An Alternative Tool Chain for DI Circuit Design

Dennis Furey
fureyd@sbu.ac.uk
June 23, 2003

Abstract

An early theoretical model for delay insensitive (DI) circuits with the help of some recent innovations has become the basis for a practical DI circuit synthesis tool, `syndi`. These innovations include a more efficient implementation technique for sparse decision wait elements than any found in the literature, automated detection and decomposition of concurrent systems within a specification, a modern approach to state space enumeration using a method similar to stubborn sets, and a very high level specification language front end. Building on previous work, this presentation will also demonstrate an integrated solution to DI circuit design and verification incorporating a novel DI circuit analysis tool, `diana`.

1 Introduction

Various approaches to asynchronous circuit synthesis are to be found in the literature [4, 3, 5, 2, 8, 6, 1, 7], but the present work is concerned primarily with the problem of a general, fully automated synthesis method for delay insensitive (DI) circuits, and takes as its starting point a lesser known idea from [9].

In the course of a theoretical argument, Patra and Fussel gave a construction whereby it was claimed that any delay insensitive specification could be implemented using an arbiter, an iwire, a decision wait, and an acyclic network of forks and merges (a.k.a. exclusive-or gates). The specification is viewed as a state machine, with a merge assigned to each output, a column in the decision wait assigned to each state, and a row to each input, implying that each output terminal on the decision wait corresponds to a particular state and input pair. The arbiter is interposed between the inputs from the environment and the row inputs to the decision wait to serialize them if necessary, and the iwire enables the column input for the initial state. When a given combination of state and input occurs, a fork from the corresponding terminal enables the column for the succeeding state, signals an acknowledgement to the arbiter, and sends any required outputs to their associated merges. In this way the decision wait effectively executes the state machine specification.

This basic construction would not appear to cope with non-deterministic state transitions or with specifications that are not initially quiescent, although these flaws are easily corrected. It should also be said that the construction was perhaps not intended as a practical circuit synthesis method, which it is not for several reasons. DI circuits have a reputation for being big and slow, and decision wait elements even more so. If decision waits could be deployed at no cost, there would still be better ways of implementing most specifications than by turning them into state machines. Finally, it may be more laborious to derive a state machine specification suitable for automated synthesis than it would be to design the circuit manually.

Current circuit fabrication technology tends toward increased wire delays with each generation, which will cause substantial engineering problems for other delay models in the long term and perhaps nullify their competitive advantage over DI circuits. In preparation for that possibility, the remaining points above are addressed by the present work. A new construction for sparse decision waits curbs their worst excesses. A generalization of the basic state machine construction to incorporate decomposition into communicating subsystems synthesizes circuits more like a human designer would, and a high level circuit description language eliminates the need for detailed state based specifications. These features are incorporated into a new DI synthesis tool, `syndi`, and a DI circuit analysis tool, `diana`.

2 Sparse Decision Waits

In most DI specifications, not every combination of state and input is valid, not just for application specific reasons but because delay insensitivity precludes consecutive transmissions of the same input without an intervening output. The naive implementation of a specification with n inputs and m states would require a decision wait with objectionably many output terminals nm, even though many and perhaps most of these terminals will never be used. A significant improvement would therefore be afforded by a decision wait supporting only the output terminals that are actually

needed.

An elegant design for a sparse decision wait based on a wavefront array was proposed in [7], but the worst case critical path in this design is proportional to the greater of n or m, making it unattractive for large state spaces. Patra and Fussel gave a design for a decision wait with a logarithmic critical path [9], but not being sparse, it could waste more time by being needlessly large than it saves. However, it turns out to be quite possible for a decision wait to be both logarithmic and sparse.

The sparse decision wait synthesis algorithm preferred by the `syndi` tool proceeds by permuting the rows and columns in the specification so as relocate as many as possible of the unused combinations to the lower right quadrant. The remaining three non-empty quadrants are then synthesized individually by recursively applying the same procedure. These three sparse decision waits are glued together by way of a network that intercepts the incoming signals and routes them to the appropriate quadrant. The recursion terminates when a specification is small enough to have a known *ad hoc* DI decomposition.

Because the signals go through only a tree-like structure to reach their destinations, the critical path is logarithmic, and because nothing is wasted on unused outputs, the result is sparse.

This algorithm has been implemented and tested on approximately thirty thousand specifications at this writing, and found to give a correct result in every case. The tests are performed by generating a netlist for each circuit, and checking it for refinement against the corresponding Petri net specification using reachability analysis.

3   System Decomposition

Traditionally a very difficult proposition, it would not be much of an exaggeration to identify the central problem of asynchronous circuit synthesis as that of efficiently partitioning a specification into a collection of subsystems with constant-bounded state spaces. A general solution would imply scalable silicon compilation comparable to that of sequential programming languages.

Some helpful techniques in this connection have been developed. Although they fall short of a general solution due to the lack of hard upper bounds on the sizes of the subsystems obtained, they are conducive to more appropriate implementations for concurrent specifications whose synthesis as state machines would be awkward.

3.1   Partial Direct Mapping

If one were to inspect a particular state machine specification, it might be possible to infer for example that some particular input may occur only in concurrent combination with another one. The pair of them could just as well be combined into a C element, whose output is fed to an implementation based on a form of the specification simplified accordingly. If a state based synthesis were used in both cases, the decision wait in the latter would not only have one less row, but also at least two fewer columns, because there would no longer be distinct states needed to represent that one input but not the other has arrived.

An implementation could benefit from the analogous transformation when two inputs found to be interchangeable are combined using an exclusive-or gate, or more generally when several suitably related inputs are combined using a majority gate. (Some benefit accrues even if no majority gate primitive is available by using a DI decomposition for one.) To a lesser extent, comparable transformations to combine concurrent or interchangeable outputs where possible using forks or randomizing elements are also beneficial.

It is not clear how these transformations would be detected and performed in general for a state machine specification. However, `syndi` initially uses a Petri net representation for all specifications, which is transformed to a state machine only at a later stage. It turns out that all of these transformations are straightforward to carry out by simple pattern matching against the specification while it is still in its Petri net form.

When these transformations are performed, not only is the quality of the resulting circuit improved, but the time spent synthesizing it may be asymptotically reduced insofar as the size of the Petri net decreases. This advantage is normally associated with direct mapping synthesis schemes, but is achieved in a state based system without any loss of generality.

3.2   Connected Components

Applying the above transformations to a specification until a fixed point is reached will in the best case leave nothing of the Petri net remaining to be implemented by state based synthesis. Alternatively, another desirable outcome is for the Petri net to become fragmented into several components, each of which can then be synthesized separately, leading to an overall implementation proportional to the sum of their state spaces rather than their product, as it would be if they were synthesized together.

A less ideal but more likely possibility, particularly when the original specification is a closed Petri net, is that the Petri net remains connected, but takes the form of an ensemble of subgraphs, each bounded by observable transitions. Each such subgraph is able to interact with the others only indirectly by way of a common environment consisting of places and unobservable transitions, and could be pictured as an island

within the Petri net.

It would be possible in this case to synthesize a circuit from each island by treating it as an open Petri net. However, synthesis of open Petri nets is usually prohibitively expensive due to state explosion, and the resulting circuits can be overkill because they will not take full advantage of their restricted environments. (E.g., a sequencer is produced where a latch would suffice.)

Instead, `syndi` makes a copy of the whole Petri net for each island in this situation, but maintains the set of observable transitions for only one island in each copy, redefining those of the others as unobservable. A circuit is then synthesized for each copy. Although synthesis from multiple copies is more time consuming, it is mitigated to the extent that rendering the majority of transitions unobservable often allows more local optimizations to the Petri net than were possible before. In any case, the quality of the resulting circuit stands to gain.

4   Language Issues

The specification language used by the `syndi` compiler is designed for the effectiveness and convenience of working engineers. The language consists of about a hundred built in functions or statements in several related groups, but perhaps only about a dozen anticipated for frequent use. Engineer friendly mnemonics similar to those in other languages and a simple uniform syntax for expressions and statements eliminates a lengthy learning curve. The language is strongly typed and capable of informative compile time diagnostics. Although it is based on a formal semantics congenial to theoreticians, an intuitive or experimental attitude toward the language is welcome. By writing small test programs and inspecting the results as a graphically rendered Petri net or finite state transducer (with the help of the `diana` tool), a user can quickly acquire a working knowledge of the language.

4.1   A Specification Example

Without going into great detail, it may be possible to convey a flavor for the language with a small example. The example in Figure 1 is that of a one-bit memory cell with dual rail data inputs named `write.0` and `write.1`, data outputs `data.0` and `data.1`, a `read` input signal, and a write acknowledgement signal. The cell responds to either write signal with a write acknowledgement, and to the `read` input with the data output corresponding to whichever write input was most recently signalled. The `getput` statement describes a system that engages in a passive handshake with its environment starting with the first input in its parameter list. The compiler directives

```
#petrinet+
#circuit+

memory_cell =

forever do<
    (until got<write.1>) doany<
       getput<write.0,ack_write>,
       getput<read,data.0>>,
    put<ack_write>,
    (until got<write.0>) doany<
       getput<write.1,ack_write>,
       getput<read,data.1>>,
    put<ack_write>>
```

Figure 1. example of `syndi` **source code**

`#petrinet+` and `#circuit+` instruct `syndi` to create two output files, one containing a Petri net listing for this circuit in `petrify` compatible format, and one containing a human readable netlist of DI primitives suitable for analysis with `diana`.

Other features of the language not conveyed by this example are the ability to invoke built in functions for families of circuits (e.g., `arbiter(7)` or `majority(3,6)`), and to build up complex specifications such as pipelines or wavefront arrays through the use of various functional and object-oriented style abstraction mechanisms.

4.2   Black Boxes

One further feature of the language worthy of particular mention, because it is likely to be a make-or-break issue for some prospective users, is its ability to coexist with circuits designed by other means. In a real world application, only the control logic might need to be delay insensitive, with the data path following a bundled data protocol or perhaps being based on synchronous IP cores with asynchronous wrappers as in a GALS style design.

A technique that could be used with no special support from the compiler would be to design the control path in isolation, treating its interface signals with the data path as external inputs and outputs. This technique is not ideal because it would make it difficult to verify the system as a whole for obvious reasons.

In the `syndi` language, the `#blackbox+` directive is suitable for this situation. Any statement annotated with this directive is taken to mean that the user intends to make other arrangements for its implementation, and that no attempt should be made to synthesize it as a DI circuit. When the netlist file is written for a circuit containing black boxes, they are listed by name as if they were primitive components,

even though the rest of the netlist may have been automatically synthesized. However, if a Petri net file is written, it will incorporate a coherent description for the whole system based on the interface information associated with the black box. In this way, it may still be possible to analyze or verify the system as a whole using `diana` or other Petri net tools.

5 Verification

As indicated already, the analysis tool `diana` can be used effectively in conjunction with `syndi` for graphical rendering of Petri nets and state machines. Its other main use is for automated verification. Systems designed by hand or different systems that have been automatically synthesized can be compared with one another and tested for equivalence or refinement. By using novel techniques for transforming a specification between different intensional process models, `diana` can compare Petri nets with Petri nets, circuits with circuits, or Petri nets with circuits. In fact, the testing of the sparse decision wait synthesis algorithm mentioned above has been carried out using it.

A key feature `diana` is its ability to generate Petri net reachability graphs with somewhat improved efficiency by using a method similar to stubborn sets [10]. In an ordinary reachability graph, each node represents a Petri net marking, and an edge connects one node to another if and only if the firing of a single transition will transform the origin to the terminus. The method used by `diana` is based on an abbreviated form of the ordinary graph wherein one marking can be adjacent to another whenever the collective firing of a set of "independent" transitions effects the transformation. Markings reached only by firing individual members of such sets need never be constructed, which reduces the necessary size of the graph and permits verification of slightly larger specifications than would otherwise be feasible.

6 Conclusions and Further Work

An alternative tool chain based on the two programs `syndi` and `diana` brings some new capabilities to the table that were previously unavailable for DI circuits. These include fully automated synthesis from high level specifications, and verification not only of equivalence but refinement.

`diana` has been beta tested by Mark Josephs and Hemangee Kapoor over the past few months, but `syndi` is in a late stage of alpha testing at this writing. Both are are planned for free distribution on the web at `http://www.sbu.ac.uk/~fureyd` by the time of the forum.

Many thanks to all participants for peer reviewing my ideas, as this will probably be the last forum I attend.

References

[1] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, April 1997.

[2] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming and silicon compilation. In *Proc. International Conf. Computer Design (ICCD)*, pages 150–166, Rye Brook, New York, 1988. IEEE Computer Society Press.

[3] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 262–265. IEEE Computer Society Press, November 1989.

[4] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[5] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1987.

[6] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[7] C. R. Jesshope, I. M. Nedelchev, and C. G. Huang. Compilation of process algebra expressions into delay-insensitive circuits. *IEE Proceedings, Computers and Digital Techniques*, 140(5):261–268, September 1993.

[8] Alain J. Martin. A synthesis method for self-timed VLSI circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 224–229, Rye Brook, NY, 1987. IEEE Computer Society Press.

[9] Priyadarsan Patra and Donald Fussel. Efficient building blocks for delay insensitive circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 196–205, November 1994.

[10] A. Valmari. *State Space Generation: Efficiency and Practicality*. PhD thesis, Tampere University of Technology, 1988.