A Guide to Migrating from Microprocessor to FPGA Coping with the Support Tool Limitations [1]

Eduardo Augusto Bezerra and Michael Paul Gough

Abstract

Microprocessors have been used as the main processing elements of embedded systems mainly because of their low cost, flexibility for system upgrade at the software level and because of the availability of efficient support tools. The configurable computing technology brought the software flexibility to the hardware level, but the limitations in support tools is a barrier for software designers. Using a microcontroller based embedded system as a study case, this paper introduces workable strategies to be used by software designers to develop a complete embedded system (software + hardware parts) using a single description language, and circumventing support tool limitations.

Keywords: Configurable computing; VHDL; FPGA; embedded system; computer architecture

Introduction

The use of field-programmable gate arrays (FPGAs) in special-purpose computer designs is no longer restricted to glue logic or specialised co-processing units [1][2][3][4]. With devices as large as one million gates, it becomes possible to design a whole processing system using only programmable logic. Furthermore, some studies use FPGAs as the main elements of general-purpose computers [5][6]. However, despite the advances in the Electronics Design Automation (EDA) field, the synthesis tools available for high level languages (e.g. VHDL, Verilog, C like languages) are still not efficient. An FPGA configuration bitstream (CB) generated from a high level language is space consuming, and represents a lower performer circuit when compared to one generated from schematic diagrams or low level languages such as VHDL structural. Low-level design entries are adequate for small designs only and, to achieve good synthesis results using a high level language, the designer has to employ pre-optimised cores which are device specific and result in a portability problem. Another concern is the time necessary for EDA tools to generate CBs. Depending on the design complexity, the synthesis and the placement and routing (PAR) execution may take as long as twenty hours or more. In time critical systems, such as space applications, effective development facilities are important because of the short time available for making changes to a faulty application. In the past several missions were saved as a result of the rapid problem identification, followed by the development of a

[1] *This research is partly supported by CNPq – Brazilian Council for the Development of Science and Technology,*

solution, ground tests and transmission of the new software to the spacecraft computer. In this class of critical application, the development tools available for a specific device must be efficient enough to allow the designer to develop a solution within an acceptable interval of time.

Two approaches for the use of FPGAs as the main processing elements in a system have been investigated in recent years. A very popular technique is to use the configurable resources to implement the functionality of a traditional microprocessor. An important incentive to this approach is the production of cores by leading microprocessor manufactures as, for example, Motorola, MIPS and ARM [3][7]. These companies can provide cores of some CPUs in the format of hard macros (device dependent netlists) or soft macros (cores available in languages as VHDL or Verilog). The main motivation for the implementation of traditional CPUs in FPGAs, is the flexibility for having a device formed only by the functional blocks and instructions required for the application. For example, in an application where there are only fixed point operations the CPU can be implemented without an FPU module. A good representative of this category is DISC [2], where instructions are loaded, or swapped, to the FPGA, as required by the application in execution, using a mechanism similar to page faults management. Other examples can be found in [3][7][8][9].

In the other approach, the internal logic of a configured FPGA is not used to execute a program but, instead, it is the program itself, which is developed for a specific application. There is no generic hardware implemented in the FPGA, all the system components are application specific. In [5] the FPGA internal logic, when configured, is considered as a single very complex instruction, developed to solve a specific problem, with massive bit-level operations executed in parallel. This approach is the most interesting way to improve FPGA resource utilisation and performance results, being the subject of the present work.

In the first approach, the designer has the option to purchase a CPU core, and develop the application using the instruction set available, in a traditional way. It is not practical to develop the core, since there is a significant choice of pre-optimised CPUs, ready for use. For the second approach, the designer has to develop the application in hardware description languages (HDLs) using concepts from the parallel-programming paradigm. Developing a program in an HDL, is arduous since one has to know how to configure the EDA tools and, most importantly, a good knowledge of the internal architecture of the target device is required. Making an analogy with the tools for RISC or DSP processors, it is possible to develop a program in C without a knowledge of the internal organisation of the processor, but the developer has to rely

on the C compilers effectiveness. A hand–made assembly program may be much more efficient than a C program, if one decides to study the processor internal organisation, optimising the use of pipelines, caches, and special hardware such as multiply-and-accumulate (MAC). The main point in this analogy is that compilers for microprocessors are at a level of development higher than EDA tools.

In order to place the present work in context, in this brief introduction, some points related to the configurable computing technology were highlighted. Now, we introduce ***the main objective of this paper, which is to explore the feasibility of developing special-purpose computer systems using only configurable logic, without general-purpose processors.*** In order to accomplish with this objective, recommendations for the development of synthesizable VHDL programs are given. These recommendations can be useful, not only for beginners in HDL and developers with software background, but also for experienced HDL developers which had never tried to synthesise a design. ***An important contribution of this paper is that it targets developers with a software background, interested in implementing systems using FPGAs and HDLs.***

The remainder of this paper is organised as follows: in section 2, configurable computing technology is situated in the computer architecture field; in section 3 there is a description of the case study; in section 4 advice for VHDL for synthesis developers are given, using the case study routines as examples; in section 5 the case study synthesis tools used are described; section 6 discusses the performance and area gains when using an FPGA to replace a microcontroller embedded system; and section 7 presents some conclusions.

General microprocessors versus FPGAs in the programming language arena

The possibility of developing hardware in a similar way to software appeals to developers with a software background. Unfortunately, VHDL books give the impression that systems can be implemented in VHDL in the same way as in languages like C or Pascal. However, it is also necessary to read manuals and documentation dedicated to VHDL for synthesis development, making texts as [10], [11] and [12] compulsory reading. VHDL is a rich language, but only a small subset of it is necessary to develop a synthesizable code. In order to develop good simulation models and test benches one is encouraged to study this language in a more detailed way. A beginner must be aware that most of the information contained in HDL books cannot be used in the design of a synthesizable code. It is common to find VHDL books that discuss several aspects of the language as, for example, procedures, functions, data types, sequential and concurrent assignments, attributes, delta delays, but, in the example presented of how to develop a

synthesizable VHDL system, only a small subset of the language is used. For instance, in [13] VHDL is very well introduced but the case study was written using only *if* and *case* constructions in sequential processes, in order to be synthesizable.

The use of popular programming languages followed by the conversion to an HDL is an approach to the language question. An example of such a system is A|RT Builder [14] that converts C/C++ programs to VHDL or Verilog. Problems appear when good synthesis results are expected, since the programmer needs to use special keywords and special instructions to be used by the compiler to produce a more optimised code. Depending on the design complexity, it may be harder to realise than developing the system directly in an HDL. Nested loops, the variety of data types, and arithmetic operators are examples of restrictions for the translation. The main point is that a compiler for a high level language can translate the program to a low-level language (assembly language), which has similar instructions, making the translation process not too difficult. On the other hand, synthesis tools have to translate the high level programs to a much reduced set of low level "instructions". For instance, while a compiler can use a *je* (conditional jump) instruction as the target for the translation of an high level *if* constructor, a synthesis tool has to use D flip-flops and multiplexers. The scarceness of options for the synthesis tools is the main obstacle for having good translators. A pro-translator argument is that developers used to a specific language do not want to waste time learning a new language. However, although it is difficult to learn a new programming paradigm, the syntax of a language like VHDL is not so different, for instance, from the C language. In addition, a C developer has to use only a sub-set of the language, because of the synthesis tools limitations. For example, it is possible to have dynamic structures in VHDL by using the *incomplete types* feature [13]. However, the translation of dynamic structures from C to VHDL is not useful, as this class of data type is not synthesizable within the current synthesis tools. C-like HDLs are a more interesting solution for those who prefer using a C like language. Handel-C [15], which is based in C and CSP [16], is a good example of such a language. The Handel-C compiler accepts as input a description of the design in a C-like format, improved with CSP constructors (primitives for parallel processing and process communication), and produces as output a netlist file for use by EDA tools. Another option is the Java language [17], which may be the next HDL generation. Verilog, one of the most used HDL today, is another language that can be classified as C-like, while VHDL inherited some of its features from ADA.

Despite all the restrictions imposed by the EDA tools, there are significant advantages to start by using

VHDL/FPGAs in new embedded designs. For instance, the sequential nature of a microcontroller, and its programming language, based on the von-Neumann model makes it difficult to introduce non-determinism to a program. One solution is to use interrupts. However, the time spent in an 8051 interruption driven design is longer than the time spent in a purely sequential one, mainly because of the scarceness of good tools able to simulate interruption events. With VHDL, the developer has to use the parallel-programming paradigm from the beginning of the design, making the system closer to real world problems.

Implementing VHDL systems for FPGAs requires a new design approach, even for those used to parallel programming. For instance, it is possible to develop a system composed of MIMD, SIMD and SISD modules, or of a combination of them. In the case of a reconfigurable design, that is, one which can have the configurable logic altered on-line, as a consequence of input or output requirements, this new kind of architecture can be described as CICD (configurable flow of instructions, configurable flow of data), following Flynn's taxonomy. This denomination is justified since for the same architecture it is possible to have differences in the data/instruction flows, at different instants of time, according to the application requirements.

An FPGA based computer system introduces the possibility of having a hardware capable of adaptation to the application, instead of the current microprocessor based methodology, where the developer has to implement the application according to the hardware specification. For instance, with the Pentium III Streaming SIMD Extensions [18] it is possible to have a significant gain in performance, **if** the system is developed according to the hardware requirements. The *add* procedure listed in Figure 1 works well because it was written following the 16-byte boundary data alignment required by the Pentium III SIMD integer/floating point instructions which use a special set of eight registers (128 bits each). A compiler can vectorize this program producing a code that fills two SIMD registers with the operands *a* and *b*, and executes SIMD instructions to add the operands and to load the result in the *c* memory position. As the contents of the arrays *a*, *b* and *c* are 32 bits long, and only four of the elements are added, it is possible to store the operands in only two registers. If the programmer needs to add, for instance, five elements, then performance problems will be noticed, as the vectorization of the code is not entirely possible.

```
void add(float *a, float *b, float *c)
{for (int i = 0; i < 4; i++) c[i] = a[i] + b[i];}
```

Figure 1. Addition procedure in C suitable for use in the Pentium III processor.

Another advantage in having the whole system, including the hardware and software parts, described at the same level of abstraction, by means of a high-level programming language such as VHDL, is the availability of facilities for improving some of the system dependability features [19]. For instance, in [20] a methodology was proposed to improve the reliability of a digital system generated from a VHDL description. Important characteristics of a digital design, such as reliability and testability, can be improved if formal verification methods are used to prove the correctness of the VHDL description that specifies the design [21]. Important points when porting a microcontroller based design to VHDL/FPGA are discussed next in the application case study description and analysis.

Case study: original microprocessor implementation

SVAL is an on-board instrumentation module flown on a NASA sounding rocket, launched from Svalbard (Spitzbergen, Norway), in December 1997. This module was designed at the Space Science Centre, University of Sussex and it was chosen as a case study because of its relevance in scientific space applications. The successful implementation of this 8051 based module, using only FPGAs, signifies that configurable logic can be used, not only in space applications, but also to replace traditional 8051 based special-purpose systems. This type of computer is usually found in embedded systems, where the advantages to using configurable hardware are well known [3][4][6]. The application performed by this board is auto-correlation function (ACF) processing of particle count pulses [22]. The ACF is a statistical method that can be used to obtain information about the behaviour of a signal, revealing the presence of periodicity in a near random signal. An ACF is constructed by sampling a signal at two instants of time (t), separated by a lag (τ), measuring the signal amplitude (x), finding their product, and averaging over the time of the record (N). This procedure is better described by the equation:

$$R_x(\tau) = \lim_{N\to\infty} \frac{1}{N} \int_0^N x(t).x(t+\tau)d\tau \qquad \textit{Equation 1.}$$

Although this ACF application was used to investigate the behaviour of energetic electrons at altitudes of up to 500 Km [23], the SVAL system as a whole, including the hardware and the software parts, is not too different from conventional embedded systems based in microprocessors or microcontrollers. The original SVAL hardware used two microcontrollers with embedded memories, FIFOs, parallel to serial converters, and state machines. A block diagram of the board is shown in Figure 2.a. The software, as in other conventional microcontroller applications, processes the inputs using arithmetic operators, as MACs in Equation 1, and sends the results to the outputs, requiring a high flow of data transfer between the embedded

RAMs. This scenario is not difficult to find in conventional embedded system applications, especially in DSP ones, which motivated the utilisation of SVAL as the case study for this work.

The SVAL board has two DS87C520 microcontrollers (8051 family), with one of them responsible for the low frequency (LF), and the other one for the high frequency (HF) ACF processing, respectively KiloHertz and MegaHertz frequencies. The input for both modules is supplied by two channels of pseudo-random 250ns electron detection pulses. SVAL is a typical memory transfer application, with a high input sampling rate and with scarceness of processing modules. There are only two MACs modules and few other arithmetic and logic operations. Basically, the information about the input signal is stored in memory, after or during the sampling, and when an output is requested, blocks of memory are processed and transferred to an output storage area. The processing activity is more demanding in the LF module because of the MAC operations required for the software ACF implementation. In the HF module, there are only simple arithmetic operations, as the ACF is partially realised in hardware by state machines.

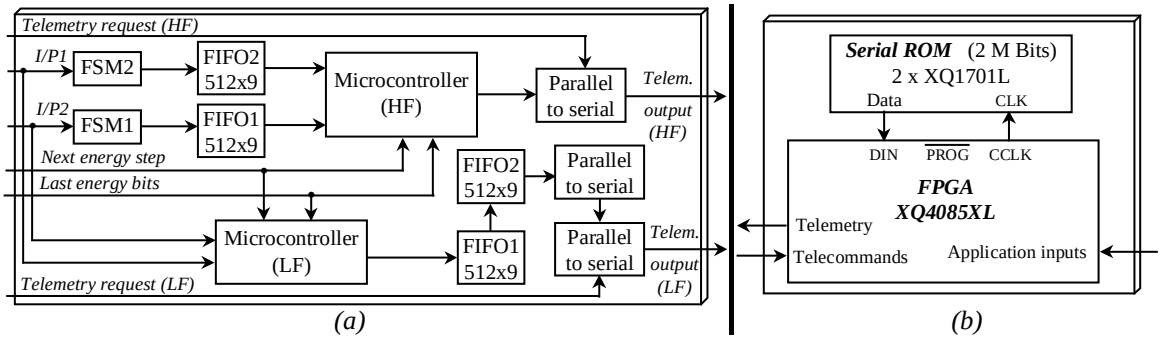

Figure 2. SVALBARD hardware components. (a) SVAL board (22 ICs); (b) SVAL-VHDL board (3 ICs).

Case study implementation

The VHDL version of SVAL was designed with three main aims: increased performance, portability, and a reduction in the number of hardware components. These goals could be met with a pure VHDL description, without pre-optimised cores instantiation, and using a generic synthesis tool. The reduction in the number of components was achieved by using only one FPGA device configured to execute the same functionality as the whole SVAL system. For instance, the external DS2009 FIFOs were replaced by circular FIFOs implemented using data structures in VHDL. The microcontrollers' functional blocks were not described in VHDL, but instead the functionality of the application was implemented directly. Figure 2.b shows the hardware components necessary for the SVAL-VHDL implementation, which is the SVAL version for FPGAs.

SVAL, the original design was first implemented at a high level of abstraction. During the development, a

series of changes were necessary because of synthesis tool limitations. Consequently, the final version is a combination of VHDL behavioural and structural, in a technology independent format but, unfortunately, it is not synthesis tool independent. When the design is synthesised using Synplify [12], for example, it is possible to generate CBs for different FPGA families and no special cores are needed. However, using FPGA Express 1.5 [11] it is necessary to include device dependent cores to replace some constructions that are not well understood, which may result in a poor resource utilisation, and performance degradation. Examples of these constructors are RAM memories and FIFOs.

Figure 3 shows the block diagram of SVAL-VHDL, from which is possible to identify important differences when compared to the original design. One of the most important improvements is in the use of distributed memory, which allows several modules to work in parallel. This feature could be implemented because of the combination of the VHDL concurrent constructors, and the FPGA array of LUTs that can be used, alternatively, as memory cells.

From Figure 3.a it is possible to observe that for the HF module there is no need for the two input FIFOs (see Figure 2.a), as the histogram generator process is executed in a non-stop way. In the original design, because of the sequential nature of the microcontroller, it was necessary to interrupt the histogram processing in order to execute the “next o/p requested” module, and to calculate the new base address for the histogram memory, in the event of a new energy step. The FIFOs were necessary in that design in order to avoid loss of randomly arriving input data during the interruption routines processing. For instance, after all the output array has been transmitted, the “next output” routine has to reload this array by copying a new block of data from the histogram memory and, in addition, this routine has to format this data in a suitable way for the telemetry activity. With the parallel execution of the three processes, the data loss problem was solved. In order to be accessed in parallel for reading and writing, the histogram memory had to be implemented as a dual-port RAM. This enormous flexibility is not found in a standard microcontroller embedded memory.

The LF module design had to be more sequential than the HF because of the data dependencies. For instance, the ACF sample processing (③) must be executed only after the sample activity has finished (②), and the ACF scaling and output must be executed only after the last ACF values had been processed (①). In order to synchronise the whole LF module processing, a manager process was implemented in the form of a state machine. Following the same ratiocination used in the HF module, the memory was broken in smaller parts in order to be accessed in parallel by the two processes present in each stage of execution. One of them is

responsible for the input 1 processing whilst the other one for the input 2. Apart from the distributed memory, other concepts from the computer architecture theory were employed in the LF module design. The sample module is an example of MIMD execution with distributed memory, as the two processes receive different data and work out them in parallel, using separated memories. In the original design, this procedure was executed sequentially. An example of SIMD processing is the ACF sample processing. The two processes execute the same operation, synchronised by the same clock, on different data, accessing different memories for reading/writing. The manager process is responsible for telling the two sample processes what to execute. In both cases, SIMD and MIMD, the communication and process synchronisation is executed by message exchange. The VHDL nomenclature for the communication media is “signal”, instead of “channel”, “link” or “socket”, used in other parallel processing systems.

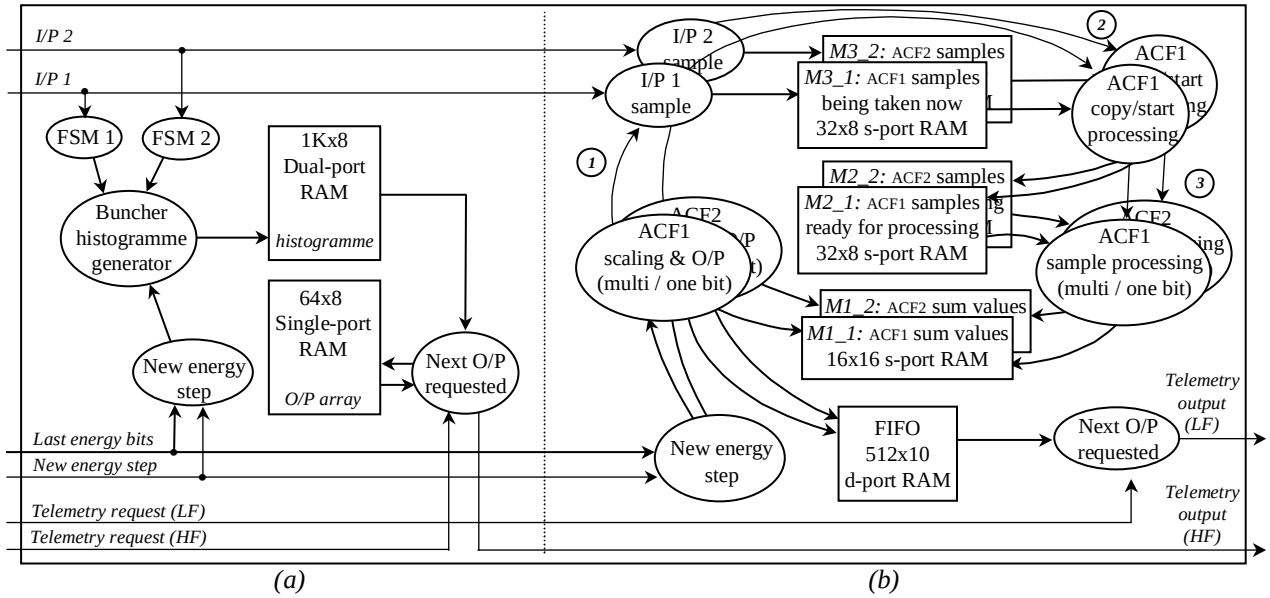

Figure 3. SVAL-VHDL FPGA: processes (circles) and memories (rectangles). (a) HF module; (b) LF module.

Case study implementation: the learning process

The first version of SVAL-VHDL was implemented without taking consideration of any synthesis criteria, with VHDL used like a programming language in a level of abstraction similar to languages as C or Pascal. Considering that the development of a simple program at such an abstraction level is not a difficult task, the high frequency module was finished in two working days, using a VHDL simulator (ModelSim). Figure 4 shows part of the code of the first version.

In the first version the implementation of the whole system using only the simulator, without synthesising it was a considerable error. These problems could be avoided if the design had been divided into small blocks,

and each block simulated and synthesised separately. Following this approach, when both versions of a block, the high level and the mapped VHDL, generated by the synthesis tool are providing an adequate behaviour, then the block can be included in the system. The ideal situation is the one where each block is a file containing only one entity/architecture pair, with the architecture split into combinational and sequential parts, and with the sequential part having no more than two processes (ideally one). In the case of inexperienced VHDL designers, it is suggested that they first execute the synthesis and then the simulation of the design, otherwise much time is spent implementing and simulating code that may not be synthesised.

The source code listed in Figure 4 is not suitable for synthesis, and it was developed in this way because the designer had the knowledge of how to develop systems in VHDL but not in VHDL for synthesis. Some of the problems in Figure 4 cause synthesis errors, whilst others cause warnings that must be fixed in order to avoid discrepancies in the simulation before and after synthesis. The most dangerous situations are those where there are neither errors nor warnings, but in spite of this the synthesis tool creates hardware with behaviour different from the expected one. For example, on line 4 there are two situations that might represent problems, although not a single note about them was provided in the synthesis tool report. The first one is the use of *inout*. It may not be a problem if the signal is associated to an I/O pin (external signal), as most of the commercially available FPGAs have resources to implement bi-directional I/O ports. However, if an *inout* signal is used in the interfacing of internal components, then it might represent a problem because not all FPGAs have latches available in their configurable logic blocks (CLBs) to store the input data. In the absence of latches, the synthesis tool has to improvise another resource, which may be very expensive. In order to avoid synthesis problems, a good advice is to use only *in* and *out* signals. The second dangerous situation in line 4 is the assignment of an initial value to the signals. It can be used for simulation purposes, but not all synthesis tools can identify and implement it.

Another important problem is the use of variables (lines 13 to 15 in Figure 4). In synthesis driven designs, it is strongly recommended that variables be avoided, however they can be used to speed up the simulation. It is strongly recommended to use only the standard logic type (*std_logic* and *std_logic_vector*) for all definitions. Two main reasons for this recommendation are: to avoid spending time in type conversions at the simulation level; and to use the same testbench for both behavioural and back-annotated netlist descriptions. In general, the back-annotated descriptions use only standard logic types (e.g. Xilinx). In order to follow these recommendations, the declaration in Figure 4, line 14 should be changed to: *signal init: std_logic;* and

moved to line 8. The test in line 17 should be changed to *if init = '0' then*.

```
1:  entity svalhf is
2:     port (clock, reset: in std_logic;
3:           P1: out std_logic_vector(7 downto 0);
4:           P0, P2, P3: inout std_logic_vector(7 downto 0) := "00000000";
5:           addr: inout std_logic_vector(7 downto 0);
                              .
6:  end svalhf;               .
                              .
7:  architecture behaviour of svalhf is
8:     type RAM_TYP is array(255 downto 0) of std_logic_vector(7 downto 0);
9:     signal w_fifo1, w_fifo2: std_logic;
10:    signal mem: RAM_TYP;      .
11: begin                        .
12:     Main: process            .
13:        variable Acc, Breg, Dpl: std_logic_vector(7 downto 0) := "00000000";
14:        variable init: boolean;
15:        variable Dptr: std_logic_vector(7 downto 0);
                              .
16:     begin                    .
17:        if not init then      .                        -- Powerup module
18:           Dptr := "00000000";
19:           for i in 0 to 16#3FF# loop                 -- Fill SRAM with RAMPS
20:              mem(i) <= Dptr;
21:              Dptr := Dptr + 1;
22:              wait for 10 ns; .
23:           end loop;          .
24:        else                  .    -- Loop indefinitely accumulating histogramme
25:           wait on w_fifo1, w_fifo2, P3(2), P3(3);
26:           if rising_edge(w_fifo1) then               -- i/p1 data ready
```

Figure 4. Partial listing of the high frequency module (first version).

The *wait* statements in lines 22 and 25 are examples of problems that may cause a synthesis error. It is not permitted to have more than one *wait* per process, and the *wait* must be either the first or the last statement of the process. In addition, the statement *wait for 10ns* is not allowed in VHDL for synthesis because of the complexity in implementing this kind of hardware. Usually synthesis tools ignore time statements, and if it is necessary to include this type of functionality in a design, then the designer should implement a timer. The *wait for time* statements are very helpful in testbenches. The loop in Figure 4 (lines 19 to 23) is used to initialise the array *mem* with 00H..FFH, 00H..FFH, and so on. From this description, the synthesis tool can not implement such circuit, and the generated one would initialise *mem* only with zeros, if the memory had been implemented in a synthesizable way. The initialisation would be done in one clock by resetting all flip-flops used to implement the memory cells every time *init* is zero.

Inferencing RAM for array implementation

The memory implementation itself is another concern. The definition in Figure 4, lines 8 and 10, is correct and can be used by the synthesis tool to infer memory, using the RAM modules available in the target device (LUTs in the XC4000 family), if there are some. However, this is not done because of the way that the memory is accessed for reading and writing (line 20). In order to tell the synthesis tool to use the memory blocks available in the target device, special code for memory management following the style suggested in the synthesis tool documentation, must be implemented. The memory component implemented according to Synplify documentation is illustrated in Figure 5. Even small alterations in this code can lead Synplify to

infer memory in a wrong way using, for instance, flip-flops instead of the available RAM/LUTs in the target device. It is important to emphasise here that some synthesis tools, as for example FPGA Express F1.5, cannot infer memory from VHDL code, and in this case it is necessary to use in the project the netlist of the memory block, generated by another tool like, for example, CoreGen [24], LogiBlox [11] or Synplify. With the purpose of having the whole design implemented only in VHDL, it was decided to use Synplify for the logical synthesis, and no cores were employed.

```
1: library IEEE;
2:    use IEEE.std_logic_1164.all;
3:    use IEEE.std_logic_unsigned.all;
4:    use work.SVAL_PKG.all;
5: entity RAM_MEM is
6:    port ( CLK_IN       : in  std_logic;
7:           WR_IN        : in  std_logic;
8:           ADDR_IN      : in  std_logic_vector(SIZE1K_C-1 downto 0);
9:           DATA_IN      : in  std_logic_vector(BYTE_C-1 downto 0);
10:          DATA_OUT     : out std_logic_vector(BYTE_C-1 downto 0));
11: end RAM_MEM;
12: architecture RAM_MEM_BEH of RAM_MEM is
13:   type RAM_TYP is array(SIZE1K_C-1 downto 0) of std_logic_vector(BYTE_C-1 downto 0);
14:   signal MEM         : RAM_TYP;
15:   signal ADDR_IDX    : std_logic_vector(BYTE_C-1 downto 0);
16: begin
17:   DATA_OUT <= MEM(conv_integer(ADDR_IDX));
18:   MEMORY_PRO: process (CLK_IN)
19:   begin
20:      if CLK_IN'event and CLK_IN = '1' then
21:         if (WR_IN = '1') then
22:            MEM(conv_integer(ADDR_IDX)) <= DATA_IN;
23:         end if;
24:         ADDR_IDX <= ADDR_IN;
25:      end if;
26:   end process MEMORY_PRO;
27: end RAM_MEM_BEH;
```

Package SVAL_PKG must contain the definitions:
constant SIZE1K_C : integer := 1024;
constant PTR1K_C : integer := 10;
constant BYTE_C : integer := 8;
Synplify will infer a 1Kbytes RAM memory.

Figure 5. RAM description suitable for Synplify memory inference.

Implementing loops and processing modules

Although the above use of the RAM_MEM component has solved the memory inferencing problem, the memory writing loop (lines 19 to 23, Figure 4) still does not work. The solution is to implement the repetition constructor in a non-conventional way, from the point of view of a high level programming language. No loop constructors are used in the design, and the repetitions are implemented with test constructors (*if* and *case*) in clocked processes, as illustrated in Figure 6, or state machines as shown in Figure 8. The main advantage of this approach is that a code with only test constructors is easier to synthesise, in contrast to a code with test and loop constructors. On the other hand, without loop constructors the implemented code becomes more distant from the original algorithm, and may be more complex to test/debug.

The code in Figure 6 executes the same functionality as the loop in Figure 4. Now the CLK_IN clock signal guarantees the repetition, as the process is executed every time when there is an event of a rising edge in this signal. The test for the end of the loop is executed on line 8, and the signal ADDR_MEM is incremented on

line 13. The memory writing statements are executed on lines 11 and 13. The signals M1_PTR and ADDR_MEM (pointer to memory) are PTR1K_C bits long, and only their 8 LSBs are written to the memory (MEM receives 00H..FFH, 00H..FFH, …).

```
1:  SVALHF_PRO: process (CLK_IN, RESET_NEG_IN )
2:  begin
3:    if RESET_NEG_IN = '0' then                    -- initialisations go here
4:       FLAG_PWRUP <= '1';  WR_MEM <= '1';  M1_PTR <= (others => '0');
5:       DATAIN_MEM <= (others => '0');   ADDR_MEM  <= (others => '0');
6:    elsif CLK_IN'event and CLK_IN = '1' then
7:       if FLAG_PWRUP = '1' then                   -- initialise on powerup
8:          if ADDR_MEM = 1023 then                 -- last memory address is 3FFH
9:            WR_MEM <= '0'; FLAG_PWRUP <= '0';     -- don't write memory/end powerup
10:         else
11:           WR_MEM <= '1'; FLAG_PWRUP <= '1';     -- fill memory
12:         end if;
13:           DATAIN_MEM <= ADDR_MEM(7 downto 0); ADDR_MEM <= M1_PTR; M1_PTR <= M1_PTR+1;
14:      elsif ...
15: end process SVALHF_PRO;
```

Figure 6. Use of if constructor in replace for loops (partial listing of the HF module).

Another way to implement a repetition without a loop constructor, is by means of state machines. This solution was employed in situations where there were more than two alternatives. Using the *case* constructor instead of *if* avoids not only the generation of hardware for priority management, but also the use of nested *ifs*. Figure 8 shows an example of a synthesizable VHDL code for the partial ACF algorithm implemented in VHDL for simulation (listed in Figure 7). On line 17, Figure 8, there is the description of an MAC, which is a basic module of DSP algorithms. The ACF algorithm processes the data in MEM2, and stores the results in MEM1, where MEM2 contains the last samples ready for ACF processing, and MEM1 contains the ACF sum values ready for output. The algorithm listed in Figures 8 and 7 is best described by Equation 2. From Figure 7, high level algorithm or Equation 2, we can see that this algorithm has a cost $O(n^2)$, but from Figure 8 this observation is not so straight-forward, being a problem of VHDL for synthesis implementations, that is, code readability and debugging. Although the original 8051 software and the VHDL implementation are both $O(n^2)$, the last one is 20 times faster than the first one. This gain in performance is a result of two main factors: the parallel nature of VHDL and the distributed memory implementation.

```
1: for i in 0 to n loop
2:   AUXS_1 := MEM1(i); PTR1_2 := HALF_SEP1;
3:   AUX1 := W_AUX1;
4:   for j in 0 to n loop
5:     AUXS_1 := AUXS_1+MEM2(PTR1_2) * MEM2(AUX1);
6:     PTR1_2 := PTR1_2+1; AUX1 := AUX1+1;
7:   end loop;
8:   MEM1(PTR1_1) := AUXS_1; W_AUX1 := W_AUX1 + 1;
9: end loop;
```

Equation 2.

$$MEM1 \leftarrow \sum_{i=0}^{N} \sum_{j=0}^{N} MEM2_j * MEM2_{j+i}$$

Figure 7. Algorithm of multi bit ACF process for input 1 (LF module), implemented in VHDL for simulation

The code listed in Figures 5,6 and 8 can be used to illustrate the code convention adopted in the project, which was based on [25]. The main advantages of adopting this convention are: improvement in the code

readability and re-usability; possibility of using the same code for different target devices; use of different synthesis tools in different platforms; and enabling the development of hierarchical designs. Some of the adopted conventions are (see [25] for a complete suggestion list): use of upper case for user defined names (e.g. signals and types); use of special suffixes, as _IN for input signals and _C for constants; and three spaces for block indentation. In [10], Chapter 6, there is a standard which is fundamental for good synthesis results. For example, the asynchronous set-reset definition, employed in almost all components of the design, states that a process must have only one clock signal (it is possible to have additional signals, but all of them should be asynchronous signals), and this clock edge must be the last *elsif* condition. No sequential statements are allowed before or after the *if* statement. Following these conventions decreases the risk of developing non-synthesizable code. The code in Figure 6 was written according to this standard.

```
1:  case NEXT_STATE_FEW_1 is
2:     when SX_C => NEXT_STATE_FEW_1 <= SX_C;
3:     when S1_C =>                  -- external loop
4:        WR_MEM1_1 <= '0';
5:        if PTR1_1 < 16 then        -- sweep MEM1 (MEM1 == 00H..0FH)
6:           ADDR_MEM1_1 <= PTR1_1; PTR1_2 <= '0' & HALF_SEP1; AUX_IDX1 <= "00000";
7:           ADDR_MEM2_1 <= W_AUX1; AUX1 <= W_AUX1; NEXT_STATE_FEW_1 <= S2_C;
8:        else                       -- end of multi bit ACF (end of external loop)
9:           DONE_SEP1_OUT <= '1'; NEXT_STATE_FEW_1 <= S1_C; -- end of external loop
10:       end if;
11:    when S2_C =>
12:       AUXS_1 <= DATAOUT_MEM1_1; NEXT_STATE_FEW_1 <= S3_C;
13:    when S3_C =>
14:     XDT_MEM2_1 <= DATAOUT_MEM2_1; ADDR_MEM2_1 <= PTR1_2; NEXT_STATE_FEW_1 <= S4_C;
15:    when S4_C =>                  -- internal loop
16:       if AUX_IDX1 < 16 then
17:          AUXS_1 <= AUXS_1 + DATAOUT_MEM2_1 * XDT_MEM2_1; PTR1_2 <= PTR1_2 + 1;
18:          ADDR_MEM1_2 <= AUX1 + 1; AUX1 <= AUX1 + 1; AUX_IDX1 <= AUX_IDX1 + 1;
19:          NEXT_STATE_FEW_1 <= S3_C;     -- repeat internal loop
20:       else
21:          NEXT_STATE_FEW_1 <= S5_C;     -- end of internal loop
22:       end if;
23:    when S5_C =>                        -- repeat external loop
24:       WR_MEM1_1 <= '1'; ADDR_MEM1_1 <= PTR1_1; DATAIN_MEM1_1 <= AUXS_1;
25:       W_AUX1 <= W_AUX1 + 1; PTR1_1 <= PTR1_1 + 1; NEXT_STATE_FEW_1 <= S1_C;
26:    when others => NEXT_STATE_FEW_1 <= S6_C;        -- invalid state, stay here
27: end case;
```

Figure 8. Use of FSMs in replace for loops (partial listing of the multi bit ACF process for input 1 - LF module).

Synthesising the case study

In order to produce a CB for an FPGA using VHDL the designer has to be aware of the fact that VHDL was originally used for description and simulation purposes only, and, in order to obtain good synthesis results, the VHDL constructors used in the design must be as simple as possible. Everything that seems difficult to implement in hardware must be avoided. Furthermore, the designer has to have not only a good knowledge of the target technology and components available in the device library, but also of how the synthesis tool works. Without this background information a designer would spend a considerable amount of time developing a system which is correct from both, the simulation and synthesis point of view, but the CB

generated could not use the resources available efficiently. Therefore the design would require a larger device, and would have a low performance. In chapter 2 of [11] there are suggestions that must be followed in order to develop a VHDL program that performs the same operation before and after the synthesis. The flow showed in Figure 9 was built following the synthesis and PAR documentation.

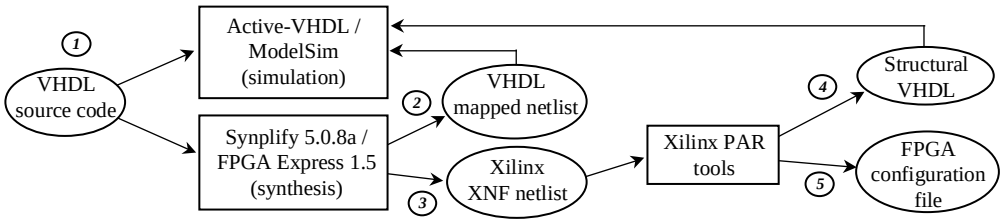

Figure 9. CB generation. The used tools, in rectangles, and the generated files.

An important activity in an HDL design flow is the simulation that can occur in five different phases, at different abstraction levels. In the SVAL-VHDL design, the simulation was executed in three of these phases identified by the numbers in Figure 9: ① *register transfer level* (RTL); ② *post-synthesis*; and ④ *post-route*. The other two possible moments are pre- and post-mapping, both before routing. The advantage of the RTL simulation is the high level of abstraction, enabling the execution of VHDL behavioural code step-by-step, analysing the signal values, and setting breakpoints in a similar way as the debug activity in a conventional programming language. The advantage of executing first the *post-synthesis* simulation is that it avoids the waste of time debugging behavioural modules which cannot be synthesised. At this stage, the design is described at the gate level, in contrast to the system level description, before synthesis. The last simulation is executed over a back-annotated description (④, in Figure 9), which contains the actual block and net timing delay data generated by the PAR process. In all simulation levels the same testbench can be used. If the post-route simulation is satisfactory, then the probability that the CB works well in the FPGA is very high.

First the simulation was executed using ModelSim. When the license expired, a copy of Activity-VHDL was received as a donation from Aldec's University program. This Simulator has a better user interface with additional features not present in ModelSim, being adopted as the official simulation tool. Synplify was chosen as the synthesis tool mainly because of its ability in generate netlists for different FPGA vendors, from a standard VHDL description. Synthesis tools as FPGA Express 1.5, for example, cannot generate adequate hardware from some VHDL descriptions, without using pre-optimised cores. The memory inference, as stated before, is an example of such a module.

Lessons learnt

VHDL was chosen for this project, mainly, because of its device independent characteristic. Unfortunately, it was noticed later that VHDL programs are not synthesis tool independent as, for example, the memory problem discussed in section 4.2. Another motivation for using VHDL is the availability of a significant number of simulators and synthesisers. FPGA vendors usually include a VHDL and a Verilog synthesiser in their EDA tools package. Also, the expressive community of VHDL users is an important incentive for using this language.

Latch inference is another hardware subject that must be considered by the VHDL for synthesis developer. When a signal is not assigned in all possible situations, then the synthesis tools infer latches in the circuit in order to allow it to "remember" a previous signal value. One problem here is the additional hardware required to implement the circuit. Another problem is that some FPGAs may not have latches in their CLBs, with flip-flops used instead. In terms of area, this solution can be very costly to the overall design. The code listed in Figure 6 was developed in a way that latche inferencing is avoided. In this code the signals are assigned in all if and else conditions, for instance if there were no assignment for FLAG_PWRUP in the RESET_NEG_IN = '0' condition, then a latch would be infered.

The original microprocessor implementation of SVAL, was lost in the sea after its mission has been completed. Therefore, in order to demonstrate the gain in performance obtained with SVAL-VHDL the number of cycles necessary to run the SVAL routines, are compared to the SVAL-VHDL equivalent ones. From Table 1 it is possible to observe that with the 8051 instruction set 4,518 cycles are required to execute the “new energy step” routine, while using VHDL it is completed in only one cycle. For the low frequency module, the VHDL implementation is more sequential, resulting in more cycles for the execution of the routines. However, the performance improvement is still significant.

	Microcontroller	***FPGA***	***Improvement rate***
New energy step (HF)	*4,518T*	*1T*	*4,518 times faster*
“Buncher” histogram (HF)	*8T .. 36T*	*1T*	*8 to 36 times faster*
Next o/p word requested (HF)	*18T .. 1018T*	*1T .. 68T*	*18 to 15 times faster*
O/P one bit ACF (LF)	*1,240T*	*48T*	*26 times faster*
O/P multibit ACF (LF)	*1,334T..3,438T*	*132T..143T*	*10 to 24 times faster*

Multibit ACF processing (LF)	*11,116T*	*288T*	*38 times faster*

Table 1. Performance comparison for the ACF application.

RAM implementation is the most consuming module of FPGA designs. However, because of the flexibility of VHDL/FPGAs that allow the developer to have in the design only the amount of memory necessary for the application, the 29 Kbits of RAM memory present in the SVAL board were reduced to 14.5 Kbits in the SVAL-VHDL implementation.

The look-up tables (LUTs) of the XC4000 CLBs can be configured to implement a pair of 16x1 RAMs, a 32x1 RAM, or a 16x1 dual port RAM [26]. Using this information and the fact that a 4x1 multiplexer can be implemented in one CLB, the amount of CLBs needed for memory implementation, including the output bit selection (using the mux) can be estimated. For instance, the number of CLBs necessary to implement the 1Kbytes dual port RAM for the HF module can be estimated as follows: 64 CLBs are used for 1Kbit memory implementation (1024 ÷ 16 = 64 CLBs) and 21 CLBs for the 64x1 mux (4x1 mux = 1 CLB; 16x1 mux = 5 CLB – four 4x1 mux + 1 CLB for the final output; 64x1 mux = four 16x1 mux + 1 CLB for the final output). So, 64 + 21 = 85 CLBs * 8 bits = 680 CLBs for the 1 Kbytes dual port RAM. Using similar calculation, for all memory blocks shown in Figure 3, it is possible to estimate that 1,010 CLBs are necessary to implement the whole SVAL-VHDL memory. After the PAR execution the actual usage is 1,112 CLBs.

Another space consuming module is the multiplication. A 16x16 pre-optimised multiplier, generated by CoreGen occupies 213 CLBs. Using the '*' operator in VHDL, Synplify can generate a multiplier that consumes 215 CLBs, in both cases considering an XC4000 FPGA, and optimising for area. As there is no significant gain in area using the pre-optimised core, and in order to have all the system described in VHDL, and at the same abstraction level, the '*' VHDL operator was used.

From a first estimation, the XC4085XL was chosen as the target device. After the implementation and the PAR execution, it was concluded that the original board could be shrunk to this single device. The area usage estimation is important in an FPGA implementation, because if the PAR execution produces results considered different from the expected ones, then it is possible to identify problems in the VHDL description.

Conclusions

In 1997, when the capacity of the largest devices was between 10K to 50K gates, limited FPGA gate capacity

was the main obstacle to their use as configurable computing system [3]. The 1M gates XCV1000 Xilinx FPGA [26] launched in 1998 represents an increase of 20 to 100 times in the gate count in only two years. Other limitations pointed out in [3] were the scarceness of memory structures and interfaces. In the XCV1000, there are 16K bytes of embedded RAM, distributed in 32 blocks, and there are enough CLBs and input/outputs blocks (IOBs) for the implementation of control logic for external memory access. Altera, another market leader in the FPGA segment, has devices as large as 250K gates [27], which is more than enough to implement designs with the dimensions of the one described in the present work. With the continuous increase in gate count and with the advances in the fabrication processes, area/speed limitations will not be a problem in a few years.

The SVAL project was a good choice for the case study, for two main reasons: with an existing design it is possible to measure the effectiveness of the new FPGA implementation, and because it represents a typical embedded system application. An important characteristic of this class of application is that, usually, only a small amount of memory is necessary, normally, for temporary data storage and for small tables. This feature is important, because in a large memory system, complex memory hierarchy strategies are needed to decrease the performance problems associated with the von-Neumann bus bottleneck. The implementation of cache or virtual memory management is possible in an FPGA, but is not usually an efficient use of FPGA.

Pipeline is an important technique used to improve the performance of FPGA designs, without significant increase in the area usage. A design is pipelined by including registers between processing modules, allowing the data to be ready for the next module, whilst the last module can be used for processing new data. In the case of SVAL-VHDL, as most of the design was implemented in a high level of abstraction, this technique was not used, at least explicitly. However, analysing the circuit generated by the synthesis tool, it is possible to identify pipeline blocks automatically generated.

The choice of Xilinx FPGAs is justified by their architecture features more suitable for arithmetic and DSP applications than those of other FPGA vendors, and because of the active Xilinx University Program which provides full functional software tools and prototyping boards with small FPGAs.

One important problem of using FPGAs to replace microprocessors in special purpose embedded systems, is the high cost involved. Depending on the FPGA size, with today's technology, it can be much more expensive than using an 8051 but this is not a problem in applications where only a couple of boards are needed, and factors such as board size, high performance with less power consumption, and improvement in

the dependability features are essential. Another concern is the time necessary for learning an HDL language and the use of associated development tools. This can be compared with microprocessors, which not only possess well developed tools, but also have a large base of people that know how to program them. Note however that the time necessary to learn how to use FPGAs and tools is not so different from learning to program a new type of microprocessor.

In order for FPGAs to be able to replace microprocessors, a major improvement is necessary in the capabilities of synthesis tools. On the other hand, to improve the tools, it is necessary to reformulate the present FPGA architectures, which are flexible and powerful, but very difficult to be configured. A good example is the XC6200, the only reconfigurable Xilinx FPGA at the block level, whose production was discontinued because of the difficulties of developing efficient tools to explore its resources in an adequate way.

One important conclusion of this work is that to develop FPGA applications using HDLs with the available synthesis tools, as stated before, the best strategy is to keep the HDL description as simple as possible.

References

[1] Graham, P and Nelson, B, **A Hardware Genetic Algorithm for the Travelling Salesman Problem on SPLASH 2**, In 5th International Workshop on Field Programmable Logic and Applications, Oxford, England, Aug. 1995, 352-361.

[2] Wirthlin, M J and Hutchings, B L, **DISC: The Dynamic Instruction Set Computer**, In Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing, 1995, 92-103.

[3] Mangione-Smith, W et al, **Seeking Solutions in Configurable Computing**, IEEE Computer, Nov. 1997, 38-43.

[4] Villasenor, J and Mangione-Smith, W, **Configurable Computing**, Scientific American, Jun.1997, 66-71.

[5] Saleeba, Z, **A Self-Reconfiguring Computer System**. PhD Thesis, Department of Computer Science, Monash University, Victoria, Australia, 1997.

[6] DeHon, A, **Reconfigurable Architectures for General-Purpose Computing**. PhD Thesis, Artificial Intelligence Laboratory, MIT, USA, 1996.

[7] Turley, J, **CPU and DSP Cores vie for ASIC Designers' Attention,** Computer Design, Jan. 1997,

75-86.

[8] Davidson, J, **FPGA Implementation of a Reconfigurable Microprocessor**, In Proceedings of the IEEE Custom Integrated Circuits Conference, 1993, 3.2.1-3.2.4.

[9] Mirsky, E and DeHon, A, **MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources**, Proc. FPGAs Custom Computing Machines, IEEE CS Press, Los Alamitos, Calif., 1996, 157-166.

[10] IEEE, **P1076.6/D1.12 Draft Standard For VHDL Register Transfer Level Synthesis,** IEEE, 1998.

[11] Xilinx, **Synthesis and Simulation Design Guide**, Xilinx, 1998.

[12] Synplicity, **Synplify Better Synthesis – User Guide release 5.0**, Synplicity, 1998.

[13] Perry, D L, **VHDL - Second Edition,** McGraw-Hill Series on Computer Engineering, New York, 1993.

[14] Frontier Design, **A|rt Builder (http://www.frontierd.com/)**, Frontier Design, 1999.

[15] Aubury, M. and Watts, R. **Handel-C Compiler, Release 1 Documentation and User's Manual,** Oxford University Computing Laboratory, Oxford, 1995.

[16] Hoare, C, **Communicating Sequential Processes**, Commun. of the ACM, v.21, n.8, Aug. 1978, 666-677.

[17] Chu, M. et al, **Object Oriented Circuit-Generators in Java,** Proceedings of the International Symposium on Field-Programmable Gate Arrays for Custom Computing Machines (FCCM '98), 1998.

[18] Intel, **Intel Architecture Optimization - Reference Manual,** Intel Corporation, 1999.

[19] Bezerra E, et al, **Improving the Dependability of Embedded Systems Using Configurable Computing Technology,** XIV International Symposium on Computer and Information Sciences (ISCIS'99), 1999, Izmir, Turkey.

[20] Vargas, F, et al, **Optimizing HW/SW Codesign Towards Reliability for Critical-Application Systems**. 4th International IEEE On-Line Testing Workshop, Capri, Italy, 1998, 17-22.

[21] Borrione, D, **From Hdl Descriptions to Guaranteed Correct Circuit Designs**, Proceedings of the IFIP, 1987.

[22] Beauchamp, K and Yuen, C, **Digital Methods for Signal Analysis,** George Allen, 1979.

[23] Gough, M P **Particle Correlator Instruments in Space: Performance Limitations Successes, and**

the Future, American Geophysics Union, Geophysical Monograph 102, 1998, 333-338.

[24] Xilinx, **Coregen (http://www.xilinx.com/products/logicore/coregen),** Xilinx, 1999.

[25] Figueiredo, M and Graessle, T, **VHDL Style Guide**. Adaptive Scientific Data Processing group (http://fpga.gsfc.nasa.gov/asdp/index.htm), 1999.

[26] Xilinx, **The Programmable Logic Data Book**, Xilinx, San Jose, 1999.

[27] Altera, **Altera Data Book**, Altera, 1998.