Advanced Intermediate Representation with Extensibility (AIRE)

John Willis
FTL Systems, Inc.
Rochester, MN 55906
jwillis@ftlsys.com

Philip A. Wilsey
University of Cincinnati
Cincinnati, Ohio
paw@thor.ece.uc.edu

Gregory D. Peterson
Wright Laboratory
Wright Patterson AFB
Dayton, Ohio
gdp@aa.wpafb.af.mil

John Hines
Wright Laboratory
Wright Patterson AFB
Dayton, Ohio
hines @el.wpafb.af.mil

Alex Zamfirescu
VeriBest, Inc.
Mountain View, California
azamfire@veribest.com

Dale A. Martin
University of Cincinnati
Cincinnati, Ohio
dmartin@ece.uc.edu

Robert N. Newshutz
FTL Systems, Inc.
Rochester, MN 55906
newshutz@ftlsys.com

Abstract

The Advanced Intermediate Representation with Extensibility, *AIRE*, supports integration of advanced HDL tool components and exchange of partially compiled HDL designs. Tool components may be integrated within a single address space using AIRE's internal intermediate representation, *IIR*. AIRE's file intermediate representation, *FIR*, supports integration of tool components and HDL designs though reading and writing AIRE files.

Efforts to build previous generations of HDL tools on a standard intermediate format did not converge or achieve the support of multiple vendors due to competing commercial concerns and technical problems. Earlier approaches no longer meet the requirements of today's advanced tool developers.

Today, advanced HDL tools require consistent representation from analysis through runtime, true portability among tools and platforms, extensibility to support both localized and industry-wide extensions, memory / processor efficiency, intellectual property security, unrestricted access to the specification / revision process and broad HDL language scope. The Electronic Industry Association's AIRE meets these requirements for advanced HDL tools.

1. Introduction

Traditionally, a minimum of several person years were required to provide full VHDL support within an EDA tool. Tool developers were generally required to create a complete tool solution without re-use of tool components. In the commercial sector, this situation increased cost, delayed tool delivery and slowed technology advancement. In the research sector, this situation diverted substantial research effort into duplicate and incomplete infrastructure development rather than research.

The availability of a common VHDL intermediate and compliant tools allows specialists to expediently insert their unique functionality into the design flow. Furthermore, a common VHDL intermediate provides hardware designers with a secure means to exchange intellectual property without compromising utility. Designers can obtain partially compiled components which retain enough information that they can be in-lined into a high performance simulation [1,2] or embedded in a globally optimized hardware synthesis process (capabilities regulated by component provider) [3]. Such capability increases the tool performance available to designers.

This paper presents an overview of the new Advanced Intermediate Representation with Extensibility, *AIRE*. Included in the presentation are discussions of the background leading to AIRE, advanced HDL tool design requirements motivating AIRE, AIRE's basic data types, AIRE's internal IR (IIR) [4], AIRE's file-based IR (FIR) [5], sources for the complete AIRE specification and the status of tools implementing and utilizing AIRE.

2. Background

CAD Language Systems Incorporated (CLSI) developed one of the earliest file-based VHDL representations. Users accessed CLSI's files thru a SPY API. Portions of the CLSI API were adopted by the IEEE DASC's VHDL Intermediate Form (VIF) working group [6]. The group was unable to bring a draft standard to ballot, and IEEE deactivated the group's PAR several years ago. Leda S.A. picked up where the DASC VIF left off, updating the API to VHDL-93 [7], and providing a commercially supported analyzer.

Other early VHDL tool developers created their own, proprietary file formats. These formats are sometimes available under confidential disclosure agreements, however a lack of commonality among tools all but prevents portability of design files or tools written to a specific API.

Logic Modeling, now a part of Synopsys, developed the Swift model interface as another approach to HDL intermediate files. Unlike the CLSI-derived intermediate forms, Swift models are compiled down to machine-specific executables. While these fully compiled models provide greater protection of intellectual property, they all but preclude simulation optimizations crossing the Swift interface (global optimizations), limit simulation to a single instruction set architecture / operating system and preclude use in synthesis.

In order to broaden support for and functionality of the Swift API, several companies banded together in 1993 to form the Open Modeling Forum (OMF). A new study group under the IEEE Design Auotmation Standards Committee (DASC) expects to develop the result of OMF work into a balloted IEEE Standard. The OMF approach is a good solution for a different set of requirements than those addressed here.

3. Design Requirements

AIRE meets eight advanced HDL tool requirements for research and hardware design. These requirements are portability, consistency, extensibility, efficiency, integration, security, availability, and language scope.

3.1. Portability

Designs need to be portable between tools and across hardware and/or operating system boundaries. Thus, differences between platforms that must be resolved include: variant integer range, floating point range and precision, ordering of bits within a byte, ordering of bytes within a word, data type alignment, and address space organization. Furthermore, even when running on the same hardware, operating systems differences may cause portability problems. Such differences include variant virtual address space structure, maximum file size, and file system structure.

Readers and writers of the file intermediate need to recognize and map elements within a file with minimal computational effort. The Sun Unix™ external data representation (XDR) translates data types into and out of a single, canonical representation. While this provides data type integrity, it imposes a penalty when communicating between two platforms with common data representations yet which don't match XDR's canonical data representation.

3.2. Consistency

Application programming interfaces accessing the IR in memory need to have a consistent data model and interface paradigm spanning source code analysis through the runtime application programming interface. Advanced tool capabilities such as symbolic debugging, incremental compilation, interactive graphical user interfaces and dynamic test benches are greatly simplified through such a spanning, consistent API.

3.3. Extensibility

Functionality required by tools typically increases over time. A successful intermediate representation (IR) must provide a means by which either new information can be associated with an existing

description or entirely new kinds of information can be represented with little effort.

In order for such an extensible IR to remain portable, the format needs to have a well-defined, stable, and common core functionality; it must also have extensibility mechanisms allowing a core-compliant tool to skip non-core compliant information with minimal impact.

3.4. Efficiency

A commercial-quality IR must be much more efficient to read and write (in both time and space) than reading or writing the equivalent HDL source code. Both space and time efficiency dictate a binary (rather than textual) IR.

Objects within the IR must be rapidly mapped into and out of the memory representation used by tools. The mapping must involve a minimal number of decisions. Whereas a direct map between IR and memory increases performance, direct mapping does not satisfy portability or HDL integration requirements. Since HDL designs often do not fit in the memory of a single processor, it is important that the IR can be written and read using a single-pass algorithm.

3.5. Integration

A practical file IR facilitates integration of separately compiled units and extraction of design fragments for re-integration with other designs. The units may include predefined language components, library components, or separate design sub-units integrated by a design team.

A practical in-memory IR facilitates integration of tool components developed by distinct parties. For example, the analyzer, code generator and graphical interface associated with a simulator may be mixed and matched from different sources.

3.6. Security

HDL designs often embody proprietary information and intellectual property. Typically some subset of this information must be exported in order to make the design useful. This exported design information must be usable for exactly what the writer intended-no more and no less.

Without some means of providing for discretionary security within the IR, the current situation, intellectual property owners are unwilling to export their design information. Since compiled forms of intellectual property have a well-known standing in the legal community, it is important that the IR be considered as a compiled form of the design and not a wrapper around the source code of a design.

3.7. Availability

In order to achieve broad use, the IR must be readily available to tool developers and end users with little (or no) cost. For maximum acceptance the IR specification must be Internet accessible and suitable for redistribution of identical copies without restriction. Since many users access the IR through tools created by others, it is also important to have multiple, widely-available tools that support the IR.

3.8. Language Scope

The AIRE's IR is initially intended to represent IEEE Std 1076-87 [8], 1076-93 [9], draft 1076.1 (VHDL analog and mixed signal) [10], other VHDL-related standards and IEEE Standard 1364 (Verilog) [11]. Extensions under development, outside the current proposed standard, include support for C, C++, and parallelization.

In a practical design environment, diverse hardware and software components must often be mixed. In order to accommodate this breadth of source representations, it is important that the IR not embed details of any specific language's predefined environment. Instead, such environments are themselves represented as FIR files and partial IIR memory images.

4. Basic Types

AIRE's IIR and FIR representations use twelve basic types. All other types are constructed as aliases or composites using these basic types. The basic types are:

- Boolean (IR_Boolean),
- Character (IR_Char),
- 32 Bit, Signed, Integer Value (IR_Int32),
- 64 Bit, Signed, Integer Value (IR_Int64),
- 32 Bit, Floating Point Value (IR_FP32),

- 64 Bit, Floating Point Value (IR_FP64),
- IR Object Kind (IR_Kind),
- IIR Pointer (IIR only),
- FIR Reference (FIR_Ref, FIR only),
- FIR Proxy Reference (FIR_ProxyReference, FIR only),
- FIR Proxy Indicator (FIR_ProxyIndicator, FIR only), and
- FIR Source Code Location (FIR_Source, FIR only).

FIR readers and writers for each instruction set architecture and operating system must be able to correctly read data items of a basic type which have been written by *any* other supported instruction set architecture or operating system. Information contained in the FIR header (such as the magic number) tells the reader about the writer's instruction set architecture and operating system characteristics. The reader must also be configured based on the instruction set and operating system it is running on.

The record kind (IR_Kind) serves to identify the composite type of an object within either a memory image (IIR) or the body of the FIR file. For example, an IR_Kind may identify a record describing a declared physical type or a variable declaration. Once a specific IR_Kind has been seen in a memory image or file, the structure of the remaining object is predefined (although it may be of variable length, such as an IR_STRING object).

The pointer (IIR*) and file reference types (FIR_Ref) respectively define relationship between objects. Predefined elements of the language, such as the integer type, are assigned references which are implicit for a specific language (such as IEEE Std 1076-93).

Proxies provide a mechanism enabling single pass reading and writing of FIR files. The proxy reference (FIR_ProxyReference) and proxy indicator (FIR_ProxyIndicator) provide for forward reference to a specific object within an FIR. Such forward and resolved references mirror the forward declaration technique used by languages such as VHDL in contexts such as an incomplete type.

The source code location (FIR_Source) provides a linkage back into the design representation most familiar to the human designer. This may be a point in textual VHDL source, a state diagram element or even refer to a drawing and location within a schematic. The FIR_Source realizes the source location abstraction within the IIR.

5. Internal Intermediate Representation (IIR Specification)

AIRE's in-memory representation, the IIR, is defined in terms of a class hierarchy. An IIR class forms the top level. Classes derived from IIR include:

- IIR_DesignFile,
- IIR_Literal,
- IIR_Tuple,
- IIR_TypeDefinition,
- IIR_Declaration,
- IIR_Name,
- IIR_Expression,
- IIR_SequentialStatement,
- IIR_ConcurrentStatement, and
- IIR_List.

Information associated with classes is accessible via class methods. Basic data types, introduced in the previous section, appear within the type signature of these methods, however the IIR does not define the organization of data within an IIR object (leading to many cost/performance tradeoffs).

Design file classes represent the information contained in a single HDL source file. Such information includes a list of comments, a list of design units / modules present in the design file, and information required for tools to cross-reference from the IIR representation back to the designer's original files.

Literal classes represent design elements such as text literals (strings), identifiers, integer literals, floating point literals, bit vector literals and enumeration vector literals.

Tuple classes represent collections of design information with a small, fixed number of elements. Examples include associations between formals and actuals, the value and delay of a waveform element, or the association of a choice and action within a case statement.

Type definition classes represent a range of data values and associated operators. Reflecting a broad view of language design, types include not only

"traditional" types such as integers and arrays, but also signatures, attribute types and group types.

Declaration classes generally represent named instances of types. Examples include VHDL's variable declarations, constants and signal interface items.

Name classes generally refer to implicit or explicit declarations. Examples include VHDL's selected names and attributes.

Expression classes represent a functional relationship which results in a value of well defined type. Examples include VHDL's calls to the time function, addition of integers and allocation of new data objects.

Sequential statement classes represent one or more complete steps within a sequential execution. Examples include VHDL's variable assignment, loops and exit statements.

Concurrent statement composite records define declarative regions with zero or more distinct threads of control. Examples include VHDL's component instantiation, block statement and process statement.

One or more extension classes may be introduced just before any derived class in order for applications to add data or functionality to the predefined IIR specification. In order to preserve binary tool compatibility, data may only be added within the extension classes immediately preceding terminal classes from which objects may be created.

While the current version of the file format is believed to represent the full VHDL language, the class hierarchy is not intended to map one to one with a language reference manual. Source code analysis and transformations readily map arbitrary source code into a canonical subset of VHDL in order to facilitate optimization or other usages. Verilog is supported as a VHDL dialect [12].

6. File Intermediate Representation (FIR Specification)

The file intermediate representation provides a portable representation for IIR information contained within a persistent, operating system file. Whereas the FIR uses classes introduced above in the context of the IIR, the FIR addresses a range of new issues associated with a portable, persistent representation.

Each FIR file consists of a header, body and trailer, as shown in Figure 1. The header defines global characteristics and context used by the following body. The body consists of zero or more objects representing a design. Objects may be as simple as an integer literal or as complex as an entity declaration. References within the body combine the objects into a design representation.

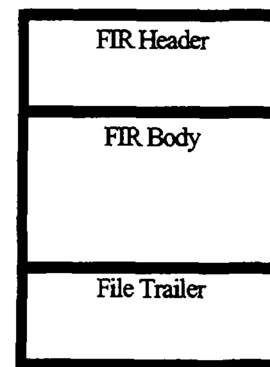

Figure 1. Structure of logical FIR file

The design representation may be as complete and readable (or opaque) as the writer desires. On one extreme, comments and formatting of the original source code may be recovered (to maximize reusability of the design). On the other extreme, identifiers may be obfusticated, then the entire design encrypted (to maximize IP protection). The FIR is a mechanism supporting a wide range of policies.

6.1. File Header

Each FIR file begins with a header. The header provides context global to the remainder of the file (body). Fields within the header include:

- Magic number,
- Guard,
- Version number,
- Number of bytes in header,
- Number of basic types,
- Size of each basic type,
- Byte alignment of basic types,
- Number of IR kinds,
- Number of bytes in each IR kind,
- Presence of source location for each IR kind,

- Number of bytes in extension identifier,
- Bytes containing extension identifier, and
- Number of predefined objects.

The magic number serves to identify the source language from which the file originated, the organization of bits within a byte and the organization of bytes within a word. In the absence of a feasible interpretation of the magic number, such as a source code file inadvertently supplied to an FIR reader, the magic number allows rejection of the file.

The guard explicitly detects the case of an improperly transferred binary file (where the eighth bit of each byte was corrupted in transfer).

The version number defines the header structure, the basic types and the globally defined composite types. Any information contained in the FIR header beyond this portable FIR subset is assumed to be some form of local extension to the FIR format. Compliant readers can then ignore such local extensions unless the structure and semantics of the extensions are known to them.

The number of bytes in the header defines the boundary between header and body. This permits local additions to the FIR header structure while maintaining the ability of compliant readers to parse the FIR file structure.

The number of basic data types field, currently set to 12, provides an opportunity for local additions to the set of basic data types (for example, a 128 bit integer might be added to support some supercomputer architectures). A pair of arrays follow the number of basic data types. The first array denotes the size (in bytes) of each data type, while the second array denotes the byte alignment restrictions associated with that basic data type.

The number of IR_Kinds field denotes the number of distinct composite data types which may be referenced in the file. A pair of arrays follow this integer field. The first array denotes the number of bytes in objects of each IR_Kind. The second array denotes the presence of an FIR_Source for each IR_Kind.

Following objects of each predefined IR kind, one or more extension classes may be added. The extension self-identification helps to distinguish which extensions a given FIR utilizes. A length followed by a character array represents the extension-identifier.

Finally, an integer denotes the number of predefined FIR objects which are implicit in the file (based on the file's source language). These include predefined language elements, such as VHDL's standard libraries. Numbering of explicit FIR objects within the file begins with the the next integer value.

In summary, the file header provides enough information to determine the file's source language, critical instruction set architecture and operating system characteristics of the writer, local additions of new basic or composite types (IR_Kinds) to the file format and addition of entirely new, local extensions to the header.

6.2. File Body

The file body contains zero or more explicit, inter-related objects denoting the design information to be communicated.

Security (asset protection) concerns are addressed by permuting names, removing source location information and encrypting subsets of one or more objects using a defacto-standard encryption mechanism. For legal reasons, the actual encryption engine is defined by reference and external to the tool sets now under development. Neither this paper nor the online specifications directly include an encryption technology.

6.3. FIR Trailer

The FIR trailer consists of a single IR_Int32 representing a modulo 2^{32} checksum of all the bytes in the FIR file (excluding the checksum itself).

7. Specification Availability

Copies of the standardized AIRE specifications are available from numerous web servers world-wide. Sites mirroring the AIRE specification include:

http://www.vhdl.org/vi/aire/
http://www.ftlsystems.com
http://www.ececs.uc.edu/~paw/savant

Following the Ada LRM policy, copies of the standardized AIRE specification may be freely copied (in their entirity) and implemented without royalty (see copyright notice for details).

AIRE is now a working group under the Electronics Industry Association, EIA. For further information on AIRE, send to the group's mail reflectors, currently aire@vhdl.org and aire-request@vhdl.org.

8. Tool Support

Implementations of AIRE have been announced by FTL Systems (Auriga[tm] analyzer / elaborator / optimizer) and the University of Cincinnati (Savant analyzer). FTL System's Auriga is a complete, commercial implementation of AIRE. The University of Cincinnati's Savant analyzer implements VHDL-93 and the AIRE IIR (FIR support planned). Savant is freely available for non-commercial use. Savant available for commercial use from Intellx. Announcement of other AIRE implementations are expected.

Several tens of HDL tools using AIRE are in development by commercial and university groups. These tools include simulators, emulation engines, synthesis tools, performance modeling tools, graphical user interfaces, PCB routers and other physical design tools.

9. Conclusions

Both the research and design community needs an effective, widely available mechanism for the exchange of partially compiled design information. A successful mechanism must address portability, extensibility, efficiency, integration, security, availability and language scope requirements.

In this paper, we provide a design overview of in-memory (IIR) and file (FIR) formats which achieves these objectives. Further, detailed information is available via the World Wide Web. Tools implementations and an associated API are under development, with availability expected later this year.

10. Acknowledgments

This work would not have been feasible without the support of Dr. Robert Parker (DARPA), Dr. John Hines (USAF Wright Laboratory), Randy Harr (DARPA), Ron Waxman (University of Virginia), Patti Rusher (EIA) and the EIA companies. Many thanks are due to early AIRE reviewers, including Serafin Olcoz (TGI), Paul Menchini (consultant), Peter Ashenden (University of Adelaide), Hal Carter (University of Cincinnati), Chuck Swart (Analogy) and others.

Auriga is a trademark of FTL Systems, Inc. VHDLyzer is a trademark of Intellx Inc. All other trademarks are acknowledged as being the property of their respective owners.

Acknowledgments

DARPA is supporting this research under contracts DABT63-96-C-0004 and J-FBI-93-116 as well as Wright Laboratory (USAF) under contract F33615-95-C-1638. This paper does not necessarily reflect the position or policy of the US Government; no official endorsement should be inferred.

Bibliography

1. Willis, J., Li, Z., and Lin T. 1995. Use of Embedded Scheduling to Compile VHDL for Effective Parallel Simulation. *In Proceedings of EuroDAC/EuroVHDL*. pages 400-405, IEEE Press.
2. Martin, D.E., McBrayer, T.J. and Wilsey, P.A. 1995. Time Warp Parallel Simulation of VHDL Descriptions and the Need for Dynamic Parameter Adjustment. *In VHDL International User's Group Fall 1995 Conference*. Pages 7.1-7.10. October.
3. Vemuri, R., Kumar, N., Vutukura, R., Rao, P.S., Sinha, P., Ren, N., Mamtora, P. Vemuri, R., and Roy, J. 1993. An Integrated Multicomponent Synthesis Environment for Multichip Modules. *IEEE Computer*. Volume 26, Number 4. April, page 62. IEEE Press.
4. Willis, J., Wilsey, A., Martin D., Malolan, C., Newshutz, R., McBrayer, T., Carter, H., Shanmugasundaram, V., Rogerman, Steve. 1996. *AIRE Internal Intermediate Representation*. Available from http://www.ftlsystems.com/aire/ and other sites.
5. Newshutz, R., Willis, J., Wilsey, A., Martin D. 1996. *AIRE File Intermediate Representation*. Available from http://www.ftlsystems.com/aire/ and other sites.
6. VIFASG 1076 VHDL Procedural Interface and Schema Definition. 1990. Draft version developed by the VIFASG.
7. Implementor's Guide for LEDA VHDL System. 1993. Available only from Leda S.A. Meylan, France.

8. *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076. 1987. The Institute of Electrical and Electronic Engineers, New York, NY.
9. *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076. 1993. The Institute of Electrical and Electronic Engineers, New York, NY.
10. *Proposed IEEE Standard VHDL Analog and Mixed Signal Language Reference Manual*, IEEE Std. 1076.1. Unpublished working document. The Institute of Electrical and Electronic Engineers, New York, NY.
11. *Draft IEEE Standard Verilog Language Reference Manual*, IEEE Std.1364. 1995. The Institute of Electrical and Electronic Engineers, New York, NY.
12. Willis, J. Moretti, G. Menchini, P. Verilog: Dialect of VHDL? 1995. In *Proceedings of the Fall VHDL International User's Forum*. VHDL International, Mountain View, California.