

Model based design of Heterogeneous Processing Architectures

Mr. Ian Alston & Dr. Bob Madahar
BAE SYSTEMS Advanced Technology Centre
West Hanningfield Road,
Gt. Baddow, Chelmsford CM2 8HN, U.K.
Tel: +44 1245 242195
Fax: +44 1245 242124

ian.alston@baesystems.com & bob.madahar@baesystems.com

ABSTRACT

The software programmable multi-processor architecture has been employed extensively for embedded signal processing applications over the past two decades. New methodologies, engineering processes and support environments are beginning to emerge for these architectures, in particular to support the concept of rapid prototyping. That is the ability to rapidly and seamlessly move from functional design, to the architectural design, to the implementation, through automatic code generation tools, onto real-time COTS test beds [6], [15]. The maturing of the technology has however been limited to supporting microprocessors and digital signal processors (DSPs) as the computing nodes within heterogeneous architectures. Recent advances in microelectronics is enabling part of the increased complexity of these embedded signal processing systems to be moved to field-programmable gate arrays (FPGAs) to offer hardware acceleration to meet the growing time critical aspects of the design. Hence 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 (HDL), 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 technical details on high-level system level design languages that can support FPGAs in a manner analogous to the support of conventional microprocessors and DSPs. It is demonstrated that such languages can be integrated into a high-level design flow for the rapid development of heterogeneous embedded signal processing systems. Proposals for an improved core-based design flow and its integration into Geda are also presented.

1. INTRODUCTION

An overall Rapid Prototyping Methodology was developed by the tri-national European EUCLID/Eurofinder defence project called ESPADON (Environment for Signal Processing Application Development and Rapid Prototyping) completed in Sept' 01 [6]. This three year project demonstrated significant improvements (reduced cost and timescales) to the overall process by which complex military digital processing systems are designed, developed and supported. The programme made significant progress on a rapid and 'seamless' high level design flow for conventional processors, but less so for FPGAs, in heterogeneous architectures.

The FPGA is a re-configurable 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 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.

However, in order to exploit the use of FPGAs in heterogeneous architectures what is needed are high level languages and tools for rapid system design to support FPGAs akin to the tool support for COTS processors. The low-level development methods, such as schematic capture or hardware description languages (HDL), employed to implement FPGA designs are wholly inappropriate (time, cost and effort, including specialists) as system level tools for rapid embedded system development. Instead we need to raise

"© BAE SYSTEMS 2003. All rights reserved."

"Unless BAE SYSTEMS (Operations) limited has accepted a contractual obligation in respect of the permitted use of the information and data contained herein such information and data is provided without responsibility. BAE SYSTEMS (Operations) Limited disclaims all liability arising from its use."

the design to higher levels of abstraction and provide an integrated approach to software and hardware design supporting 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 languages are derived from traditional programming languages such as C, C++, 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.

The paper also presents an extended design flow for heterogeneous platforms developed under the ESPADON project employing one of these system-level design languages. The benefits and limitations of this design flow are reported. In order to overcome these limitations, the paper also presents an improved design flow based on an IP core approach. We also discuss some of the issues in connection with integrating such a design flow with the Gedae tool [8] for 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. In order to ensure that systems meet the 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, designers (system, hardware, and software) have tended to create an executable specification for their systems. For the most part, these are functional models written in a language like C or C++. Their use 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 as they 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 to model timing, concurrency, and reactive behaviour, all of which are needed to create accurate models of systems containing both hardware and software.

In order 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 three common system-level design languages available today. Other languages in this arena are SpecC [7] and HardwareC [11].

2.1 Handel-C

The Handel-C language and supporting design environment [5] is developed by Celoxica, which was previously known as Embedded Solutions Ltd (ESL), 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 [9] and the supporting OCCAM language [10]. 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 that incorporates simulator, debugger, compiler and implementation generation, in either EDIF netlist format or VHDL.

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 Figure 1.

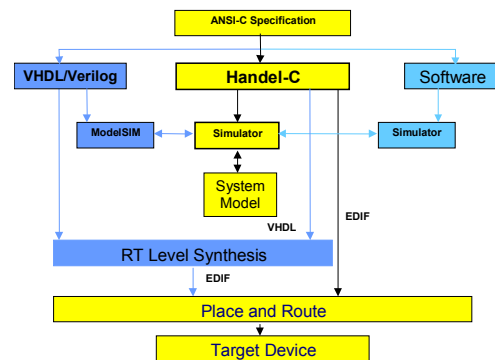


Figure 1: Handel-C Design Flow

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 languages parallel construct, the PAR statement, enable area vs. performance optimisation. Special I/O constructs, the Port and Channel construct, 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 place & route and timing analysis tools. Interfaces (or plugins) 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 & software partitioning via interfaces between the Handel-C simulator and software simulation tools such as ARMulator [3] and Windriver's SDS Single Step debugger [16]. Optional VHDL code generation supports IP integration through interfaces with HDL development tools for co-simulation (e.g. ModelSIM [13]) and RTL synthesis (e.g. Exemplar Leonardo Spectrum [12]).

While Handel-C supports a rich sub-set 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 for 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.

2.2 A|RT [4]

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-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 to scan the available libraries for additional resources. The user can also write their own pragmas 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 sub-set 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 a HDL development environment.

The design flow used by A|RT is shown in Figure 2.

While Figure 2 implies that the system specification can be written in ANSI-C, there are however, restrictions imposed by the behavioural synthesis tools of A|RT Designer and A|RT Builder. Nonetheless, a rich sub-set of the C language. The restriction 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.

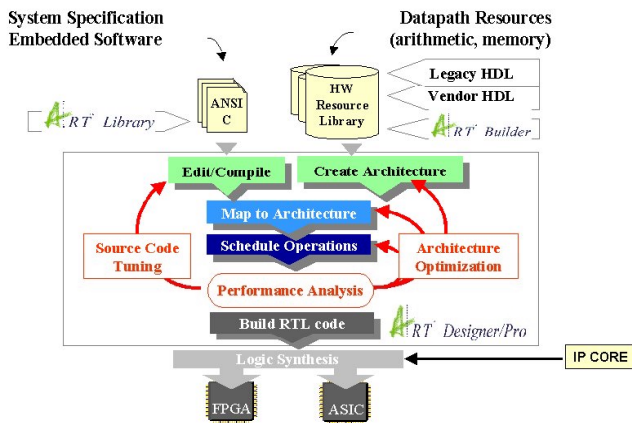


Figure 2: A|RT Design Flow

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.

Since the evaluation of the A|RT suite of tools during the ESPADON programme, the tools have been removed from the market place as a commercially available toolset.

2.3 SystemC

SystemC [17] 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 vendors are thus able to create tools that are automatically interoperable.

During the early days of this initiative, lack of co-operation and patent rights 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 a HDL, such as Verilog or VHDL. Similar 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 which resumes execution when some set of events occur 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 pre-specified before simulation starts.

Figure 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 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.

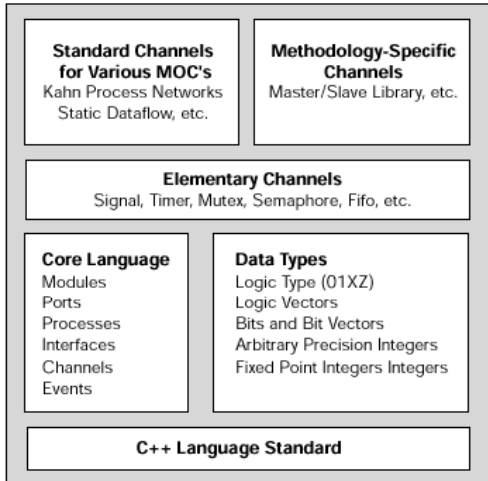


Figure 3: SystemC Language Architecture

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. ESPADON DESIGN FLOW

The use of a Rapid Prototyping Methodology and its supporting tools have been well documented and is becoming a mature process for COTS based processors. 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 developments 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 project is shown in Figure 4 [6]. 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.

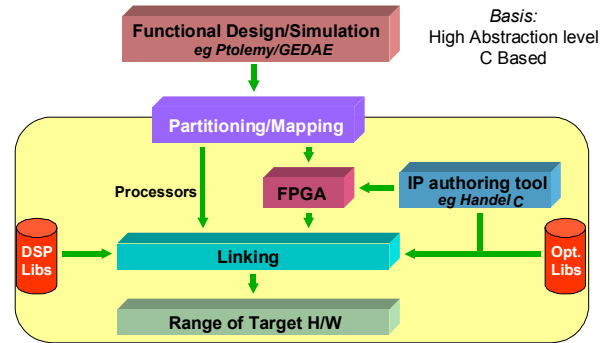


Figure 4: Heterogeneous Platform Design Flow

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 Handel-C 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.

A practical implementation of this type of design flow integrated with the Gedae tool [8] has been demonstrated [2]. It was also shown that a comparison of the overall development time using this type of design flow with conventional FPGA developments for the same application provides an improvement of x 7.5. That is the former is a factor of 7.5 faster with commensurate benefits in costs and the ability to rapidly iterate 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. 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.

4. EXTENDED FPGA DESIGN FLOW

While the design flow produced on the ESPADON project enabled functional models to be implemented on heterogeneous architectures, the two main limitations of the approach were:

- The FPGA implementation route was essentially a separate sub-flow within the overall design flow. Thus the rapid architectural mapping process achieved with processor based systems couldn't be achieved as a change in mapping required additional manual coding within the Handel-C language.
- The use of optimised IP cores was not fully integrated into the process thus making the implementation far less efficient than one produced using VHDL.

Therefore we have begun to extend the FPGA work much further to address these two issues while still keeping within the ESPADON methodology and risk-driven iterative development process.

In order for the design flow to be usable by system designers, we must ensure that any design flow implemented for FPGA provides us with the flexibility and features that are already in existence for processor based systems. Therefore the general aim is to be able

to map elements of a functional model to efficient implementation on FPGA. This leads to the following design flow requirements:

- Provide a graphical workstation algorithm development and simulation environment and a seamless route to implementation.
- Support hybrid targets with a combination of RISCs, DSPs and FPGAs.
- Provide support for performance trade-off studies during the mapping of the algorithms to the heterogeneous architecture with minimum rework during mapping changes.
- FPGA implementation:
 - Provide an FPGA firmware programming capability to software engineers.
 - Enable the use of optimised IP cores at various levels of the functional specification hierarchy.
 - Provide a means to generate new IP cores.
 - Take advantage of the inherent parallelism in FPGAs.

There are a number of approaches to achieving these requirements:

- Low-level implementation

Provision of a traditional low-level implementation route using HDLs or schematic capture within the graphical modelling environment. This approach has been adopted in several tools before and while it usually provides a fairly efficient means of implementing algorithms for hardware, a special subset of the building blocks must be used. This requires the user to translate a generic algorithm specification into an equivalent specification using this special subset of building blocks. It is therefore difficult to perform architectural exploration as a specific model must be built each time.

- System-level design languages

System-level design languages allow implementations to be synthesised from a high-level language, such as C. Many of these languages allow different levels of parallelism to be incorporated within the implementation to take into account the inherent parallelism within the algorithm. However, some of these languages require refinement through various levels of abstraction to arrive at a register transfer level specification prior to implementation. In addition, efficient route to implementation is limited in many cases.

- Core-based flow

As the functional design is at a high level of abstraction, a mapping of generic functional models to IP cores provides an efficient implementation route assuming that the IP cores are optimised for the FPGA device being considered. Also at these high abstraction levels, the inherent parallelism within the functional block can be exploited within the FPGA implementation.

- Embedded processor cores

Processor cores for embedding within an FPGA are now becoming common place allowing existing processor based design flows to be used. Efficient implementation will only be achieved if the compiler technology for the chosen

embedded processor core is sufficiently mature. Also, it is likely that the clock frequency of such an embedded processor core will be lower than an equivalent external processor. However, it offers a quick route for implementation of a particular functional block within an overall design flow when an IP core or equivalent isn't available.

Each of these approaches has a place within the support for FPGAs, but as described in Section 3 the core-based approach mirrors the requirements of the conventional rapid prototyping flow and of the approaches discussed will lead to the most efficient implementation route, enable efficient architectural trade-off studies and provide a flexible design flow. An additional benefit, is that the majority of such a design flow can be independent from the high level functional design tool. The level of dependence is a function of the capabilities of the tool and how it supports multiple processor targets.

In order to meet the above design flow requirements the FPGA Core Based Design flow presented in Figure 5 is proposed.

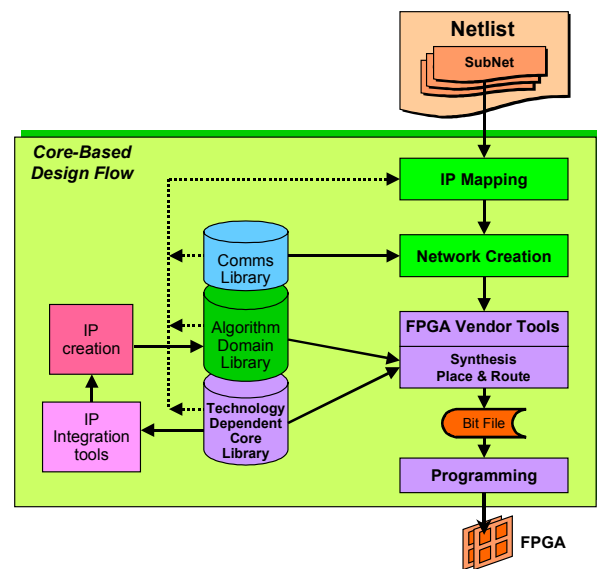


Figure 5: Extended FPGA Design Flow

The design flow consists of the following elements:

- *Input Netlist* - this is essentially a connectivity netlist rather than an executable netlist. The main objective of the netlist is to define a network of functional building blocks without necessarily defining their implementation. For multi-processing implementations the netlist will contain a series of subnets each one defining a network mapped onto a particular processing element. The netlist format will be hierarchical. It is assumed that the netlist will contain any references to inter-FPGA or inter-processor communication blocks.
- *IP Mapping* - this is a process to convert the functional building block reference into an IP core from the core library. The mapping will be available at all levels of the netlist hierarchy thus allowing the highest level of functional abstraction to be mapped where possible. This mapping will

be “switchable” (on/off) with the off condition implying that the implementation must be referenced within the netlist.

- *Network Creation* - this is the process of providing the communications between the referenced building blocks in order to maintain the functional properties of the overall network. The language could be a low level HDL or one of the system-level design languages discussed in Section 2. The network creation will also take into account the use of multiple instances of the same functional building block. Two approaches will be available consisting of either instantiating as many instances as contained in the netlist (thus using more resources) or instantiating a single instance with sharing of this instance by the complete network on the FPGA. This latter case will involve the implementation of some form of “scheduling” algorithm i.e. the routing of inputs/outputs through multiplexers.
- *IP Core Libraries* – this is a series of libraries containing communications cores, domain specific algorithmic cores and vendor/technology dependent cores. The ability to efficiently map a given functional model onto an FPGA will be a function of the extent of these libraries. In addition, the libraries will contain multiple implementations of a given functional building block optimised for different scenarios e.g. power verses size.
- *IP Generation* – in order to be able to populate the IP Core Libraries an efficient means of creating new high level IP cores is an important element of the design flow. The languages for the creation would range from HDLs to system-level design languages and also include specific IP creation tools.

In developing this flow the two main elements are the IP Core Libraries and the Network Creation. The IP Core Libraries will have to be designed in order that the IP cores have a common interface linked to the way in which the network is constructed. A study of interfacing standards for such IP cores has highlighted that at present there is no low-level interfacing standard for IP cores. The Virtual Socket Interface Alliance (VSIA) [18] has developed a Virtual Component Interface (VCI) standard to enable IP cores, Virtual Components (VCs), to be interconnected within SoC. The standard addresses the particular issues of data communications and the interfacing to an on-chip bus (OCB). Independence from the actual bus used is achieved by the development of a Bus Wrapper to interface to the VCI of the VCs. The Open Core Protocol (OCP) [14] have developed a high-performance, bus-independent interface between IP cores that is a superset of the VCI standard that also supports configurable control signalling and test harness signals. While these standards have been developed for SoC architectures many of the features may well be applied to FPGAs. However, it is likely that the standards are too generic for the interfacing requirements of the network creation. It is therefore likely that some form of external wrapper code may be required to convert third party IP cores into suitable candidates for inclusion within the domain libraries.

The Network Creation will be dependent on the model of computation contained within the netlist. At present the development of the extended design flow is concentrating on the data-flow paradigm of computation. In this case the data-flow network can be viewed as a process network with each of the functional processes connected via a first-in/first-out (FIFO)

queue. The properties of this FIFO are governed by the data-flow properties contained in the netlist. As no standard interface exists for connecting IP cores, the FIFO interface could be used as the appropriate standard. Note that in some cases the FIFO may degenerate to a direct connection for suitably produced IP cores.

Another feature of the proposed design flow is that optimisation of the FPGA implementation using third party tools can be used to transform the input netlist into an equivalent functional netlist but optimised for particular implementation scenarios, such as timing or pipelining. It is likely that this optimisation would be performed during the final refinement stages of the implementation once a suitable mapping has been found.

From the above description it can be seen that the design flow at this level of specification is independent from the high level design tool creating the input netlist. The only pseudo-dependency that exists is the mapping of the high-level building blocks, which are explicitly defined by the high level tool, onto the IP cores within the core libraries.

5. GEDAE INTEGRATION

There is some work already in progress to support FPGAs directly within Gedae and this at present is based on the low-level implementation method described in Section 4. The approach is based on a BHDS proprietary language called Register Transfer Language (RTL) used to create single-sample primitives. These RTL primitives produce a network of registers, combinatorial logic and clock enable signals. Data, contained in signals, moves through this RTL network in synchronism to a local clock. As an RTL primitive may only use 4 atomic operations (assign, decimate/interpolate, clock and register) the conversion of these into implementation languages is fairly straightforward. A C implementation route already exists for simulation purposes with other languages (such as VHDL and Handel-C) in preparation. While the original version of the RTL graphical language was a separate program, it has now been integrated into the main Gedae tool where an RTL graph is converted into a single static dataflow primitive.

While RTL provides a means of producing an implementation of synchronous networks for FPGA, there are some limitations to its use. The three main disadvantages are:

- The language is essentially a schematic capture language and thus assumes that the user has some knowledge of developing hardware.
- Its productivity is low due to its low level language syntax.
- Architectural analysis becomes difficult because:
 - A processor implementation of an RTL graph (based on say C) will be inefficient compared to the optimised e_functions employed within standard dataflow primitives.
 - Specific RTL networks will have to be created each time a functional element of the overall Gedae model is mapped onto an FPGA.

Therefore, it is the authors belief that the inclusion of a link to the extended FPGA design flow presented in Section 4 will not only enable more efficient FPGA implementations from Gedae but allow greater productivity improvements through the support of FPGAs at the highest levels of functional abstraction.

It should be emphasised that the RTL language has a place within the proposed design flow in two specific areas:

- Creation of specific reusable low level networks.
- An IP core design capability in order to populate the IP core domain libraries.

In the former case the ability in the proposed FPGA design flow to allow implementation to be included within the netlist specification would be employed.

The integration of the proposed FPGA design flow within Gedae is shown in Figure 6. The changes to the main Gedae host programme are minimal to include the integration required with much of the work involved within an FPGA board support package (BSP).

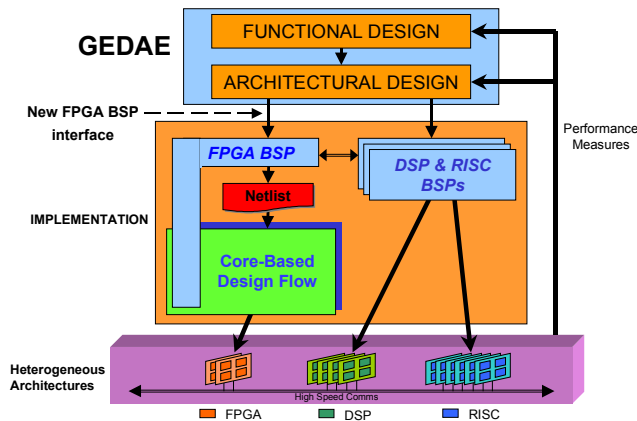


Figure 6: Integration of extended FPGA Design Flow with Gedae

The following subsections provide a commentary on some of the issues faced with the proposed approach. It should be noted that many of these issues also apply to other methods of supporting FPGAs within Gedae.

5.1 Netlist Export

As indicated in Section 4 the proposed design flow relies on a netlist being exported by Gedae. The netlist should provide the connectivity network at the following abstraction levels:

- *Primitive level* - in addition to acting as a reference to the implementation, the specification should also contain input/output specifications, granularity and memory requirements.
- *Partition level* - this will contain all the primitives within a single partition and is used to indicate how the primitives are mapped to a particular processing element. At this level the netlist will contain inter-partition (which will imply inter-processor) communication specifications either as primitive level entries or as specific send and receive entities.
- *Model level* - this describes the complete Gedae model composed of its many partitions.

In order to meet these needs, the netlist should be exported post partitioning and scheduling. It may also be advantageous to contain scheduling information. The use of hierarchical netlists is encouraged to aid understanding and simplify implementation issues.

Gedae already generates this information in alternative forms during its current build process so the production of a netlist export should be fairly simple and non-destructive to the current implementation flow. There are many existing netlist formats that could be adopted to aid the reading and writing of the netlist.

5.2 Building and Running an Application

The Gedae build process for conventional processors is fully automatic based on a script (perl & make) mechanism. The situation for FPGAs and in particular linking to the proposed design flow is more complex and time consuming. However, a seamless build process, with possible manual intervention as required, should be possible through the BSP. The basic needs are as follows, assuming that the partitioning, mapping and scheduling are equivalent for an FPGA implementation and conventional processors:

- *Build Application* - this will commence with the generation of the exported netlist and conclude with a number of FPGA bit-files. This is analogous to the current code generation, compilation and linking stages for processor based systems. For the proposed design flow, this will imply an interface from an FPGA BSP to the core-based flow. As the FPGA build process could take from a few minutes to a few hours, the process should be triggered only when required (i.e. when a particular FPGA partition has changed) and be made available in background mode so that other work can be carried out during the build. In addition, the status of any build process should be made available to the developer via a display rather than through the normal Gedae console.
- *Load Application* - this will be a function of the BSP for the FPGA architecture chosen. This is likely to be achieved through a system bus (e.g. VME or PCI) or via a JTAG connector.
- *Start Application* - this is the process of loading parameters, performing a reset and actually starting execution of the application on each FPGA by enabling the system clock.

The activities of *Stop*, *Continue*, *Reset* and *Terminate* would have the current meaning as for processors. The currently defined *Run* activity for processor based systems could be replaced by the above defined activities.

5.3 Data Communications

There are a number of key areas where data communications is important:

- *Run-Time Parameters* - for conventional processors, the Gedae host and embedded kernel provide a means to read/write the values of run-time parameters via a host control port controlled through the command programme API. This functionality should also be mirrored on the FPGA. Parameters within an FPGA will generally be in the form of registers and so there must be some method of equating the registers with the parameters within the Gedae model. This could be achieved by the creation of an "addressing table" during the build process for the FPGA. Work at BAE SYSTEMS has already shown that a simple parameter control programme, written in Handel-C, can be embedded within an FPGA to enable host control of parameters using such an addressing table.

- *Data Transfer to/from Host* - apart from control information as explained above, specific modes of communicating using data ports to the host must be provided. These data ports act as a means of transferring data from/to a host partition to/from an FPGA. These transfers should be blocking read/write to stop "execution" of the associated Gedae partitions until data is read/written. It is likely that suitable buffering of the data during the data transfer will take place in order to cater for unbalanced processing and asynchronous behaviour.
- *Inter-processor communications* - send and receive functions to enable dedicated point-to-point, virtual channel and shared memory communications should be provided within the BSP. Dedicated point-to-point links will most often be used where a set of pins of one FPGA can be connected directly to pins on another FPGA or processor on a target board. This will provide a high performance data path suitable for single stream data flow. Virtual channel links offer a lower data rate and/or longer latency communications where a set of virtual data channels is multiplexed onto a single physical channel. Shared memory communications is possible on many modern FPGA cards and is suitable for bulk data transfers.

An important element in data communications for FPGA is the ability to convert the data type, particularly between conventional processors and FPGA, during the transfer. The conversion should be located in the send/receive functions with parameters either automatically passed to these functions by the build process or via the Gedae Group Control-Set Transfer Methods dialogue.

5.4 Library Support

Apart from the concurrent nature of FPGA devices, one of their significant differences between conventional processors is the difference between the range of supported/possible data types. Gedae libraries should be updated to include N-bit signed and unsigned integers and fixed point representations as a minimum. In order to perform functional analysis on the host, bit true simulation of these library primitives should be provided. It has already been shown [1] how fixed point processing support can be included within Gedae as a set of library primitives using the SystemC language. This concept should be extended to provide a basic library of processing primitives to cover simple arithmetic operations (including scalar, vector and matrix) and typical signal processing functions (e.g. FFT, FIR, IIR, etc.).

In addition, consideration should be given to a "multi-representation" library structure for Gedae primitives. This could be achieved using multiple Apply methods within the primitive definition. Alternatively, if we consider a primitive specification as consisting of two parts, an interface specification and an implementation, then it is possible to have multiple implementations for a primitive with a single specified interface. This would allow a mixture of code and hierarchical implementations for the same interface specification. One implementation may be a highly optimised library function whereas another may be a hierarchical specification using lower level components. This would enable the RTL language to be effectively used within Gedae.

Another consideration for the extended library is the generation of performance data for use within the GSIM tool. As the implementations within a core based design flow will be

generated by third parties, a means of importing performance data into the Gedae database should be provided.

5.5 Heterogeneous Support

While Gedae is currently capable of supporting heterogeneous processor based architectures there are a number of limitations:

- The ability to include multiple BSPs from multiple vendors within the same Gedae executable is not guaranteed to work. This is due to a potential clash of global symbols (both variable and function names) within the vendor platform BSPs that are used to create the Gedae BSPs for a particular platform and linked into the Gedae executable.
- Any communications between multi-vendor platforms using the Gedae BSPs must be via the host. This is likely to be over a slow interface and thus not appropriate for high speed communications.

Other tools have overcome the first limitation by ensuring that the vendor BSPs are isolated from each other by the use of independent processes interacting with specific platforms with each process communicating in a common manner to the main host application. This approach could also be adopted within Gedae.

At present, the transfer mechanisms available for a particular platform are pre-compiled within the Gedae BSP for that platform. Therefore the second limitation should be addressed by providing a method of extending transfer mechanisms within a BSP via the embedded configuration file and additional library code (for host and embedded processing elements) to support the additional transfer mechanism. Any specific parameters of the extended transfer mechanisms could be provided through the Group Control-Set Transfer Methods dialogue.

5.6 Monitoring and Debugging

Two areas are important here:

- *Trace Data Collection*
- *Interface to Signal Monitoring*

The Gedae embedded kernel executing in each embedded processor target has the ability to collect trace information for collection and display by the host. This information provides performance measurements and is useful in performing load balancing for multi-processor architectures. The information shows the execution time of each primitive, data being sent and received between partitions, dynamic data flow queues being filled and emptied, data flow blocking and schedule state changes.

For FPGA implementation there are two issues with this form of data collection:

- There will be no Gedae embedded kernel in an FPGA.
- The conventional trace information may not be suitable due to the concurrent and continuous execution of the functional building blocks within an FPGA implementation.

The build process must therefore be capable of inserting suitable performance measure collection points within the FPGA network. It is likely that an area of shared memory be set aside for this purpose. In this case, the host could access the shared memory without needing direct interface to the FPGA. The type of performance data collected will need tailoring to the FPGA

implementation. Data such as length of FIFOs actually used, processing times of IP cores (based on interaction with FIFOs) and numeric overflows would be of particular use in debugging the FPGA implementation.

Another general feature of Gedae is the ability to connect data-flow monitoring primitives (e.g. scope or echo) to nodes within a data-flow graph and is particularly useful during demonstrations and debugging. As the build time for FPGA implementations is considerably longer than conventional processors, a means of adding monitoring primitives to a data-flow graph without having to rebuild the implementation should be provided. Each monitorable node will thus need to be exported to the FPGA build process and a means added to transfer the data for monitoring to the host for display as appropriate.

6. CONCLUSIONS

We have shown that a maturing technology known as system-level design languages are becoming available that enable FPGA hardware and software to be co-designed and synthesised directly at higher levels of abstraction compared to the conventional HDLs and methods. From a user perspective, the languages are 'C like' and the programming environment analogous to the programming of conventional microprocessors. These languages enable the rapid prototyping of embedded systems and rapid trade-off analysis for heterogeneous architectures.

A practical implementation of a design flow based on one of these system-level languages integrated with the Gedae tool, as produced on the ESPADON programme, has been described. While a comparison of the overall development time using this type of design flow with conventional FPGA developments for the same application provides an improvement of x 7.5, a number of limitations have been identified.

A discussion on ways of overcoming these limitations has been presented and an extended design flow based on IP cores has been proposed. The integration of the proposed design flow within Gedae for FPGA implementation has been presented and a number of specific and general issues discussed. From work conducted on the ESPADON programme and current research it is believed that this approach, linked with Gedae, will lead to the most efficient implementation route, enable efficient architectural trade-off studies and provide a flexible design flow for both processor and FPGA architectures. In addition, it doesn't preclude the use of alternative approaches for FPGA implementation or generation of the IP cores. In particular the RTL language being proposed by BHDS is complementary to this approach.

7. Acknowledgements

The authors would like to acknowledge the support and technical contributions of BAE SYSTEMS business groups such as Future Systems Division, the Sensor Systems Division of Avionics, and the Underwater Systems Division of Sea Systems towards this area of work.

The extended design flow is being produced in collaboration with QinetiQ under a MoD Corporate Research Programme entitled Core-Based DSP for FPGA (ref. CRP/TG10/02/01/016/2001).

The authors acknowledge the technical contributions from QinetiQ and the Queens University of Belfast during this ongoing programme.

Part of this work was presented at the internal BAE SYSTEMS Signal and Data Processing Conference, 5-7 March, 2002, Dunchurch Park Conference Centre, UK.

8. REFERENCES

- [1] Alston, I.D., "Fixed Point Support within Gedae", BAE SYSTEMS Presentation, Gedae Training Course, June 2002.
- [2] Alston, I.D. and Madahar, B.K., "From C to Netlists: Hardware Engineering for Software Engineers?", IEE Electronics & Communication Engineering Journal, August 2002.
- [3] ARMulator, ARM Ltd., <http://www.arm.com/>.
- [4] A|RT, Adelante Technologies, <http://www.adelantetechnologies.com/>.
- [5] DK1 & Handel-C, Celoxica Limited, <http://www.celoxica.com/>.
- [6] ESPADON Project, EUCLID/Eurofinder programme: Project RTP2.29, <http://www.espadon.org/>.
- [7] Gajski, D.D., et al, "SpecC: Specificatin Language and Methodology", Kluwer Academic Publishers, March 2000.
- [8] GEDAE: A Graphical Programming and Autocode Generation Tool for Signal Processing Applications, Blue Horizon Development Software, <http://www.gedae.com/>.
- [9] Hoare, C.A.R., "Communicating Sequential Processes", International Series in Computer Science, Prentice-Hall, 1985.
- [10] Inmos, "The occam2 Programming Manual", Prentice-Hall, 1988.
- [11] Ku, D.C. and DeMichelli, G., "HardwareC – a language for hardware design", Stanford University, Technical Report, CSL-TR-90-419, 1988.
- [12] Leonardo Spectrum, Exemplar Logic Inc., <http://www.exemplar.com/>.
- [13] ModelSIM, Model Technology, <http://www.model.com/>.
- [14] OCP, <http://www.ocpip.org/>.
- [15] Pridmore, J., et al, "Model-Year Architectures for Rapid Prototyping", Rapid Prototyping of Application Specific Signal Processors, Jnl. Of VLSI SIGNAL PROCESSING SYSTEMS for Signal, Image and Video Technology, Kluwer Academic Publishers, Vol. 15, 83, Feb, 1997.
- [16] SingleStep Debugging Solutions, Wind River Systems Inc., <http://www.windriver.com/>.
- [17] SystemC Version2.0 User's Guide. Available at <http://www.systemc.org/>.
- [18] VSIA, <http://www.vsi.org/>.