**Technical Report No: 2002/07**

# *Targetting PLDs for high-level High Integrity Systems Development*

***Adrian Hilton***

***Jon G. Hall***

*2002*

---

***Department of Computing***
**Faculty of Mathematics and Computing**
**The Open University**
**Walton Hall,**
**Milton Keynes**
**MK7 6AA**
**United Kingdom**

***http://computing.open.ac.uk***

The Open University

# Targetting PLDs for high-level High Integrity Systems Development

Adrian Hilton and Jon Hall

The Open University

**Abstract.** Combined hardware/software systems are increasingly being used for safety-critical systems, with hardware taking processing load off the software. We have produced a design template that allows SPARK Ada, a programming language used for safety-critical systems development, to be compiled and interpreted on programmable logic devices.
In this paper we describe a high-integrity PLD-based interpreter for compiled Ada code. The intention is to allow developers to select sections of a SPARK Ada program to be interpreted on hardware, rather than run in software, in parallel with the rest of the program running on a conventional CPU. This extends the reach of high integrity systems development to encompass PLDs.
**Topic Areas:** Theory, Mapping and Parallelization; CAD
**Keywords:** Ada, safety-critical, FPGA, PLD, compiler

## 1 Introduction

In [3] we argued that programmable logic devices were becoming an important component of safety-critical systems, taking processing load off the conventional CPU-based software. To this end we proposed a formal methods based development process for PLD and FPGA programs suitable for systems of the highest safety integrity levels (SIL-3 and SIL-4).

A key component of this process was the use of the SPARK Ada high-integrity programming language as a high-level specification language for PLDs. In this paper we describe work in progress on the use of SPARK Ada as a suitable high-level language for programming, directly, PLDs.

The approach depends on the high-integrity nature of SPARK Ada, which it preserves. Therefore, using our development path, SPARK Ada programmed PLDs are suitable for use to the highest integrity levels. In addition, by constructing an effective high-level language compiler we simultaneously make programming PLDs a simpler task. The gains in time-to-market reductions could be comparable to those made in software now that assembly language has been all but replaced by languages such as C, C++, Java and Ada.

Importantly, we do not have to restrict the parallelism present in a SPARK Ada program: coarse-grained parallelism is managed at the level of the CPU-PLD interface; fine-grain parallelism is managed by instruction dependency relations derived from the SPARK data-flow information.

**Section 2** gives a short introduction to SPARK Ada. In Section **Section 3.3** we show how specific sections of a conventional program written in SPARK Ada can be compiled directly into a PLD with no alterations to the remainder of the program. Section 3 details the design of an interpreter executing simple opcodes. The design depends crucially upon the features of SPARK Ada which contribute to the simplicity and dependability of the interpreter. Through an example, we show the correspondence between SPARK Ada constructs and opcodes. Finally we describe on-going and further research work planned in this area.

By 'PLD', unless specifically mentioned otherwise, we include FPGA and CPLD.

## 2 SPARK Ada

SPARK Ada 95 (shortly SPARK) is an annotated subset of the Ada language, as defined in the Ada 95 Language Reference Manual [4, 1]. The use of SPARK for safety-critical system development has been widely validated; for instance, the development process of the SHOLIS helicopter landing guidance system, fitted to the Duke-class Type 23 frigates, made substantial use of SPARK. The development was run under UK Defence Standards 00-55 (software) [8] and 00-56 (system safety). A subsequent study [5] found that the use of SPARK enhanced confidence in the system's reliability.

Of relevance to the development of safety-critical systems, SPARK supports substantial static analysis of programs including strong typing, proof of absence of run-time exceptions, data and information flow analysis, and proof of correctness in the form of pre- and post-conditions on subprograms. Because of the expressive mismatch between SPARK and PLDs, we must be able to tightly constrain expression in the source language. Fortunately, these constraints are statically checkable properties of SPARK programs; SPARK is, therefore, particularly suited to the role we have chosen for it.

The constraints we require are listed them here for future reference:

1. the package and subprogram calling orders forms directed acyclic graphs;
2. no type conversions or expression evaluations will overflow the containing type;
3. loop exit points are restricted;
4. data flow for each statement and subprogram is known at analysis time;
5. all variable types are known at analysis time;
6. all type sizes are known at analysis time; and
7. it is free of run-time exceptions.

A SPARK program that satisfies these criteria we will call *well-formed*. The starting point for the work of this paper is a well-formed SPARK program.

### 2.1 From SPARK to PLD

We begin with a (well-formed) SPARK program together with an identified set of packages therein (the *target list*) that are to be implemented in programmable

logic. The full program transformation process is illustrated in **Figure 1** for a single package P within a program Program. (By sans serif terms in the following we indicate elements of the figure.)

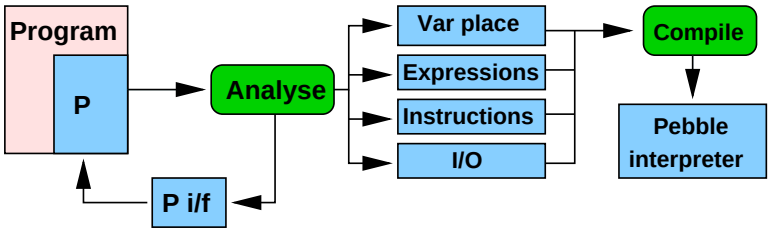


**Fig. 1.** Program transformation

To generate the replacement code, the package contents are analysed to identify subprograms, variables, types and loops. From this analysis, transformation-time decisions are made about pipeline and storage widths and lengths, and the replacement interface for P (P i/f).

The end point of the transformation is another SPARK program in which each of the packages of the target list has been replaced with new SPARK code. The new code has identical specifications and control and data flow annotations to the original code which means that the operational semantics of the original and transformed code are the same. The new code, however, places variables, transforms expressions and instructions, and manages communication with the PLD, sending and receiving data via memory-mapped I/O and on-PLD I/O handlers to replicate the Ada subprogram calling process.

The transformed package is then compiled as a separate block on the PLD; data lines link packages together, and a block on the front of the FPGA manages communication with the CPU. Package and subprogram variables are sized and placed in a local RAM block. Each Ada expression is transformed to a specialised block. Code within subprograms is compiled into a small set of opcodes and data to be stored in a ROM block within the appropriate package.

P i/f, which replaces package P, is a proxy that manages communication with the PLD of the compiled P. Communication may be synchronous (awaiting returned data) or asynchronous (starting a computation and polling for completion) depending on system requirements.

The outcome of this transformation process is a Pebble program, representing the interpreter, which can be interpreted in a simulator or further compiled into a device. We next describe the architecture and behaviour of the combined software/hardware system.

## 3 Interpreter architecture

We model a generic programmable logic device as a set of cells — each of which can compute a simple logic function — and a set of RAM and ROM blocks of configurable size. We assume a single clock across the entire device.

This model was adopted for simplicity, and fits well with the Pebble hardware programming language [6]. Pebble can be compiled into VHDL or directly onto a chosen target device, and the Pebble program model makes it easy to configure programs according to the primitive gates available on a particular device. It gives us a way to map our interpreter onto existing devices without writing an extra compiler stage.

In this section we show how a package is transformed into a form suitable for the interpreter. We assume that packages A and B are inherited and directly used by package P, with other packages C and D inherited by P but not used directly. These packages constitute the target list for an enclosing program. In **Section 3.3** we give an example of the internal structure of a subprogram in package A.

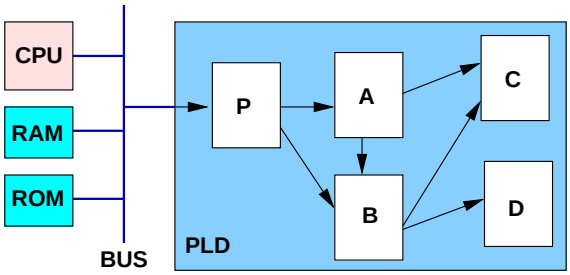


**Fig. 2.** Interpreter Architecture

The architecture of the interpreter determines the form that the compiled code will take. For our example program Program, the resulting interpreter architecture is illustrated in Figure 2. It is based on PLDs linked by a bus with a conventional CPU-RAM configuration, storing the program object code in ROM as well as interfaces to one or more embedded system devices (typically sensors and/or actuators). Under the general translation, we allow each PLD to contain the compiled versions of any number of packages. However, to avoid problems including deadlock and race conditions, we do not allow two PLDs to communicate directly. Communication between CPU and PLD is via memory-mapped I/O.

### 3.1 The interpreter in action

When Program needs to execute P, it sends a 'start' message that identifies a starting program counter value and a set of start data. It finishes when it reaches a return-from-subroutine instruction when its program counter stack is empty.

### 3.2 Input/Output

The flow of data between called and calling package is achieved in a number of ways:

**CPU-PLD** CPU-PLD communications can be synchronous or polled asynchronous in either direction. The appropriate form is chosen by the user before transformation. The CPU-PLD data exchange protocol makes no assumptions about relative clock timings or the amount of data.

**Inter-Package** The guaranteed non-cyclic dependencies between subprograms of packages in SPARK is mirrored in inter-package communications: a package requesting an event from a sub-package blocks until that event is received. Each package has an arbitrator on its inputs which only allows access to one caller at a time.

**Intra-Package** Access to the ROM containing the instructions is controlled by a program counter stack. The stack is fixed-size, determined at transform time according to the number of loops and subprogram calls in the package.

Instructions pass through a decoding pipeline which handles dropping conditional instructions, blocking on unmatched dependencies and routing the instruction data to the computation unit matching the opcode. Instructions are of various lengths.

RAM holds all package variables, internal subprogram variables, actual parameters, and external subprogram parameters. Strong typing implies that each variable can have a permanent start address and word length in RAM. Since subprograms do not recurse, there will never be more than one instantiation of each subprogram variable. SPARK information about subprogram calling order means that independent variables can often be overlapped, reducing the required RAM space.

### 3.3 Compiled code

Package A actually contained a single public subprogram `Closest`, with code given on the left side of Table 1. `Closest` calculates the closest target in a list. On the right hand side of table 1 are the compiled interpreter instructions. (Instructions preceded by `c` are conditional on the result of the last CMP instruction.)

Due to reasons of space we do not fully describe many of the instruction mnemonics; most are self-explanatory. Of the less transparent are IDXRD `a b c d e`, which is an indexed read of element `b` of variable `a` into variable `e`, where `c` is the size of the range type and `d` is the size of the element type. SUBEXT calls a subroutine in another package; here, subprogram `Gap` in package C. The third parameter of a COPY instruction is the number of words to copy.

| | |
|---|---|
| `TList : array(Target) of C.Coord;` | Package variable |
| `procedure Closest(` | |
| ` Crd : in C.Coord;` | `C.Coord` is 8 words |
| ` Tgt : out Target;` | `Target` is 2 words |
| ` Min : out C.Dist)` | `C.Dist` is 3 words |
| `--# global in TList;` | |
| `--# derives Tgt,Min from Crd, TList;` | |
| `is` | |
| ` D : Dist;` | |
| ` Idx : Target := Target'First;` | COPY `Target'First Idx 2` |
| `begin` | |
| ` Tgt := Target'First;` | COPY `Target'First T 2` |
| ` Min := Dist'Last;` | COPY `Dist'Last M 3` |
| ` loop` | LOOP |
| ` D := C.Gap(Crd,TList(Idx));` | COPY `C GAP1 8` |
| | IDXRD `TList Idx 2 3 GAP2` |
| | SUBEXT `1` |
| | COPY `GAP3 D 3` |
| ` if D < Min then` | CMP `< D M` |
| ` Tgt := Idx;` | cCOPY `Idx T 2` |
| ` Min := D;` | cCOPY `D M 3` |
| ` end if;` | |
| ` exit when Idx = Target'Last;` | CMP `= Idx Target'Last` |
| | cLPEXIT |
| ` Idx := Target'Succ(Idx);` | IDXRD `IdxA Idx 2 2 Idx` |
| ` end loop` | LPRTN |
| `end Closest;` | COPY `T TOUT 2` |
| | COPY `M MOUT 3` |

**Table 1.** An example compilation of a package to interpreted instructions

This subprogram illustrates most of the basic SPARK Ada control and data flow constructs. The translation to interpreter instructions was done by hand, but no difficult computation was needed.

## 4 Related Work

Previous research has examined the problem of compiling programs from a range of conventional high-level languages into programmable logic devices (PLDs). Java [7], C and Ada [10] have all come under scrutiny. Modified versions of these languages have been adapted with some success; the canonical example is Handel-C [9] which is targeted solely at programmable logic.

There has been previous work on the specific problem of compiling Ada programs into hardware. In particular, [11] examined the tradeoffs of executing an Ada program on a distributed system, noting that a significant difficulty in the mapping was the undefined behaviour of some Ada programs. Programs in the SPARK Ada subset may not have such behaviour, so we avoid this obstable.

## 5 Conclusions

SPARK Ada is a programming language recommended for safety-critical system development. In this paper we have shown how its use can be extended to safety-critical systems that include embedded PLDs. This extends the reach of current high-integrity software development methods to hardware/software systems incorporating programmable logic devices.

We have described how a well-formed SPARK program with identified packages may be transformed into hardware and software components, running on a conventional CPU and a PLD in parallel. We have described the resulting architecture of the interpreting PLD, and illustrated the instruction set that is the target of the SPARK code compilation. The simplicity and modularity of the interpreter lends itself to straightforward verification.

### 5.1 Further Work

Our current work is the implementation and testing of the transformation process and the SPARK interpreter, including the concommitant proofs of correctness. Further to this, we will develop quantitative and qualitative metrics for application scenarios, data channel widths, and program size and speed.

Transform-time instruction dependency analysis is key to the effective use of fine-grained parallelism, and its concommitant efficiency improvements. This is another area open for investigation.

Placing dependent packages on separate PLDs would be beneficial to reduce the need for large and expensive PLDs. This will be important if it turns out that current PLDs do not have sufficient resources for typical SPARK programs.

## Acknowledgements

This work was undertaken as part of study for a Ph.D. with the Computer Science department of The Open University. Financial support from Teleca Ltd. is gratefully acknowledged.

## References

1. Gavin Finnie and Ross Wintle. SPARK 95 – the SPADE Ada 95 kernel. Technical Report 1.0, Praxis Critical Systems Ltd., October 1999.
2. R. Hartenstein and A Keevallik, editors. *Proceedings of the 8th International Workshop on Field Programmable Logic (FPL'98)*, volume 1482 of *Lecture Notes In Computer Science*. Springer-Verlag, September 1998.
3. Adrian Hilton and Jon Hall. On applying software development best practice to FPGAs in safety-critical systems. In Reiner W. Hartenstein and Herbert Grünbacher, editors, *Proceedings of the 10th International Conference on Field Programmable Logic and Applications (FPL'00)*, volume 1896 of *Lecture Notes In Computer Science*, pages 793–796. Springer-Verlag, August 2000.

4. Intermetrics Inc. *Ada 95 Reference Manual International Standard ANSI/ISO/IEC-8652:1995*. U.S. Department of Defense, January 1995.
5. Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor. The value of verification: Positive experience of industrial proof. In *FM'99 — Formal Methods; Proceedings*, volume 1709 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1999.
6. Wayne Luk and Steve McKeever. Pebble — a language for parametrised and reconfigurable hardware. In Hartenstein and Keevallik [2], pages 9–18.
7. Robert Macketanz and Wolfgang Karl. JVX — a rapid prototyping system based on Java and FPGAs. In Hartenstein and Keevallik [2], pages 99–108.
8. Defence Standard 00-55 issue 2, 1997. Requirements for Safety-Related Software In Defence Equipment.
9. Ian Page and Mike Spivey. How to program in Handel. Technical report, Oxford University Computing Laboratory, December 1993.
10. R. J. Sheraga. ANSI C to behavioural VHDL translator, Ada to behavioural VHDL translator. *The RASSP Digest*, 3, September 1996.
11. Richard A. Volz, Trevor N. Mudge, Gregory D. Buzzard, and Padmanabhan Krishnan. Translation and execution of distributed Ada programs: Is it still Ada? *IEEE Transactions on Software Engineering*, 15(3):281–292, March 1989.