A Hardware Definition Language for Quick Implementation of Messaging Protocols in Field-Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) provide an alternative to microcontrollers for embedded development. They often provide faster speeds and lower power consumption than their counterparts [1]. However, this can come at the cost of increased development time: our lab decided against use of an FPGA in a recent embedded project, despite the project calling for high data throughput and low power draw, precisely because of a time crunch. Existing hardware definition languages (HDLs) used to create FPGA circuits [2] [3] are very general-purpose, and programmers must spend a large amount of time concerned with implementation details that are largely secondary to their objectives. In a word, they are too low-level to enable fast development of solutions to simple problems by small teams. A potential solution to this problem is to borrow from UNIX philosophy: have many interoperable languages of narrow scope built on top of a common low-level foundation, each with syntax optimized for brief resolution of the problems to which they are tailored. I propose the following coarse specification for such a purpose-specific HDL designed for easy implementation of hardware communication protocols, inspired by precisely the problem we ought to have used an FPGA for.

The basic task of interpreting a hardware message is this: “I see some streams of binary data on some pins. I have a specification the data should conform to. Perform actions based on the given binary data when interpreted according to the specification.”

Before streams of binary data are interpretable, it is usually necessary to shuffle around the order of incoming bits into intelligible units based on the physical link used. For example, in standard serial communication one must take each sequence of eight bits off a single pin as a byte, whereas in eight-bit parallel communication one must compose the eight bits from each of the data pins into a byte in a specified order. This is the purpose of the first section of the program: to perform these kinds of stream reshape operations on specified pins. The result is a stream of structured data that is fed to the next section of the program.

After the input is interpreted in the correct general format, it can be compared to a specification. Inspiration as to precisely how is drawn from Haskell’s pattern matching system [4] and the BNF grammar specification language family [5]. The hope is that the intuitive, mathematical syntax will allow effortless translation from protocol specification documents to this part of a program. In this section, data from the input streams is taken until either it matches the specified pattern or it stops matching the specified pattern (depending on whether greedy or non-greedy behavior is desired). Once the final match is found, identifiers appearing in the pattern are bound to values in the stream, and execution proceeds to the next section.

The final step most closely resembles standard imperative programs. Based on the variables produced in the above step, a series of commands are issued to perform whatever actions need be done as a result, such as outputting the received data in a different format or calling a computational routine on it.

The following program is an example for a very simple message structure, showcasing how these elements might work.

```
input a1:
  chunked(a1, 8) // produce stream of bytes from pin a1

match:
  0x5f        // start of packet MSB
  0xf0 | 0xf1 // start of packet LSB options
  packetID // bind identifier packetID to the byte here
  msgSize[2] @length  // 16-bit value, decorated to influence matching
  headerCHK

  payload[+] // A payload can be any positive number of bytes

  payloadCHK

do:
  if checksum(payload) != payloadCHK or checksum(0, 5) != headerCHK:
      return
  else:
      computation(payload)
```

I intend to fully specify this language, including its module system, typing, primitives, standard library, and so on. Once that is completed, I'll plug the grammar into the ANTLRv4 parser generator [6] to produce a program capable of lexing and parsing source files for the language. I'll then fill in the parser actions to produce the compiled SystemVerilog program. This may then be directly applied to FPGAs via the vendor-supplied compiler toolchains. I know very little about SystemVerilog and FPGA computation in general, and hope to greatly expand my capacities in that area as well as compiler and programming language design. I do have extensive experience using a wide variety of programming languages, as well as writing embedded programs that handle protocol translation, which I hope will enable me to create a very practical and useful HDL.

The long-term goal of working on this and similar projects is to form a whole class of interoperable domain-specific languages, as mentioned above. A significant amount of work has been put into the development of so-called reconfigurable computing systems over the years [7] due to the wild parallelism possible with FPGAs, but activity appears concentrated in 2000-2010, with projects like ReconOS [8], a reconfigurable computing operating system, seeing no commits in four years. I conjecture the decline is due in large part to recalcitrant silicon providers refusing to open-source HDL compilers or binary formats for reprogramming, which prevents academics from producing systems capable of generating optimized binaries and flashing FPGAs with them automatically ([7] draws its example architectures entirely from the turn of the millennium despite being written in 2008; those are the most recent examples whose low-level design characteristics are public). The modularity enabled by such free and open tools is almost required for developing disruptive computational technology, and there are a small number of people working on open FPGA compiler technology with the LLVM project for this reason. I doubt the personal computer would have been developed if no CPU providers let the individual consumer have the assembly language manual! It seems there's a wealth of opportunity being stifled by chip vendors protecting their existing electrical engineering market share at the expense of a chance at a massive new market in computation. Beyond enabling further work to improve outcomes in high-performance scientific and mathematical computing, this work would indicate to FPGA manufacturers that their customers are capable of generating useful innovations in compiler architecture, and, failing to convince them to change practices on that basis, generates experience that could be used to bring a satisfactory alternative to market.

References

[1] A. Malinowski and H. Yu, "Comparison of embedded system design for industrial applications," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 244–254, 2011.

[2] "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[3] "Ieee standard for vhdl language reference manual," *IEEE Std 1076-2019*, pp. 1–673, 2019.

[4] "Haskell 2010 language report."

[5] J. Backus, F. Bauer, J. Green, *et al.*, "Revised report on the algorithmic language algol 60," *Numerical Mathematics*, vol. 4, p. 420–453, 1962.

[6] T. Parr, *The Definitive ANTLR 4 Reference*. 2013.

[7] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. 2008.

[8] E. Lübbers and M. Platzner, "Reconos: Multithreaded programming for reconfigurable computers," *ACM Trans. Embed. Comput. Syst.*, vol. 9, Oct. 2009.