Parallel Computing 34 (2008) 231–244

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

OpenFPGA CoreLib core library interoperability effort

M. Wirthlin [a], D. Poznanovic [b,*], P. Sundararajan [c], A. Coppola [d], D. Pellerin [e], W. Najjar [f], R. Bruce [g,h], M. Babst [i], O. Pritchard [j], P. Palazzari [k], G. Kuzmanov [l]

[a] Brigham Young University, 448 CB, Provo, Utah, 84602, USA
[b] SRC Computers, Inc., 4240 N. Nevada Avenue, Colorado Springs, CO 80907, USA
[c] Xilinx Inc., 2100 Logic Drive, San Jose, CA 95124, USA
[d] OptNgn Software, LL, 2828 Corbett Avenue, Portland, OR 97201, USA
[e] Impulse Accelerated Technologies, 550 Kirkland Way, Suite 408, Kirkland, Washington 98033-6240, USA
[f] Department of Computer Science & Engineering, University of California Riverside, Riverside, CA 92521, USA
[g] Nallatech, Boolean House, One Napier Park, Glasgow G68 0BH, UK
[h] Institute for System Level Integration, The Alba Centre, Livingston, Scotland EH54 7EG, UK
[i] DSPlogic, Inc., 13017 Wisteria Drive, #420, Germantown, MD 20874, USA
[j] Altera Corporation, 110 Cooper St, Suite 201, Santa Cruz, CA 95062, USA
[k] Ylichron Srl, c/o C.R. ENEA Casaccia, Via Anguillarese, 301, 00123 S. Maria di Galeria, Rome, Italy
[l] Computer Engineering, EEMCS, TU Delft, Mekelweg 4, 2628 CD, Delft, The Netherlands

ARTICLE INFO

Article history:
Received 16 September 2007
Received in revised form 14 March 2008
Accepted 25 March 2008
Available online 29 March 2008

Keywords:
Reconfigurable computing
HDL cores
Core library
Standards

ABSTRACT

This paper begins by summarizing the goals of the OpenFPGA CoreLib Working Group to facilitate the interoperability of FPGA circuit cores within a variety of FPGA design tools, including high-level programming tools targeting FPGA architectures. This effort is contrasted with other IP reuse efforts. The paper reviews the current approach used by several high-level language compilers to integrate IP within their tool. The CoreLib approach for standardizing this IP integration is proposed followed by an example that demonstrates its utility. Finally, the current state of the effort and future plans are presented.

1. Introduction

Field Programmable Gate Arrays (FPGAs) are increasingly used to perform application-specific, hardware-accelerated computations. By customizing the data path, memory, and computing elements, FPGA circuits can be made to perform application-specific computations much more efficiently than traditional programmable processors. Many commercial applications, in a wide variety of domains, have been successfully implemented using FPGAs for performance-critical processing. These applications have included video processing, data communications, scientific and financial computing, and many others. The accelerated portions of these applications are typically characterized by the use of massive parallelism at the circuit level. Recent advances in FPGA architectures, as well as advances in FPGA-based computing platforms and related tools, have played strong roles in the increased use of FPGAs for such applications.

Most FPGA designs used for application-specific computation are created using hardware description languages (HDLs) such as VHDL and Verilog. The HDL-based design methodology, however, provides a relatively low level of abstraction when compared to traditional programming languages. The HDL level of abstraction requires that the application programmer

* Corresponding author.
E-mail address: poz@srccomp.com (D. Poznanovic).

0167-8191/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2008.03.004

have a solid understanding of fundamental digital design techniques as well as knowledge of the underlying FPGA architecture. The use of HDLs for designing application-specific accelerators is very time consuming and limits the use of FPGAs to those with specialized digital design skills.

In order to provide higher levels of abstraction and improve the ease of programming the FPGAs, a number of high level languages (HLL) and related compilers have emerged. These higher level FPGA programming tools allow a programmer to create a circuit within an FPGA from a high-level specification such as C. This provides increased productivity, shorter development times and greater ease of use to FPGA application designers. While there may be a performance, power or size penalty associated with using such tools, an increasing number of application developers are finding them to be productive and capable additions to their existing FPGA development environments.

HLL programmers obtain the best results by combining use of high-level tools with traditional HDL. Hardware description languages are used to describe high-performance circuits at a relatively low level of abstraction. Designers prefer to reuse such circuits because of their high performance and the high-cost of developing these cores from scratch. High-performance circuits that are designed in a way to facilitate reuse are often called Intellectual Property cores or IP cores for short. In such a hybrid approach to FPGA design, certain critical components may be hand-crafted using HDLs, while other parts of the system are either described at a high level using an HLL compiler, or assembled using off-the-shelf IP cores.

This hybrid approach to FPGA design is similar to practices common in software development. A common practice during software development is to use architecture-specific or platform-specific libraries, with the goal of increasing performance and reducing development time. For example, software programmer targeting such widely divergent platforms as DSPs (in an embedded system) or the Linux operating system running on a Pentium-class processor may be using the same programming language (be it C, C++ or some other language), but will almost certainly be using libraries and application programming interfaces (APIs) that are unique to that target, or unique to their specific type of application. In extreme cases, such as in DSP applications, the software programmer may be required to drop into assembly language in order to achieve a required level of performance.

While this approach of mixing high-level target independent code with low-level application-specific cores is effective, there are currently no accepted standards that address the integration of IP cores into high-level programming languages. In the absence of standards, the IP providers, tool vendors and application developers must spend extra time and effort to port and integrate existing or new IP cores into HLL tool environments. Further, the approach for integrating IP cores into HLL tools varies from vendor to vendor. The lack of a standard has limited the growth and adoption of HLLs in FPGA-based computing. A standard is needed to simplify the process of integrating IP within higher-level design tools. This paper will summarize the efforts of the OpenFPGA[1] CoreLib working group to identify and create such a standard.

2. Project goals

The overall goal of this effort is to facilitate the interoperability of FPGA circuit cores within a variety of FPGA design tools, including high-level programming tools targeting FPGA architectures. The specific goals of this effort include the following:

- Develop a standard for describing the interface and operating characteristics for cores facilitating the integration of cores within high-level programming language compilers and other FPGA design tools.
- Develop a standard for libraries of cores that facilitates the building, distribution and integration of cores across tools and hardware platforms.
- Encourage the use of these standards by compiler developers, core developers, universities, and other tool vendors.
- Create a synergistic environment where application programmers and IP core developers can create the most effective implementations of applications on FPGA-based systems.
- Create a representative set of general purpose and application specific core libraries for use across the available tools and platforms.

The CoreLib working group of OpenFPGA has developed a strategy for achieving these goals. Each of these goals will be described in more detail in the following sections.

2.1. Developing a standard for cores

In order to encourage the use of standards and to avoid constraining the development of high level language tools, the CoreLib group's strategy is to adopt a descriptive standard that allows the tools to integrate a variety of core types. Each high-level design tool has differing capabilities and requirements when integrating custom cores. These differing requirements limit the ability to reuse cores among different high-level language tools. By being able to describe the characteristics of all available cores, we seek to encourage support a variety of cores among multiple high-level language tools.

A significant amount of work has already taken place in defining interface standards and in describing cores. To avoid duplication of effort the existing work is being reviewed. Section 3 summarizes several of the prominent standardization efforts. The CoreLib group will adopt or seek to modify existing standards to meet the group's objectives.

It is expected that a small set of interface standards will develop over time and through availability of conforming cores, the tools will support the most effective and available standards. The application developers will ultimately pick the standards through use of tools and acquisition of the most effective libraries.

2.2. Developing a standard for libraries

The use of reusable libraries for software development has significantly improved software design productivity. This concept of reusable libraries for hardware, however, is lacking. It is much more difficult to create a reusable circuit module than reusable software. The interface between two circuits involves much more detail than the interface between two pieces of software (i.e. specific ports, timing interface, handshaking protocol, etc.). If a standard for circuit libraries is created, the benefits of reuse that are taken for granted in software can be available for hardware.

Creating a standard for core libraries will enable the creation of high quality cores as well as simplify the distribution and use of cores within applications. The library standard is intended to allow a variety of tools to easily access the cores. By defining a standard structure, a set of procedures can be developed to collect cores for distribution in a packaged form. Such standardization also allows the development of quality assurance procedures for cores.

2.3. Encouraging use of the standards

By providing a descriptive standard for cores, and a prescriptive standard for libraries, the widespread use of cores in applications is made possible. The wider availability of cores is intended to facilitate the use of compilers and development of applications. By having a descriptive approach to standards for core interfaces and characteristics, core designers have a clearer understanding of the requirements of each of the tools. It is expected that the compilers will eventually adopt the standard that provides access to the largest supply of cores and community of core developers.

2.4. Creating a synergistic environment

One of the most exciting outcomes of standardizing the process of designing and delivering cores for integration in high level language compilers is the opportunity that it provides to bring logic designers and application developers together in a more productive environment. With FPGA-based processors and the use of application-specific cores, programmers can interact with a large community of logic designers in order to obtain optimized cores in a timely fashion.

2.5. Creation of libraries

The wide availability of optimized cores for use in applications implemented in high level languages is the ultimate goal of the CoreLib group's work. By supporting standards the creation and distribution of libraries is enabled. The standardization also permits the creation of centers for collecting, building, qualifying, and distributing cores. Ensuring that high quality cores are available is critical, benefiting the whole community.

3. Related IP reuse efforts

There have been many efforts to improve design productivity by facilitating the reuse of circuit cores. These efforts, targeted primarily to the design of ASICs, simplify the process of reusing and integrating existing circuit IP within a complex ASIC design. The OpenFPGA CoreLib group will reuse existing concepts and infrastructure to support the goals of the effort. This section will summarize and contrast several of these efforts with the goals of this project.

opencores.org [2]: This effort is led by a group of volunteers dedicated to creating and providing reusable circuits for ASICs and FPGAs. This loosely organized group provides a variety of HDL reusable cores based on the wishbone interconnect. The organization provides a variety of online resources including cores, design standards, and specifications for the wishbone interconnect.

OCP-IP [3]: This group is dedicated to specifying and advancing a common standard for intellectual property (IP) core interfaces, and inter-core communication protocols called sockets, for System-on-Chip (SoC) design. This group maintains and sponsors the Open Core Protocol (OCP) socket standard to ensure rapid creation and integration of interoperable virtual components.

SPIRIT Consortium [4]: The SPIRIT Consortium develops specifications that enable increased IP re-use methodologies built on by unifying ways to describe IP and to capture its use in the design flow. The SPIRIT Consortium is best known for the IP-XACT v1.2 specification. IP-XACT provides a well-defined XML Schema for meta-data that documents the characteristics of Intellectual Property (IP) required for the automation of the configuration and integration of IP blocks.

Virtual Socket Interface Alliance (VSI) [5]: The VSIA was created in 1996 by leaders in semiconductor, IP, and EDA companies to define and address the challenges of SoC development. VSIA has created many documents, specs and standards focused on IP integration including documents and standards addressing IP protection through tagging and encryption and IP transfer.

Si2 [6]: The Silicon Integration Initiative (Si2) is an organization of industry-leading companies dedicated to improving productivity and reducing cost in creating and producing integrated silicon systems. This group provides a common set of tools and interface software for facilitating the interoperability of EDA tools.

Collectively, these efforts have demonstrated significant improvements in design reuse for ASIC design flows. As suggested by the growing ASIC and FPGA IP markets, the tools, standards, and methodologies introduced by these groups are indeed improving design productivity. It is likely that the success of these efforts will continue and the ability to reuse circuit IP will allow ASIC and FPGA designers to create larger, more complex circuits by easily incorporating third party IP into their designs.

While these efforts are laudable, they are targeted for the traditional ASIC and FPGA designer and are based on traditional ASIC design flows. The goal of the CoreLib effort is to facilitate design reuse by non-traditional FPGA designers and software programmers that are using high-level language tools rather than traditional HDL design tools to describe an FPGA circuit. To support this goal, additional standards and reuse techniques must be defined and demonstrated. Further, these standards must be adopted and integrated into existing HLL tools.

4. Importing IP within current HLL compiler and tools

A number of compilers and tools are emerging to generate FPGA circuits from programs written in a high-level language. These tools vary in their approaches to integrating IP cores into their environment. Just as there is an effort to study existing IP reuse efforts, there is also an effort to study the various approaches used by HLL tools to integrate user IP cores. This section summarizes the approaches used by several tools to integrate IP cores.

4.1. ROCCC

ROCCC [7] is a C to VHDL compilation tool intended for code acceleration on FPGAs. It applies an extensive set of loop, array, procedure and storage optimizations aiming at maximizing parallelism and clock frequency while minimizing the number of off chip memory accesses by exploiting data reuse on the FPGA [8]. ROCCC supports a user defined IP core wrapper [9] that abstracts away the timing and storage requirements of an IP core making it look like a function call in the C source code. The wrapper is defined by the user in C (augmented with a wait statement) and compiled to VHDL.

4.2. Impulse C

The Impulse C compiler [10] allows the use of customized platform or application-specific hardware components that are defined using either C or HDL programming methods. In support of this, the Impulse C compiler is capable of translating user-specified C function calls into instantiations of predefined, external blocks of hardware. This mechanism also supports the overloading of standard C operators.

For any external function call or overloaded operator in an Impulse C application, the compiler will look for a hardware primitive associated with that function and generate a reference to a lower-level block representing the primitive. The compiler allows these hardware primitives to be explicitly referenced, for example as C-compatible function calls to external HDL code, or be indirectly referenced as lower-level primitives located in an XML format library. In the latter case, XML descriptor files are used to specify the function interfaces as well as define the latency, pipeline behavior and handshake protocols of each primitive.

4.3. Dime-C

DIME-C [11] currently has the capability to integrate cores that exist as one or more VHDL, NGC or EDIF files. DIME-C can integrate both pipelined and non-pipelined cores. Cores must follow the DIME-C naming convention for top-level ports and be accompanied by an XML library descriptor file. This file describes the properties of each core in the library: the data types of its ports, its pipeline quality and the latencies of all the output ports. The user instantiates cores from DIME-C in a manner identical to a C function call. DIME-C can export functions described in ANSI C as low-level cores, suitable either for use in any kind of design flow.

4.4. Trident

Trident [12] is a compiler for floating point algorithms written in C that produces circuits in reconfigurable logic that exploits the inherent parallelism of the algorithm. The Trident compiler supports multiple floating point libraries and imports custom floating point arithmetic units using function calls. Each custom functional unit is described with a definition file that defines its ports, timing, and result. The user specifies the use of the functional unit using a procedure call and intermediate passes of the compiler map the procedure call to the corresponding functional unit.

4.5. Carte

The Carte Programming Environment [13] supports compiling C and Fortran to logic in Xilinx and Altera FPGA-based reconfigurable processors. The compiler translates HLL code into a control flow/data flow graph hybrid composed of interconnected cores (functional units). The cores are maintained in a library and available to the HLL as operators, intrinsics, and procedure calls. Cores are integrated into the programming environment through "info file" entries that provide descriptions of each core's interface, and characteristics such as latency, statefulness, pipeline quality, as well as C and Fortran code equivalents to the HDL core. Programmers can provide their own core libraries by including HDL in source or synthesized logic together with "info file" entries for each core. User core libraries can be used to add new cores or to provide alternate implementations for the standard cores provided with Carte. Compilation of an application can target CPU and FPGA hardware for performance, CPU and clock accurate simulation for debugging cores, or CPU alone for application debugging.

4.6. Reconfigurable Computing Toolbox

The Reconfigurable Computing Toolbox [14] for Matlab/Simulink is a general purpose, high-level graphical programming language for accelerating software using hybrid CPU/FPGA platforms. IP Cores can be integrated into the RC Toolbox to create software-callable functions, or to be used as part of a more complex program. Cores can be included as VHDL, Verilog, NGC, or EDIF files. All cores require a block-box configuration M-file, which can be automatically generated by importing VHDL/Verilog. Cores are included by instantiating a black box library function into a graphical program and connecting the input and output variables as desired. The entire program can be simulated and debugged from within the Matlab/Simulink environment. In addition, Xilinx System Generator blocks, as well as any core created using Xilinx System Generator, are automatically integrated with the RC Toolbox by simply instantiating the core into the user's graphical program. All required simulation and implementation files are automatically generated.

4.7. C2H

C2H is a hardware acceleration compiler that boosts the performance of time critical C functions by converting them into hardware accelerators in an FPGA [15]. C2H can interface to any block instantiated with Altera's SOPC Builder. It uses information about the blocks interface, such as read-latency, to optimize flight-path scheduling. Future work will also take burst characteristics into account to maximize transaction efficiency. Because of this information and the SOPC Builder framework, C2H can automatically build instantiate and connect to other IP without the user typing a line of HDL.

4.8. HARWEST

The HARWEST framework [16] is a C to VHDL source to source compiling environment. Starting from input specifics expressed as a C program, it generates a synthesizable VHDL code implementing the original program. The HARWEST compiling environment is implemented as a set of C++ libraries and requires a C++ compiler. In this framework, the reuse of pre-designed IP cores is possible through the following steps: first, defining the C function prototype according to the IP I/O behavior, second, define the module C++ class representing the IP core according to the IP properties (I/O ports, static information and synthesis support files), third, define the mapping between the module C++ class and the node C++ class. Once defined, an IP core can be used in a C program by calling the C function defined in step one.

4.9. DWB

The Delf WorkBench (DWB) is a set of design and compilation tools that support the Molen architecture [17]. The Molen reconfigurable accelerators are coupled with general purpose processors and employ application-specific functionalities referred to as Custom Computing Units (CCU). Two major DWB tools are the Molen compiler and DWARV[18] – the Delft Workbench Automated Reconfigurable VHDL generator. The DWB allows easy compilation of application code and integration of IP cores as co-processors in the Molen architecture. To integrate an IP core, it has to be designed as a Molen custom computing unit (CCU). In addition, the DWB allows automatic generation of IP cores with the DWARV tool, which translates C kernels into VHDL circuits with proper CCU interface. At software level, an IP is integrated automatically using the Molen Compiler. The DWB is validated on a prototyping HW platform based on Xilinx Virtex II Pro chip and is available for on-line experimentations [19].

4.10. CHiMPS

CHiMPS compiler compiles high level languages into hardware circuits that can be implemented on a Xilinx FPGA [23]. As an intermediate step when generating the hardware circuits, the compiler compiles an application into CHiMPS Target Language (CTL). CTL is a predefined instruction set and there is one-to-one correspondence between the final hardware and CTL generated for an application.

To facilitate importing IP cores CTL defines "foreign" instruction. When an IP core needs to be used within the CHiMPS compiler, a foreign instruction is generated by the compiler and the hardware IP corresponding to that particular foreign instruction is imported into the final hardware. A data file and behavioral implementation of the IP (for instance implementation of the IP in C or C++) is needed in addition to the IP core. The CHiMPS Simulator uses the behavioral code and latency and area estimates from the data file when simulating an application.

5. Proposed IP interface approach

A major challenge facing this effort is to define a standard that allows complex IP cores to be easily used and instantiated within a high-level specification. At the same time, this standard should be expressive enough to describe all of the technical details of the IP core that are necessary for interfacing the core to the other circuits synthesized from the high-level code. This task requires the careful representation of many details associated with a given circuit library while abstracting these details in a form that allows a programmer to instantiate and use the IP.

For example, an end user may use a high-level language such as C or C++ to describe a complex DSP function that includes an FFT. The user would like to use a high-performance core developed by a third party but does not want to understand all of the structural and temporal properties of the core. If done properly, this specification will allow the user to instantiate the function in a simple manner such as a function call or arithmetic expression within the high-level code.

During the compilation process, when the compiled code is linked to the library element, the circuit must be connected structurally to the rest of the system. To do this, the library element must be described with enough information and meta-data to allow generation tools and compiler tools to interface the specifics of that particular function call to allow for its integration into the synthesis of the whole design. This methodology will support mixed hardware (cores) and software (functions) models, fully controlled by the user. The information necessary for a compiler includes a description of the structural interface, temporal interface, control interface, and other IP specific details for creating and instancing the circuit within a larger and more complex system.

5.1. Structural interface

The most important properties of a core that need to be published in a standard are the properties that describe the structural interface of the core. This interface describes the pins, signals, and structural information needed by a tool to hook up the core to the rest of the synthesized circuit. Some structural interface is required by any interoperability specification.

The first and most important structural definition of a core is the list of signals used by the core. These signals should be named and classified into one of the following categories: clocks, resets, control, and data signals. Core specific properties could be added to each signal as necessary.

For use by a high-level tool, raw signals are not sufficient. Instead, detailed "type" information is needed for every data signal. The type information would indicate, in hardware detail, the exact type of the signal (i.e. bitwidth of the given type including fixed-point unsigned, unsigned, floating point, bit, Boolean and any application-specific type). A standard set of parameterized types would have to be defined and recognized by both the core developer and the compiler.

5.2. Temporal interface

The temporal interface of a core is also necessary for interfacing the core within a HLL tool. The temporal interface describes the timing of the signals of the core. The timing would include the data signal arrival times, pipelining depths, data introduction intervals, control signal timing, etc. This timing information is necessary to interface the core into the rest of the datapath of the synthesized circuit. The compiler could use the timing information to decide whether or not the resource could be shared, etc.

For SOCs, the temporal interface is usually defined by a bus standard. While this bus-centric model makes IP interfacing much easier, this is much too limited for the goals of this effort. Traditional ASIC IP reuse strategies rely on the use of well-defined bus protocols to facilitate reuse (i.e. wishbone, OCP-IP, AMBA, etc.). These bus protocols describe the signals, timing, and communication rules for sharing information between cores, processors, and memory.

While the bus-centric form of circuit reuse is very useful, it is too restrictive for the style of reuse that we are pursuing in this work. FPGA architecture provides abundance of routing resources. This enables FPGAs to provide enormous amount of internal bandwidth. Imposing a shared bus-centric model takes away one of the fundamental advantages provided by the FPGA architecture. Complex bus protocols require expensive hardware resources and bus protocols typically involve multiple cycles for a single transaction. For this work, the circuit IP must be integrated tightly within an optimized datapath circuit using lightweight, inexpensive communication channels. This standard must support a wide variety of communication models such as point to point, pipelined, multiplexed, and bus. A key challenge of this effort would be to properly describe/support a wide variety of communication models.

5.3. Control interface

The control interface describes how the core is synchronized with the rest of the circuit. This would involve describing any handshaking needed to start/stop the operation, signaling to indicate completion of the computation, etc. We would definitely want to support both statically scheduled operations (i.e. fixed number of clocks) and dynamically scheduled operations (i.e. completion times not known at compile time).

5.4. Other IP information

There are a variety of other attributes about circuit cores that may be needed in order to integrate the core into a more complex system. For example, many IP cores are parameterizable. Any core definition should expose these parameter options for a given core (or module generator). This interface would indicate the various options that can be used to generate or configure a specialized core from a parameterizable core.

Many cores and core generators rely on third-party tools. An interface definition would need to describe these tools and provide a way to "execute" these tools when necessary (i.e. something like Xilinx Coregen or the BYU JHDL tool that needs to be run in order to generate the optimized cores). If an HLL was used to create the IP, this tool interface would be needed.

Timing and area estimations are essential for high-level synthesis and the interface would need to provide a mechanism for describing such core estimates. This could be flexible enough to add new estimation parameters such as power and routability.

Some tools support compiling codes to run in either simulation or on traditional microprocessors with some form of emulation. To support this capability the HLL tool must have access to a behavioral description or possibly a HLL form of the IP core that can be used to simulate and debug applications without running on the FPGA-based hardware.

6. Core library standard

In addition to a standard for describing reusable circuit IP cores, this effort is developing a standard for packaging these cores into reusable libraries. The goal of this library effort is to facilitate the collection and distribution of cores to a large population of application developers. Fig. 1 demonstrates this concept. A variety of libraries are created that meet an accepted standard. These libraries are made by independent vendors for a variety of purposes and applications. At the same time, a number of high-level language tools are created that support the same standard. Each of these tools are able to exploit the large and varied set of libraries that were created to meet this standard and do not have to rely on a small set of vendor specific libraries.

While the actual structure and makeup of libraries standard is in its infancy, there are several characteristics of libraries that must be considered. First, libraries will likely come from third-parties and may require IP protection through some form of encryption. Further, these libraries may be delivered in a variety of formats including open source HDL as well as binary only implementations.

If provided in source form, the source code for libraries should be able to be controlled via recognized change management software. Further, the libraries must be organized to be consistent with structures required by place and route tools, including constraint files so that the circuits can be built from source code and a specified set of tools.

Second, libraries may be specific to a single FPGA architecture, reconfigurable computing platform, or tool chain. Alternatively, the libraries may be FPGA or platform independent for greater reusability. The library specification should be flexible enough to support both varieties and should provide sufficient information to describe the limitations of the given IP.

Third, the circuit libraries must also include the appropriate software interfaces for use by the software compiler. This may include .h files or object module libraries that are linked by the compiler. These libraries may provide specialized replacements for standard cores. Libraries must contain descriptions of interfaces, and core characteristics to allow integration with compilers and may include software and hardware simulation models.

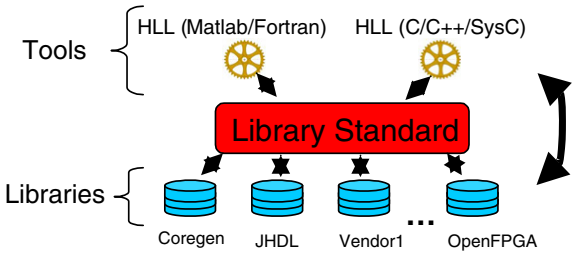

Fig. 1. Interoperability of core libraries.

7. XML-based standard

The approach we are currently pursuing is to create an extensible markup language (XML) specification for representing the details necessary for describing cores and libraries. Further, we are basing this specification on the Spirit Consortium IP-XACT format. This section will summarize our proposed approach for building upon the XML-based IP-XACT specification to represent FPGA core libraries.

There are many data objects and views associated with any given circuit library. These views include end-user views, library views, core and verification developer views, along with data object definitions. In addition, IP specific methods are required to generate and integrate the cores into an end-user's design. A methodology must be available for managing the transformations of the design data to be described.

The extensible markup language (XML) is an ideal way for describing the infrastructure of the circuit libraries. A custom Schema can be created to define all of the data types necessary for a given purpose. For this activity, a schema will be created for defining all of the data fields and methods necessary for defining the core-specific data as described above.

7.1. Spirit consortium IP-XACT

An XML Schema for representing circuit IP is currently available. This specification is called IP-XACT [20] and was created by the Spirit Consortium to address both the data-centric aspect of the IP core description problem, along with the actions of code generation, compilation and linking needed by the mixed hardware/software tool flow. The consortium is made up of a number of IP providers and CAD vendors. A variety of tools are currently available that support this standard for SOC design.

The purpose of IP-XACT is to provide a well-defined XML Schema for meta-data that documents the characteristics of Intellectual Property (IP) required for the automation of the configuration and integration of IP blocks; and to define an Application Programming Interface (API) to make this meta-data directly accessible to automation tools. The primary goals of IP-XACT are to:

1. create common ways to describe IP through XML meta-data;
2. provide standard ways of linking IP into design tools (configuration and parameterized generation of IP).

7.2. OpenFPGA IP-XACT extensions

While the IP-XACT specification is sufficient for defining circuit cores within an SOC design framework, the current standard is insufficient for all of the needs of the OpenFPGA CoreLib effort. Our strategy is to leverage this specification and provide extensions to this specification to support the goals of the OpenFPGA CoreLib effort. Extensions are needed to this standard to support the integration of IP cores within a high-level language. These extensions are needed to map the relatively low-level abstraction of the circuit IP (i.e. HDL) to the higher level language based abstraction (i.e. C or C++).

The first set of extensions needed in this specification include the extensions for specifying higher level data types that are more useable by a compiler. Bit-level types that are used to specify circuit "ports" are too low-level for a compiler. Instead, arithmetic types are needed to express arithmetic values. Examples include the types specified in the VHDL-2006 3.0 standards effort [21] which include arbitrary width fixed-point types as well as standard floating point types. Fixed point (parameterizable size, round, overflow, guard) and floating point (parameterizable size, round, err, denorm, guard) will be supported.

The second set of extensions we will add to this standard is a set of extensions for specifying the temporal interface of circuit cores. These extensions will be used to specify the latency and arrival times of input and output signals interfacing to the core. They will also be used to specify any core-specific communication protocol.

A third set of extensions relate to the specifics of interfacing HLL routines with cores. A mapping to the object referenced in HLL must be made to the instantiated core. A HLL sees the operators (+, −, *, etc.) and function calls. An operator such as "+" is often given an internal name which is later mapped to a particular instruction in a CPU or in the case of FPGA-based computing to a core. The description of a core used as an operator must have the internal name associated with it. The same mapping must exist for function calls that result in a instantiated core. The function name must be included in the core description along with the module name for the core. To properly support a HLL a complete set of cores and associated internal names is required for the operators and intrinsic functions that make up the standard library.

Practical experience with HLL compilers and tools which utilize building block cores will undoubtedly lead to additional extensions to IP-XACT. Vendor specific information may be required. Dealing with encrypted cores and validating core identity for certified libraries will have to be dealt with. Resource utilization information will be required to allow compilers to select cores that meet optimization criteria. An extensible standard will be required due to the evolving nature of HLL interaction for core.

8. Example

Floating point IP core implementations can vary in their optimizations strategies and resource utilization. [24–28]. Two IP core examples are presented in this section to demonstrate the need for a standard that allows seamless interface between IP cores and high-level languages. The first example is a simple square root and the second is CORDIC (Coordinate Rotation Digital Computer) [28]. Both examples involve the seemingly simple square root function but demonstrate how library standardization can significantly improve programmer productivity for reconfigurable computing.

8.1. Standard square root

In standard C, a square root operation is normally performed using a call to the **sqrt( )** function as shown below:

```
double a,b;
...
b = sqrt(a);
```

In a software implementation targeting a traditional processor, the programmer can expect that the **sqrt( )** function prototype is defined in the **math.h** standard include file and that the function binary is available in the precompiled math library. This math library contains an efficient implementation of the square root function for the target processor platform via a library realization of the function.

In most cases, the **sqrt( )** function provided by the compiler provider will be the most desirable one to use, and the programmer does not have to be concerned about the performance of the function or how the function actually operates.

This is not always the case as the programmer may know of a better way to perform this function based on application-specific knowledge. For example, if the application programmer knows that a specific instance of this square root function will operate on a function with a narrow set of input possibilities or small bit precision, there may be opportunities for increasing overall performance with an alternative, custom implementation. The user may replace the standard **sqrt( )** function with a custom look-up based square root as shown below:

```
double a,b;
...
// Use lookup table for a known range
my_fast_sqrt(a, & b);
```

In fact, this sort of optimization – which might be done in an iterative fashion as the programmer seeks higher performance and smaller code size – is often performed by developers of embedded and high-performance systems. It is easier to perform this kind of optimization if the language and software compiler being used has a well-defined function interface as well as flexible methods for creating and maintaining libraries. In the FPGA world, allowing platform-specific and application-specific optimizations to be easily managed by programmers is critical to the success of software-to-hardware design flows. Software programmers bringing their prior experience and their existing applications into FPGAs must be provided with library and core usage mechanisms that feel intuitively like the software design methods with which they are familiar.

Three implementation examples of the square root function demonstrate how this design methodology would work for an FPGA application developer writing an algorithm in C.

1. As a first implementation, the programmer might simply call a default **sqrt( )** function provided by the compiler provider. This **sqrt( )** function might be implemented in a generic platform independent HDL. The compiler provider documents the **sqrt( )** function and ensures that expected performance and characteristics are delivered to the user.
2. As a first-level optimization, the programmer might want to take advantage of a platform-specific library component. An example of this can be found in the Xilinx Coregen library, which includes a double-precision floating point implementation of a square root function. While in some cases the programmer can expect that the compiler will automatically take advantage of such libraries, there may be cases in which more explicit references must be made to these platform-specific functions.
3. As a next-level optimization, the programmer might decide to make use of an alternative implementation of the **sqrt( )** function, either by hand-crafting their own HDL design or by obtaining optimized version from a hardware platform vendor or third-party IP supplier. In either case, it is critical that a well-defined, well-documented interface be provided between the C-to-FPGA compiler and the lower-level hardware library or component.

Although **sqrt( )** may seem as a simple example, this function can in fact have many possible implementations ranging from a simple combinatorial lookup table to a multi-cycle, variable latency pipelined implementation.

8.1.1. Coregen square root core

The following VHDL code represents just one possible implementation of this function, in which the application or core developer has made use of a Xilinx Coregen library component:

```
entity my_sqrt_double is port (
    reset,clk: in std_ulogic;
    x: in std_ulogic_vector(63 downto 0);
    sq: out std_ulogic_vector(63 downto 0);    ce: in std_ulogic);
end my_sqrt_double;

architecture behavior of my_sqrt_double is
    signal x_slv: std_logic_vector(63 downto 0);
    signal result_slv: std_logic_vector(63 downto 0);
    signal operation_nd, rdy : std_logic;
begin
  operation_nd <= '1';

  xilinx_sqrt : floating_point_v2_0
    generic map(
      c_has_sqrt => 1,
      c_a_width => 64,
      c_a_fraction_width => 53,
      c_result_fraction_width => 53,
      c_result_width => 64,
      c_has_ce=> 1,
      c_has_operation_nd => 1,
      c_rate => 1,
      c_latency => 12)
    port map (
      a => x_slv,
      operation_nd => operation_nd,
      clk => clk,
      result => result_slv,
      ce => ce);

  x_slv <= std_logic_vector(x);
  sq <= std_ulogic_vector(result_slv);

end behavior;
```

In an example such as this, there are a number of questions that must be answered before an HLL compiler can take advantage of the core as a function callable from C code. These questions include:

- What is the format of the data? In this case the data type is clear, but that is not always the case, for example when dealing with fixed-point numbers.
- In the case of a core with multiple outputs, which of these represents the function return value?
- What is the latency and introduction rate of the function? This is important to allow the software-to-hardware compiler to efficiently schedule this hardware function in among other parallel and/or sequential operations.
- What is the estimated resource utilization and maximum clock that can be expected? This information may also be critical for efficient hardware generation, or may simply be desirable to present to the programmer as part of a more comprehensive estimation tool.

More complex C library functions and corresponding cores may include the need for stateful behaviors or require some level of multiple-instance resource sharing to reduce total area requirements. These behaviors and capabilities must all be exposed to the HLL compiler.

8.1.2. Carte IP core description example

The Carte programming environment integrates IP cores as operators or callable functions to C or Fortran HLL routines. To inform the compiler of the characteristics of the IP core that is to be integrated a Carte unique ‘‘info file’’ is used. The info file

defines the interfaces and the characteristics of the specific core that is to be used. The following demonstrates an "info file" used for the square root core described in the previous section:

```
BEGIN_DEF "my_fast_sqrt"
    MACRO = "my_sqrt_double";
    STATEFUL = NO;
    EXTERNAL = NO;
    PIPELINED = YES;
    LATENCY = 13;

    INPUTS = 1:
      I0 = DOUBLE 64 BITS (x)  // explicit input
      ;
    OUTPUTS = 1:
      O0 = INT 64 BITS (sq)  // explicit output
      ;

    IN_SIGNAL : 1 BITS "clk" = "CLOCK";
    IN_SIGNAL : 1 BITS "reset" = "RESET";

    DEBUG_HEADER = #
        void my_fast_sqrt__dbg (double v0, double *res);
    #;

    DEBUG_FUNC = #
        void my_dsqrt_debug (double v0, double *res);
        void my_fast_sqrt__dbg (double v0, double *res) {
           my_dsqrt_debug(v0, res);
        }
    #;
END_DEF
```

The above "info file" provides the required information for the Carte C compiler to construct a circuit with the "my_sqrt_double" core instantiated and called in HLL as the "my_fast_sqrt" double precision function. Additionally, information is provided to allow the compiler to compile the application to run in a traditional microprocessor and access the behavioral routine "my_dsqrt_debug". This definition is specific to Carte and would be of no use to other HLL tools or compilers. A standard description of this macro would allow the same Verilog or VHDL defined core to be used in more that one HLL tool, making the core interoperable across a number of HLL environment.

8.1.3. Modified IP-XACT IP core description example

An example of an IP-XACT style definition for this square root core is presented below.

```
<spirit+:component> xmlns:spirit+=…>
        <spirit+:name>    my_sqrt_double  </spirit+:name>
        <spirit+:callname my_fast_sqrt    </spirit+:callname>
        <spirit+:busInterface>
                <spirit+:name> CarteBus </spirit+:name>
                <spirit+:signalMap>
                        <spirit+:signalName>
                                <spirit+:componentSignalName> clk
                                </spirit+:componentSignalName>
                                <spirit+:busSignalName> CLOCK
                                </spirit+:busSignalName>
                        </spirit+:signalName>
                        <spirit+:signalName>
                                <spirit+:componentSignalName> reset
                                </spirit+:componentSignalName>
                                <spirit+:busSignalName> RESET
                                </spirit+:busSignalName>
                        </spirit+:signalName>
                </spirit+:signalMap>
```

```
                </spirit+:busInterface>
                <spirit+:signals>
                        <spirit+:signal>
                                <spirit+:name>      x </spirit+:name>
                                <spirit+:direction> in </spirit+:direction>
                                <spirit+:left>      63 </spirit+:left>
                                <spirit+:right>     0  </spirit+:right>
                        </spirit+:signal>
                        <spirit+:signal>
                                <spirit+:name>      sq </spirit+:name>
                                <spirit+:direction> in </spirit+:direction>
                                <spirit+:left>      63 </spirit+:left>
                                <spirit+:right>     0  </spirit+:right>
                        </spirit+:signal>
                </spirit+:signals>
                <spirit+:characteristics>
                        <spirit+:characteristic>
                                <spirit+:name>   pipelined </spirit+:name>
                        </spirit+:characteristic>
                        <spirit+:characteristic>
                                <spirit+:name>   latency   </spirit+:name>
                                <spirit+:clocks> 13        </spirit+:clocks>
                        </spirit+:characteristic>
                </spirit+:characteristics>
                <spirit+:debugMode>
                        <spirit+:name> my_fast_sqrt__dbg </spirit+:name>
                        <spirit+:prototype>
                                void my_fast_sqrt__dbg(double v0, double *res)
                        </spirit+:prototype>
                        <spirit+:definition>
                                void my_fast_sqrt__dbg(double v0, double *res){
                                        my_dsqrt_debug(v0, res);
                                }
                        </spirit+:definition>
                </spirit+:debugMode>
        </spirit+:component>
```

This XML-based definition is rather verbose for the reader but is designed for implementing interfaces to tools and compilers and is normally accessed through an editing tool. By using XML and a well-defined schema for external libraries, functions as simple as this **sqrt( )** example or as complex as a partial differential equation solver or FFT can be created in a portable, redistributable way.

8.2. Alternative square root Cores

Many implementations of the square root function exist for FPGAs. [24–28]. They may be implemented using CORDIC, iterative, unrolled loop, or pipelined techniques, each with different latencies, pipeline characteristics, and FPGA resource usage. The selection of a particular implementation may made by the programmer based on precision, speed, or resource requirements. It may also be selected by the compiler based upon the context of the requested operation within the circuit being generated. In either case a description of the available cores and their characteristics is required. The calling sequence for square root in this example may be exactly the same for all of the potential cores. Only the specific interface definitions and characteristics like latency, pipelining, and resource consumption may vary. The description of the core may then only differ as follows:

```
<spirit+:component> xmlns:spirit+=…>
        <spirit+:name>    my_sqrt_double_serial  </spirit+:name>          ←****
        <spirit+:callname my_fast_sqrt    </spirit+:callname>
        <spirit+:busInterface>
                <spirit+:name> CarteBus </spirit+:name>
                <spirit+:signalMap>
                        <spirit+:signalName>
                                <spirit+:componentSignalName> clock    ←****
                                </spirit+:componentSignalName>
                                <spirit+:busSignalName> CLOCK
                                </spirit+:busSignalName>
                        </spirit+:signalName>
```

```
                    <spirit+:signalName>
                              <spirit+:componentSignalName> rst      ←****
                              </spirit+:componentSignalName>
                              <spirit+:busSignalName> RESET
                              </spirit+:busSignalName>
                    </spirit+:signalName>
          </spirit+:signalMap>
  </spirit+:busInterface>
  <spirit+:signals>
          <spirit+:signal>
                    <spirit+:name>      x </spirit+:name>
                    <spirit+:direction> in </spirit+:direction>
                    <spirit+:left>      63 </spirit+:left>
                    <spirit+:right>     0  </spirit+:right>
          </spirit+:signal>
          <spirit+:signal>
                    <spirit+:name>      sq </spirit+:name>
                    <spirit+:direction> in </spirit+:direction>
                    <spirit+:left>      63 </spirit+:left>
                    <spirit+:right>     0  </spirit+:right>
          </spirit+:signal>
  </spirit+:signals>
  <spirit+:characteristics>
          <spirit+:characteristic>
                    <spirit+:name>   nonpipelined </spirit+:name>   ←****
          </spirit+:characteristic>
          <spirit+:characteristic>
                    <spirit+:name>   latency   </spirit+:name>
                    <spirit+:clocks> 25        </spirit+:clocks>    ←****
          </spirit+:characteristic>
  </spirit+:characteristics>
  <spirit+:debugMode>
          <spirit+:name> my_fast_sqrt__dbg </spirit+:name>
          <spirit+:prototype>
                    void my_fast_sqrt__dbg(double v0, double *res)
          </spirit+:prototype>
          <spirit+:definition>
                    void my_fast_sqrt__dbg(double v0, double *res){
                              my_dsqrt_debug(v0, res);
                    }
                </spirit+:definition>
        </spirit+:debugMode>
</spirit+:component>
```

The description items marked with "←****" differ between the two descriptions. However, the HLL C code remains identical for calling the two distinct implementations. The compiler takes care of instantiating the appropriate code. By providing a standard description the code can be compiled using multiple alternate cores as well as alternate compilers, providing an interoperability across tools, core implementations, FPGA chip types and hardware platforms.

8.2.1. Conclusions and future work

This paper has summarized the major goals of the CoreLib working group effort, and has provided an overview of the actual XML-based implementations being proposed. There is much work left to be completed. Fortunately, the CoreLib working group includes end-users and potential core developers as well as HLL compiler vendors, FPGA vendors, and FPGA integrators. There is a shared vision and a common understanding of the importance of this effort.

The effort to establish a standard is broken into phases. Currently, the CoreLib group envisions the effort will proceed in the following phases:

1. Investigate emerging standards for IP cores.
2. Select an existing standard if possible.
3. Investigate existing HLL tools requirements for IP core integration. Integrate a set of examples into each of the tools.
4. Define extensions required to existing standards to meet needs of HLL tools.
5. Define the core standard and implement it in the tools.
6. Define the library standard.
7. Select and implement example IP core libraries.

The current work is involved in phases 1–4. Future work is involved in phases 5–7.

The CoreLib working group strongly encourages more participation by the readers, and welcomes new members (commercial and academic) who may have very different technology backgrounds and goals. We encourage readers to review our web site [22] and participate in the CoreLib IP interoperability standard.

References

[1] OpenFPGA Consortium, www.openfpga.org.
[2] OPENCores Consortium, www.opencores.org.
[3] Open Core Protocol International Partnership (OCP-OP), www.ocpip.org.
[4] SPIRIT Consortium, www.spiritconsortium.org.
[5] VSI Alliance, www.vsi.org.
[6] Silicon Integration Initiative (Si2), www.Si2.org.
[7] Riverside Optimizing Compiler for Reconfigurable Computing (ROCCC), <http://www.cs.ucr.edu/~roccc>.
[8] Z. Guo, A.B. Buyukkurt, W. Najjar, Input data reuse in compiling window operations onto reconfigurable hardware, in: Proceedings of the ACM Symposium on Languages, Compilers and Tools for Embedded Systems (LCTES 2004), Washington, DC, June 2004.
[9] Z. Guo, A. Mitra, W. Najjar, Automation of IP core interface generation for reconfigurable computing, in: 16th International Conference on Field Programmable Logic and Applications (FPL 2006), Madrid, Spain, August 2006.
[10] Impulse Accelerated Technologies, www.ImpulseC.com.
[11] Nallatech, www.nallatech.com.
[12] Justin L. Tripp, Kristopher D. Peterson, Christine Ahrens, Jeffrey D. Poznanovic, Maya Gokhale: Trident: an FPGA compiler framework for floating-point algorithms, in: Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL 2005), pp. 317–322.
[13] SRC Computers Inc., www.srccomp.com.
[14] DSPLogic, www.dsplogic.com.
[15] C-to-Hardware Acceleration (C2H), www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html.
[16] Ylichron Harwest Complier, <http://www.ylichron.it>.
[17] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, E. Moscu Panainte, The MOLEN polymorphic processor, IEEE Transactions on Computers 53 (11) (2004) 1363–1375.
[18] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Yi Lu, S. Vassiliadis, DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator, in: Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL'07), Amsterdam, The Netherlands, August 2007, pp. 697–701.
[19] The MOLEN Prototype <http://ce.et.tudelft.nl/MOLEN/Prototype>.
[20] IP-XACT User Guide v1.2 for RTL, SPIRIT Consortium, July 2006.
[21] VHDL 2006 Packages, <http://vhdl.org/vhdl-200x/vhdl-200x-ft/packages/files.html>.
[22] OpenFPGA CoreLib, <http://isl.ncsa.uiuc.edu/twiki/bin/view/OpenFPGA/CoreLib>.
[23] Dave Bennett, Prasanna Sundararajan, Jeff Mason, Eric Dellinger, An FPGA-oriented target language for HLL compilation, The Reconfigurable Systems Summer Institute, 2006.
[24] Y. Li, W. Chu, Implementations of single precision floating point square root on fpgas, in: Proceedings, the 5th Annual IEEE Symposium on FPGA Custom Computing Machines, 1997, pp. 226–232.
[25] E. Roesler, B.E. Nelson, Novel optimizations for floating-point units in a modern FPGA architecture, in: Proceedings of the 12th International Conference on Field-Programmable Logic, 2002, pp. 637–646.
[26] X. Wang, B.E. Nelson, Tradeoffs of designing floating-point division and square root on virtex FPGAs, in: Proceedings of the 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2003, pp. 195–203.
[27] X. Wang, S. Braganza, M. Leeser, Advanced components in the variable precision floating-point library, in: IEEE Symposium on Field-programmable Custom Computing Machines (FCCM'06), April 2006, pp. 249–258.
[28] R. Andraka, A Survey of CORDIC Algorithms for FPGAs, in: Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays (FPGA'98), Monterey, CA, Feb. 22–24, 1998, pp. 191–200.