Use of Multi-Port Memories in Programmable Structures for Architectural Synthesis

C. Mandal R. M. Zimmer
Department of Computer Science
Brunel University, Uxbridge UB8 3PH, U.K.

Abstract

In this paper we make a study of the capabilities required of memories to support the synthesis of designs using structured architectures. We explore the advantages of using multi-port memories with two write ports as an architectural component over conventional memories with a single write port in such a synthesis environment. A study the of the memory resources available in some of the current Field Programmable Gate Arrays (FPGA) is made. We then propose a multi-port memory structure that could be suitable for use in programmable structures such as FPGAs, to facilitate implementations of designs through HLS. The principal advantages of the proposed memory structure are its flexibility, simplicity and its ability to support more efficient execution of operations than existing memory structures.

Keywords: Semiconductor Memory, High-Level Synthesis (HLS), Field Programmable Gate Array (FPGA), VLSI.

1 Introduction

Electronic design automation is supporting design at increasingly higher levels of abstraction. For designs involving algorithmic computations and complex data flows the design techniques to synthesize the architecture of the target system is usually referred to as high-level synthesis (HLS). HLS starts with the behavioural specification (BS) of the target design and goes on to find a schedule of operations in the BS and construct the architectural data paths and the controller to implement the BS. The data path is composed of register transfer level components, such as adders, subtracters, registers, buses and switches. The construction of the data paths is usually referred to as data path synthesis (DPS). The prime tasks are scheduling, and component allocation and binding. FPGAs and other programmable structures are an attractive platform for implementing designs synthesized using HLS techniques as they permit quick prototyping of target designs.

In this paper we explore the role of multi-port memories in aiding HLS targeted towards programmable structures, such as FPGAs. We show that multi-port memories with two write ports are required to have efficient structured implementations of designs. We then propose simple structures to implement such multi-port memories on programmable devices.

In section 2 we briefly examine some HLS techniques. The use of multi-port memories in structured architectures is considered in section 3. Here we bring out the importance of the multi-port memories with two write ports as a building block in programmable structures. Memory resources of a few FPGAs are examined in section 4. We propose the design of our memory section 5 and close the paper with our conclusions.

2 Current HLS techniques

Much work has been done on scheduling, allocation, and also on integrated scheduling and allocation. Maha [1] performs scheduling of operations, and allocation and binding of hardware operators. Force directed scheduling [2] attempts to minimize the cost of hardware operators while trying to schedule within a specified number of time steps. In Cloutier et al. [3] scheduling is combined with allocation and mapping. A method for integrated scheduling and binding has been presented in Balakrishnan et al. [4]. A problem space genetic algorithm has been proposed in Dhodhi [5] which does concurrent scheduling and allocation. The above techniques address optimizations for the data path with respect to its performance or its cost. Most of these techniques use individual registers as storage elements and employ a random interconnect structure. While such data paths may be easily fabricated using semi-custom fabrication techniques, they may place excessive demands on interconnect resources if they are implemented using programmable structures.

There have been efforts to develop HLS techniques to produce data paths which will have more economic physical implementation. Balakrishnan et. al. [6] have developed techniques to cluster storage through the use of multi-port memories. Duncan et al. [7] have developed a simulated annealing algorithm for synthesizing a data path where all interconnections take place over a few global buses. This is a useful feature for FPGA based implementation. Mandal et al. [8] have developed a genetic scheduling algorithm to synthesize structured data paths which have a predictable layout. Data paths obtained using these techniques often rely on the use of multi-port memories with two write ports. In the next section we study the role of memories for implementing designs on structured architectures, with the help of an example.

3 Memories in Structured Architectures for HLS

Let us consider the architectural schematic in figure 1. The main components are the architectural block (A-block) with local functional unit and memory, the interconnecting buses, the global memory (if any) and the controller. Each functional unit implements a set of operations derived from the behavioural specification. The key feature here is that data transfers between the A-blocks can only to take place over the global bus. This particular feature endows the design with a predictable layout structure. It is desirable that the internal storage and interconnection structure of A-blocks should also be compact.

The FU and memory in each block are used to execute operations of the form $x = y \; op \; z$. This usually requires two read accesses and one write access of the local storage. In addition, it cannot be guaranteed that the required data will always be present locally. Thus, at times, it will be necessary to fetch data from other blocks or the global memory. These transfers have to take place over the global buses. The memory available in each block should support the required local storage accesses. In example 1 we examine the

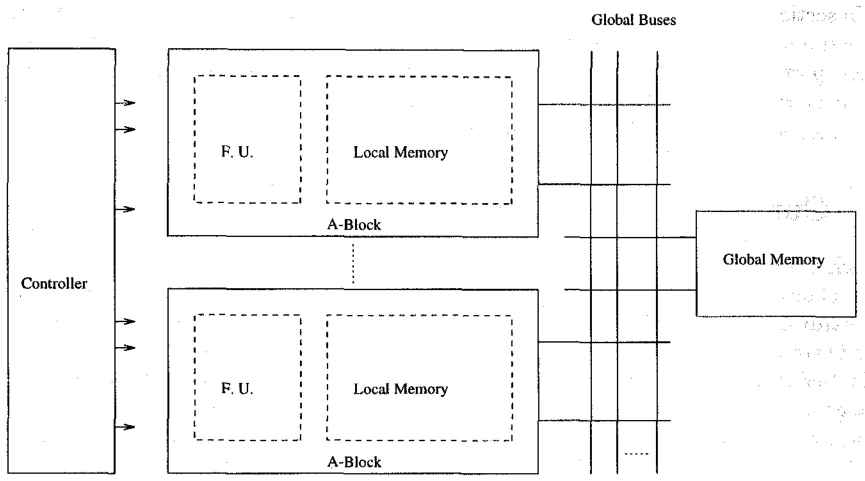

Figure 1: Structured Architecture.

data transfer requirements that could arise in the implementation of differential equation solver proposed by Paulin et. al. [9].

Example 1 The code sequence for the Diffeq. example in Paulin et al. [9] is given below.

```
loop
    v0 = dx *  u        v1 =  3 *  x        x = dx +  x
    v2 = v0 * v1        v3 =  3 *  y            x <  a
    v4 =  u - v2        v5 = dx * v3       v6 =  u * dx
     u = v4 - v5         y =  y + v6
loop end
```

The partial order arising from this code sequence is shown in figure 2. For example, $v2 = v0 * v1$ can be performed only after $v0 = dx * u$ has completed execution. Let us assume that we would like to implement the above computation on a structured architecture involving three blocks and suppose that we permit up to one transfer between the block and the buses. Let us also assume that each multiplication takes two clock cycles and that we would like to obtain a schedule in seven time steps.

We present in table 1 a schedule of operations and transfers which could be implemented on such a structured architecture. We note that a loop computation is involved. Variables used in an iteration but not defined by a preceding operation should be present at a predefined location. The first row of the table indicates the blocks where the loop variables are located. The second part of the table, below, indicates the schedule of operations and the blocks where they are located. The last part of the table indicates the schedule of inter-block data transfer. The annotations "in" and "out" indicate that the variable is either entering or leaving the block. This particular schedule minimizes the cost of the FUs, while at the same time performing all the required transfers within the given number of time steps, seven in this case. The value dx is constant and is replicated in all the three blocks.

	(dx)	(x, y, dx)	(u, dx)
1	v0 = dx * u	v1 = 3 * x	
2			
3	v2 = v0 * v1	v3 = 3 * y	x = dx + x
4			x < a
5	v6 = u * dx	v5 = dx * v3	
6			v4 = u - v2
7		y = y + v6	u = v4 - v5
1	(u: in)		(u: out)
2	(v1: in)	(v1: out)	
3		(x: out)	(x: in)
4	(v2: out)		(v2: in)
5		(x: in)	(x: out)
6		(v5: out)	(v5: in)
7	(v6: out)	(v6: in)	

Table 1: Operation and transfer schedules for Diffeq.

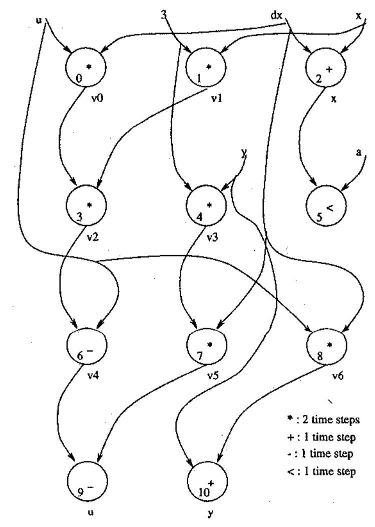

Figure 2: Partial order for Diffeq.

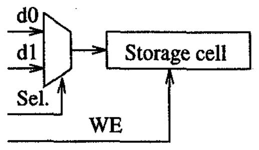

Figure 3: Basic Write Control Signals for a Storage Cell.

In example 1 we note that there is one inter-block data transfer in each time step. Specifically in time steps 2 and 6 we note the following. In time step 2, in the first block the values $v0$, generated by the multiplier, and $v1$, coming from another block, both need to be stored. This means that the memory configuration of a block should permit two simultaneous write operations. A similar situation is also observed in time step 6, where $v5$ and $v4$ both have to be stored in the third block. Here two read accesses to u and $v2$ also need to be supported. In time step three we notice that there are three reads from the memory, for x, y and the constant 3. The total number of memory accesses in any block does not exceed four.

Thus, to implement the schedule shown in table 1, the memory architecture of a block should support two read, one write and one read/write accesses independently. It may be desirable to support greater number of accesses, but that would be more expensive. We now examine a few memory configurations using conventional storage structures. First in section 3.1 we consider the use of directly controlled registers. In sections 3.2 and 3.3 we consider the use of dual port and multi-port memories with more than two ports, respectively.

3.1 Block Using Individually Controlled Registers

This is the fundamental structure for storage. Figure 3 shows the data and control lines for an individual register. Suppose that there are n individual registers and m sources for data to these n registers. Then for each register $\lceil \lg m \rceil$ lines will be needed to identify the source of the data and one line to control writing to the register will be needed. Thus $n\lceil \lg 2m \rceil$ signals need to be generated for controlling write operations to these cells. If these cells are organized as in a memory then up to $m\lceil \lg 2n \rceil$ signals will suffice. This is considerable reduction in the number of output functions of the controller as $m \ll n$ usually. A local decoder is required to decode each encoded cell address. We believe that this is usually a reasonable trade-off for most designs and we shall now consider only the memory type organization for storage cells.

3.2 Block Using Dual Port Memories

We first note that at least two memories will be needed to provide the required accesses. The memories may have one read and one write port. Alternatively, one or both of the ports may have read/write capability.

Case 1: Using two memories each with one read and one write port. At most one value may be read out of a memory. Thus variables need to be allocated to the memories

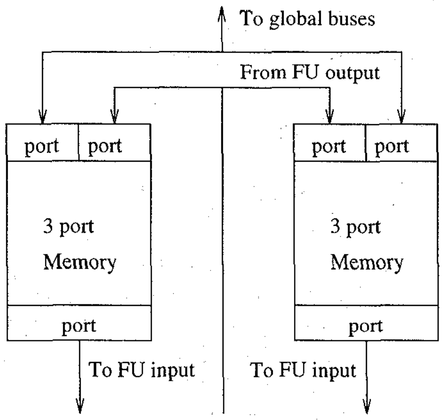

Figure 4: Two Triple Port Memories.

so that those being accessed at the same time are on different memories. Such an allocation is not always possible and some of the variables will have to reside simultaneously on both the dual port memories. This can be ensured only by storing new values in both the memories blocking the write port of each memory. Thus, all the ports get used up for data transfers within the block. An access from outside the block, in general, will have to wait.

Case 2: Using two memories each with two read/write ports. In this case it may be shown that all access for operations within the block may be satisfied using three of the four ports available between the two memories. It does not matter in which of the two memories the operands are stored. The main advantage over the previous case with single read and write ports is that the computational task of finding a feasible variable assignment to the memories is eliminated. This comes at the expense of duplicating many values in both the memories. However, the fundamental problem of being able to accommodate storing a value from another block along with a value generated internally remains unsolved.

Thus, with the use of these memory configurations in blocks micro-operations requiring simultaneous storage of values in a block will have to be distributed over two time steps. This makes the implementation slower. We now examine memories with three ports.

3.3 Block Using Memories With More Than Two Ports

We consider two triple port memories each using one read, one write and one read/write port, as shown in figure 4. This is essentially a 4-port memory made up using the triple port memory. With this configuration it is easy to permit two reads, one write and one read or write outside the block. This is an expensive solution to the problem because two triple port memories need to be used in each block, with copies of most values having to reside in both the memories.

We can also use a 4-port memory that will permit up to three reads and two writes, the total number of accesses not exceeding four. The key factor to be noted is that to permit a general access to data outside the block alongside normal accesses within the block, *the* *memory should support two write accesses.* Where permissible, four port memory, using two write ports, could be used in place of two triple port memories. We now examine at the memory resources available in two common FPGAs.

4 Memories in FPGAs

Both Xilinx^{TM} [1] and Actel^{TM} [2] produce FPGAs which include on-chip RAMs. The Xilinx XC4000 series FPGAs [10] employ two SRAMs, each of sixteen cells, in their configurable logic block (CLB). Internally each SRAM uses separate circuitry, with independent addressing, for read and write. The two SRAMS can be configured as a dual port RAM (16 bits) with two read ports and one write port. This is achieved by tying the data and address lines of the write circuitry of the two SRAMs in a CLB. The SRAMs may be used as lookup tables to implement logic functions or just to store and retrieve a bit of data. The two SRAMs can also be used as two 16-cell SRAMs or a single 32-cell SRAM. In the dual port mode the writes are edge triggered.

The Xilinx XC8100 family of FPGA [11] has a very different architecture, employing an array of programmable cells. Each cell may be configured as an AND or as a sum of products (SOP), a tri-state buffer or a latch. On-chip memories are not directly provided.

The Actel 3200DX FPGA [12] uses independent on-chip SRAM units alongside three types of logic modules. These are the combinatorial (C-modules), sequential (S-modules) and some decode (D-modules). The D-modules implement decode circuits and are located at the device periphery. The C-module is a multiplexer based logic module, shown in figure 5. The multiplexer is a well known universal logic module. The S-module is similar to the C-module, with the addition of a D-type flip-flop, connected to the multiplexer output. The Actel 3200DX also provides on-chip SRAMs having one read and one write port, both independently addressable. The address and data input lines of two SRAMs can be tied up work as a dual port RAM with two read and one write ports.

We thus note that the FPGAs mentioned do not directly permit the storage facilities that have been shown to be desirable in section 3. It is interesting to note that some of these FPGAs support on-chip multi-port memories, but these memories are restricted to having only one write port. In the next section we propose the implementation of a multi-port memory using primitives similar to those used in these FPGAs.

5 Proposed Memory Structure

Figure 6 depicts the standard implementation of a dual port SRAM. Current FPGAs implement the decoder, latches and the multiplexer at the output as an integrated unit. This makes it impossible to programatically incorporate additional write ports to an SRAM. Figure 7 shows the write circuitry for a dual port RAM with two write ports and eight cells. Cells in up to two rows can be independently addressed through the address ports. A cell is written if it is selected through one of the address ports. The source of the data bit to be written to the cell is determined by the address port through which the cell is selected. For correct operation of a dual port RAM with two write ports it is expected that

[1] Xilinx is the trademark of Xilinx Inc.
[2] Actel is the trademark of Actel Corp.

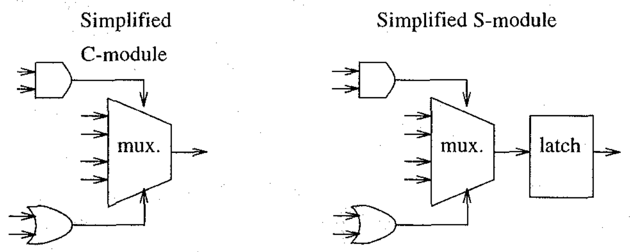

Figure 5: Two Actel 3200DX FPGA logic modules.

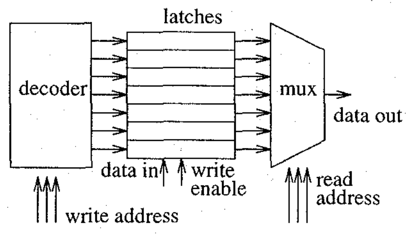

Figure 6: Dual Port Memory with One Write Port.

simultaneous writes through both the ports will not be attempted on cells in the same row. The write control logic in figure 7 performs a cell selection and provides the correct data for writing under this condition. In case the cell is selected by both the address ports then the data to be written to the cell represents $D0 \mid D1$. We now consider implementations for the building blocks shown in figure 7. The salient feature of our implementation is that the structures used are flexible enough to be used for general logic synthesis when they are not required to implement the memory. The implementation using the proposed structures is also compact. Duplication of memories to permit additional read ports, as indicated in section 4 is inefficient and is not warranted with the proposed structure.

Figure 8 shows a re-programmable 4-to-1 multiplexer/demultiplexer. Such a multiplexer/demultiplexer could be used in some of the configurable logic cells (CLC) of a multiplexer based CLC in an FPGA. The demultiplexer function permits direct implementation of the decoders for the write circuitry shown in figure 7. In addition, by hierarchical cascading, larger decoders can be easily built up. When used as a decoder, each output of the demultiplexer needs to fan-out to only one cell. The structural compatibility of the multiplexer/demultiplexer with the C-module of the Actel 3200DX FPGA may be noted. The write control logic requires implementation of the two functions $S0.D0 + S1.D1$ and $S0 \mid D0$. A flip-flop is required to store the bit. Figure 9 shows an integrated implementation of the write control circuit along with the flip-flop. This can be compactly implemented as a cellular unit. A multiplexer based implementation is used for $S0.D0 + S1.D1$ and $S0 \mid D0$ is directly implemented using an OR gate. This makes the implementation compatible with the structure of the S-module of the Actel 3200DX FPGA. The write control logic can be easily implemented using the programmable cells of both the Xilinx FPGAs studied here. However, the implementation of the decoder could be more involved. The

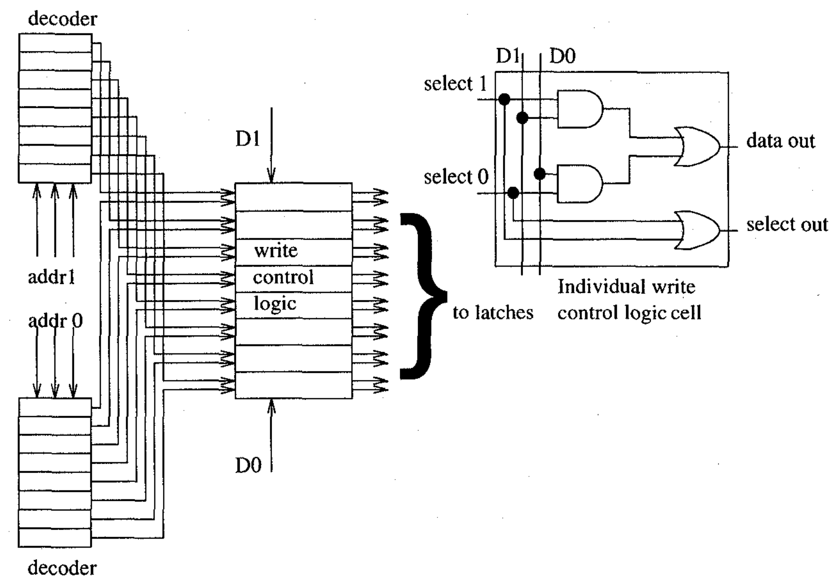

Figure 7: Write Circuitry for Dual Port Memory.

implementation of the read circuitry is simple, it only involves connecting additional multiplexers in tandem with the multiplexer shown in figure 6.

It is apparent that the CLCs indicated need to be programmed in many modes. These cells may be implemented to be re-programmable, like the XC4000 type FPGAs from Xilinx, permitting ease of use and reuse. Alternatively they may be made one time programmable, restricting their re-usability but considerably enhancing the device density and the performance.

The delay of the proposed memory structure may be predicted as follows. The delay of the decoder may be taken as that of the 3200DX C-module which is 2.5ns. The delay of the write control logic along with the flip-flop may be taken as the delay of the S-module which is 2.8ns. The routing delay between the decoder and the cell may be taken as 1.3ns, bringing the total write delay to 6.7ns. If the logic is to be implemented using standard FPGA cells then the basic circuit of figure 3.1 would have a delay of 6.7ns. If the storage organization indicated in this paper is used then the decoder implementation would have a delay of 6.8ns, assuming two levels of CLCs with interconnect in between. Total delay for write would now come to 14.8ns. If the storage organization indicated is not used then there is a vast increase in the output functions and the total complexity of the controller. The on-chip RAM of 3200DX has a delay of 5ns. It has the restrictions of memories discussed in section 3.2. Additional delay for multiplexing and interconnection may be taken as 3.8ns bringing the total delay to 8.8ns. It thus appears that the proposed structures lend a definite advantage for high-level synthesis based designs.

Standard examples such as the differential equation solver [2], fifth order elliptic wave filter [13] and discrete cosine transform [14] have been synthesized for implementation with

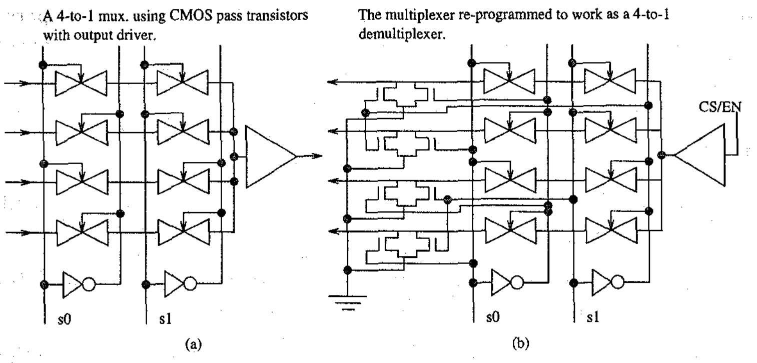
A 4-to-1 mux. using CMOS pass transistors with output driver.
The multiplexer re-programmed to work as a 4-to-1 demultiplexer.
CS/EN
s0
s1
(a)
s0
s1
(b)

Figure 8: A re-programmable 4-to-1 Multiplexer/Demultiplexer.

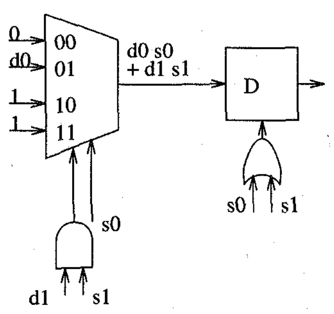

Figure 9: Implementation of a single RAM cell with its write control circuit.

structured architectures. All these examples are realizable using local storage structures needing to support only two concurrent writes. This would lead us to conclude that the proposed structures are useful for application to practical synthesis applications.

6 Conclusions

Some of the existing high level synthesis techniques discussed in this paper rely on the use of memories with two write ports. Also we have shown through an example that memories with two write ports are required to satisfy data transfers that arise while synthesizing structured architectures for target designs. The FPGAs examined in this paper neither directly support memories with two write ports nor do they permit efficient implementation of such memories. Current FPGAs provide on-chip single/dual port SRAMs with a single write port. We have proposed two simple primitives for the implementation of random access storage for use in programmable devices. One is a multiplexer/demultiplexer to implement the decoder. When used as a multiplexer, it serves as a general purpose programmable logic structure. The other is a storage cell using multiplexer based write control circuit and a flip-flop. This structure too can be used for general logic synthesis. Together these provide a compact implementation for random access memory structures permitting concurrent write to up to two cells. The proposed structures are also compatible with the programmable logic blocks used in some of the current FPGAs.

References

[1] A. C. Parker, J. T. Pizarro, and M. Mlinar, "Maha: A program for data path synthesis," *Proceedings of the 23rd Design Automation Conference*, 1986.

[2] P. G. Paulin and J. P. Knight, "Force-directed scheduling for asics," *IEEE Transactions on Computer Aided Design*, June 1989.

[3] R. J. Cloutier and D. E. Thomas, "Thc combination of scheduling, allocation and mapping in a single algorithm," in *Proceedings of the 27th ACM/IEEE DAC*, pp. 71–76, June 1990.

[4] M. Balakrishnan and P. Marwedel, "Integrated scheduling and binding: A synthesis approach for design space exploration," in *Proceedings of the 26th ACM/IEEE DAC*, pp. 68–74, 1989.

[5] M. K. Dhodhi, F. H. Hielscher, R. H. Storer, and J. Bhaskar, "Datapath synthesis using a problem-space genetic algorithm," *IEEE Transactions on Computer Aided Design*, vol. 14, no. 8, pp. 934–944, 1995.

[6] M. Balakrishnan, A. K. Majumdar, D. K. Banerjee, J. G. Linders, and J. C. Majithia, "Allocation of multiport memories in data path synthesis," *IEEE Transactions on Computer Aided Design*, vol. 7 No 4, pp. 536–540, April 1988.

[7] A. A. Duncan and D. C. Hendry, "Area efficient dsp synthesis," in *Proceedings of the 1995 European Design Automation Conference*, pp. 130–135, September 1995.

[8] C. Mandal and R. M. Zimmer, "A genetic scheduling technique for the synthesis of structured data paths," technical report, Department of Computer Science, Brunel University, Uxbridge UB8 3PH, U.K., August 1996.

[9] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," *Proceedings of the 24th Design Automation Conference*, 1987.

[10] Xilinx Inc., *XC4000 Series Feild Programmable Gate Array*. 1996.

[11] Xilinx Inc., *XC8100 FPGA Family*. 1995.

[12] Actel Corporation, *3200DX Field Programmable Gate Arrays – The System Logic Integrator Family*. 1995.

[13] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*. Prentice Hall, 1984.

[14] J. P. Neil and P. B. Denyer, "Simulated annealing based synthesis of fast discrete cosine transform blocks," in *Algorithmic and Knowledge Based CAD for VLSI* (G. Taylor and G. Russel, eds.), ch. 4, pp. 75–93, Peter Peregrinus, 1992.