties such as throughput, maximum clock rate, latency, and hardware size, amongst others. Higher level components are then composed from these primitive components. This provides the hardware description with a certain level of granularity. The composition is then operated on by architectural techniques to ensure that minimal performance degradation is imposed on the high level component, over and above that of the fundamental components, because of the chosen architecture. In the cases we consider, we are primarily concerned with maintaining high component throughput, hence we use pipelining techniques to minimize the critical path through the higher level components. When high level components are constructed from these fundamentals, the high level components themselves become part of the library. Once again, various configurations of these may be created by combining varying configurations of the lower level components.

The input to the methodology is a GEDAE graph representation of the high level component, described by connected behavioural primitives representing components in the component library. This is converted to a structural SystemC component (see section 4.2). This is followed by architectural manipulation to minimized composition side effects in a certain sense. If necessary, architectural synthesis (see section 3.2) is performed to combat the effects of architectural manipulations on lower level components. The resulting component is then implemented. This component can subsequently be added to the component library, to raise the level of hardware component description in future projects.

3.2 IRIS and Architectural Synthesis

Architectural synthesis is the process of transforming an algorithm to an RTL level expression. In translating the GEDAE component to an architecture composed of lower level components, we have violated some assumptions made in the algorithm in constructing the architecture. For example, the algorithm assumes zero latency computation. However, in the architecture component latency is most definitely finite, variable with component pipelining levels, and variable on distinct paths from input to output on the processor. Also, the architecture utilizes components which employ reduced precision arithmetic. In connecting these we must resolve numerical truncation issues

to ensure accurate algorithm implementation. Resolution of these issues is part of architectural synthesis.

IRIS [9] is an architectural synthesis tool developed at Queen's University Belfast. The input form is an algorithm expressed as a signal flow graph (SFG). With every component in the algorithm is associated a processor model. This details data formats at the processor interfaces, and also latencies on the paths through the processor. IRIS uses this information to generate a properly timed and truncated architecture ready for implementation.

IRIS was originally developed with its own graphical interface. Recent evolution [11] has interfaced the tool to Xilinx System Generator. This allows an easy interface to the Core Generator tool for Xilinx FPGA, and a dynamic simulation capability.

4. FIR Filter Synthesis

The design approach described in section 3 is now applied to the creation of a FIR filter component, as commonly employed in a wide range of applications. Specifically, the implementation of a digital receiver for use in an adaptive beamformer system, as described in a complementary paper [13], is of interest. This FIR filter will be based on fixed point arithmetic, and will be parameterisable in terms of input wordsize, and filter length. It is assumed that the FIR filter is to be integrated into a statically scheduled system. Hence the aim of the design process will be to maximize throughput. A measure of the success of the architectural manipulations employed will be evidence of no performance degradation of the final architecture when compared to the fundamental components, in terms of throughput in Msamples/s. The target architecture for this component is a Xilinx Virtex-II family FPGA [10]. It is also assumed data can be supplied and removed as quickly as it is consumed and produced by the processor.

Figure 4 shows the GEDAE representation of a four tap FIR filter used in this exercise. The repetitive nature of this architecture makes it a good candidate for architectural manipulation.

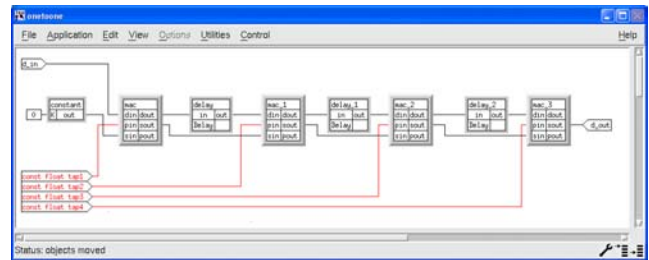


Figure 4: Four Tap FIR Filter GEDAE Graph

Given the structure of the FIR as shown in figure 4, it is evident that this development will proceed in three steps:

1. Fundamental Component Library Population
2. MAC Processor Construction
3. FIR Filter Construction

4.1 Fundamental Component Library Population

This will contain the components used to construct a fixed point MAC processor, i.e. a input wordwidth fixed point adder and multiplier. Since we are to implement this architecture on VirtexII

FPGA, we can take advantage of the embedded multiplier components on this family of device to implement the fixed point multiplication.

To implement the fixed point adder, a carry-ripple adder structure will be utilized. This structure is shown in figure 5.

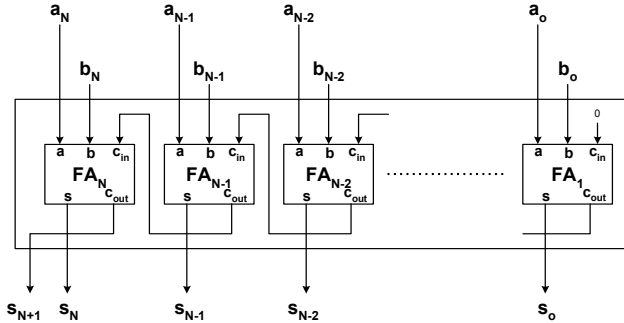


Figure 5: Fixed Point Adder

A fixed point adder component with an input wordsize of N requires N full adders. Hence, the parameterisation of the adder component in terms of input wordwidth determines the number of instantiated full adder components in the architecture.

A parameterisable fixed point adder is described in SystemC as below.

```
#include <systemc.h>

template<int add_bitwidth>
SC_MODULE(adder)
{
    sc_in< sc_uint<add_bitwidth> > A; //input ports
    sc_in< sc_uint<add_bitwidth> > B;
    sc_in< bool > CIN;
    sc_out< sc_uint<add_bitwidth> > S; //output ports
    sc_out< bool > COUT;

```

```
    SC_HAS_PROCESS(adder);

```

```
    adder(sc_module_name name)
        : sc_module(name)
    {
        SC_METHOD(add);
        sensitive(A);
        sensitive(B);
        sensitive(CIN);
    }

```

```
    void add(){
        sc_uint<add_bitwidth+2> sum_i;
        sc_uint<add_bitwidth> sum_u;

        sum_i = A.read() + B.read() + CIN.read();
        sum_u = sum_i.range((add_bitwidth+1),0);
        S.write(sum_u.range((add_bitwidth-1),0));
        COUT.write(sum_u[add_bitwidth-1]);
    }
};

```

Synthesis results for a parameterisable fixed point adder are shown in table 1 for input words of width 16, 24 and 36 bits.

These, like all the synthesis results in this paper, are quoted for the Xilinx VirtexII 6000 FPGA.

Table 1. Fixed Point Adder Synthesis Results for Varying Wordsize

Wordsize (bits)	Resource Usage (LUTs)	Throughput (Msamples/s)
16	38	94.6
24	57	70.5
36	88	54.1

Note that as the size of the input word increases, the throughput decreases dramatically. In this case, it is very likely that the long carry propagation chain is limiting component throughput.

4.2 MAC Processor Construction

Having obtained a multiplier and an adder, we can now construct the MAC blocks. These are multiply accumulate components, built of a single multiplier and adder. The description of such a component in GEDAE is shown in figure 6.

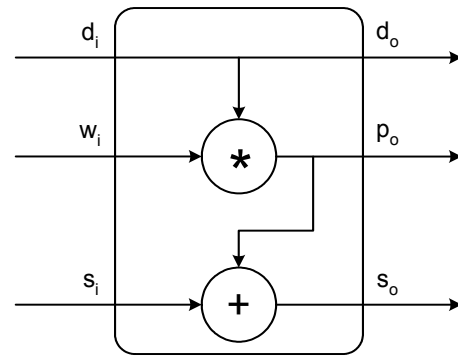


Figure 6: MAC block

4.2.1 Primitive Component Network Autogeneration

Given that we now have fixed point adder and multiplier resources to call on, and behavioural models of these we can implement as primitives in GEDAE, we can compose the two to create a fixed point MAC unit, as shown in figure 7.

Use of the Perl script automation route converts the graph of the MAC block in GEDAE as shown in figure 7, to the SystemC network level description as listed below.

```
#include <systemc.h>
#include "multK.h"
#include "add.h"

SC_MODULE(tap)
{
    /*Input and output ports declared here*/
    sc_in< /*declare signal type*/< /*length*/> > sin;
    sc_in< /*declare signal type*/< /*length*/> > coeff_in;
    sc_in< /*declare signal type*/< /*length*/> > din;
    sc_out< /*declare signal type*/< /*length*/> > dout;

```

```
sc_out</*declare signal type*/</*length*/> > sout;
```

```
sc_signal</*Put type here */ > add_a;
sc_signal</*Put type here */ > multK_K;
sc_signal</*Put type here */ > multK_in;
sc_signal</*Put type here */ > add_b;
```

```
/*-----
The behaviour of the box is defined here, use this
definition to write the systemC code
-----*/
```

```
multK multK_;
add add_;
```

```
SC_HAS_PROCESS(tap)
```

```
public:
```

```
tap(sc_module_name name)
:sc_module(name),
multK_("multK_"),
add_("add_"){
```

```
/*declare threads and processes here */
```

```
add_.out(sout);
multK_.out(add_a);
add_.a(add_a);
multK_.K(coeff_in);
din(multK_in);
multK_in(multK_in);
din(dout);
add_.b(sin);
};
```

```
}
```

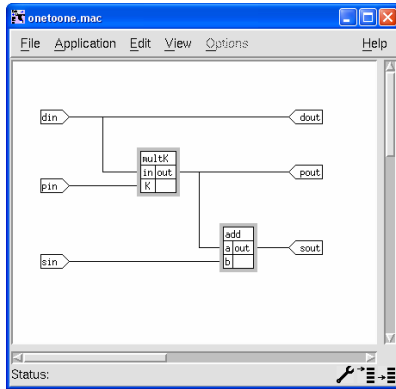


Figure 7: GEDA MAC Description

4.2.2 Implementation Optimisation

If implemented as viewed in figure 4, the MAC processor would have low throughput, due to a long critical path through the component. This is composed of a multiplier and an adder. A measure of the success of our architectural techniques is the level of performance degradation (in terms of throughput) of the MAC over the fundamental components. We wish to be able to obtain throughput levels for the MAC equal that of the lowest throughput fundamental component (the adder in this case). The present structure would not allow this. To fragment the critical path of multiplier and adder, pipelining is applied between to the two.

If we were to create a long chain of MAC components, as we will in the FIR filter, a long critical path of N adders (where N is the

number of taps) would limit the throughput of the chain. To break this, pipelining the adder output will reduce the critical path in a chain of adders of a single adder. Finally, to ensure data reaches the adder input from the multiplier and the s_i input at the same time, the second adder input is pipelined also. The resulting MAC processor is shown in figure 8.

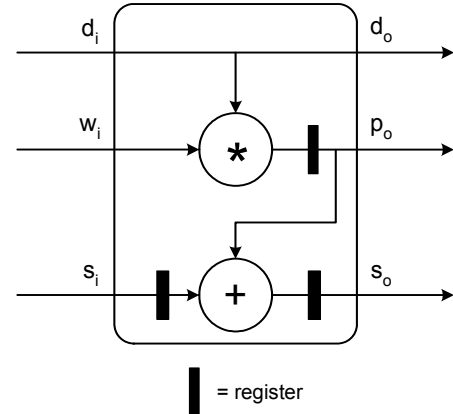


Figure 8: Pipelined MAC Processor

The SystemC interface for this component is generated using the Perl script. At present, this interface is defined in terms of standard software sizes and data types (due to the lack of bit true modeling in GEDA). However, the substitution process is simple and the generated code shows where this should be done. The word sizes for the MAC defines the wordsizes for the adder and multiplier sub-components, which are instantiated in place of the generic add and multiply operations in the behavioural description. Hence, composing the circuit as shown in figure 8 is a semi automated process, which includes a small amount of designer interaction to apply appropriate pipelining and insert component wordsizes. The MAC component has the characteristics stated in table 2 on synthesis.

Table 2. Fixed Point MAC Synthesis Results for varying Wordsize

Worsize (bits)	Resource Usage (LUTs)	Resource Usage (18x18 Mults)	Throughput (MSamples/s)
8	36	1	94.6
12	55	1	75.4
16	87	1	53.3

Note that the throughput levels are almost equal that of the slowest fundamental component (the adder). From this we can conclude that the architectural manipulations applied (pipelining) have been effective in maintaining high throughput.

4.3 FIR Filter Architectural Synthesis

At this point we have created the MAC component of which the FIR (of figure 4) is to be composed (this MAC has now been added to the component library for use in other designs).

The creation of the FIR filter, however, is not simply a matter of connecting a string of MAC blocks together and inserting delays

as appropriate to the algorithm. Consider the MAC component created in section 4.2. Figure 9 shows the latencies of the output paths, relative to the inputs on which they are dependent.

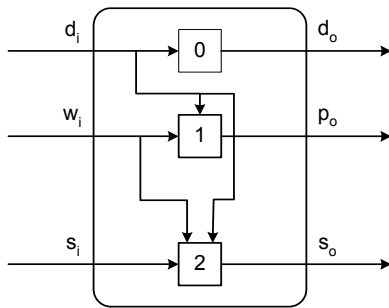


Figure 9: Relative Latencies of Paths Through MAC Processor

It is clear that the output is available on different outputs at different times. The correct operation of the next MAC unit in the chain is dependent on data being available on all inputs at the same time. Hence a measure of ‘data alignment’ must be performed. This involves the integration of additional delays to ensure correct operation.

IRIS is used for this task. The SFG for a 4 tap FIR filter described in IRIS is shown in figure 10. The blocks marked ‘sc_mac’ are the MAC blocks designed as described in section 4.2. The blocks marked ‘delay’ are delays inserted between the MACs according to the algorithm (i.e. assuming zero latency on all paths through the MACs). The numbers in the delay blocks represent the size of the delay in clock cycles.

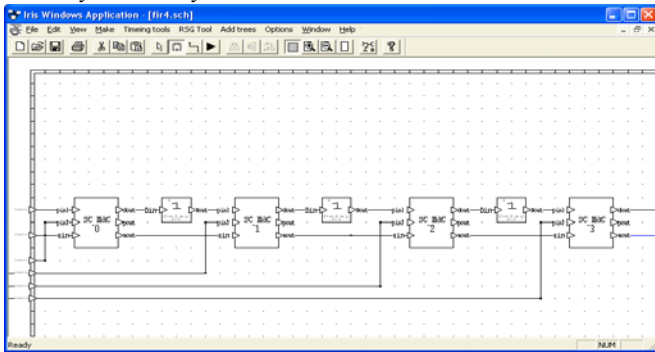


Figure 10: 4-Tap FIR Filter in IRIS

In practice the latency on all paths through the MAC units is not zero, as discussed earlier. Upon insertion of additional delays to ensure proper algorithm operation, whilst taking account of the

physical latencies of the MAC processors, the implementation looks like that of figure 11. Analysis of this along with the representation of the MAC unit in figure 9 reveals the consistency of timing on every path from circuit input to output. Note the inserted delays.

Synthesising this component produces the characteristics of table 3. Tables 4 and 5 show the synthesis results for 20 and 50 tap FIR filters respectively.

Table 3. 4 Tap FIR Synthesis Results

Wordsize (bits)	Resource Usage (LUTs)	Throughput (MSamples/s)	Computation Rate (MMAC/s)
8	109	92.2	368.8
12	168	73.8	295.2
18	259	53.3	213.2

Table 4. 20 Tap FIR Synthesis Results

Wordsize (bits)	Resource Usage (LUTs)	Throughput (MSamples/s)	Computation rate (GMAC/s)
8	680	94.7	1.894
12	1063	73.8	1.476
18	1632	54.1	1.082

Table 5. 50 Tap FIR Synthesis Results

Wordsize (bits)	Resource Usage(LUTs)	Throughput (MSamples/s)	Computation rate (GMAC/s)
8	1745	94.7	4.735
12	2758	72.3	3.615
18	4236	53.3	2.665

Notice the similarity in throughput of all FIR filter lengths, and that of the slowest component (the adder). From these results we can discern that the architectural manipulations applied to the MAC processor, and the architectural synthesis techniques applied for the correct construction of the FIR filter have been successful in achieving our design goal of maximum throughput rate. Also note the increasing computational complexity of the FIR filter systems. Given that the Analog Devices TigerSHARC DSP benchmarks quote computation rates of 2.4 billion MAC/s [14], it is clear that FPGA implementation of dedicated hardware processors can achieve computational performance at throughput rates that software solutions simply cannot match.

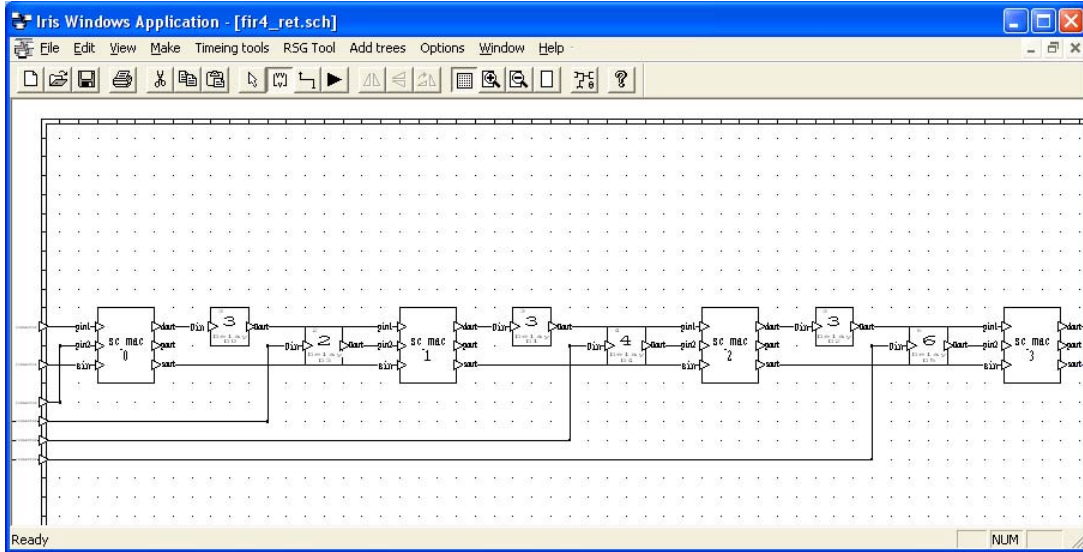


Figure 11: Retimed 4-Tap FIR Filter in IRIS

5. Architectural Exploration

We have proven a methodology for implementing a FIR filter component hierarchically from a library contained fixed point multiplier and adder components. By intelligent use of the target architecture and its capabilities, we have extended this library to include a MAC processor and a FIR filter.

For this size of input word however, we only have one hardware processor implementation. This is a restricted choice, and may severely inhibit a partitioning methodology such as [1] which requires a large space of implementation options from which to select the ideal component. These options vary in size, throughput, latency and power dissipation, and depend on the system constraints.

Options can be created in a variety of ways. For example, floating point arithmetic may be used in the place of fixed point, although the result may be of larger hardware size. Hardware sharing may be used to share the same hardware processing resource amongst multiple data streams. This limits the size of the hardware, but also the throughput of the system.

The notion of hardware sharing is that of sharing processing resource amongst multiple input data streams. This limits the size of on chip hardware processing resource, but also the throughput of processors. To achieve this, a multiplexing technique is applied to the inputs of the processor, and an amount of control circuitry added to select between input streams at various points in time. This control circuitry amounts typically to an N -value counter where there are N input streams. In the case of systems with internal data storage, such as the FIR filter delay line, this must be increased N -fold.

Finally, multiple instances of the same component may be instantiated to increase throughput. This however, will also increase hardware usage. The specific architectural techniques employed depend on the deficiencies of the existing implementations.

By intelligently representing the hardware network, the use of intuitive transformational techniques can easily transform the network construction. For example, techniques such as loop unfolding, can be applied to a Nested Loop Program (NLP) representation of a system. By applying a simple transformation, many instances of the same hardware component may be instantiated. Techniques for the synthesis of these networks and the automatic generation of controller circuitry is introduced in [12] for the case of a QR factorization processor. Also of significance is the use of loop skewing techniques to alter the order in which operations are performed, altering the latency. This too is discussed in [12].

These techniques, allow a range of implementation options to be created, allowing the selection of a processor from a variety of such options, to achieve a more efficient system integration.

6. Algorithmic Synthesis

Given the availability of intuitive techniques for representing hardware processor network architectures (for example NLP), the realms of algorithmic synthesis can be considered as a viable option for hardware creation. Using standard transformations on loops (unrolling/skewing) we can manipulate the size (parallelism levels) in a processor network, and their execution schedule.

Given a streaming data computation problem, described as a NLP, where the fundamental computations at the 'root' of the NLP are representative of hardware processors used, loop transformations can be used to alter the processor network. If the network of processors is defined to be implemented in a certain manner (i.e. data transfer methods between processors, parameterized control mechanisms for multiplexing and demultiplexing at the processor inputs and outputs), the problem space for the processor network can be mapped out. The specific characteristics of memory sizing and control schedule can be calculated for each hardware processor in the context of that network configuration. This provides a method for quickly evaluating the quality of processor network solutions to a particular problem, and allows easy iteration and efficient hardware component synthesis. This

approach is a geometric algorithmic synthesis approach. This is distinct from traditional behavioural synthesis, since it requires description of the form of the hardware processor, and employment of the algorithmic synthesis techniques to calculate the levels of control and memory required.

These algorithmic synthesis techniques can be applied on any form of input specification, opening up the option of simple hardware processor network representation and manipulation in GEDAE, translation to SystemC and refinement to RTL for synthesis.

7. Conclusions/Future Work

This paper has addressed the use of a fast but efficient hardware component creation approach to create hardware components from GEDAE graph specifications. The graph representation can be quickly and automatically converted to a SystemC network level specification, instantiating components from a primitive components library in the place of the behavioural description. This implements a 'meet-in-the-middle' technique which has been shown to be viable. The one-to-one conversion implementation of a FIR filter processor, constructed hierarchically from this library has been examined. Architectural manipulation (in this case pipelining) was employed to ensure the high throughput of processors composed of low level combinational components. The results of this, i.e. variable path latencies on processor outputs would cause the high level algorithm implementations composed from these pipelined processors to be incorrectly implemented. The IRIS tool was used to resolve this situation, applying architectural synthesis techniques.

In the case where a range of implementations is required for exploration, to allow optimal component selection in the light of system constraints, a number of techniques have been suggested to provide the necessary solutions. Exploration of the use of the aforementioned architectural exploration techniques is the subject of further investigation. In a rapid design methodology, it may be required that as yet unsynthesised components be modeled for inclusion in the system. In this case it would be useful, since synthesis may be time consuming, to have a high level modeling technique for hardware components for properties such as hardware size, throughput, latency etc. The hierarchical design approach adopted in this paper lends itself well to such a method as demonstrated, for example, in the consistency in throughput rates of the FIR filter given that of the MAC components, and of the basic adder. Finally integration of components into a standard implementation framework, such as that in [13] could help provide a complete design and integration methodology.

It is believed that GEDAE could provide an excellent springboard from which to launch a fast IP-based implementation scheme. The aim of an overall partitioning approach is solution of the extended partitioning problem. This requires the ability, not only to implement one-to-one actor substitutions, as well as composition and architectural transformation techniques to explore the implementation option space.

8. Acknowledgements

GEDAE is a trademark of Blue Horizon Development Software.

This work has been sponsored by the UK Ministry of Defence Corporate Research Programme, and the Department of Education and Learning (DEL) NI. This work has been undertaken in

collaboration with QinetiQ Ltd., Great Malvern, UK and BAE SYSTEMS ATC, Gt. Baddow, UK. The authors are grateful for contributions by Ying Yi and Darren Reilly at Queen's University in Belfast.

9. References

- [1] Kalavade, A and Lee, E.A. The Extended Partitioning Problem: Hardware/Software Mapping and Implementation-Bin Selection. *Journal of Design Automation for Embedded Systems*, vol. 2, March 1997, pp. 125-163
- [2] DeMan, H., Cathoor F., Goossens, G., Vanhoof, J., Van Meerbergen, J., Note, S. and Huisken, J. Architecture-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms. *Proc. IEEE*, vol. 78, No.2, 319-335, February 1990.
- [3] Hoare, C.A.R. *Communicating Sequential Processes*. Comm. ACM, vol. 21, No. 8, 666-677, August 1978.
- [4] The Open SystemC Initiative (OSCI). <http://www.systemc.org>
- [5] Grötter, T., Liao, S., Martin, G., Swan, S. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
- [6] Lee, E.A. and Sangiovanni-Vincentelli, A. A Framework for Comparing Models of Computation. *IEEE Trans. On CAD*, Vol. 17, No.12 1217-1229, December 1998.
- [7] Lee, E.A. and Messerschmitt, D.G. *Synchronous Dataflow*. Proc. IEEE, September 1987.
- [8] CoCentric SystemC Compiler Datasheet Available at <http://www.synopsys.com>
- [9] Trainor, D.W., Woods, R.F. and McCanny J.V. Architectural Synthesis of Digital Signal Processing Algorithms Using "IRIS". *Journal of VLSI Signal Processing* 16, 41-55 (1997)
- [10] Virtex-II Datasheet Available at <http://www.xilinx.com>
- [11] Yi, Y. and Woods, R. FPGA-based System-level design framework based on the IRIS synthesis tool and System Generator. *Proc.2002 IEE International Conference on Field-Programmable Technology (FPT)*, 85-92, December 2002.
- [12] Harriss, T., Walke, R., Kienhuis, B. and Depretter, E. Compilation from Matlab to Process Networks Realised in FPGA. *Journal of Design Automation for Embedded Systems*, April 2002.
- [13] Walke, R. FPGA Acceleration of an Adaptive Beamformer within GEDAE. *Gedae Users Conference*, 2003.
- [14] TigerSHARC DSP Benchmark Statistics. Available at <http://www.analog.com>.