Kernel Ada to Unify Hardware and Software Design

Sy Wong
Adaware
5200 Topeka Drive
Tarzana, Ca 91356-3923
1/818/345-6247
sywong@markv.com

Gertrude Levine
Fairleigh Dickinson University
1000 River Road
Teaneck, NJ 07666
1/201/692-2020
levine@alpha.fdu.edu

1. ABSTRACT

This paper is a call to SIGAda members to make a determined thrust to broaden Ada usage in the commercial world. More specifically, we wish to exploit an opportunity in the Electronic Design Automation industry (EDA) to use (a kernel of) Ada as a hardware description language (HDL) for the design and programming of today's System On a Chip (SOC). The Ada subset referred to in this paper is intended for the EDA domain to solve practical problems as an HDL, and, in addition, as the interfacing programming language used for testing and simulations (a market currently dominated by C/C++).

Simplicity can be a significant aid in penetrating the market of users and makers of EDA tools. This community consists mostly of electrical engineers and tool manufacturers who would have both a cultural orientation and a commercial interest in a simple kernel of Ada.

We use examples to illustrate the appropriateness of Kernel Ada for the development and testing of reusable hardware components.

1.1 Keywords

Ada, HDL, EDA, VHDL, Hardware Description Language

2. INTRODUCTION TO HDL

"Varied-level computer languages to represent analog, microwave, and digital hardware are becoming increasingly important, which is partially due to a need to predict correct operation of complex ICs and their integration into multi-chip module parts. The problem is especially acute in circuits and systems with mixed-signal functionality. A significant problem area exists in the capturing of information, where the carrier of such information is called a hardware description language (HDL). We need to develop HDL theory and techniques, which support the entire product development cycle. This cycle includes specification, synthesis, test, formal verification, and manufacturing data support, as well as the traditional modeling and simulation arenas. We should also consider coupling HDL developments to computer-aided engineering and computer-aided design tools and backplanes." [5]

HDLs are presently used to capture the architectural design of hardware units, and then to refine it repeatedly through the stages of first modeling the system architecture, then capturing the concepts of the functional elements, then to the detailed logic, and finally to the lowest levels, the circuit elements and their interconnections to high level systems[9]. An HDL's capability of supporting top-down design and of interfacing to simulation and testing programs during various stages of development is essential for the design of very large integrated circuits, and assists in the modification and reuse of design elements.

There are many HDLs used by the Electronic Design Automation (EDA) industry. Most prominent are VHDL and Verilog, both IEEE standards. VHDL was developed by the DoD Very High Speed Integrated Circuits (VHSIC) program office which actually included most of the draft Ada that existed in the early 1980's [3]. Unfortunately, VHDL does not use the Ada constructs that support development of components. Entirely different and apparently disconnected constructs were added that have resulted in a very large and complex language. This might be responsible for the major VHDL shortcomings that are mentioned in Skahill [7, p.7]:

"Design engineers express three common concerns about VHDL: (1) You give up control of defining the gate-level implementation of circuits that are described with high-level, abstract constructs, (2) the logic implementations created by synthesis tools are inefficient, and (3) the quality of synthesis varies from tool to tool."

VHDL relies on using Boolean expressions as abstractions and leaves detailed gate level implementations, called synthesis, to tools or intellectual property(IP) vendors. Using Kernel Ada, the designer can exercise control at all levels when desired.

Verilog was originally developed by an EDA tools vendor, and became an IEEE standard in 1987. Most integrated circuit development tools support Verilog. The tools are relatively expensive compared to VHDL tools.

VHDL or Verilog coding is very different from coding in a computer programming language. Most development organizations use an HDL to design a component, and a programming language, usually C or C++, for simulation and testing. With Kernel Ada, design can be integrated with testing, and an iterative design process can be greatly simplified. For Systems on a Chip (SOC), processor(s) and memories are all contained within a chip, together with other peripheral functions. Design trade-offs between hardware and software implementations often must be made. A simple Kernel Ada unifying both HDL and programming will make trade-offs easier.

The news that a major Japanese electronic manufacturer has extended C for design purposes [2] reinforces our position on the usefulness of a programming language for design purposes. Kernel Ada should be more acceptable to the VHDL community, since much of it is already contained in VHDL, assuming that SIGAda quickly takes action.

3. EXAMPLES OF KERNEL ADA AS AN HDL

We assume no prior knowledge of computer architecture or detailed logic design by our readers for what follows.

The Ada Package is a direct analogy of circuit component packages. Ada Boolean expressions with operators AND, OR, and NOT are completely adequate as abstractions to define the functionalities of circuit components. These two, when augmented by generics and private types, suffice to encapsulate components in package form for reuse.

The examples, oversimplified to illustrate the form of a development environment, are sufficient to support the concepts. The programs use ANSI 1815 Ada-83 and were compiled with gnat, with and without a gnat83 switch, and also with Ada83 compilers [6]. The examples, plus additional code and information, can be found at the site http://alpha.fdu.edu/~levine/wong and also at http://www.cdl.wong.com (the latter oriented towards the EDA community). We begin with a package that contains type definitions for our device designs:

```
-- package HDL provides definitions used by all devices.
package HDL is
    subtype input  is boolean;
    subtype output is boolean;
    type bus is array (natural range <>) of boolean;
end HDL;
```

Note that strong typing is not enforced. Input and output are declared subtypes so that the terms input/output appear in type declarations for readability by the designer but do not prevent the assignment of input values to outputs.

3.1 Nand Gate Design

The basic circuit elements are not AND or OR, but NAND and NOR, which are NOT AND and NOT OR respectively. One important circuit design principle is to minimize signal inversions in a chain of logic elements. To generate AND or OR, an inverting circuit (NOT) must be added with a total of two inversions and a tiny amount of extra component. To paraphrase Ben Franklin, "a transistor saved is a few pico-seconds shaved." We try to think NAND and NOR. Limiting ourselves to two-input devices for simplicity purposes, we adopt a standardized format:

```
with HDL;
package NAND is
    type device is record
      input1, input2: HDL.input:= TRUE;
      data_out        : HDL.output:= FALSE;
    end record;
    procedure update (d: in out device);
end NAND;


package body NAND is
    procedure update (d: in out device) is
    begin
      d.data_out:= not (d.input1 and d.input2);
    end update;
end NAND;
```

The package specification only specifies the externally visible interface including the update procedure that generates new output from new inputs. The functionality of the NAND device appears in the package body. Package NAND is comparable to VHDL's Entity and Architecture, but there are major differences. VHDL Entity is like the pin-out naming in TTL IC catalogs, popular at the time

VHDL was first defined. Package NAND exports an update function so that any number of nand devices (objects) can be updated with new inputs to get new outputs. The actual implementation of NAND.update is encapsulated in the body, distinct from VHDL Architecture. The implementation can be changed and only requires relinking the simulation programs referencing the nand device implementation(s).

Packages NOR and INVERT have similar device and update declarations and can be found at our web sites. Two-input nand or nor devices each require four transistors, and invert requires two. Most tools have standardized layout for such low-level devices and further decomposition down to single transistors is neither necessary nor desirable.

3.2 Dual Ranked Flip_Flop Design and Test

The dual ranked flip-flop (or latch), commonly called D-Flipflop or DFF, is widely used, particularly the "edge triggered" type. A DFF without preset or preclear can be made with three 2-input nand gates connected as three flip-flops.

```
-- Specification for a dual ranked flip-flop
with HDL;
package DFF is
   type device_state is limited private;
   type device is
   record
      data_in, clock: HDL.input:= FALSE;
      data_out        : HDL.output:= FALSE;
      state            : device_state;
   end record;
   procedure update (d: in out device);
private
   type device_state is
   record
      clock: boolean:= FALSE;
   end record;
end DFF;
```

The flip-flops in DFF are memory devices and their current states combined with new inputs determine the next state. A DFF is designed to make output equal to input only when the clock goes from false to true. After the change or clock edge, the input data_in may change without affecting the output. Data_in must be stable during the clock change from false to true, which is always finite in time.

Detailed analysis of what happens during the clock change is not a subject for this paper. The important point is to illustrate the representation of the device_state in package DFF. In order to cope with change, the update procedure must know the previous clock state. The use of a limited private type provides encapsulation for the device.

```
-- First DFF implementation
package body DFF is
   procedure update (d: in out device) is
   begin
      if not d.state.clock and then d.clock  then
           d.data_out := d.data_in;
      end if;
      d.state.clock:= d.clock;
   end update;
end DFF;
```

With Kernel Ada, every component cell can be tested prior to inclusion in other circuits. First we provide a utility package:

```
with HDL;
package UTILITY is
   procedure put (s: boolean);
   procedure putms1st (b: HDL.bus);
   -- left bit is most significant in a string of 1's and 0's.
end UTILITY;


with TEXT_IO;
package body UTILITY is
   procedure put (s: boolean) is
   begin
      if  s  then
          TEXT_IO.put ('1');
      else
          TEXT_IO.put ('0');
      end if;
      TEXT_IO.put("       ");
   end put;


   procedure putms1st (b: HDL.bus) is
   begin
      for i in reverse b'range
       loop
          if b(i) then
              TEXT_IO.put ('1');
          else
              TEXT_IO.put ('0');
          end if;
       end loop;
       TEXT_IO.put("       ");
   end putms1st;
end UTILITY;
```

Test results that should be obtained by running the following testing procedure are included as comments. These are shown after each corresponding display procedure call and are not repeated separately.

```
with HDL, UTILITY, TEXT_IO, DFF;
procedure test_dff is
   d              : DFF.device;
   clock_was: HDL.input := FALSE;
```

```
    procedure display is
    begin
         UTILITY.put (d.data_in);
         UTILITY.put (d.data_out);
         UTILITY.put (clock_was);
         UTILITY.put (d.clock);
         DFF.update (d);
         UTILITY.put (d.data_in);
         UTILITY.put (d.data_out);
         clock_was:= d.clock;             -- state
         TEXT_IO.new_line;
    end display;
begin
   TEXT_IO.put_line ("before update                 " &
                                    "after update");
   TEXT_IO.put_line  ("data_in data_out clock_was" &
                                    " clock data_in  data_out" );
   display;          -- 0        0        0        0        0        0
   d.data_in:= TRUE;
   d.clock:= TRUE;
   display;          -- 1        0        0        1        1        1
   d.data_in:= FALSE;
   display;          -- 0        1        1        1        0        1
   d.clock:= FALSE;
   display;          -- 0        1        1        0        0        1
   d.clock:= TRUE;
   display;          -- 0        1        0        1        0        0
   display;          -- 0        0        1        1        0        0
end test_dff;
```

Now that we have implemented DFF with Boolean expression abstractions, we can write an alternate body for DFF using nand components:

```
-- Second DFF implementation
with NAND;
package body DFF is
    procedure update (d: in out device) is
       unit1,    unit2,    unit3,    unit4,    unit5,    unit6:
NAND.device;
    begin
       if not d.state.clock and then d.clock then
       -- setup input state before clock turns true
          unit5.input2:= d.state.clock;
          unit4.input2:= d.state.clock;
          NAND.update (unit5);
          NAND.update (unit4);
          unit6.input2:= d.data_in;
          unit6.input1:= unit5.data_out;
          NAND.update (unit6);
          unit3.input2:= unit6.data_out;
          unit3.input1:= unit4.data_out;
          NAND.update (unit3);
       -- complete feedbacks
          unit4.input1:= unit3.data_out;
          unit5.input1:= unit6.data_out;
          -- now let clock turn true
          unit4.input2:= d.clock;
          NAND.update (unit4);
          unit5.input2:= d.clock;
          NAND.update (unit5);
          if not unit4.data_out  then
             unit1.input2:= unit4.data_out;
             NAND.update (unit1);
          elsif not unit5.data_out then
             unit2.input2:= unit5.data_out;
             NAND.update (unit2);
             unit1.input1:= unit2.data_out;
             unit1.input2:= unit4.data_out;
             NAND.update (unit1);
          end if;
          d.data_out:= unit1.data_out;
       end if;
       d.state.clock:= d.clock;
   end update;
end DFF;
```

Linking the test program test_dff to the new implementation of DFF, we obtain the same results (else we would know that this unit was incorrect). While designing this second implementation, we made numerous errors because the feedback circuits complicate visualization. Ada's constructs in support of the design of reusable software components allowed us to substitute an alternate design and check the output, thus assisting in the design of reusable hardware components. This two stage design step avoids a major shortcoming of VHDL. Designers, using Kernel Ada, have a choice between relying on tool vendors, or specifying tests and implementation details themselves where appropriate.

With Kernel Ada, assignment statements are used to represent signal flow and perforce circuit connections. Tools could then extract this information to obtain mask layouts. They could also count the number of signal inversions to estimate worst case delays.

3.3 Register Design

With DFF, it is possible to define a register:

```
with HDL;
generic
   N: positive;           -- the number of bits in the register
package REGISTER is
   type device_state is limited private;
   type device is record
      data_in, data_out : HDL.bus (0..N-1):=
                                    (0..N-1 => FALSE);
      clock              : HDL.input:= FALSE;
      state              : device_state;
   end record;
   procedure update (d: in out device);
private
   type device_state is
   record
      clock: HDL.input:= FALSE;
   end record;
end REGISTER;
```

The package specification illustrates the usefulness of generics for an HDL. The package body for REGISTER can be found at our web sites.

3.4 Adder Design

We next design an adder:

```
with HDL;
package FULL_ADD is
   type device is
   record
      carry_in, input1, input2: HDL.input;
      sum, carry_out          : HDL.output;
   end record;
   procedure update(d: in out device);
end FULL_ADD;

package body FULL_ADD is
-- An alternate implementation of FULL_ADD using
-- NAND, NOR, INVERT and X_OR devices (code for
-- which can be found at our web sites).
   procedure update (d: in out device) is
   begin
      d.sum := d.input1 xor d.input2 xor d.carry_in;
      d.carry_out:= ((d.input1 or d.input2) and d.carry_in)
                   or (d.input1 and d.input2);
   end update;
end FULL_ADD;
```

3.5 Accumulator Design

Now we declare a parallel accumulator, with its implementation (found at our web sites) referencing an adder and register, basic components for most processors.

```
with HDL;
generic
   N: positive;           --number of bits in accumulator
package ACCUM is
   type device_state is limited private;
   type device is
  record
      carry_in     : HDL.input:= FALSE;
      data_in,
     accumulator:  HDL.bus(0..n-1):=  (0..N-1  => FALSE);
     -- bit 0 is least significant
      carry_out    : HDL.output:= FALSE;
      add          : HDL.input:= FALSE;
      state        : device_state;
   end record;
   procedure update (d: in out device);
private
   type device_state is
   record
      add: boolean:= FALSE;
   end record;
end ACCUM;
```

With package ACCUM, we have tested a 200,000-bit accumulator implemented down to gate levels for proper carry propagation. Using GNAT compiling on an Alpha machine, a 262144 byte executable image was created that took .9 seconds to execute. A small 10,000-bit accumulator was also tested on an old 25 mhz 386 PC under DOS using an old Ada-83 compiler [6]. The executable size was slightly over 20,000 bytes and took less than a second to execute. This illustrates that Kernel Ada is effective on very simple platforms.

This concludes our demonstration that Kernel Ada is adequate for use as an HDL.

4. WHAT IS INCLUDED IN KERNEL ADA?

A kernel of Ada constructs, as we have indicated, is sufficient for use as an HDL. A suggested list of reserved words is available at our web sites. But are there reasons to limit ourselves to these constructs? Obviously, there are advantages, including simplicity and learning ease, in defining a small language for an HDL. Electrical engineers who program real-time systems for small real-time target platforms, in particular, have need for efficient languages that generate small executable code images and that execute quickly. Synthesis tools may not support the syntax of some of the constructs of a large HDL [8]. There are cost and time savings for developers of tools that interface to the language. Most important, the need for reliable constructs is crucial for the design of integrated circuits.

What is retained in Kernel Ada is based solely on necessity. For our needs, we have excluded all use of heap storage. Concurrency is excluded because it is not necessary for code implementation; all devices have to wait for their inputs. We have also excluded fixed and floating-point types as not applicable for this application area.

Precise definitions of the Kernel Ada subset can best be arrived at by a joint effort of a working group composed of SIGAda members who are interested in a wider use of Ada and of EDA tools and user communities who are interested in a simple language that is effective for all phases of their product development.

5. KERNEL ADA FOR OTHER APPLICATION AREAS

The Ada-95 Language Reference Manual includes an Annex H for Safety and Security Software [4] that recommends limiting certain constructs when system safety is paramount. In addition, the Ada community that creates high integrity systems [1, 10] has restricted similar constructs. These efforts define informal subsets (although the term is rarely used); standard Ada compilers are used with either pragmas or tools to restrict features that are considered potentially dangerous. Interestingly enough, these subsets are similar to Kernel Ada.

Any computer programming educator that teaches beginning programming in Ada chooses an Ada subset, of necessity. The language is too large to teach in one semester, particularly when we consider that the primary subject matter should be design and development, not syntax. The large number of computer science departments that introduced, and in many cases, still use Pascal for introductory programming (even though C/ C++/Java are the favorites of the commercial world) indicates the usefulness of a small, well-designed language for educational purposes. Kernel Ada is comparable to Pascal in its simplicity, but improves on some constructs and is superior for the development of reusable components.

Reliability should also be a goal for beginning students; indeed, the constructs that are excluded from Kernel Ada are exactly those that are the most unreliable for teaching purposes. (Java, for example, has eliminated user defined access types.) Electrical engineers, who frequently program for small target machines, should not learn bad habits on the seemingly unlimited heap storage that may be provided by a personal computer's development environment. Even if a cross-compiler recognizes the target limitations, errors that are not caught by the test suite or a Storage_Error exception can cause catastrophic failures. Thus a (perhaps informal, but surely reliable) Ada subset can be useful in unifying both the computer science and electrical engineering programming disciplines.

The problems with accepting various, perhaps de facto, subsets are numerous. For example,

1) Users must master the larger language, and also each specific subsetting document as needed. It seems unlikely that we can win new Ada users with such an approach.

2) Developers of tools are not likely to commit resources for different subsets, yet developing for full Ada seems unlikely in some communities, particularly EDA tools manufacturers.

3) Critics of Ada can select from any of these subsets to criticize the whole.

4) The Ada community is increasingly fragmentized.

We urge SIGAda members to help define a single subset for the EDA domain, which will be of value to computer science and electrical engineering educational departments as well. One of the authors of this paper has used similar subsets to translate missile assembly codes; it is thus likely that the same subset will be useful for hard real-time embedded systems which also demand high-integrity software. Certainly, tools that are developed for the EDA domain will be beneficial to other areas.

6. CONCLUSION

Ada, being designed with reusable components in mind, anticipated the era of semi-conductor intellectual property (IP), a fancy name for design components. Currently IP vendors must describe their products in an HDL, with their design documents interfacing to simulation and testing programs in a programming language.

The examples we have provided indicate that Kernel Ada as an HDL is sufficient for the entire system development project, including design and testing.

Kernel Ada has a window of opportunity that may not last [2]. SIGAda members must make a concerted effort to agree upon a single (perhaps de facto) subset in order for the EDA industry to adopt it as a stable standard for use as an HDL. We request that SIGAda form a subgroup for the standardization of Kernel Ada.

7. REFERENCES

[1] Barnes, J., *High Integrity Ada, the SPARK Approach* Addison-Wesley, England, 1997.

[2] Cataldo, A., NEC extends C language in bid to speed Design. Electronics Engineering Times, (July 13, 1998), p.1.

[3] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, VHDL. (revised from 1076-1987) June 6, 1994

[4] ISO/IEC 8652, Annex H.4 Safety and Security Restrictions, 1995

[5] Perlman, B. S. Hardware Description Language Theory. NRC Research Assoc. Prog, Fort Monmouth, NJ, http://rap.nas.edu/lab/ARL/76231507.html

[6] R.R. Software, www.rrsoftware.com, Janus compiler version 2.2.2c. Version 2.2.1b (used to compile the examples and has again been placed on sale).

[7] Skahill, K. *VHDL for Programmable Logic* Addison-Wesley, 1996

[8] Smith, M. J. S. More logic synthesis for ASICs. IEEE Spectrum, 29, 11 (Nov. 92), 44-48.

[9] Waxman, R., Saunders, L., and Carter, H. VHDL links design, tests and maintenance. IEEE Spectrum, 26,5, (May 1989), 40-44.

[10] Wichmann, B. A. et al. Guidance for the use of the Ada Programming Language in High Integrity Systems, Ada Letters, 18, 4 (July, Aug. 1998).

Dr. SY Wong began design work at the Institute for Advanced Study Computer Project directed by Professor John Von Neumann, and designed the first transistorized computer in the United States at Philco, Radar DSP and single chip processor designs at Hughes Aircraft. Currently he is using Ada to describe core processors for SOC applications.

Dr. Gertrude Levine is a professor of computer science at Fairleigh Dickinson University, She has published several articles on Ada and has been writing a column for Ada Letters since 1990.