

# A Comparison of High Level Synthesis and Register Transfer Level Design Techniques for Custom Computing Machines

Adam Postula\* David Abramson† Ziping Fang\* Paul Logothetis†

\*Department of Elec. and Computer Engineering, Univ. of Queensland, Brisbane Qld 4172, Australia.  
email: (adam, zfang)@elec.uq.edu.au

† Department of Digital Systems, Monash University, Clayton Campus, Melbourne Vic 3168, Australia.  
email: (davida,paul)@dgs.monash.edu.au

## ABSTRACT

*The most expensive component in the process of building a Custom Computing Machine is the time consuming and highly qualified work of hardware designers. This hinders the wide proliferation of CCMs and pushes this innovative technology into a niche market of research applications. To widen the range of potential applications, the large community of software developers must be attracted.*

*A potential solution to the problem is Behavioural or High Level Synthesis (HLS) which promises the compilation of algorithms into hardware.*

*The experiment reported in this paper was aimed at the evaluation of general purpose HLS tools in building CCMs. We focused on identifying areas where the synthesis methods and tools should be improved in order to cope with CCM's specific design problems. The design case, a specialised computer for the simulation of the sintering process, has been carefully chosen to exhibit the problems that are likely to be encountered in advanced designs for CCMs.*

*Our experiment shows that the HLS tools, when supplied with an appropriate input description, are capable of producing a design not too different from a RTL level manual design. HLS needs to be improved, especially regarding loop unwinding, pipelining, and the synthesis of suitable memory configurations. However, many of the required techniques are already available from the field of parallel compilation.*

*The conclusion is that improved HLS tools will bring the performance of compiled CCMs very close to that of manual designs. This can put the high performance of CCM technology at the fingertips of computer programmers without extensive hardware expertise.*

## 1. Introduction

The surging interest in Custom Computing Machines [1],[2],[3],[4],[9] is fuelled by the promise of very high performance at a moderate price. Field Programmable Gate Array (FPGA) technology [15],[16] and advanced hardware design tools [19],[20] help to make the cost of building a specialised computing machine justifiable. Still, the

main problem is the high dependence on the hardware designer's expertise and this can be considered the most serious drawback of the Custom Computing Machine concept.

Advanced design tools based on simulation and synthesis help to make the design work easier, and to some extent also lower the expertise required.

Commercially well established Register Transfer Level and logic synthesis tools [19], [20] speed up the design process but still require detailed design decisions. The scheduling of operations and allocation of resources must be done by the designer, who is also responsible for handling loops and pipelines. The crucial decisions about mapping the data structures to appropriate memory organisations are also left to the designer.

An attractive alternative is offered by High Level Synthesis (HLS) - the field of research that aims at the automatic conversion of algorithms, usually described in a procedural language, into hardware implementations [10], [11]. The promise of HLS is the compilation of hardware in a very similar way to the compilation of software.

In an ideal situation all the lower level design decisions are left to the intelligent tools and the designer is in a position of a programmer working with a high level language compiler.

There is a wide spread belief that HLS has not yet matured enough to play a role in the development of CCMs [3], [4] but there are few reports of HLS tools used for building Custom Computing Machines [6], [7], [8]. This is why we decided to explore the capabilities of the available, general purpose HLS tools and the future potential of this design methodology. Our main goal was to identify areas where the HLS tools should be improved to cope with CCM specific design problems.

The design chosen for this experiment was a specialised processor for the simulation of sintering on an atomic level with a MonteCarlo method. The sintering simulation problem has the following characteristics which makes it representative of a larger class of CCM designs:

- it uses irregular (dynamic arrays) data structures
- it requires gather and scatter operations on array elements
- it offers a large amount of low level parallelism

- solving real problems are computationally demanding. The strategy in the experiment was to design the same CCM manually on a RTL level, and then with HLS tools. Analysis of both designs and comparison of the design decision made by expert designers and HLS tools should provide insight into CCM specific design problems. The starting point for both RTL and HLS synthesis was a program written in Fortran and run on a Sparc workstation.

## 2. CCM development environments

An impressive list of Custom Computer machines can be found in [9]. Many of the reports listed there also cover the design environment issues. We compare our hardware and software environment to two well known systems - DECPeRLe-1 and Splash2.

DECPeRLe-1 [5] is a reprogrammable co-processor configured and provided with data by a host computer. The processor is organized as an array of 16 FPGAs connected by direct and global interconnections; this emulates the structure of a large FPGA. The memory, 4MB in four banks, is connected through the dedicated FPGAs to the array and is also accessible to the host machine. All the connections between the chips are fixed and the hardware reconfiguration takes place only in the FPGA chips.

The design development environment consists of: the driving software used to download data and configuration, the C++ based netlist generator also capable of processing placement directives, the run time library, and the debugging tools.

The designer/programmer is supposed to define, connect and place hardware modules in the C++ description. High level language constructs such as conditionals, for loops, procedures and data abstractions in the form of arrays ease the design task. Nevertheless, it is still a pure hardware design since the developer must have a very clear view of the hardware implementation of an algorithm/program.

Splash2 [6] is a linear array of processing elements which makes it a good candidate for linear systolic applications. Each of the 16 computational FPGAs on a board has 36 connection to its two local neighbours and to a local 512Kbyte memory. A crossbar switch connects all the chips allowing some limited communication between non-neighbours. 16 boards can be daisy chained to provide a linear array of 256 FPGAs.

The standard development environment consists of VHDL based tools for simulation and synthesis. The designer/programmer usually works on the Register Transfer Level (RTL) and produces VHDL input to the Synopsys synthesis tools. Working on the RTL level still burdens the designer with the allocation of resources and scheduling of

operations - the crucial decisions in mapping an algorithm to hardware.

The dbC-to-Splash2 compiler [6](chapter 7) is a specialised High Level Synthesis system allowing the designer to work with a version of the parallel C language. The results reported in [6] (pp94-95) show that the synthesised genetic database search performed more than one hundred times slower than the manual design.

In contrast to DECPeRLe-1 and Splash2 we consider the process of building CCM as a compilation of a procedural program (algorithm) onto a reconfigurable board. No initial structure or interconnections between hardware modules are assumed. The design tools have total freedom in building, connecting and reconfiguring resources such as ALUs, controllers, random number generators, memories, etc. This way the compilation process suits the general purpose HLS tools well and is very similar to the manual design of specialised hardware when the designer is not constrained regarding architectural decisions. The designer can guide the tools and achieve a desired structure by the appropriate modularisation of the program.

Advances in hardware technology make our model very close to reality since logic is reconfigurable with FPGAs, interconnections are reconfigurable with Field Programmable Interconnection Circuits (FPICs) [16] and memory can be made reconfigurable using standard chips and FPICs or modified memory chips. There are already FPGAs with dedicated memories on chip [17] and this trend can develop into FPGAs with a large on-chip reconfigurable memory. Also, with the present pace of technology progress one can expect that in a few years FPGAs will reach densities of 25 000 configurable logic blocks running at speeds of 200Mhz [12].

Our development environment is composed of the Mentor Graphics design system with VHDL simulation and RTL synthesis, NeoCad tools for mapping the netlists to FPGAs and Aptix tools for reprogrammable interconnections. The High Level Synthesis environment is provided by the MEBS tools [18]. We use behavioural VHDL as a development language.

The hardware platform is based on the Aptix AXB-AP4 [16] reprogrammable board with up to 16 FPGAs interconnected by 4 FPICs. Each FPGA connects to the FPICs switch with about 150 pins, and FPICs are also interconnected to each other by 100 signals. Memory modules can be inserted into any of the FPGA sockets. All the pins of five 128x8 memory chips on the module are accessible, which allows reconfiguration of the memory system down to the chip level.

The APTIX AP4, with memory modules, can be considered as a good approximation of an ideal CCM hardware platform.

The APTIX board is connected to a Sparc 5 workstation

- a host processor which downloads data, reconfigures the board and provides the graphical user interface.

### 3. RTL Design of sintering machine

#### 3.1 The Sintering Problem

Sintering is the metallurgical process in which an object is formed by heating a metal powder to a temperature below its melting point. The powder is modelled as a hexagonal

```

WHILE not finished DO
  read address of hole
  choose random neighbour
  read hole and neighbour
  IF neighbour is atom THEN
    retrieve neighbours of hole
    retrieve neighbours of atom
    calculate deltaN -- difference in number of atoms
    calculate value of probability function P(deltaN)
    choose random number R(0-1)
    IF R < P THEN
      write swapped hole and atom
      update hole address
    END IF
  END IF
  increment hole array counter
END WHILE

```

Figure1. Pseudo-code of the sintering algorithm

onal grid of atoms. If a grid cell does not contain an atom, then it is occupied by a “hole”. The simulation proceeds by moving atoms in the grid according to statistical mechanics, which can be modelled as a Monte Carlo process [13]. However, even unrealistically small problems contain millions of elements whose movements must be calculated in every Monte-Carlo step. Since the number of holes is generally less than one percent of the number of atoms in the powder, it is more convenient to consider the motion of holes in the simulation algorithm. Even using this optimisation, simulations can take days of CPU time on a conventional workstation and there is an obvious need for acceleration of those computations. The pseudo-code in Figure 1 describes the sintering process as implemented in software [13]. This algorithm was implemented in hardware to accelerate the computation.

#### 3.2 The design considerations

The design started from an analysis of the data structures needed for an effective implementation of the above

algorithm. This analysis has shown that the memory organisation should reflect the physical arrangement of atoms on a hexagonal grid. The optimal organisation from the performance point of view would be a seven way interleaved memory. If the cells (atoms or holes) were assigned to different banks, according to Figure 2b, only two memory read cycles would be needed for fetching all neighbours of a hole to be moved. The other extreme is the standard memory in Figure 2a with ten reads for the same operation.

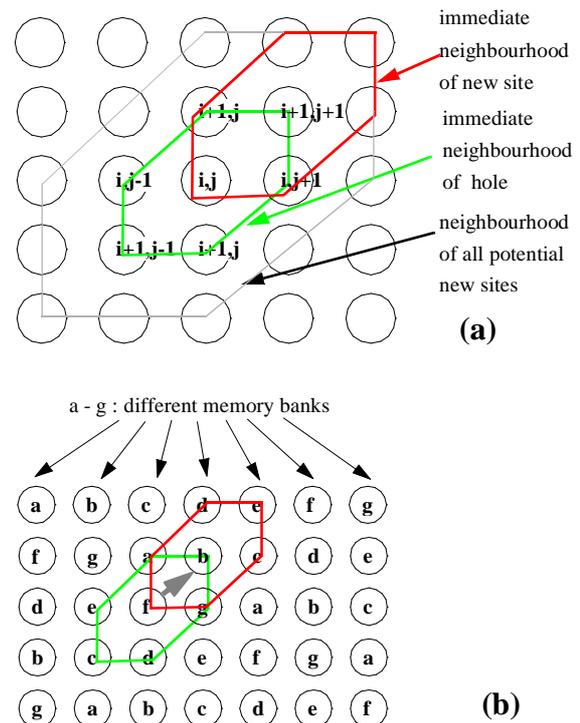
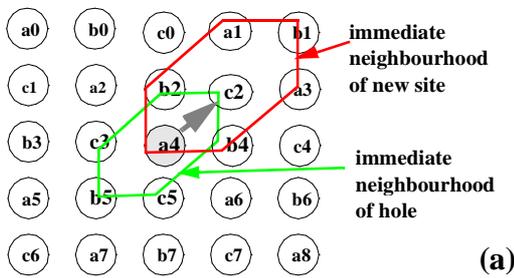


Figure2. Organisation of hexagonal grid in software (a) and in hardware (b).

As a trade-off between the hardware complexity and performance a three way interleaved memory has been designed [14]. As is shown in Figure 3 only four reads are needed. The address generation is not straightforward and a look up table must be used, but the physical design is much simpler than for a full seven way interleaved memory.

The frequency of memory access is a major factor in the optimisation of the sintering machine. Figure 1 reveals that the bulk of operations are conditioned on a hole being movable i.e. having an atom as a neighbour. The statistics from sintering simulations shows that the ratio of the moveable holes to all holes in the system is at most about 7% for larger problems. Another condition in the algorithm is used to decide whether move a moveable hole. The statistics shows that only about 0.1% of all holes



Operation	Frequency
<b>FETCH 1 : a4, b4, c2</b>	<b>100%</b>
<b>FETCH 2 : a3, b5, c5</b>	<b>7%</b>
<b>FETCH 3 : a1, b1, c3</b>	<b>7%</b>
<b>FETCH 4 : b2</b>	<b>7%</b>
<b>WRITE : a4, c2</b>	<b>0.13%</b>

Figure3. Organisation of hexagonal grid on the three way interleaved memory (a) and the access sequence (b)

are actually moved.

This means that the effort in optimising memory access should be directed towards shortening the basic cycle of reading a neighbour of a hole and checking if it is an atom. The seven and three way interleaved memories provide this in one read cycle, while the standard memory needs two read cycles.

Figure 4 shows the structure and the control sequence of the first sintering machine designed on the RTL level. The machine needs four states to decide if a hole is moveable and the processing of a moved hole is finished in twelve states.

Performance of this machine is limited by the time taken to decide if a hole is movable despite the memory organisation allowing it to obtain this information in just one read cycle. The reason is the intrinsically sequential operation of picking a hole, generating addresses and reading the memory .

Analysis of this design revealed that the most efficient way to improve performance was to pipeline the operations related to memory access. This resulted in a one cycle operation for deciding if a hole was movable. However, this also caused longer processing times of movable holes in case the move could not be performed and the pipeline contents had to be restored. As the latter happens much less frequently the overall performance has been significantly increased as it is shown in Table 2.

The pipelined design was coded in VHDL, simulated,

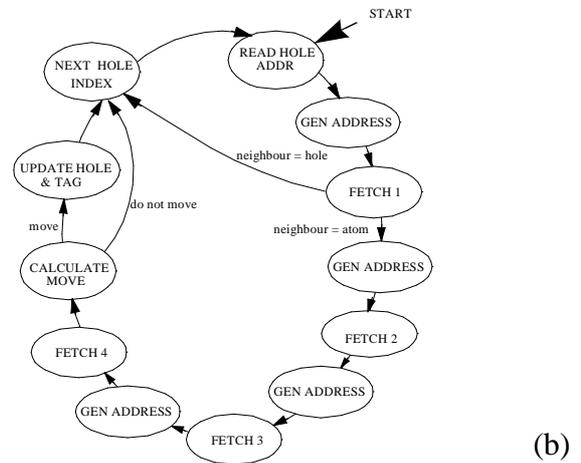
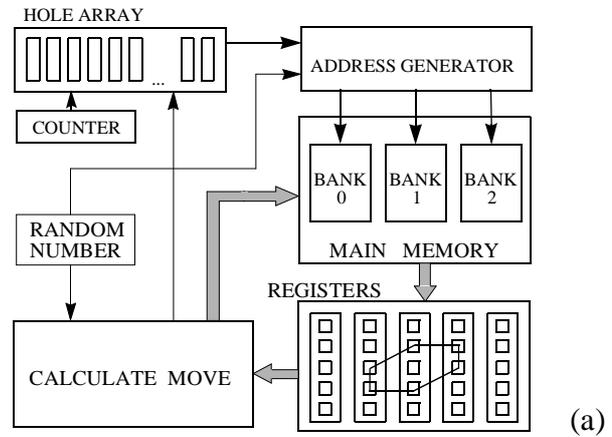


Figure4. Architecture (a) and state diagram (b) of the sintering processor

synthesised, and finally tested in hardware.

## 4. Behavioural synthesis of sintering machine

### 4.1 Tool limitations and code translation

The MEBS synthesis tools [18] can accept design descriptions in VHDL on behavioural, RTL and structural level. The input to the behavioural synthesis is a VHDL process and the output is VHDL code describing the synthesised design on the Register Transfer Level. MEBS synthesises arrays onto separate memories but assumes that those memories are implemented on the FPGA chip. Manual intervention of the designer into the RTL output of MEBS is required if the memories are to be implemented as physically separate modules.

The sequential Fortran code developed for sintering simulation on a workstation had to be translated to behavioural VHDL code required by the high level synthesis

tools.

At first we adopted a strategy of translating the code without considering any hardware specific optimisations. In this straightforward translation all the loops and data types were preserved from Fortran.

The main difficulties in translation were caused by the restrictions on VHDL for behavioural synthesis. Two dimensional arrays from Fortran code had to be converted to one dimensional arrays. Furthermore, the entire description has been put into one VHDL process since the arrays for synthesis are only supported within one process.

The procedures from Fortran had to be removed since they used arrays passed as parameters. The Fortran do-continue loop constructs were translated to loop-exit-next in VHDL.

In general, the translation although heavily restricted by the VHDL subset posed no more difficulties than a conversion between two different procedural programming languages.

## 4.2 The simple and the optimised designs

The VHDL code has been verified with the simulator and synthesised by MEBS. The memories have been separated from the rest of the design manually in the VHDL code produced by MEBS.

The final design after mapping to XILINX FPGA occupied 429 logic blocks (CLBs) which is just over one XC4010 chip. In comparison with the RTL design shown in Table 1 the result was poor both in area and performance.

The reason for the poor results, was our simplistic translation of Fortran to VHDL, not taking into account the inefficiencies in handling VHDL code by the High Level Synthesis tools. The inefficiencies turned out to lie mainly in the analysis and transformation of the sequential code to a form more suitable for the generation of hardware constructs. The encountered limitations and the ways to overcome them are discussed below:

### 1. Memory read/write dependencies

Whenever an array variable appears on the right side of an assignment or in a conditional statement a read operation is implemented by MEBS at the RAM location specified by the array index.

This straightforward approach is inefficient when the same array location, with an unchanging value, is read within a loop. In this case it is better to fetch the array values into temporary variables (registers) and use those rather than accessing memory every time.

Whenever an array variable appears on the left side of an assignment statement, a write operation to memory is

implemented by MEBS.

This is inefficient when the same array locations are referred in the non-mutually exclusive conditional statements. In this case the memory accesses can be reduced by using intermediate variables and writing the results to the memory at the end of the processing block.

We modified the VHDL code whenever the situations described above occurred by manually inserting appropriate intermediate variables. This resulted in large savings in the overall memory access time.

This type of memory read/write optimisation can be built in the high level synthesis tools. Such methods are already used in optimising software compilers.

### 2. The data types

The data types need to be optimised for efficient hardware implementation. Both data encoding and storage must be considered. The HLS tools can easily map integer variables to bit vectors with length dependent on the variable range. Another issue is the encoding of data for optimisation of the computations. This is much more difficult and requires estimators to measure the effects of different encoding on the complexity of computational units.

In our case the Fortran code used an integer variable (32 bits) with values 0 to 3 to represent four different types of cells. Straightforward translation to VHDL resulted in an inefficient hardware design due to the overhead of storing and processing 32 bit data.

We replaced the integer variable with a bit vector of only three bits for the enumeration of the cell types. The encoding of the cell types on three bits was devised to minimise the logic of computational units. This also resulted in much smaller storage registers.

Also, it should be possible to implement this kind of optimisation using the schemes in optimising compilers. It would be necessary to augment the algorithms with hardware specific encoding of data.

### 3. Loop unravelling

Loop unravelling is one of the optimisations that results in dramatic improvements in performance of the synthesised hardware. It is an established method [10], [11], [22] but the difficulty lies in deciding whether a loop should be unravelled and if unravelling can be complete or partial. From the designer's point of view an interactive process of loop unravelling would be the most usable. Unfortunately the MEBS tools used in this experiment do not unravel loops.

In our case there was a nested loop in the Fortran code for fetching all the neighbours of a cell and computing the number of atoms among them. It was a reasonable solution

in software but could be replaced in hardware by a simple combinational block for generating addresses and another combinational function for counting the atoms.

We replaced these loops by combinational logic described in VHDL. This transformation reduced the number of required control cycles from seventeen to eight for this part of the design. The hardware cost has also been reduced since the controller became smaller and the combinational blocks were well optimised.

**Table 1:**

Design	RTL Manual	HLS Simple	HLS Optimised
controller states	12	41	17
CLBs	208	429	265
compilation time	20 minutes	22 hours	1 hour

The changes described above resulted in a new, optimised VHDL description of the sintering machine. This new design has been simulated and synthesised. The results are presented in Table 1 and show that the code optimisations resulted in a dramatic improvement of both area and performance over the initial design

The sintering machine synthesised from HLS description is presented in Figure 5.

## 5. Analysis and comparison of both designs

The HLS design resulted in the same overall architecture as the RTL design.

The main difference lied in the memory structure which was highly optimised in the manual design. The sophisticated memory organisation reflecting the spatial structure of the data in the problem was the main strength of the RTL design.

The three way interleaved memory allowed to fetch all the neighbours of a moving cell in four fetches while the single memory needed ten cycles for the same task.

Comparison of Figure 4b and Figure 5b shows that the difference in the number of states in both controllers is only due to the memory access sequence.

The controller in the RTL design has been manually built and the number of states was minimal. The controller in Figure 5b was automatically extracted by the high level synthesis tools. Extraction of the control sequence was very efficient and the reason seemed to be the optimisation of the input VHDL code as described in the previous section. As discussed before, loop unravelling was the most crucial operation for achieving good results.

Pipelining was relatively easy to introduce in the manual RTL design and resulted in dramatically improved performance. The hardware cost was very low since only three additional registers and a slightly more complicated controller was needed. .

**Table 2:**

Technology	Design	Cycle Time	Total Cycles	MCS/sec <sup>a</sup>	Speed-up
Sparc 5	Fortran	13 ns	--	57	1.0
Xilinx 4010 FPGA	RTL	100 ns	$1.96 \times 10^{10}$	1183	20.8
	RTL (pipelined)	100 ns	$6.12 \times 10^9$	3268	57.3
	HLS	140 ns	$2.35 \times 10^{10}$	613	10.7

a. Monte Carlo Steps per second



loop unravelling, code hoisting, inter-procedural analysis, code in-lining, etc. are already used in software compilers for sequential and parallel computers [22], [23].

Our experiments show that an application of this already available technology to CCM synthesis can result in a dramatic improvement of the synthesised designs.

Memory synthesis is the most important and difficult part as it involves the optimisation of array references in the code, recognition of spatial structures of data in the problem, parallelisation of memories to obtain required speed-up, and trade-offs between performance and complexity of the memory system. In this study we did not address the synthesis of regular arithmetic structures, and this is an area for future research.

Our future work will concentrate on improving HLS tools to meet the requirements of automatic synthesis for Custom Computing Machines.

## Acknowledgements

The authors would like to acknowledge the contribution of our colleagues at the University of Queensland, namely Professor Kevin Burrage, Dr. Graham Schaffer and Dr. Ivan Podolsky.

This project is supported by the Australian Research Council.

## References

- 1 W. Hartenstein et al., Custom Computing Machines (invited opening keynote), Proc. DMM'95 - Int'l Symposium on Design Methodologies in Microelectronics, Smolenice, Slovakia, September 1995.
- 2 D. Abramson, "High Performance Application Specific Architectures", Proceedings of 26th Hawaii International Conference on System Sciences, Kauai, Hawaii, Jan 1993.
- 3 W.H.Magione-Smith, B.L.Hutchings, Configurable Computing: The Road Ahead, RAW'97- 4th Reconfigurable Architectures Workshop 1997. Web site <http://xputers.informatik.uni-kl.de/RAW/RAW97.html>
- 4 R.Kress, Configurable Computing: The Software Gap, RAW'97- 4th Reconfigurable Architectures Workshop 1997. Web site <http://xputers.informatik.uni-kl.de/RAW/RAW97.html>
- 5 J.E.Vuillemin, P. Bertin, D.Roncin, M.Shand, H.H. Touati and P.Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age, IEEE Trans on VLSI Systems, Vol.4, No1, March 1996, pp 56-68.
- 6 D.A.Buell, J.M.Arnold, W.J.Kleinfelder, Splash2 - FPGAs in a Custom Computing Machine, IEEE Computer Society Press, 1996.
- 7 P.M.Athanas and H.F.Silverman, Processor Reconfiguration through Instruction Set Methamorphosis; Architecture and Compiler, Computer, Vol.26, No.3, Mar.1993, pp.11-18.
- 8 I.Page and W.Luk, Compiling Occam in FPGAs, in W.Moore and W.Luk, eds. "FPGAs", Abingdon EE&CS Books, Abingdon, England, UK, 1991, pp.271 - 283.
- 9 S.A.Guccione, List of FPGA-based Computing Machines, [http://www.io.com/~guccione/HW\\_list.html](http://www.io.com/~guccione/HW_list.html), 1996.
- 10 D.Gajski, N.Dutt, A.Wu, S.Lin, High Level Synthesis - Introduction to Chip and System Design, Kluwer Academic Publishers, 1992
- 11 R. Camposano and W.Wolf, High-Level VLSI Synthesis, Kluwer academic Publishers, 1990.
- 12 J.E.Vuillemin, On Computing Power, Programming Languages and System Architectures, in Lecture Notes in Computer Science No.782, J.Gutknecht, Ed., Springer-Verlag, 1994, pp. 69-86.
- 13 T. Sercombe, "A Monte Carlo Simulation of Sintered Microstructures", Bachelor of Engineering Thesis, Department of Mining and Metallurgical Engineering, The University of Queensland, 1994.
- 14 A.Postula, D.Abramson and P.Logothetis, The Design of a Specialised Processor for the Simulation of Sintering, Proceedings of the 22 Euromicro Conference, September 1996, Prague, Czech Republic.
- 15 XILINX, The Programmable Logic Data Book, 1994. Web site: [www.xilinx.com](http://www.xilinx.com).
- 16 APTIX, System Data Book 1993. Web site: [www.ap-tix.com/products/mp4.html](http://www.ap-tix.com/products/mp4.html)
- 17 ALTERA, web site [www.altera.com/html/literature/litprod1.html#f10k](http://www.altera.com/html/literature/litprod1.html#f10k).
- 18 University of California, Riverside, CA, MEBS User Guide, 1995.
- 19 Mentor Graphics Corporation, Design Architect System Documentation, V8.2, 1994
- 20 Synopsys Inc., Mountain View, CA, Design Compiler Reference Manual, 1993.
- 21 A.V.Aho, R.Sethi, J.D. Ullman, Compilers: principles, techniques, and tools, Addison-Wesley, 1986.
- 22 F.E. Allen, Optimising Compilers for Parallel Computers, Stanford University, California, 1991.
- 23 H.Zima and B.Chapman, Supercompilers for Parallel and Vector Computers, Addison-Wesley, 1991.