From C to netlists: hardware engineering for software engineers?

by Ian Alston and Bob Madahar

The software programmable multiprocessor architecture has been employed extensively over the past two decades for embedded signal-processing applications. However, the increased complexity of such systems has, in many cases, required the use of hardware acceleration to meet the growing time-critical apsects of the design. Today's field-programmable gate arrays (FPGAs) offer an alternative or additional acceleration platform, especially to an application-specific integrated circuit (ASIC). However, the traditional low-level development methods, such as schematic capture or hardware description languages (HDLs), employed to implement these hardware accelerated parts of the design result in a design lifecycle mismatch between the rapid development techniques available for the software programmable parts. This paper presents high-level design languages that enable users to generate netlists for FPGAs directly from high-level C-like languages, thereby offering an equivalent programming solution to that available with microprocessors. It details how one of these languages can be integrated into a high-level design flow for the rapid development of heterogeneous embedded signal-processing systems and presents results from a benchmark.

1 Introduction

The increasing complexity of digital signal-processing (DSP) algorithms in embedded applications, including image and control processing algorithms, requires high processing power to satisfy the real-time constraints often imposed by such applications. This processing power can be achieved by parallel processing devices and parallel multiprocessor architectures. For the latter, DSP designers and commercial vendors have developed high-performance (communications and processing), modular, scalable, and low-latency architectures based on commercial off the shelf (COTS) processors. For the former, DSP designers have traditionally chosen to implement the time-critical aspects of the design in an application-specific integrated circuit (ASIC).

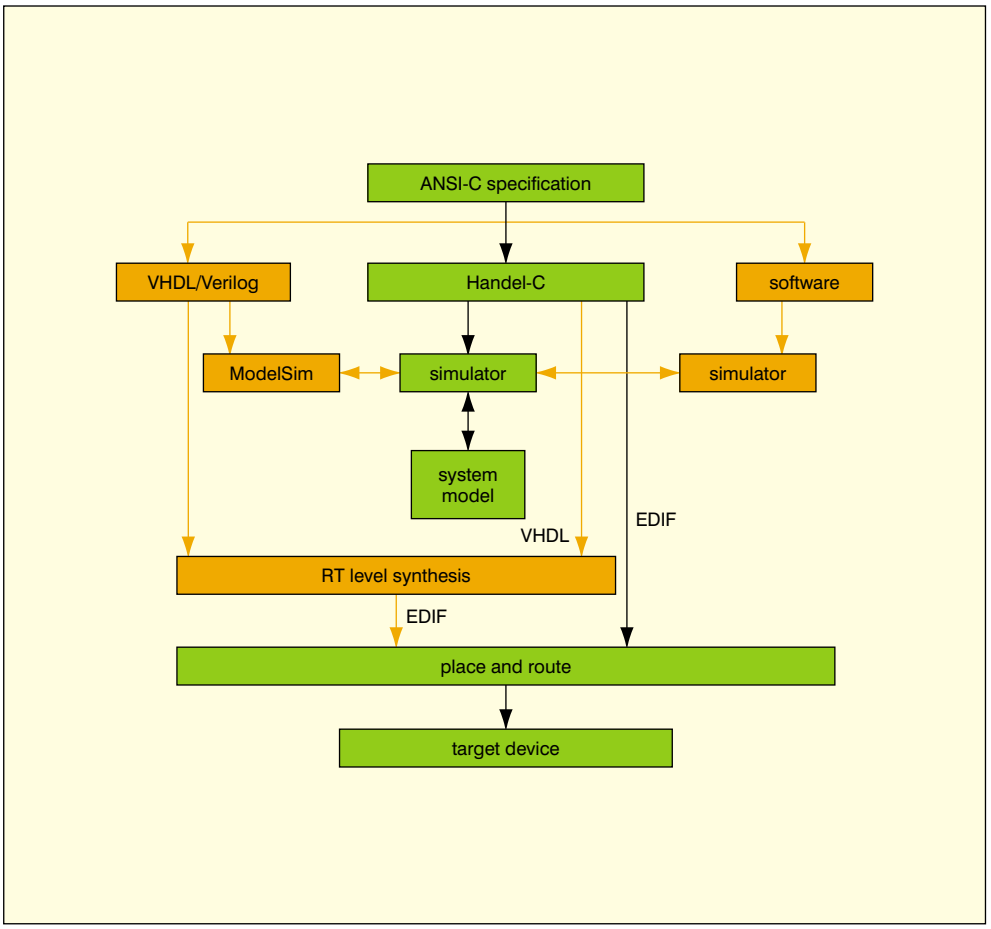
The significant growth in the performance and logic capacity of today's 'low cost' field-programmable gate arrays (FPGAs), however, now offers a competitive alternative to ASICs and COTS processors in multiprocessing architectures, especially for front-end processing.

The FPGA is a reconfigurable device that allows designers to build part, or all, of their design in hardware rather than software. By exporting functionality and embedding it into the hardware, significant performance improvements can be realised because the functionality doesn't have to be split into individual instructions for the central processing unit (CPU) to fetch, decode and apply. It also provides the opportunity to exploit the inherent concurrency of digital circuits, i.e. the device can be configured, or partitioned, into multiple pipes or subsystems all of which could run concurrently with each other. In this way any inherent parallelism in the algorithms can be exploited to its full extent.

What are however needed are high-level languages and tools for rapid system design to support FPGAs in heterogeneous architectures akin to the tool support for COTS processors[1]. The low-level development methods, such as schematic capture or hardware description languages (HDLs), employed to implement FPGA designs are wholly inappropriate (in terms of time, cost and effort, including specialists) as system level tools for rapid embedded-system development. Instead we need to raise the design to higher levels of abstraction and provide an integrated approach to software and hardware design that supports heterogeneous systems.

In this paper we describe a maturing language technology known as system-level design languages which have the potential to describe both the software and hardware elements of the design at a high abstraction level using a common high-level language. These

Fig. 1 Handel-C design flow



languages are derived from traditional programming languages such as C, C++ and Java, and hence allow designers to use familiar language syntax for FPGAs. They also enable the direct generation of the HDL code and the netlists needed to 'program' the devices. Hence we can attain equivalence with the programming environment for microprocessors and benefit from unified development environments.

We also provide examples of these languages, and the results of our evaluations in terms of ease of use and efficiency optimisation. An example design flow and signal-processing function development employing one of these languages is presented. This is completed with measurements of the metrics to quantify the usefulness of this type of approach in the rapid development of embedded signal-processing applications.

2 System-level design languages

With the increased complexity of systems in terms of functionality and processing platforms, the need for an integrated approach to software and hardware design is becoming more and more important. To ensure that systems meet customer requirements, design decisions need to be made at the system level and as early as possible within the design process. Therefore, in order to allow interaction between the various components of a system, (system, hardware, and software) designers have tended to create executable specifications for their systems. For the most part, these are functional models written in a language like C or C++. The use of these languages has become ubiquitous for a number of reasons. Firstly, they hide the hardware complexity of the processing devices, are simple to use, and enable users to rapidly develop compact and efficient system descriptions, including the necessary control and data abstractions, the external interfaces, the algorithms and the processing. Secondly the languages are supported by efficient compilers, across a broad range of devices, and provide code portability. Thirdly there are a large number of development tools and support tools associated with them that help to improve the overall system software engineering process. Finally they are familiar to new engineers, who are taught these languages at university. Hence it is logical to extend these developments to include programmable logic devices as well as microprocessors. However this is problematic as the languages do not have the constructs necessary for model timing, concurrency, and reactive behaviour, all of which are needed to create accurate models of systems containing both hardware and software.

To overcome these limitations, language developers have adopted two approaches. The first relies on C/C++ syntax extensions and thus requires the development of separate compilers to parse and process the new syntax. The second approach relies on the addition of class libraries to an extensible language, such as C++ or Java, which model the hardware aspects of the design. This approach has the advantage that standard compilers and development tools can be used for the

simulation of the design.

The following sections briefly describe the three most common system-level design languages available today.

Handel-C[2]

The Handel-C language and supporting design environment is developed by Celoxica (previously known as Embedded Solutions Ltd. (ESL)), which was formed by the University of Oxford in 1996 to commercialise its research into the Handel-C high-level programming language. It is based on the work on Communicating Sequential Processes[3] and the supporting Occam language[4]. Handel-C is aimed at compiling high-level algorithms directly into gate level hardware. In order to support the use of the language the vendor also supplies a graphical design environment called DK1, which incorporates simulator, debugger, compiler and implementation generation, in either EDIF (Electronic Design Interchange Format) netlist format or VHDL (Very high speed integrated circuit Hardware Description Language).

Handel-C uses the syntax of conventional C with the addition of inherent parallelism. Sequential programs can be written in Handel-C, but to gain maximum performance benefit from the target hardware parallel constructs have been added to the language. Handel-C is designed to allow you to express your algorithm without knowing how the underlying computation engine works. This philosophy makes Handel-C a programming language rather than a hardware description language. That is, Handel-C is to FPGAs what a conventional high-level language is to microprocessors. The design flow is shown in Fig. 1.

It is important to note that the hardware design that Handel-C produces is generated directly from the source program. There is no intermediate 'interpreting' layer as exists in assembly language when targeting general-purpose microprocessors. The logic gates that make up the final Handel-C circuit are the assembly instructions of the Handel-C system. Unlike similar behavioural synthesis tools that stop at the register transfer level (RTL), Handel-C generates gate level (EDIF) netlists ready for FPGA placement and routing using the tools supplied by the FPGA vendor.

The Handel-C compiler provides estimates of the design complexity and the simulator provides information on the clock-cycle-based performance. The design of the language and the compilation process ensures that all assignments are performed in a single clock cycle. Furthermore, the control logic of the various language constructs imposes no additional clock cycles on the implementation. Thus assignment takes exactly one clock cycle and all other language statements add precisely zero additional clock cycles, although they add to the combinatorial delays, which can lengthen the fundamental system clock cycle. The incorporation of the language's parallel construct—the PAR statement—enables performance to be optimised with respect to area. Special input/output (I/O) constructs, the Port and Channel constructs, are provided to interconnect to external interfaces and to parallel branches/segments, respectively. As an example for the latter, one parallel branch outputs data onto the channel and the other branch reads data from the channel. Channels also provide synchronisation between parallel branches because the data transfer can only complete when both sender and receiver are ready.

Hardware interfaces can be defined and the compiler-generated EDIF netlist passed to the FPGA vendor's (Xilinx or Altera) place-and-route and timing analysis tools. Interfaces (or plug-ins) are also available to link the Handel-C simulator with other applications to provide a co-simulation capability that supports system-on-chip (SOC) development. This co-simulation environment allows for hardware and software partitioning via interfaces between the Handel-C simulator and software simulation tools such as ARMulator[5] and Windriver's SDS SingleStep™ debugger[6]. Optional VHDL code generation supports IP (intellectual property) integration through interfaces with HDL development tools for co-simulation (e.g. ModelSim[7]) and RTL synthesis (e.g. Mentor Graphics' LeonardoSpectrum™[8] [formerly offered through Exemplar Logic]).

Although Handel-C supports a rich subset of the ANSI-C language, the following syntax restrictions are imposed:

- Floating point is not supported.
- Functions may not be recursive.
- Variable-length parameter lists (...) are not supported.
- The union object is not supported.
- You may not change the width of a variable by casting.
- You cannot convert pointer types except to and from void, between signed and unsigned and between similar structs.
- Statements in Handel-C may not cause side-effects. This has the following consequences:
  - Local initialisations are not supported.
  - The initialisation and iteration phases of loops must be statements, not expressions.
  - Shortcut assignments (i.e. += −= *= /= %= <<= >>= &= |= ^= ++) must appear as standalone statements and not in the middle of more complex expressions.
- Limited standard library.

While Handel-C does not provide any fixed-point analysis it is possible to integrate other tools into the design flow for this purpose.

In general the behavioural synthesis capability allows software engineers to progress rapidly from C to gate level without any knowledge of the underlying hardware. This overall is the strongest point in favour of using Handel-C. However, to create parallelism within the design and thereby exploit the concurrency offered by the hardware, manual editing of the code is required. Hence an understanding of the code's execution sequencing is necessary to ensure that this doesn't result in a change to the functional behaviour of the application.

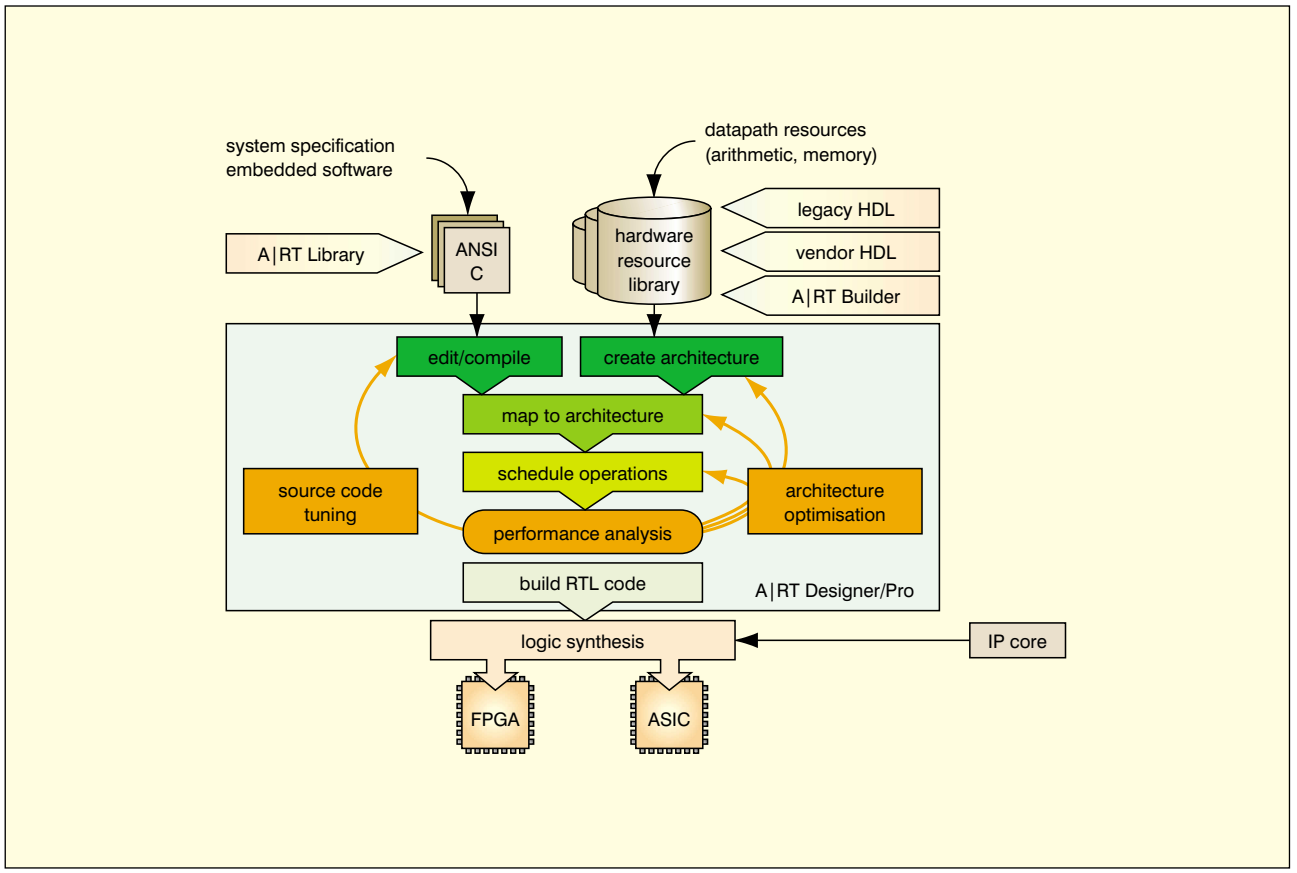


Fig. 2 A|RT design flow

A|RT[9]

Frontier Design was spun out of the Mentor Graphics' European Development Centre, Leuven (Belgium), in 1997. Frontier Design has applied its 15 years of experience in transforming DSP algorithms (written in C code) into working silicon to create a methodology that is called 'Algorithm to Register Transfer', or A|RT. Within this methodology are a number of tools that support the translation of algorithms into implementation.

A|RT Designer is a tool that implements an algorithm into digital synchronous hardware. The algorithm is expressed in the C language and assists the designer in the development of a processor or processor-like architecture, customised for the algorithm that has to be executed on this architecture. The generated processor consists of a set of datapath resources, controlled by a pipelined VLIW (very long instruction word) type controller. These datapath resources may be shared over different clock cycles. This means that A|RT Designer is able to apply resource sharing to operations in the source description. The resulting modularity and parallelism of the architecture can be used to create a design that is optimised for specific needs regarding throughput, area cost or power consumption.

During the translation of the C specification to the RTL, the C source code is automatically analysed to determine which C operators are used and to select a suitable resource or resources from the A|RT hardware library with which to implement them. Unless specified by the user, only one instance of each resource type selected will be included in the final processor architecture. A pragma editor allows the user to modify the selection process, give the resources more informative instance names and scan the available libraries for additional resources. The user can also write their own pragma statements, which offers a powerful mechanism for design optimisation.

The A|RT Designer tool comes with two other tools:

- A|RT Library provides extensions to the C language for fixed-point arithmetic and analysis. A|RT Library helps the user to optimise the design by minimising the size of variables and to detect errors that might occur when converting from floating-point arithmetic.
- A|RT Builder is a C-to-HDL translation tool. It supports the same fixed-point extensions as A|RT Designer and a similar subset of the ANSI-C language. A|RT Builder has its own graphical user interface (GUI) and can be used to develop optimised cores for addition to the A|RT Designer hardware resource libraries or for use within an HDL development environment.

The design flow used by A|RT is shown in Fig. 2. Although this figure implies that the system specification can be written in ANSI-C, there are restrictions imposed by the behavioural synthesis tools of A|RT Designer and A|RT Builder. Nonetheless, the A|RT language is a rich subset of the C language. The restrictions or features not supported are:

- No floating point types.
- Functions may not be recursive.
- Variable-length parameter lists (...) are not supported.

- The union object is not supported.
- The sizeof operator is not supported.
- Pointers and pointer arithmetic are not supported. You should use indexing in an array instead of a pointer.
- String literals are only supported for the initialisation of A|RT Library type variables.
- Arrays with incomplete type descriptions (e.g. a[ ]) are not supported.
- There is no support for division (/) and only limited support for modulo (%).
- For shift operations, if the shift value is negative, the left operand will be shifted in the other direction.
- The goto statement is not supported.
- External functions are not supported.
- The declaration of variables as extern is not supported.
- The standard C library is not supported.

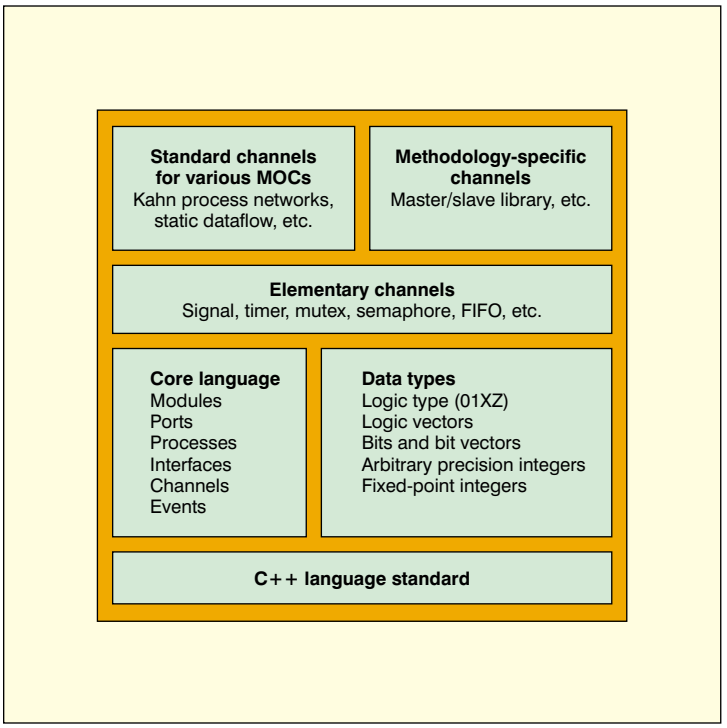


Fig. 3 SystemC language architecture

As a C-based design tool, with provision for fractional arithmetic, A|RT Designer and its associated tools are suited for embedding DSP algorithms in hardware. The automated architectural synthesis and scheduling capabilities will allow engineers to progress rapidly from C to HDL without any knowledge of the underlying hardware. Manual resource allocation and assignment is necessary, however, to create parallelism within the design and thereby exploit the concurrency offered by hardware. Knowledge of the tool's synthesis methodologies is necessary to make full use of the optimisation features (such as loop folding) which may require modification of the source code to be effective.

The performance analysis features and quick run-time of the tool made it very easy to determine the impact of design changes on the performance in terms of clock cycles. However, because the tool stops at the RT level the impact on logic complexity and clock rate can't be determined without completing the logic synthesis step and possibly the placement and routing as well.

Overall, we conclude that the A|RT Designer is more suited to hardware engineers who wish to increase the level of abstraction of their system designs.

SystemC[10]

SystemC is a modelling platform consisting of C++ class libraries and a simulation kernel for design at the system-behavioural and RT levels. There are abstract definitions for the fundamental components of programmable hardware such as communications, memory and processing. Designers create models using SystemC and standard ANSI C++. The Open SystemC Initiative (OSCI) is a collaborative effort among a broad range of companies to support and advance SystemC as a *de facto* standard for system-level design. OSCI provides an interoperable, modelling platform to exchange very fast system-level C++ models and develop seamless tool integration. The contributing EDA (electronic design automation) vendors are thus able to create tools that are automatically interoperable.

During the early days of this initiative, lack of co-operation and patent-right difficulties delayed progress. However, these original difficulties have now been resolved and the latest offering of the SystemC standard, version 2.0, was released in 2001.

SystemC provides a set of modelling constructs that are similar to those used for RTL and behavioural modelling within an HDL, such as Verilog or VHDL. In a similar way to HDLs, users can construct structural designs in SystemC using modules, ports and signals. Modules can be instantiated within other modules, enabling structural design hierarchies to be built. Ports and signals enable communication of data between modules, and all ports and signals are declared by the user to have a specific data type. Commonly used data types include single bits, bit vectors, characters, integers, floating-point numbers, vectors of integers, etc. SystemC also includes support for four-state logic signals (i.e. signals that model 0, 1, X, and Z).

An important data type that is found in SystemC but not in HDLs is the fixed-point numeric type. Fixed-point numbers are frequently used in DSP applications that target both hardware and software implementations, since floating-point operations usually consume more hardware resources. An example fixed-point operation might be to add two signed numbers that have three bits of integer precision and four bits of fractional precision and assign the result to a similar fixed-point number. Often users wish to specify rounding and overflow modes (e.g. saturate or wrap on overflow) when using fixed-point numbers. It is easy and natural to model fixed-point numbers in SystemC, but this is very difficult to do in HDLs. In SystemC, concurrent behaviours are modelled using processes. A process can be thought of as an independent thread of control that resumes execution

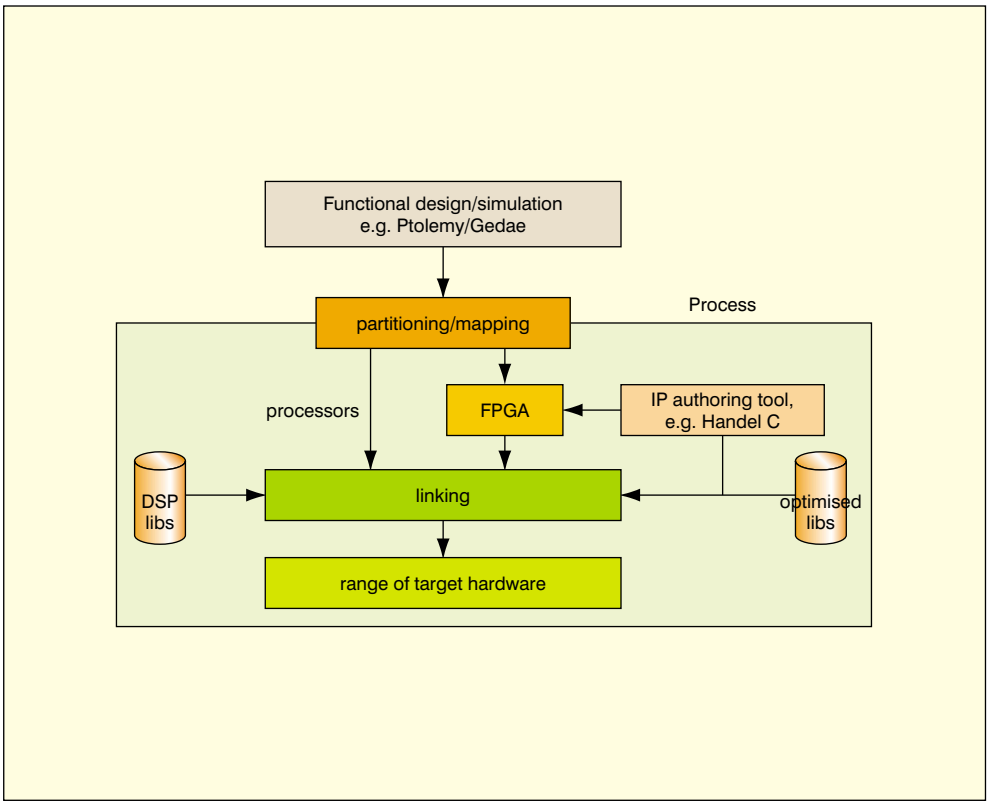


Fig. 4 Heterogeneous platform design flow

when some set of events occurs or some signals change, and then suspends execution after performing some action. However there is a limited ability for specifying the condition under which a process resumes execution—the process can only be sensitive to changes of values of particular signals, and the set of signals to which the process is sensitive must be prespecified before simulation starts.

Fig. 3 summarises the SystemC language architecture. There are several important concepts to understand from this diagram.

- All of SystemC builds on C++.
- Upper layers within the diagram are cleanly built on top of the lower layers.
- The SystemC core language provides only a minimal set of modelling constructs for structural description, concurrency, communication, and synchronisation.
- Data types are separate from the core language and user-defined data types are fully supported.
- Commonly used communication mechanisms such as signals and FIFOs (first-in, first-out memories) can be built on top of the core language. Commonly used models of computation (MOCs) can also be built on top of the core language.
- If desired, lower layers within the diagram can be used without needing the upper layers.

Although the SystemC modelling platform is freely available from the OSCI, routes to implementation rely on the EDA vendor offerings. Though these will be driven by the requirements of the SOC market, the technical evolution is likely to have wider implications and opportunities for the defence embedded-systems market.

3 Example design flow

The use of a rapid-prototyping methodology and its supporting tools has been well documented and is becoming a mature process. This process is based on the C language and relies on optimised signal-processing libraries in order to improve efficiency of the final implementation. The development of system-level design languages, also based on C, thus provides us with the ability to support heterogeneous platforms, i.e. those consisting of DSPs, general-purpose processors and FPGAs, within an extended rapid-prototyping methodology. A typical extended design flow for such heterogeneous platforms developed under the ESPADON (Environment for Signal Processing Application Development and Prototyping) project is shown in Fig. 4[1]. In this design flow the functional design tool is used to partition and map the functional behaviour onto the heterogeneous processing elements.

Mirroring the requirements of the conventional rapid-prototyping flow (shown on the left of the diagram), the route to FPGA relies on:

- extrapolating the functional specification to the highest abstraction level in order to specify functional blocks as domain-specific elements
- using a system-level design language as an IP integration platform and authoring tool
- having the ability to integrate optimised IP cores within the system-level design language.

Within the ESPADON programme, a detailed evaluation of Handel-C and A|RT Builder has been performed. Although the languages and associated design environments provide similar features, the previous

discussions on the languages have highlighted that the Handel-C language allows software engineers to progress rapidly from C to gate level without any knowledge of the underlying hardware. As such it was chosen within the ESPADON programme as it mirrors the rapid-prototyping methodology employed for general-purpose processors and DSPs.

4 Example development

In keeping with the objectives of the ESPADON demonstrator programme, this design flow has been used to implement a radar and sonar beamformer targeted for a heterogeneous platform. The initial element of the algorithm to be placed on the FPGA was chosen to be the fast Fourier transform (FFT) used within the beamforming process. A standard complex radix-2 FFT example, written in C, was selected so as to demonstrate the design flow and to evaluate the ease of porting standard C applications to FPGAs. The only modification to the original FFT code made prior to entering the design flow described above was to remove the calculation of the FFT butterfly coefficients from within the main FFT loops. These only need to be generated once per pass, and hence the code was modified to use precomputed ones. Though this type of FFT algorithm is not the most efficient to implement on an FPGA, it was used to show how a parameterisable function could be generated using Handel-C.

The first steps in the porting process were:

- removal of the floating-point data types
- modification of the code to remove the unsupported C features.

The final change to the code was required due to a problem in the multiply mechanism in the nested loop within the FFT. During standard multiplication the least significant bits are chosen, but as one of the values to be multiplied was a fraction the most significant bits needed to be used. This problem was overcome by

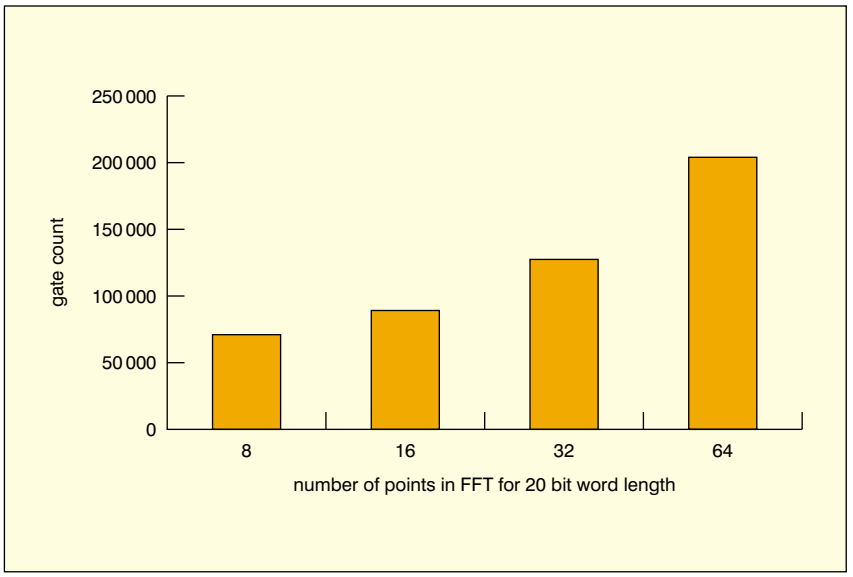


Fig. 5 Gate count and FFT size

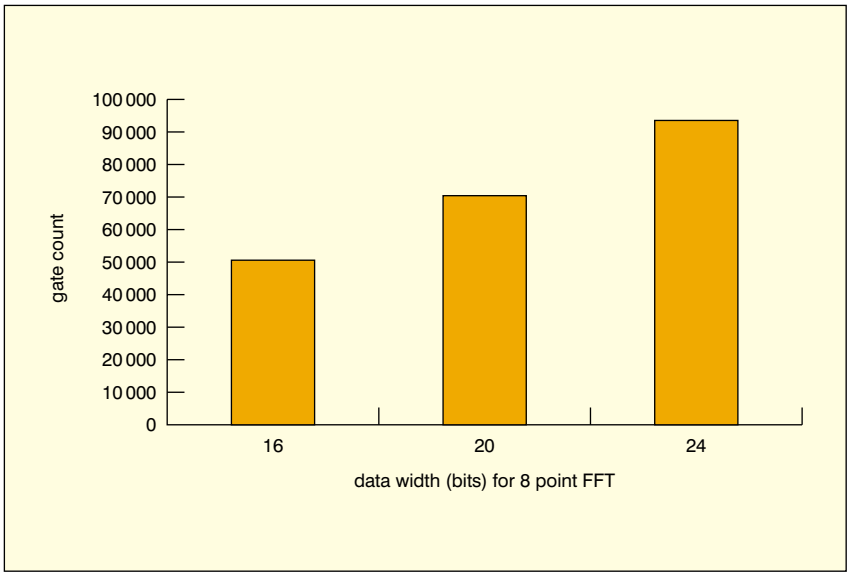


Fig. 6 Gate count and word length

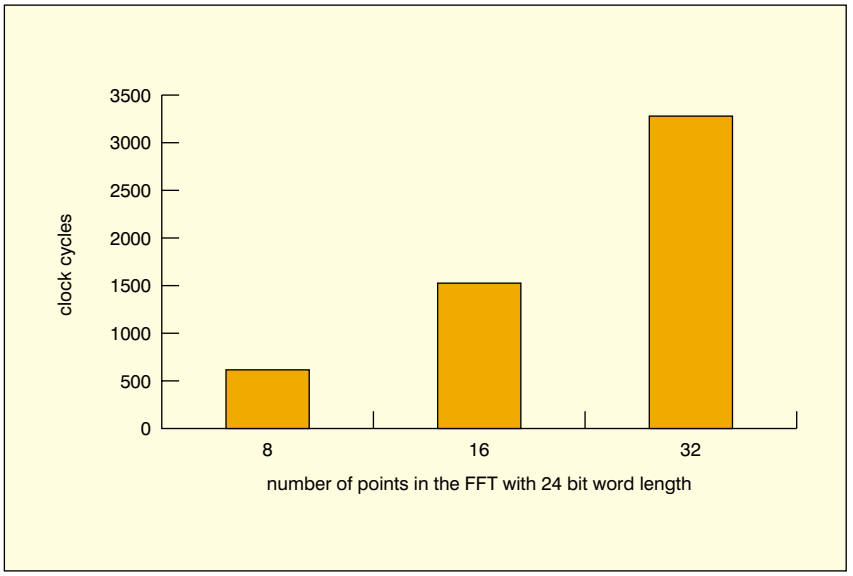
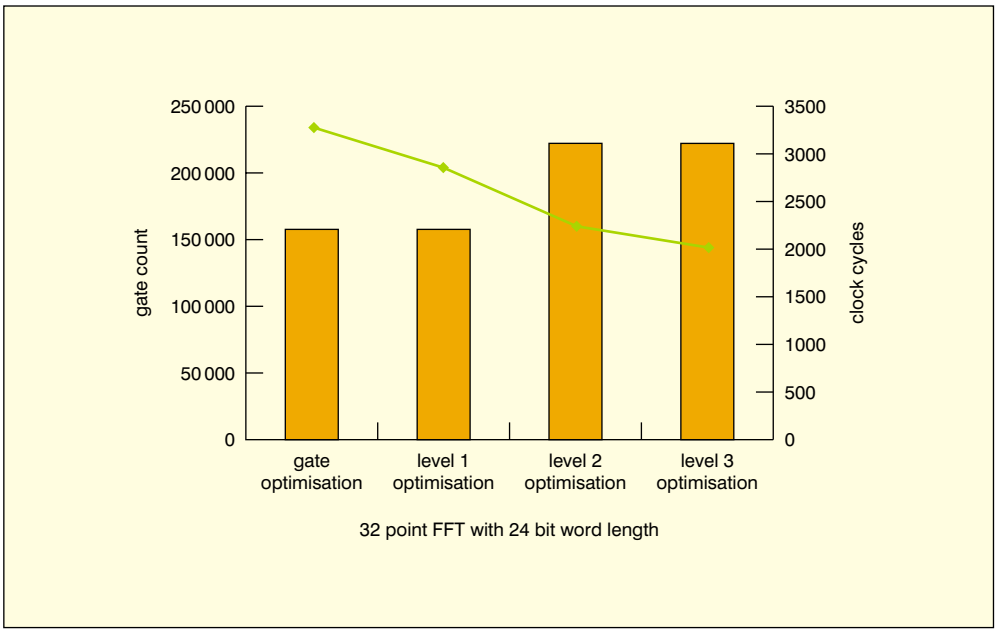


Fig. 7 Execution speed for various FFT lengths

Fig. 8 Graph to show the optimised results



creating a function which would do the multiply and then correctly choose the most significant bits of the result.

The performance characteristics of the FFT were measured for different data lengths and different word lengths as reported in the next subsections. Throughout this porting process, the debugging capabilities of the tool were found to be extremely useful in being able to single-step through all the code as per a standard software-debugging environment.

Performance results

The results that are shown in Figs. 5–7 were obtained from the Handel-C simulator before either timing or size optimisation. The devices targeted for the implementation were the Xilinx 4000XV series of FPGAs.

Figs. 5 and 6 show how the complexity (gate counts) of the FFT increases with the number of data points and the word length, respectively. Fig. 7 shows how the speed of operation of the FFT algorithm changes with the data length of the FFT. The number of clock cycles was obtained from the simulator by stopping the simulator at the end of the simulation and viewing the master clock cycles.

Optimising the code for speed

A number of optimisation steps were available for improving the execution speed. These were applied to a 32-point FFT with a 24-bit word length as follows. The overall results from the simulator are shown in Fig. 8.

- Level 1: The first step was to use the PAR parallel construct to optimise the parallel execution of (*a*) the add and subtract statements and (*b*) the variable assignments within the main FFT routine. This had an immediate and significant impact on execution speed (Fig. 8, Level 1 optimisation) with only a small increase in gate size.
- Level 2: The second step was to place the non-standard multiply routines within a PAR statement. This meant that the multiply routine had to be altered so that there was an array of multipliers as opposed to a single one. Consequently the execution speed improved significantly but the gate count also increased significantly.
- Level 3: The final step was to combine the optimisations from Levels 1 and 2, giving a 39% reduction in clock cycles with a 41% increase in gate count.

It should be noted that *the optimisation was achieved in only a few hours*, thus demonstrating the strength of the language and the simulation tools.

Practical use of the FFT implementation

Having completed the generation of a working implementation of an FFT for an FPGA, the design flow described in Section 3 was benchmarked with the ESPADON sonar beamformer application. This particular beamformer employs an 8-point FFT and a functional model of the complete algorithm was available in the Gedae tool[11].

Gedae is a tool that enables users to design signal-processing applications in a hierarchical data flow structure. It provides a workstation environment to develop applications, tools to support multiprocessor scheduling and mapping, and a run-time environment to execute efficiently on scalable embedded processors.

To use the FFT FPGA implementation, information needs to be communicated between the Gedae data-flow graph, executing either on the host platform or an embedded processor, and the FPGA hardware. For the particular embedded system available, VME (Versa Module Eurocard) was the only interboard communication medium available. Two additional developments were made in order for the design flow to be realised. Both were written in Handel-C:

- Gedae primitive functions were produced to send data to and from the VME memory space. An additional primitive function was also produced to poll the VME space for a control register that indicated when the

algorithm being processed on the FPGA hardware had completed.
- An SRAM (static random access memory) controller was developed to enable the FPGA FFT application code to input and output data to memory and hence the VME.

Fig. 9 shows the difference between the FFT executed on a floating-point processor and the FPGA. Due to the fixed-point implementation used on the FPGA the magnitude of the FFT data was scaled to be of the order of 10^6 and this resulted in an error from the two implementations of the order of 10. These errors are entirely due to the differences between floating-point and fixed-point implementations.

Comparison of the overall development time using Handel-C and the process described in Section 3 with conventional FPGA developments for the same application shows an improvement of ×7·5. That is, the former is faster by a factor of 7·5, with commensurate benefits in costs and the ability to iterate rapidly to functionally correct solutions on heterogeneous embedded systems. The disadvantage of using this approach is that the resulting implementation is far less efficient than one produced using VHDL. Work is continuing to quantify the inefficiencies for real applications. However, it has been shown that optimised IP cores can be integrated into Handel-C and thus efficiency can be regained where needed.

5 Conclusions

We have shown that a maturing technology known as system-level design languages is becoming available that enables FPGA hardware and software to be co-designed and synthesised directly at higher levels of abstraction compared to the conventional HDLs and methods. The code can be optimised simply, and in rapid steps, to improve the efficiency of the implementation. From a user perspective, the languages are 'C like' and the programming environment analogous to the programming of conventional microprocessors (hence we coined the phrase 'hardware engineering for software engineers'). Therefore the same system functional models can be used for heterogeneous architectures and any function partitioned and mapped to the appropriate hardware resource and the application code generated automatically. This enables the rapid prototyping of embedded systems and rapid trade-off analysis. With the addition of optimised libraries for the system as a whole, efficient implementations can be expected enabling hard real-time requirements to be met. The principal disadvantage at present is the same as that experienced when high-level languages were emerging for microprocessors. The high-level language 'C to netlist' implementations are grossly inefficient, in terms of resource utilisation and execution, compared to hand-crafted code in HDL. As with microprocessors, we expect these inefficiencies to be ameliorated in time, especially because of the SystemC developments and the interest of the EDA vendors, albeit for the SOC market.

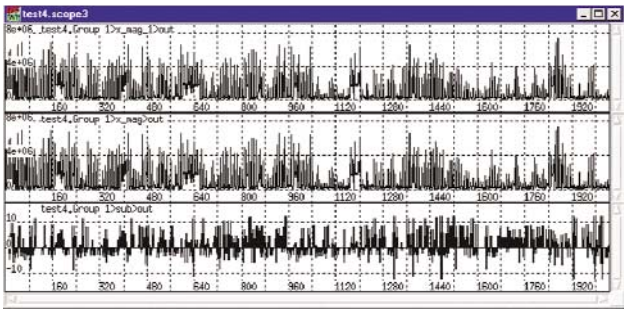


Fig. 9 Comparison FFT results

Acknowledgments

The work reported in this paper has been carried out on the ESPADON, EUCLID/Eurofinder programme, Project RTP2.29, with support from the UK, French and Dutch ministries of defence and participating companies. The authors are grateful for this support and would like to acknowledge the contributions of all the ESPADON team members.

This paper was first presented at the internal BAE SYSTEMS Signal and Data Processing Conference, 5th–7th March 2002, Dunchurch Park Conference Centre, UK (Conference Proceedings p.2–1).

References

1 MADAHAR, B. K., *et al.*: 'Environment for Signal Processing Application Development and Rapid Prototyping—ESPADON'. NATO IST Panel Symposium on COTS products in defence: 'The ruthless pursuit of COTS', 3rd–5th April 2000, Brussels, Belgium
2 DK1 and Handel-C, Celoxica Limited. See http://www.celoxica.com/
3 HOARE, C. A. R.: 'Communicating sequential processes' (Prentice-Hall Int. Series in Computer Science, 1985)
4 Inmos: 'The occam2 programming manual' (Prentice-Hall, 1988)
5 ARMulator, ARM Ltd. See http://www.arm.com/
6 SingleStep debugging solutions, WindRiver Systems Inc. See http://www.windriver.com/
7 ModelSim, Model Technology. See http://www.model.com/
8 LeonardoSpectrum, Mentor Graphics. See http://www.mentor.com/synthesis/
9 A|RT, Adelante Technologies. See http://www.adelantetechnologies.com/
10 SystemC Version 2.0 User's Guide. Available at http://www.systemc.org/
11 Gedae: a graphical programming and autocode generation tool for signal processing applications, Blue Horizon Development Software. See http://www.gedae.com/.

Received 11th April 2002

The authors are with BAE SYSTEMS Advanced Technology Centre, Systems Department, West Hanningfield Road, Great Baddow, Chelmsford, CM2 8HN, UK.
E-mail: ian.alston@baesystems.com;
bob.madahar@baesystems.com