Formal Aspects of VLSI Research Group
University of Utah, Department of Computer Science

From Process-Oriented Functional Specifications to Efficient Asynchronous Circuits

VENKATESH AKELLA* (*akella@cs.utah.edu*)
GANESH GOPALAKRISHNAN† (*ganesh@bliss.utah.edu*)

Dept. of Computer Science
University of Utah
Salt Lake City, Utah 84112

Keywords: asynchronous circuits, high-level synthesis, performance-directed synthesis

Abstract. *A methodology for high-level synthesis and performance optimization of asynchronous circuits is described. A specification language called hopCP which is based on a simple extension to classical flow graphs is introduced. The extension involves the addition of* expression actions *to a flow graph, to model computational aspects of hardware behavior in a purely* functional *framework. Control and Communication aspects are modeled explicitly just as in Hoare's CSP. A systematic methodology to synthesize asynchronous circuits from hopCP based on the notion of a self-timed block is presented. The compilation methodology based on self-timed blocks coupled with the functional flavor of hopCP gives us the ability to exploit several optimizations like quick return, intra-loop pipelining and speculative evaluation of conditional expressions. The specification language hopCP, the synthesis procedure and the optimizations are illustrated in design of an asynchronous iterative multiplier.*

To Appear in the " Fifth International Conference on VLSI Design", Bangalore

*Supported in part by University of Utah Graduate Research Fellowship

†Supported in part by NSF Grant MIP-8902558

1 Introduction

Asynchronous circuits which are based on a explicit request-acknowledge protocol have the following advantages: (i) absence of a global clocking signal and the associated problems of reliable (skewless for example) clock distribution across large ICs (ii) ability to design locally synchronous and globally asynchronous circuits which help retain the advantages of synchronous synthesis (eg: no handshake overhead) within the local subsystems and could have performance gains over purely synchronous circuits and (iii) ability to derive average-case performance instead of worst-case performance from a circuit, because the design is not constrained by the worst case delays of constituent modules unlike in a synchronous design. In an asynchronous circuit the constituent modules can function at a rate governed strictly by *local delays.*

These advantages have revived the interest in synthesis of asynchronous circuits recently [8, 4]. In the next section we will give a very brief introduction to our specification notation hopCP (details are presented in [1]). We then briefly introduce *action-refinement*, which is our technique to transform hopCP descriptions into asynchronous hardware. Finally, we describe several high-level optimizations that can be performed to improve the efficiency of our circuits. We illustrate our ideas on the specification driven design of an asynchronous iterative multiplier.

2 Overview of the Proposed Approach

We take the behavioral specification in hopCP, and translate them into a *hypergraph* (petri-net) notation called hopCP Flow Graph (HFG). This is the intermediate representation for our work. The implementation of a hopCP specification involves *refining* the actions in a HFG into an interconnection of primitive asynchronous circuit blocks. Resource allocation is incorporated into the refinement rules. Data Flow analysis techniques are then used to discover optimizations. We consider three specific optimizations: *pipelining*, *relaxing synchronization* requirements, and *speculative evaluation* of conditional expressions.

2.1 The language hopCP

hopCP specification takes a *sequence domain* view of hardware where the behavior of a system is captured by the *causal relationships* between a set of *actions.* Actions in hopCP could denote control, value communication, and computation. Computation is captured by an expression in a simple first order functional language. Functional languages which are *referential transparent* and free of *side-effects* are appropriate to describe computational aspects because of their *inherent* parallelism (which means parallelism need not be extracted from sequential descriptions as in conventional imperative HDLs like ISPS, VHDL, Hardware C etc.) In addition absence of side-effects leads to elegant formal verification techniques too. Thus hopCP enables us to integrate a process-oriented view of hardware useful in specifying synchronization and value communication with a functional (or abstract datatype) view of hardware which is elegant to capture computational aspects of hardware. The integrated view is the hopCP Flow Graph or HFG. However, hopCP differs from conventional process calculi like CSP [5] and CCS in that actions could be *nonatomic* (temporally refinable). This gives us the ability to model (and reason about) a hardware system at different levels of timing in the same specification formalism. It also leads to a simple and intuitive synthesis procedure based on refining the actions. The specification formalism does not prescribe any specific *timing discipline.* The designer is free to adhere to any *timing discipline* like speed independent, delay-insensitive, transition signaling, level-based signaling etc. [8] during the synthesis, as long as it does not violate the *causal relationship* between the actions.

We will introduce the language with the specification of an iterative multiplier in hopCP. The details of the language and its operational semantics can be found in [1]. Though this is a very simple example it is representative of a wide class of iterative algorithms one encounters in an application like signal processing. The *structural specification* of the multiplier is a module defined as: $\langle MULT, \{a?, b?, c!\}\rangle$ where $MULT$ is the behavioral specification (HFG) and $a?, b?$ are the *input ports* (communication channels) while $c!$ is an output port. The behavioral description is given by the user through the following hopCP program which is then compiled into its corresponding HFG:

$MULT \Leftarrow a?p,\ b?q \rightsquigarrow c!\ multiply(p, q, 0) \rightsquigarrow MULT$ where

```
fun multiply x y z = if (y = 0) then z
                     else case (odd y)
                          true ⇒ multiply x (y-1)(z+x)
                          false ⇒ multiply 2*x (y div 2) z;
```

Here, $a?p$ is an input action, $c!multiply(p,q,0)$ is an output action and $\rightsquigarrow$ captures the sequencing between actions $(a?p, b?q)$ and $c!multiply(p,q,0)$. The comma "," in the action $(a?p, b?q)$ denotes that the input actions $a?p$ and $b?q$ can proceed *concurrently*. *MULT receives* two values p and q from its input ports $a?$ and $b?$ respectively and outputs $multiply(p,q,0)$. Also note that the actual computation involved in the iterative multiplier is elegantly captured by a tail recursive functional program. The initial translation of the behavioral description into a HFG is shown in figure 1. The symbols depicting the HFGs can be interpreted as follows: The circles denote *control states* and the horizontal lines denote *actions* (control, data or expression) The double horizontal lines denote *conditional* expression actions. The arguments in square brackets annotating a state represent the internal *datapath state* of the module.

2.2 Action Refinement Based Compilation Strategy

In this section we present a brief summary of our compilation strategy. Details can be found in [2]. The compilation of hopCP specifications into asynchronous circuits involves deriving implementation for every action in the corresponding HFG. We have three action categories in hopCP: *control actions* which denote synchronization, *data actions* which capture synchronization plus value communication and *expression actions* which capture computation. Every action in hopCP is implemented by an *circuit-abstraction* called an **action-block**. An **action-block** is a piece of hardware (implementing a given action) characterized by an explicit **initiate** and **complete** signal. There are three types of action-blocks: (i) *primitive action-blocks* denote leaf cells of the compiler like the C element, MERGE element, REGISTER (ii) *predicate action-blocks* are circuit elements that implement conditional expressions (iii) *function action-blocks* which implement standard functions like add, subtract, shift etc.

The compilation scheme essentially involves *rewriting* every action using the following **action-**

Illustrating Refinements of Actions in Iterative Multiplier Specification

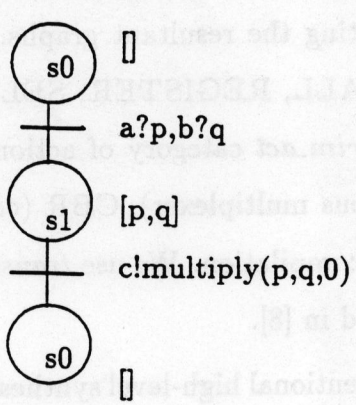
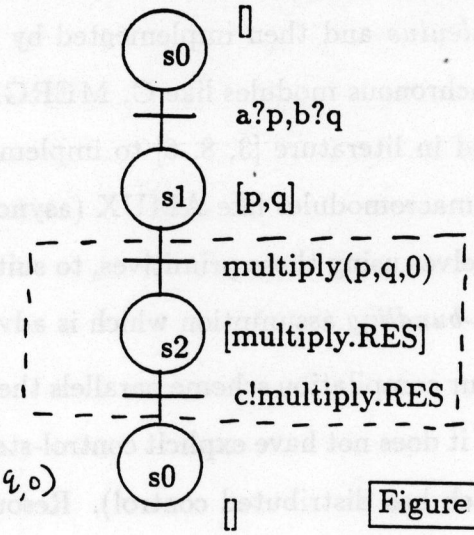


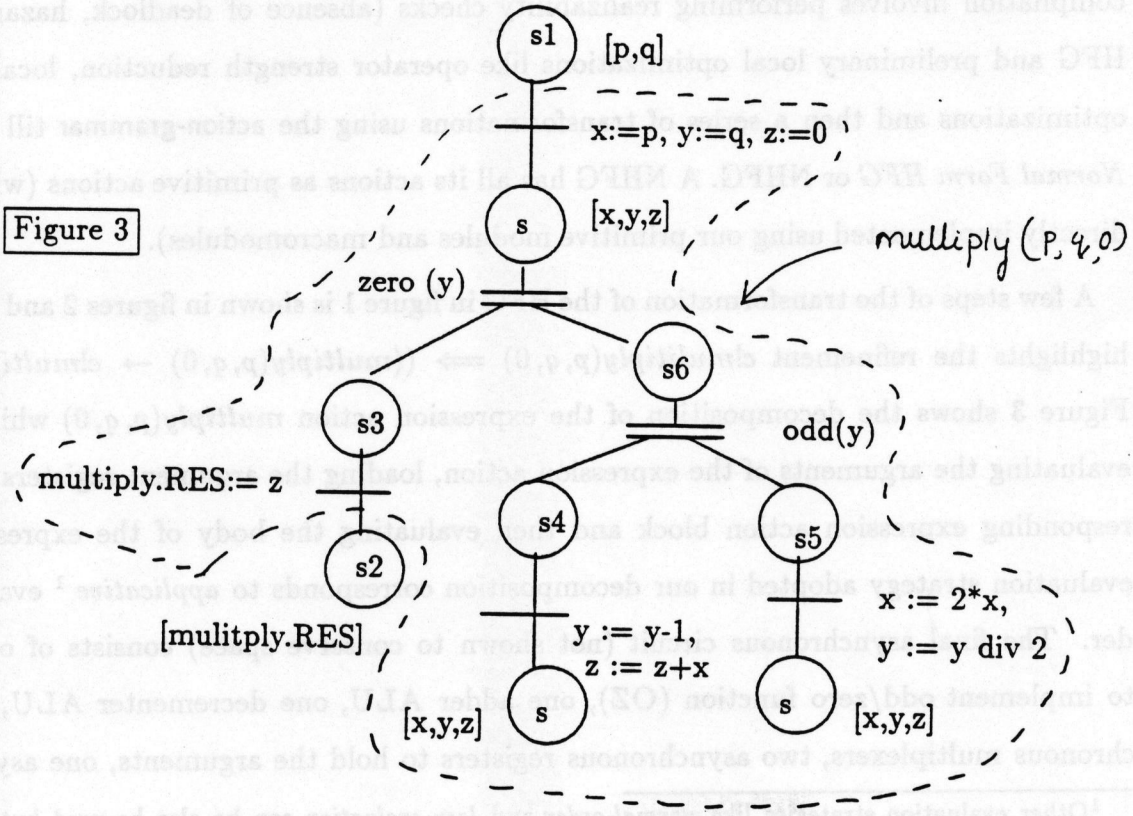
Figure 1



c! multiply (p,q,o)

Figure 2

Figure 3



grammar.

$$Act ::= prim\_act \mid Act, Act \mid Act \rightarrow Act \mid Act \mid Act$$

where, *Act* is the set of possible actions (control, data or expression) in hopCP.

The *control* and *data* actions are directly implemented by rewriting using the above action grammar. The expression actions are first translated into expressions in a first order untyped *λ-calculus* and then implemented by rewriting the resultant graphs. We use the standard asynchronous modules like C, MERGE, CALL, REGISTER, SELECT, ATS, CAL etc. found in literature [3, 8, 6] to implement *prim_act* category of actions. We also designed a few macromodules like AMUX (asynchronous multiplexer), CBR (call-with-boolean-result) ourselves using these primitives, to suit our compilation. We use *transition signaling* with the *data-bundling* assumption which is advocated in [8].

Our compilation scheme parallels the conventional high-level synthesis algorithms [7] except that it does not have explicit control-step scheduling since we are in a asynchronous framework (which has distributed control). Resource allocation is built into the refinement rules. The compilation involves performing realizability checks (absence of deadlock, hazards) on the HFG and preliminary local optimizations like operator strength reduction, local load-store optimizations and then a series of transformations using the action-grammar till we reach a *Normal Form HFG* or NHFG. A NHFG has all its actions as primitive actions (which can be directly implemented using our primitive modules and macromodules).

A few steps of the transformation of the HFG in figure 1 is shown in figures 2 and 3. Figure 2 highlights the refinement $c!mulitiply(p, q, 0) \Longrightarrow ((multiply(p, q, 0) \rightarrow c!multiply.RES))$. Figure 3 shows the decomposition of the expression action $multiply(p, q, 0)$ which involves evaluating the arguments of the expression action, loading the argument registers of the corresponding expression action block and then evaluating the body of the expression. The evaluation strategy adopted in our decomposition corresponds to *applicative* [1] evaluation order. The final asynchronous circuit (not shown to conserve space) consists of one module to implement odd/zero function (OZ), one adder ALU, one decrementer ALU, two asynchronous multiplexers, two asynchronous registers to hold the arguments, one asynchronous

[1]Other evaluation strategies like *normal-order* and *lazy-evaluation* can be also be used but they would result in tradeoffs in area (resources) and time (length of critical path)

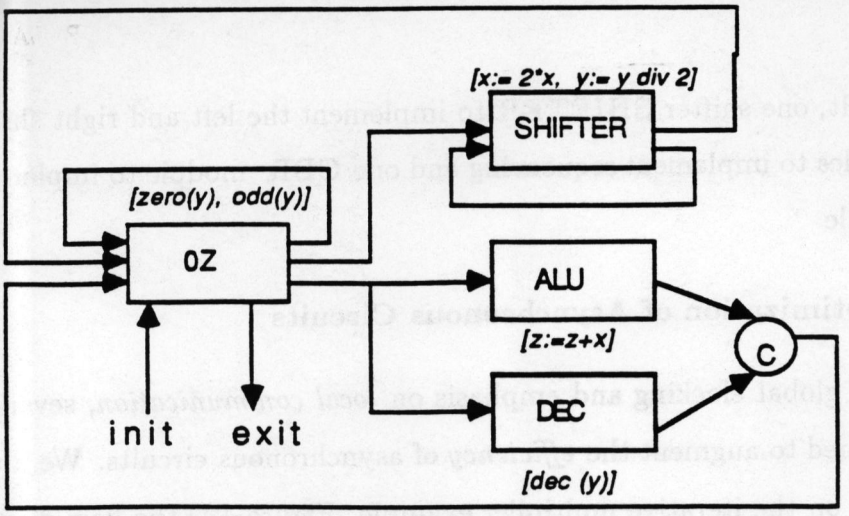


Figure 4

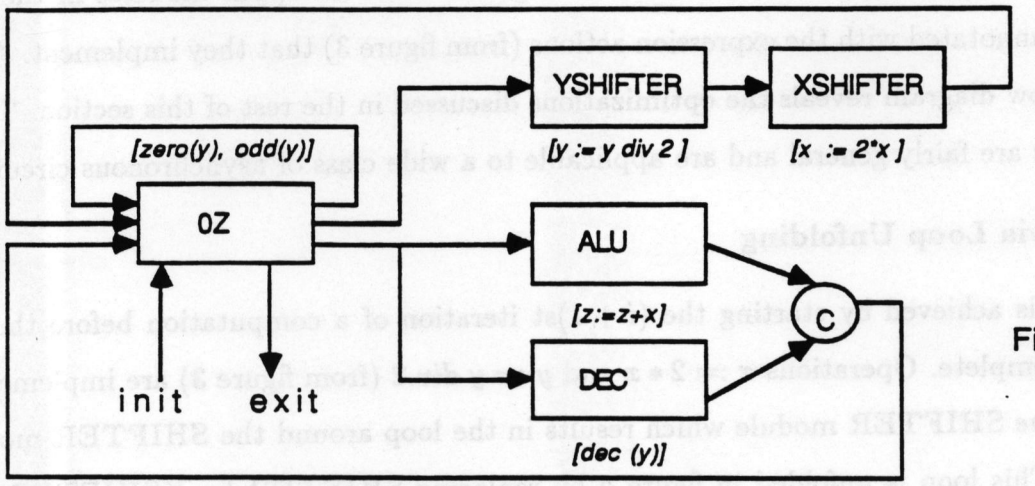


Figure 5

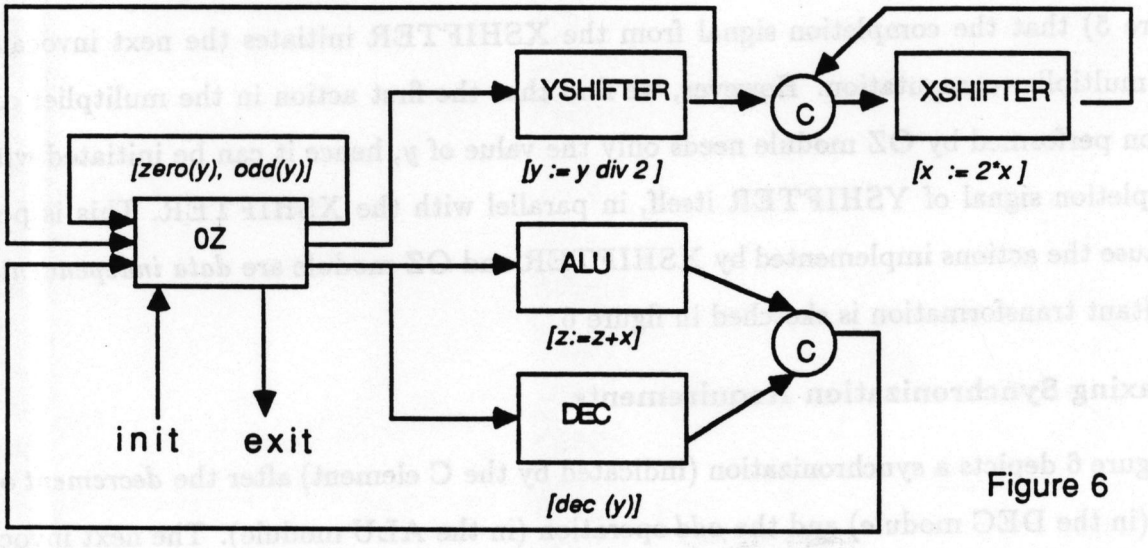


Figure 6

register to hold the result, one shifter SHIFTER to implement the left and right shift functions, four CALL modules to implement sequencing and one CBR module to implement the sharing of the OZ module

2.3 Performance Optimization of Asynchronous Circuits

Due to the absence of global clocking and emphasis on *local communication*, several optimizations can be performed to augment the *efficiency* of asynchronous circuits. We illustrate three such optimizations on the iterative multiplier example. Figure 4 is the flow diagram of the datapath of the circuit. It denotes the implementation of the paths $(s \rightarrow s6 \rightarrow s4 \rightarrow s)$ and $(s \rightarrow s6 \rightarrow s5 \rightarrow s)$ in the HFG shown in figure 3. The datapath modules in the flow diagram are annotated with the expression actions (from figure 3) that they implement. Analysis of this flow diagram reveals the optimizations discussed in the rest of this section. These optimizations are fairly general and are applicable to a wide class of asynchronous circuits.

Pipelining via Loop Unfolding

Pipelining is achieved by starting the $(i+1)$st iteration of a computation before the ith iteration is complete. Operations $x := 2 * x$ and $y := y \ div \ 2$ (from figure 3) are implemented using the *same* SHIFTER module which results in the loop around the SHIFTER module in figure 4. This loop is unfolded in figure 5 by replacing SHIFTER by XSHIFTER and YSHIFTER which implement $x := 2 * x$ and $y := y \ div \ 2$ respectively. We notice (from figure 5) that the completion signal from the XSHIFTER initiates the next invocation of the multiplier computation. However, we find that the first action in the mulitplier computation performed by OZ module needs only the value of y, hence it can be initiated with the completion signal of YSHIFTER itself, in parallel with the XSHIFTER. This is possible because the actions implemented by XSHIFTER and OZ module are *data independent.* The resultant transformation is sketched in figure 6.

Relaxing Synchronization Requirements

Figure 6 depicts a synchronization (indicated by the C element) after the *decrement* operation (in the DEC module) and the *add* operation (in the ALU module). The next invocation of the multiplier loop will not take place unless both these operations are completed. This

synchronization is a performance penalty because the absolute time taken by the *decrement* operation is significantly smaller than the time taken by the *add* operation (because *add* has to fetch two operands while *decrement* has to fetch only one operand). This penalty can be avoided by relaxing the synchronization requirement (or *delaying the synchronization*) without violating the data dependecies. One way of doing it in our example is to initiate the OZ action with the completion signal of the DEC module and synchronizing with the consumer of the value produced by the ALU module.

Speculative Evaluation of Conditionals

Finally, we will illustrate how we could employ *speculative evaluation* to optimize our circuits. Figure 7 shows the *conditional dataflow* in our multiplier example. Actions a_1 and a_2 denote the boolean *expression actions* (also known as guards), while a_3, a_4, a_5 denote the corresponding next actions depending on the values of the guards. Our refinement procedure would implement it as shown in figure 8, where guards a_1 and a_2 are evaluated sequentially. (a_2 is not evaluated if a_1 evaluates to true). Figure 9 shows the implementation which *speculatively* evaluates a_2 in parallel with a_1. The result of a_2 is discarded (using the CAL component) if a_1 is true. This optimization can be very useful in *asynchronous instruction pipelines*, where the actions a_1, a_2 denote conditional branches. Optimizations such as this are possible in the hopCP framework because of the referential transparency in the underlying functional language. Speculative evaluation becomes very expensive and complicated with imperative specification languages.

Implementation Details

We developed a compiler and a *functional* simulator for the hopCP specifications. The simulator is *generated* by translating HFGs into Concurrent ML source code, and executing it directly in the Standard ML environment. This enables us to simulate concurrency and detect errors like deadlock and synchronization failure during simulation. Work is underway to *automate* action-refinement and implement the asynchronous circuits in a FPGA (field-programmable gate array).

Illustrating Speculative Evaluation Optimization

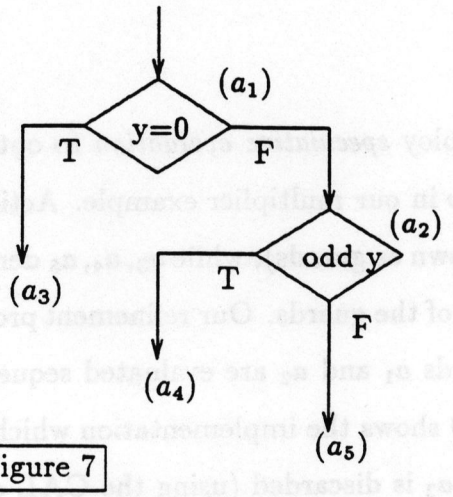


Figure 7

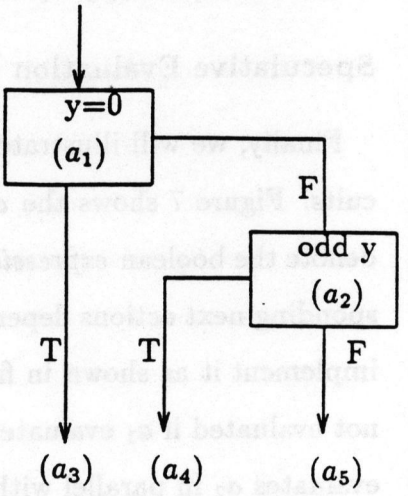


Figure 8

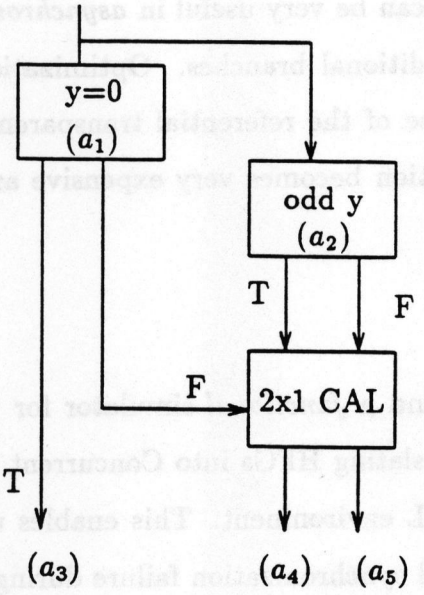


Figure 9

3 Main Contributions and Significance of the Work

There are two main contributions in this paper. Firstly we have shown how the ideas of conventional (synchronous) high-level synthesis work can be modified to synthesize *asynchronous* circuits. Secondly, we contribute to the area of *performance-directed* synthesis of asynchronous circuits. The only existing work in this area of which we are aware is that of Brunvand [3] but it only employs *peephole* optimizations to remove local redundancies. If asynchronous circuits are to become practical and popular, it is very important to address global optimization and performance issues in a specification-driven design environment. Our work can be interpreted as an initial effort in that direction.

References

1. Venkatesh Akella. *hopCP: Language Definition, Semantics and Examples.* Tech Report UUCS-91-XX, Dept. of Computer Science, University of Utah.

2. Venkatesh Akella and Ganesh Gopalakrishnan. *Hierarchical Action Refinement: A Methodology for Compiling Asynchronous Circuits from a Concurrent HDL.* Tenth International Symposium on Computer Hardware Description Languages and their Applications, Marseille, France, April 1991.

3. Erik Brunvand and Robert F. Sproull. *Translating Concurrent Communicating Programs into Delay-Insensitive Circuits.* In International Conference on Computer-aided Design, ICCAD 89, November 1989.

4. Ganesh Gopalakrishnan and Prabhat Jain. *Some Recent Asynchronous System Design Methodologies.* Technical Report UU-CS-TR-90-016, Dept. of Computer Science, University of Utah. *(Submitted to the ACM Computing Surveys).*

5. C. A. R. Hoare. *Communicating Sequential Processes.* Published by Prentice-Hall International Series in Computer Science, 1985.

6. Robert M. Keller. *Towards a Theory of Universal Speed-Independent Modules.* IEEE Transactions on Computers, C-23(1):21–33, January 1974.

7. Michael C. McFarland, Alice C. Parker, and Raul Camposano. *The High-Level Synthesis of Digital Systems.* In Proceedings of the IEEE, pages 301-317, February 1990.

8. Ivan Sutherland. Micropipelines. *Communications of the ACM*, June 1989. *The 1988 ACM Turing Award Lecture.*