HIGH LEVEL OPTIMIZATIONS IN COMPILING PROCESS DESCRIPTIONS TO ASYNCHRONOUS CIRCUITS

GANESH GOPALAKRISHNAN* (*ganesh@cs.utah.edu*)

University of Utah

Dept. of Computer Science

Salt Lake City, Utah 84112

VENKATESH AKELLA† (*akella@ece.ucdavis.edu*)

Department of Electrical and Computer Engineering,

University of California,

Davis, CA 95616

Keywords: Asynchronous/Self-timed Systems, High Level Synthesis, High level optimizations

Abstract. Asynchronous/Self-Timed designs are beginning to attract attention as promising means of dealing with the complexity of modern VLSI technology. In this paper, we present our views on why asynchronous systems matter. We then present details of our high level synthesis tool SHILPA that can automatically synthesize asynchronous circuits from descriptions in our concurrent programming language, hopCP. We outline many of the novel features of hopCP and also sketch how these constructs are compiled into asynchronous circuits, and then focus on the high level optimizations employed by SHILPA, including *concurrent guard evaluation* and *concurrent process decomposition.*

---

*Supported in part by NSF Award MIP 8902558

†The work reported here was part of this author's PhD dissertation work, and was supported by a University of Utah Graduate Fellowship

1 Introduction

It has been pointed out by many researchers recently that asynchronous circuits—circuits that do not employ global clocks—have a number of advantages over synchronous circuits when it comes to building large and complex sequential systems [1, 2, 3, 4]. In this paper, we summarize recent developments in asynchronous circuit design and then present our high-level synthesis system, SHILPA[1]. We will focus on the *high-level optimizations* used by SHILPA. High-level optimizations are similar to "flow-graph level optimizations" in programming language compilers [5]; they should not be confused with *circuit level optimizations* which are similar to machine code optimizations.

Synchronous *vs.* Asynchronous Circuits

Synchronous circuits are employed virtually everywhere. They have a number of desirable characteristics, some of which are the following. The clock period of a synchronous circuit is chosen to be long enough to allow its combinational stages to settle down, thereby preventing failures due to hazards. In asynchronous circuits, hazards can be mistaken for genuine signal transitions. Hence, it is of paramount importance to eliminate hazards, for instance by employing special purpose Boolean minimization procedures [6]. Synchronous circuits do not have the overhead of handshaking. Very many simulation and testing techniques, as well as Computer-Aided Design (CAD) tools, are available for them. Synchronous circuits also have many shortcomings. Large synchronous circuits employ *high frequency* and *low skew* global clocks, driving which can consume considerable amounts of power [7]. The design of synchronous/asynchronous interfaces—for example, peripheral interfaces—must be done with great care, for fear of inviting failure due to metastability [1].

There are many specific kinds of asynchronous circuits, some of which are: *self-timed* circuits (those that generate completion signals), *delay insensitive* circuits (whose behavior

[1]System for the High-level synthesis of Process to Asynchronous circuits

is invariant over module- and wire-delays), and *speed independent* circuits (whose behavior is invariant over module-delays, but not necessarily wire-delays). These distinctions depend largely on the *granularity* of the *circuit primitives*. For example, a synchronous system that communicates externally using handshake signals can be regarded as a self-timed component in a larger context. Asynchronous circuits are attractive in many ways. To a large extent, they allow one to focus on *functionality* and not on *timing details*. This makes the task of high-level synthesis centered around asynchronous circuits much easier in many respects. For example, there is no need to perform clock scheduling. Operations whose durations are data dependent as well as I/O dependent can be more cleanly and efficiently handled in the asynchronous high level synthesis framework. Asynchronous circuits can also exhibit better average case performance, unencumbered by clocking rules [8].

Despite these promises, many designers have have shunned away from asynchronous circuits. It is feared that asynchronous circuits are excessively larger than synchronous circuits. Asynchronous circuits offer the designer with even more freedom to explore the design space. The designer has the choice of numerous *concurrent algorithms* to begin with; upon each chosen algorithm, he can effect numerous high level optimizations; each lead to circuits to which different *circuit level optimizations* can be applied; finally, each specific circuit has its own "best suited" circuit design style; and, all these tasks are inter-related. For instance, if an addition operation is used in a thread whose *average execution* time should be kept low, carry-completion addition would be a viable alternative. This may, in turn suggest a DCVSL CMOS style implementation with its own associated transistor sizing rules. Without adequate design space exploration support tools, this added freedom offered by the asynchronous style can be a burden for the designer.

It is hoped that many of these limitations of asynchronous circuits can be overcome very soon through additional research. Many of the early failures involving asynchronous circuits can now be avoided through careful design [9] or verification [10]. Area overheads are be-coming less severe, especially if a slight increase in area can actually buy reduced design time. Recently, there have been many convincing demonstrations of the practicality of large asynchronous designs [11, 12]. Many high-level [11, 13, 14, 15] and low-level synthesis tools [16, 17, 18] have been developed.

The clean separation between "synchronous", and "asynchronous" systems has already begun to blur. Mixed synchronous/asynchronous circuits [19, 20], Q-modules [21], locally clocked asynchronous systems [18], and the "asynchronous style" synchronous control networks used in Olympus [22] are indicative of this trend. Whichever course the hardware design community may ultimately follow, it seems inevitable that asynchronous design will play an increasing role as time goes by. Based on this assumption, we are justified in taking the approach of studying asynchronous designs in isolation, in this paper.

Context and Motivation for our work

A prominent category of efforts in asynchronous design deals with compiling behavioral descriptions in high-level languages based on the *communicating sequential process* paradigm into asynchronous circuits. In these efforts, asynchronous design is viewed as *concurrent programming*, where the computation to be implemented is expressed in a high-level concurrent HDL. This approach is more suitable for system level synthesis. This is in contrast to the works of [23, 24], as well as more recent works of [25, 10, 9, 26], which are more suited for *low level* synthesis and verification of *asynchronous state machines.*

Our system, SHILPA, belongs to the former category. To the best of our knowledge, systems similar to ours that have been fully implemented and tried out in practice are those by Brunvand [14], van Berkel [27, 13], and by Martin and Burns [11, 28]. Improvements in SHILPA over these works are primarily the following. hopCP, the source language for SHILPA, is more expressive than Martin's input language 'CHP', Brunvand's version of 'Occam', or van Berkel's language 'Tangram'. We use a class of annotated Petri net-like *flow graphs* (called hopCP flow graphs, or HFGs) as our intermediate form. This intermediate form is very amenable to flow-analysis. Optimizations for resource sharing can be easily carried out on HFGs. Life-time analysis for variable reuse is also easy to carry out on HFGs. A tool called *Concur* [29], that can determine if two actions are serially ordered or not, could be developed fairly easily, thanks to the HFG based notation. *Concur* is central to many of the optimizations performed by SHILPA. The HFG based intermediate representation also helps in smoothly integrating all the SHILPA tools (a compiled code simulator, *Concur*, and, in future, performance evaluation tools). SHILPA compiles circuits by taking each action in the HFG and rewriting it to a *normal form HFG* (NHFG) fragment (to be explained later) as well as the associated resources; this *graph rewriting* based compilation keeps the SHILPA compiler modular, easier to understand, and (it is hoped) easy to verify (in future). Flow analysis based optimizations, a common intermediate form for a variety of asynchronous design tools, and compilation through graph rewriting have not been addressed before in asynchronous high level synthesis.

There are two classes of approaches for realizing asynchronous circuits in hardware: *Boolean gate* based, and *macromodule* based. Asynchronous macromodules implement functions such as rendezvous, arbitration, procedure call and return, and control merging. Many approaches using macromodules view the given design problem as a *concurrent programming problem*—more specifically, one of mapping a given concurrent program into an interconnection of macromodules. There are also many efforts in which macromodules are used directly for realizing state machines (*i.e.* for low level synthesis). Some examples are [25, 9]. Some of these distinctions are also rapidly blurring, with the use of *complex gates* that directly realize multi-input multi-output Boolean functions as macromodules. In SHILPA, macromodules are the target of compilation, at present. Our set of macromodules were originally developed by Brunvand [30] using the Actel field programmable gate arrays (FPGAs); we have made numerous extensions to this cell set.

Organization

In Section 2, we briefly sketch the syntax and semantics of hopCP. In Section 3, we illustrate SHILPA on a two-stage pipeline. In Section 4, we examine concurrent guard evaluation in some detail. In Section 5, we present an example of *parallel decomposition*, a useful technique for obtaining pipelined designs. Concluding remarks are provided in Section 6.

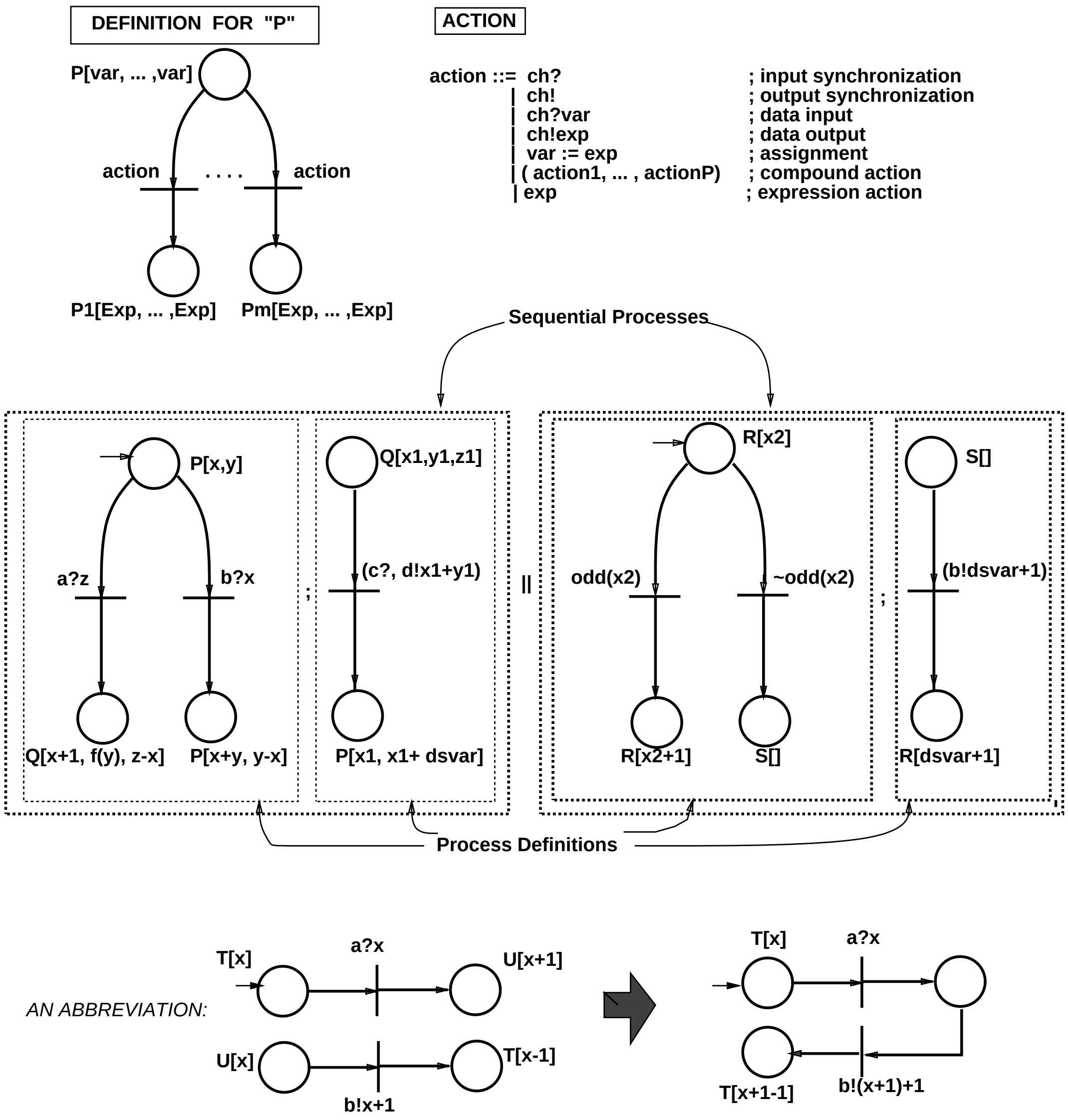
2 hopCP System Overview

Syntax

A hopCP description consists of one or more *sequential processes* composed in parallel (using the || operator). Two sequential processes are shown in Figure 1. through HFGs as well as using a textual notation. A sequential process is one or more *process definitions* composed in series (using the ; operator), such that for every *process call*, there is a corresponding process definition. Each sequential process shown in Figure 1 consists of two process definitions each. The processes defined are P, Q, R, and S.

A process definition consists of a *choice node* annotated with a *process name* and a list of *formal parameters*. Arcs lead off from the choice node (a "circle") to one or more *alternative transitions* that are annotated with *actions*. These actions are commonly known as *guards*. Arcs lead off from the guards to nodes that perform *process call*s.

The left-most process definition defines process P that has two formal parameters x and y. (We will use the words *process* and *state* synonymously.) The guards of P are a?z and b?x. These actions belong to the category data input. These transitions are, in turn, followed by the process calls Q[x+1, f(y), z-x] and P[x+y, y-x]. Note that every process call has a corresponding process definition in the same sequential process. When a *process call* is made, the actual parameters are passed *by value*.




```
   (  P[x,y]       =   a?z -> Q[x+1, f(y), z-x]
                     | b?x -> P[x+y, y-x]
   ;  Q[x1,y1,z1] =   (c?, d!(x1+y1)) -> P[x1, x1+dsvar]
   )
|| (  R[x2]        =   odd(x2)  -> R[x2+1]
                     | ~odd(x2) -> S[]
   ;  S[]          =   b!(dsvar+1) -> R[dsvar+1]
   )
```

Figure 1: Overview of hopCP

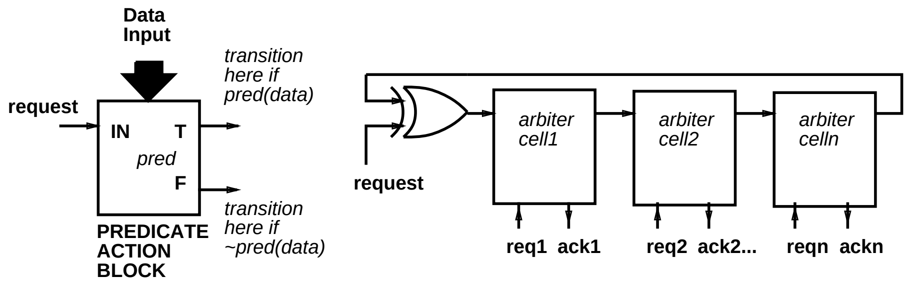


Figure 2: Predicate Action Block and Ring-style Arbiter

The guard of process Q is the *compound action* (c?, d!(x1+y1)). A compound action can appear as a guard if it is the only guard of a choice node. (Further restrictions on hopCP's guards are noted later.) This guard requires the input synchronization action c? and the data output d!(x1+y1) to be both finished before the process call to P is made. All the constituent actions of a compound action must be *disjoint*, *i.e.*, must not share channels, registers, or other resources, so that they may run in parallel without interference. Compound actions are useful for specifying a collection of primitive actions to be done in parallel. They are also very useful for specifying the compilation rules of SHILPA which break up high level actions into collections of simpler actions that can be done concurrently.

The guards of process R are the expression actions odd(x2) and ∼odd(x2). These form *Boolean guards* that decide where control passes from state R[x2]. We encourage designers to specify Boolean guards in a mutually exclusive manner using the form formula and ∼formula, as this situation arises very frequently. It compiles such guards using *predicate action blocks* (Figure 2). A predicate action block evaluates *pred*(*data*) and steers the request transition to either the T (if *pred(data)*) or the F (if ∼*pred(data)*) output. If SHILPA does not find the pattern "formula" and "∼formula," it assumes that the Boolean guards are not mutually exclusive, and uses an arbiter to select one of the true Boolean guards (Figure 2). We use the ring-style arbiter from [30] which functions (roughly) as follows: after a request is applied, a token is circulated within the arbiter; more than one reqi input may be asserted at any time; one of these requests is acknowledged. In general, arbiters occupy more area to realize than predicate action blocks. They also use circuits such as the *interlock* [2] that cannot be realized in many technologies, such as most of today's FPGAs. (Note: The FPGA realization in [30] is only an approximation, to permit rapid prototyping.)

The different categories of variables, channel names, and their scoping rules, are as follows. Variables can either be local to a process definition (*e.g.*, x,y are local to P), or declared to be globals (in this example, dsvar has been declared as a *global variable*). Variables used in data input actions (*e.g.*, z) are local to the process definition in which they appear; their scope begins at the data input action and lasts till the ensuing process call. Channel names are local to a sequential process. Global variables can be shared across process definitions as well as sequential processes. Other comparable description languages disallow sharing global variables across parallel threads for a good reason: they have no tool support to determine if global variable accesses can be potentially concurrent. In hopCP, we allow such shared variables because (a) it has been our common observation that many real world systems frequently communicate over shared registers (or busses); (b) procedure *Concur* can determine whether two actions in an HFG are serially ordered or potentially concurrent. Using *Concur*, all accesses to global variables can be checked and made sure that they are serial. (Notes: The serial ordering itself is imposed by the *synchronization*s between the sequential processes.)

Algorithm *Concur* works as follows. When invoked with two actions a and b as arguments, *Concur* first *composes* the sequential processes into one HFG by merging transitions that can rendezvous. Then it performs a reduction of the HFG by removing *places* and *transitions* in such a way that the causal orderings between a and b are unaffected. Then, *Concur* performs reachability analysis on the reduced HFG, to determine all the reachable markings. It then checks whether there exists a marking y such that the union of the preconditions of a and b is a subset of y, but the intersection of the preconditions of a and b is empty; if so, actions a and b are potentially concurrent; if not, these actions are serial. *Concur* assumes that all Boolean guards are true. Therefore, although it can tell that only one guard will be picked, it cannot tell which one will be picked. Hence, its results are pessimistic. Despite this caveat, in practice, we find that it is relatively easy to determine whether two actions are serially ordered or not. Though its worst case complexity is exponential in the HFG size, *Concur* has performed reasonably fast on many practical examples.

A commonly used notational abbreviation is as follows: if a process definition has only one reference (*i.e.*, only one process calls the process being defined), then it is possible to in-line substitute the process definition in place of the process call. This abbreviation is illustrated on an example consisting of two process definitions for T and U that are mutually recursive. Here, process U has exactly one reference, while process T has two references, because it is also the initial state. We can eliminate an explicit definition for process U. The textual syntax for this abbreviated definition would be (after simplifications):

```
T[x] = a?x -> b!x+2 -> T[x]
```

Informal Execution Semantics

The informal execution semantics of the example in Figure 1 are as follows (the formal semantics of hopCP are given in [31]). Suppose the execution is begun at P and R. These processes begin their execution concurrently. Process P first makes a choice between the guards a?z and b?x. This alternative ("choice") command has the same meaning as in CSP-like languages. For example, if action a?z is to take place, a matching action of the form a!exp must also be enabled in another sequential process; in this case, a?z and a!exp are said to *rendezvous*, whereupon the value of exp gets bound to z. In our example, the data input action b?x of P is matched by the data output action b!(dsvar+1) of S; a matching action for a?z is not shown. Input and output synchronization actions are value-less counterparts of data input and data output, respectively.

In hopCP, data output follows the *multicast* semantics: a data output action such as b!(dsvar+1) can rendezvous with more than one data input action that uses the same channel. For example, suppose that three concurrent processes P_1, P_2 and P_3 attain a state in which P_1 offers action b!(dsvar+1), while P_2 and P_3 offer b?v and b?x, respectively. According to the multicast semantics, P_2 can proceed as soon as b!(dsvar+1) is offered by P_1; likewise, P_3 can proceed as soon as b!(dsvar+1) is offered by P_1. However, P_1 can proceed only after both b?x and b?v have been offered by P_2 and P_3, respectively.

Valueless communication actions in hopCP follow the *barrier synchronization* semantics: an output synchronization action such as e! can synchronize with more than one e? action in as many sequential processes. In this case, all the actions e? as well as the single e! action must wait for each other and proceed only after they all have been enabled.

A designer may use barrier synchronization when "time alignment" is called for. Since, in hopCP, interactions between concurrent threads can occur through value assignments on global variables (as noted earlier) or through rendezvous, it makes a semantic difference whether barrier synchronization is followed or multicast. In addition, the effect of multicast can be obtained even for valueless communications, by suitably "faking" a value communication (for example, following the syntax e!nullvalue and e?ignore).

The availability of barrier synchronization as well as multicast offers considerable flexibility in specifying system level behavior, as we have shown through numerous large examples, notably the specification of the high level protocols obeyed by Intel 8251 USART [32]. These constructs are also useful for specifying concurrent algorithms [33]. These features of hopCP are absent from comparable languages that are used for asynchronous high level synthesis.

Coming back to process P, consider a situation in which the communication actions a?z and b?x can arrive potentially concurrently. In this situation, an arbiter would be used to pick one of these communications (for example, as in [14]). However, *if* it can be determined (using *Concur*) that these actions are *mutually exclusive*, SHILPA compiles a circuit using the *concurrent guard evaluation* technique. This technique also uses a circuit that is smaller and easier to realize than an arbiter. This is one of the high level optimizations to be discussed later.

Coming back to the guards of P, if action b?x is chosen, the existing value of x is overwritten during the data input action b?x. Then, control goes back to P through a process call P[x+y, y-x], when the current value of x gets replaced by the value of x+y and the value of y by the value of y-x. Note that this particular process call P[...] cannot have used variable z in its actual parameter expressions because z is visible only in the scope of action a?z.

If guard a?z of P is chosen for execution, control reaches process Q. In the process, formal parameters x1,y1,z1 are bound to the values of expressions x+1, f(y), z-x, respectively. Process Q performs a *compound action*; *i.e.*, it waits for the input synchronization c? and the data output d!(x1+y1) to both finish before it engages in the process call P[x1, x1+dsvar]. While the value of the global variable dsvar is being acquired during the computation of expression x1+dsvar, dsvar must not be concurrently changed by another process. We can determine whether this is the case, using *Concur*. The execution semantics of processes R and S are similar. Process R involves Boolean guards that are mutually exclusive. If control passes to S, it performs the data output which can synchronize with action b?x of process P.

Notice the common subexpressions dsvar+1 in process S. Currently SHILPA cannot avoid recomputing dsvar+1; however, this optimization can be incorporated in a straightforward way, as done in standard compilers. However, SHILPA can be made to do *resource sharing*: for example, since the two uses of '+' are in the same process definition, the designer can request SHILPA to share the adder, if he/she so desires. The two invocations of *add* used in process definitions Q and S can be shared only if they are guaranteed to occur serially. Again, *Concur* can be used to determine if these two usages are always serial or not.

Restrictions on Guards

Guards in hopCP have to obey a number of restrictions. These restrictions help in many ways: they help avoid potentially dangerous situations (*e.g.* deadlocks). They also help in obtaining efficient circuits without compromising the expressive power too much. Some of the restrictions on guards are now listed. If a compound action is used as one of the guards, it must be the only guard going out of the choice node. Similarly, if a data output action is one of the guards, it must be the only guard going out of the choice node. Also, if an assignment action is one of the guards, it must be the only guard going out of the choice node. Two guards must not use the same input channel. All input channels used in guards must be point-to-point: in other words, broadcast or multicast channels should not be used in guards. The guards associated with a choice node may consist of expression actions, data input actions, and input synchronization actions. During execution, however, all the expression actions are examined before any of the non-expression actions within guards are examined.

Summary of Features

To sum up, our work makes a number of advances over comparable works. hopCP has been designed for supporting the specification of large hardware systems at a high level. It is more expressive than the HDLs used in comparable works. Although Martin [11] also makes the distinction between mutually exclusive and non-exclusive "guards", his approach is slightly different. In Martin's approach, an input guard is turned into a *input probe* which is then made part of the Boolean guard. We do not use *probe*s in hopCP for several reasons. First, we believe that not having *probe*s keeps the HDL simple. Second, many of the proposed uses of *probe*s can be replaced by corresponding uses of *global variable*s. The synthesis systems developed by Martin, van Berkel, or Brunvand do not support flow analysis or sharing analysis. Last, but not the least, we have built an integrated design system that includes a flow analyzer, an efficient compiled code simulator, and a high level synthesis system. It generates circuits ready for implementation in *Actel* FPGAs, supported by *Viewlogic* tools.

3 Overview of SHILPA

SHILPA generates transition style circuits using bundled data, as presented in [1]. We illustrate SHILPA through the design of a two-stage pipeline:

```
(P[x] = a?y -> b!(x+y) -> P[y])
||
(Q[z] = b?z -> c!z -> Q[z])
```

The top-level command in SHILPA for compiling this specification is function `compile`. This function first turns the textual description of hopCP into HFGs. The HFG for the pipeline is shown below in textual form:

```
      HFG for Process P                      HFG for Process Q
1   precondition  :{s__3[x,y]}         1   precondition  :{s__5[z,z]}
    actions       :(b!(x + y))             actions       :(c!z)
    postcondition :{P[y]}                  postcondition :{Q[z]}

2   precondition  :{P[x]}              2   precondition  :{Q[z]}
    actions       :(a?y)                   actions       :(b?z)
    postcondition :{s__3[x,y]}             postcondition :{s__5[z,z]}
```

Each action in the HFG is then refined into simpler actions which consist of *signal transitions* on *allocated resources*. This results in NHFGs, which were introduced in Section 1. For process `Q`, the NHFG is as follows:

```
1   precondition  :{START[ ]}   4   precondition  :{s__5[ ]}    7   precondition  :{s__42[ ]}
    actions       :(start??)        actions       :(c_out!!)        actions       :(REG_14_ld!!)
    postcondition :{Q[ ]}           postcondition :{s__37[ ]}       postcondition :{s__43[ ]}

2   precondition  :{s__37[ ]}   5   precondition  :{s__41[ ]}
    actions       :(c_in??)         actions       :(C_12_out??)
    postcondition :{Q[ ]}           postcondition :{s__42[ ]}

3   precondition  :{Q[ ]}       6   precondition  :{s__43[ ]}
    actions       :(C_12_in1!!)     actions       :(REG_14_ldack??)
    postcondition :{s__41[ ]}       postcondition :{s__5[ ]}
```

Actions that end with two question marks are input signal transitions that are awaited. Actions that end with two exclamation marks are output signal transitions that are generated. For example, `C_12_in1!!` is a signal transition generated on input `in1` of C-ELEMENT number 12. Notice that the allocated resource instances for process `Q` include one C-ELEMENT and one register. SHILPA can explain why each resource is being allocated, in the following form:

```
C_12:data assert for b!(x + y)          REG_5:argument for AB_4_arg1
C_13:data query for a?y                 REG_6:argument for AB_4_arg2
REG_3: datapath for x                   FAB_4:2 for (x + y)
REG_14:query var for z                  REG_7:result for 4
REG_10:const for y                      CTREE_8:2 for AB_4_arg2 <- y
```

One example from this printout, `FAB_4:2 for (x + y)`, explains that function action block (`FAB`) number 4, of arity 2 has been allocated to support `(x+y)`. Control circuitry is now generated in SHILPA by detecting *shared resources*—resources that are triggered from two different places. In the pipeline, there are no shared resources. Next, excess registers are eliminated based on user's interactive commands. For example, users may like to retain *result* registers to function blocks, so as to share the results of evaluating common subexpressions. Retaining registers can also help pipeline the evaluation of nested expressions (*e.g.* $(x+(y+z)+w)$).

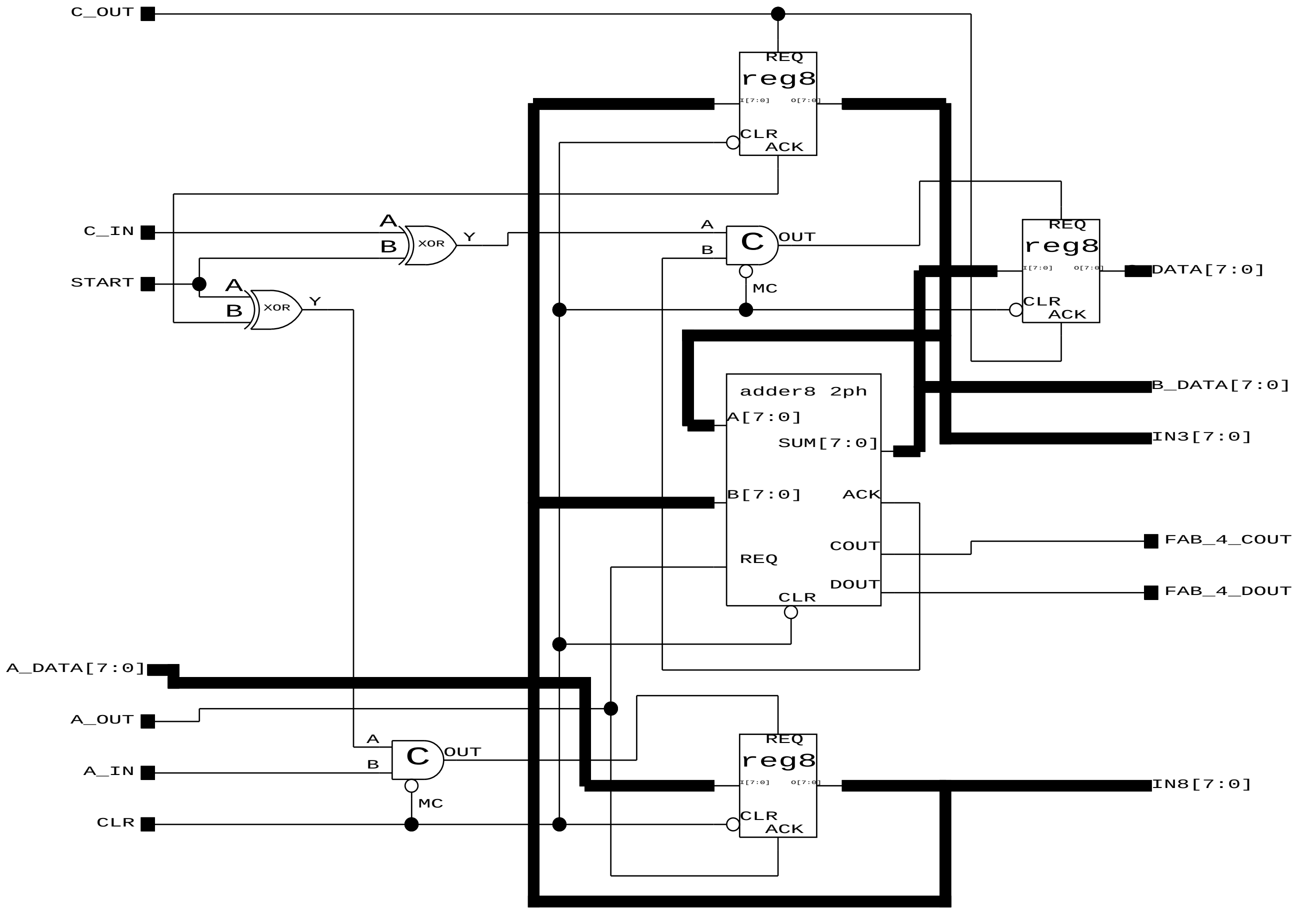


Figure 3: A Two Stage Pipeline

Sometimes, result registers have to be retained to prevent combinational loops from forming. For instance, if value of the actual parameter expression in process `Q` is `z+1`, then the result of `z+1` is held in a result register and then only loaded back to `z`. Detection of these situations is straightforward (though not automated at present).

After eliminating the desired number of registers, an *abstract netlist* can be generated:

```
connect(XOR_19_out,C_12_in1)    connect(REG_7_ld,FAB_4_+ack)    connect(REG_10_out,REG_3_in)
connect(XOR_18_out,C_13_in1)    connect(C_12_out,REG_14_ld)     connect(REG_14_in,b_data)
connect(REG_10_ldack,FAB_4_+)   connect(start,XOR_19_in2)       connect(REG_7_ldack,C_12_in2)
connect(REG_7_in,FAB_4_out)     connect(REG_3_ldack,XOR_18_in2) connect(REG_3_out,FAB_4_in1)
connect(REG_10_in,a_data)       connect(REG_10_out,FAB_4_in2)   connect(start,XOR_18_in1)
connect(REG_14_out,c_data)      connect(REG_7_out,b_data)       connect(REG_10_ldack,a_out)
connect(REG_14_ldack,REG_3_ld)  connect(C_13_in2,a_in)          connect(C_13_out,REG_10_ld)
connect(REG_14_ldack,c_out)     connect(c_in,XOR_19_in1)
```

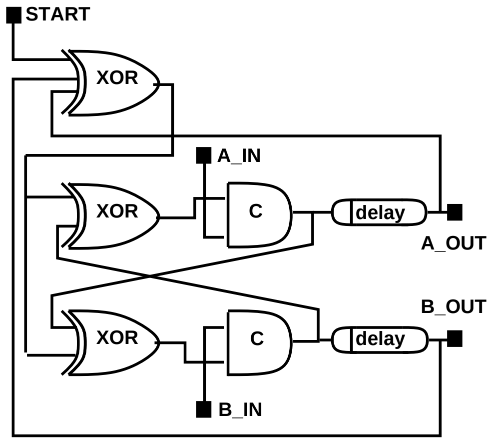


Figure 4: Circuit for the Concurrent Evaluation of Mutex Guards

SHILPA checks whether this netlist is structurally well-formed (for example, whether two outputs are connected together, *etc.*); this check is redundant (but quite re-assuring). Then it technology maps the netlist, currently to Actel FPGAs.

The resulting circuit is shown in Figure 3. The circuit works as follows. First CLR is lowered to reset the components. Then START is applied. This "arms" both the C-ELEMENTs, which then await A_IN as well as B_IN. When A_IN comes from the external world, the lower C-element fires. It causes the lower reg8 module (variable y) to store the data coming through the A_DATA port. The acknowledge signal of this register is forked to A_OUT as well as starts the addition of x (held in the upper reg8 module) and y. Completion of this addition triggers the upper C-element, thus finishing the synchronization involved with b?z. This causes the rightmost reg8 (variable z) to be loaded, thus finishing the data acquisition part of b?z. Finally, C_OUT is generated, register x gets loaded with the value of y, and process P is resumed. Process Q is resumed by the arrival of C_IN.

4 Concurrent Guard Evaluation

We shall illustrate concurrent guard evaluation through process P given below:

```
P[] =  a? -> P[] | b? -> P[]
```

Assume that $Concur$ has determined that `a?` and `b?` are mutually exclusive. SHILPA then generates the circuit shown in Figure 4. Here, A_IN and A_OUT are the handshake lines for channel `a?` while B_IN and B_OUT are for `b?`. The circuit is started by applying a transition on START. This, in turn, puts transitions (through the XORs) into the inputs of both C-ELEMENTs. Depending on whether an A_IN or a B_IN transition comes, that C-ELEMENT fires. For example, if A_IN comes, the upper C-ELEMENT fires. It first subjects the lower XOR to another transition, which results in the bottom input of the lower C-ELEMENT seeing two successive transitions. This C-ELEMENT is therefore reset. Following this (the delay ensures this), control is returned back to the top XOR.

The circuit comprising the lower two XORs, the two Cs and the two delays actually forms a 2×1 CAL component [9] (first introduced by Molnar). An $M \times 1$ CAL component can be considered to be a *generalized* C-ELEMENT. It has inputs $a_1, \ldots, a_M$ and b, and outputs $c_1, \ldots, c_M$. In any cycle of operation, a transition is received on exactly one of the a_i inputs and on b; the $M \times 1$ CAL then produces a transition on c_i. The circuit in Figure 4, with the CAL sub-circuit treated as a primitive (a "black-box"), belongs to the family of delay insensitive circuits.

A natural question at this stage is why we synthesize a CAL component each time, and not use a library primitive for a CAL component. The answer is given by the following example:

```
P[] =  a? -> b? -> P[] | b? -> a? -> P[]
```

In this example, after engaging in an `a?` action, `P` engages in a `b?` action (and after a `b?`, it does an `a?`). There are two invocations of the synchronous input action on channel `a?` (and likewise on `b?`). The usual semantics of channels requires that these invocations share the same resources (C-ELEMENTs, and handshake wires, in our case). Usually this is achieved by using a CALL module [1].

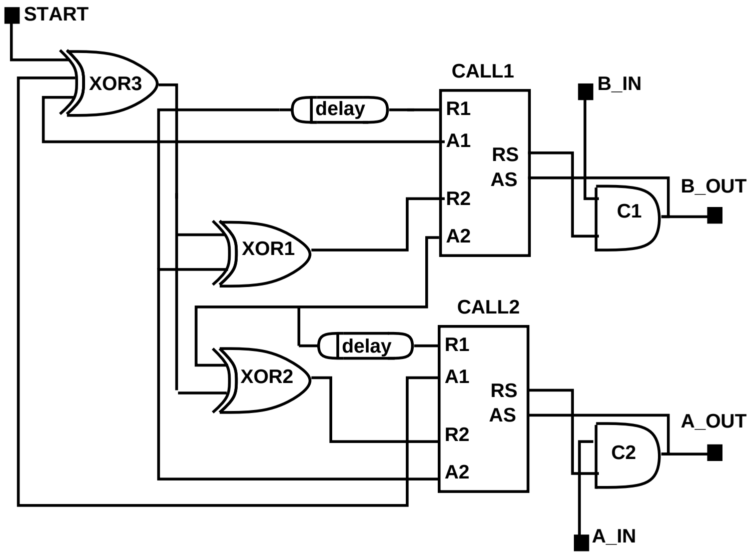


Figure 5: Mutually Exclusive Guards with Sharing

Coming back to our example, assuming that the guards are mutually exclusive, SHILPA generates the circuit shown in Figure 5. The circuit works as follows. After CLR, when START is applied, both the CALL elements make a "procedure call" onto the C-ELEMENTs through the respective R2 inputs. Suppose A_IN happens first; then C2 fires. It generates A_OUT and also returns the "call" through AS and A2 of CALL2. This transition first triggers XOR1 through its lower input. This causes another R2 on CALL1.

We have selected the CALL implementation of [30] in which the sequence R1; R1 causes a sequence RS; RS (and likewise R2; R2 also causes RS; RS), and in addition, the CALL element is reset at the end of this sequence. Therefore, CALL1 is reset by the two R2 transitions it sees. Since the R2; R2 sequence causes an RS; RS sequence at the output of CALL1, C-ELEMENT C1 also gets reset!

The value of delay must be large enough to make sure that this resetting happens before a call is made through the A1 input of CALL1.

Though the generated circuit is large, the situation shown is fortunately rare. Thus, in most instances, we will have to generate a circuit similar to that in Figure 4; in those circumstances, we can indeed use a library CAL component.

The purpose of presenting this example was to demonstrate (a) how non-obvious the interactions between features—for example concurrent guard evaluation and sharing—can be; (b) to show that by imposing one-sided delay constraints, often "clever" designs can be obtained. As discussed in Section 2, Martin's implementation of mutually exclusive guards uses *probe*s and hence may avoid some of the difficulties we are facing. However, an exact comparison is not possible between our approach and Martin's approach, because we synthesize two-phase transition style circuits (which have a large number of desirable characteristics [1]) while Martin synthesizes four-phase (level based) circuits.

5 Concurrent Process Decomposition

It is easy to come up with iterative specifications for many computations. We have identified a useful heuristic for implementing iterative computations through *concurrent process decomposition.* Concurrent process decomposition is a very convenient way to achieve *software pipelining.* To make this clear, consider the iterative specification of a multiplier:

```
MULT[x, y, z] =   (isZero y) -> result!z -> MULT[x, y, z]
                | (not (isZero y))
                    ->  (odd y)        -> MULT[x, (y-1), (z+x)]
                      | (not (odd y)) -> MULT[(lshift x),(rshift y),z]
```

Notice that the value of the actual parameter `z+x` is not needed until the corresponding formal parameter, `z`, is used in the body of `MULTF`. Also notice that `z` is used *only* in certain threads; while `(not (odd y))` is true, this updated value of `z` is not needed! (The situation `(not (odd y))` being true for many iterations can happen if the number being multiplied has a string of 0s in it.) Thus, holding up the recursive invocation of `MULT` till `(z+x)` has finished computing can be wasteful in time.

A modified `MULT` algorithm can take advantage of this situation. Expressing such algorithmic modifications in traditional sequential HDLs (*e.g.*, VHDL) can be tricky. Fortunately, a CSP-style language is very expressive in this regard because, using rendezvous style communications, the desired interactions between various threads of computation can be conveniently specified. Note that *concurrent decomposition*, as we propose here, is different from Martin's *process decomposition*, which essentially only gives the ability to call a subroutine and return to the place of call—and not spawn two concurrent threads as we do.

The modified specification is as follows:

```
   MULTPIPE [x, y] =    (isZero y) -> sz! -> MULTPIPE [x, y]
                      | (not (isZero y))
                          ->   (odd y)       -> azx!x -> MULTPIPE [x, (y-1)]
                             | (not (odd y)) -> MULTPIPE [(lshift x),(rshift y)]
||
   PZ[z]           =    sz?    -> result!z -> PZ[z]
                      | azx?x1 -> PZ[x1+z]
```

We first factor out variable `z` from `MULT`, and make it a local variable of a new process `PZ`. `PZ`'s role is to treat variable `z` as an abstract data type object, allowing it to be accessed only through two operations: operation `sz` that stands for "send `z`", and operation `azx` that stands for "add `z` to `x`". These operations can be conveniently implemented through rendezvous communications `sz?` and `azx?x` respectively. Notice that the second rendezvous communication involves data `x` that is sent from `MULT` to `PZ`. The operation of `MULTPIPE` is as follows. If `(not (isZero y))` and `(odd y)`, it sends `x` to `PZ` and *immediately goes back* to state `MULTPIPE`. While `(not (isZero y))` and `(not (odd y))`, it ignores `PZ`, allowing it to complete the previous add operation, if any. When `z` is needed inside `MULTPIPE` (occurs when `(isZero y)` is true), `MULTPIPE` orders `PZ` to send the value of `z` through the channel `result`. This process transformation achieves the effect of software pipelining. We show the results of compiling process `PZ` in Figure 6. We do not show the rest of the circuit to conserve space. Process `PZ` has been compiled to take advantage of concurrent guard evaluation because it is clear (as was checked using *Concur* also) that the guards `sz?` and `azx?x1` are mutually exclusive. Process `PZ` functions as follows. Upon receiving `START`, a

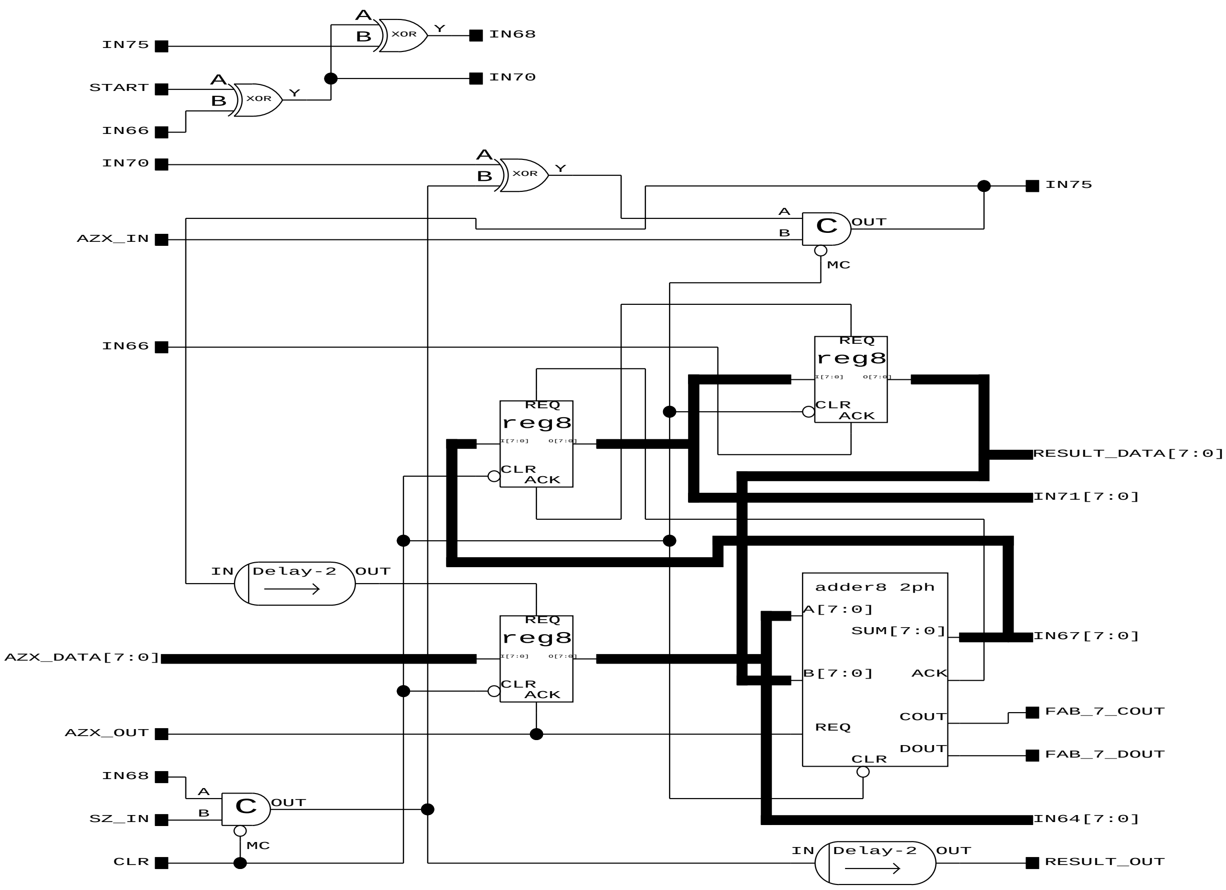


Figure 6: Process PZ of the Multiplier

transition is injected into IN68 and IN70. These, in turn, arm the two C-elements to look for request transitions on SZ_IN and AZX_IN. If AZX_IN is triggered, the bottom C-element is first reset; then, AZX_DATA is loaded into the bottom reg8 module. Its acknowledgement starts the addition and also generates AZX_OUT. When the addition finishes, the results of addition are first loaded into the result register (top-left reg8), and then transferred into register z (top-right reg8) before restarting process PZ.

We are currently studying the process of semi-automating concurrent process decomposition in SHILPA [34]. Until we have tool support for concurrent process decomposition, we believe that this technique can still be applied manually without too much trouble.

	Circuit	Number of View*logic* Modules	
		SHILPA	VHDLDes
1	Micropipeline with processing (2 Stages)	418	822
2	Iterative Multiplier (Non-Pipelined)	1176	1353
3	GCD (Euclid's Algorithm)	895	999

Figure 7: Comparative Study of Circuits Produced by SHILPA and *VHDLdes*

6 Concluding Remarks

In this paper, we have tried to demonstrate that asynchronous VLSI design can be greatly facilitated by designing a high level synthesis system that uses an expressive HDL and incorporates numerous optimizations. We have detailed such a system in this paper. It is well known that writing and debugging concurrent programs is difficult without tool support. The hopCP notation avoids many of the possible pitfalls in writing concurrent (HDL) programs by offering many high level descriptive mechanisms. To facilitate design debugging, the hopCP system offers CFSIM, a compiled code functional simulator, and *Concur*, a flow analyzer. hopCP also allows low level hardware features, such as global variables, to be used in hardware descriptions, and such usages checked for safety using *Concur*. Last, but not the least, SHILPA tries to keep the designer fully informed about its actions, and allows the designer to influence the final circuit in many ways, through many interactive commands.

Unit-delay simulation of the pipelined multiplier showed that despite its pipelined nature, it will run slower than its non-pipelined counterpart! The reason is that the '+' operation finishes too soon, thereby not allowing the '+' to overlap in any significant way with other operations. However, with other examples, we have actually observed significant speed-ups due to pipelining. As is clear from these examples, the main problem we are facing currently is in *performance estimation*. In our experience, a high level synthesis framework for asynchronous circuits offers the designer with even more freedom to explore the design space. Tool support for conducting design space exploration in this manner is sorely missed in SHILPA; but that is exactly what we will begin working on, next. At present, we have synthesized many small circuits using the SHILPA system. Sizes of SHILPA generated circuits seem to compare favorably with the results produced by one VHDL synthesis system that generates synchronous circuits (the VHDLDesigner tool of the *Viewlogic* family was fed VHDL descriptions obtained through hand-translation of hopCP descriptions), Figure 7. These results, though by no means definitive, are at least reassuring.

Acknowledgements: SHILPA was developed as part of the PhD dissertation work of the second author, partly supported by a University of Utah Fellowship. The first author was supported through NSF award MIP 8902558.

References

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

2. C. A. Mead and L. Conway. *An Introduction to VLSI Systems.* Addison Wesley, 1980. *Chapter 7, entitled "System Timing".*

3. John Brzozowski and Carl-Johan Seger. Advances in Asynchronous Circuit Theory: Part I: Gate and Unbounded Intertial Delay Models; and Part II: Bounded Intertial Delay Models, MOS Circuits, Design Techniques. Technical report, University of Waterloo, 1990.

4. Ganesh Gopalakrishnan and Prabhat Jain. Some recent asynchronous system design methodologies. Technical Report UUCS-TR-90-016, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1990.

5. Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers, Principles Techniques and Tools.*

Addison-Wesley, 1986.

6. Steven Nowick and David Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 626–630, November 1992.

7. D. W. Dobberpuhl. A 200-MHz 64-b Dual-Issue CMOS Microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1567, November 1992.

8. Ted E.Williams and Mark Horowitz. A zero-overhead self-timed 160ns 54bit cmos divider. *IEEE Journal of Solid State Circuits*, 26(11):1651–1661, November 1991.

9. Jo C. Ebergen. *Translating Programs into Delay Insensitive Circuits*. Centre for Mathematics and Computer Science, Amsterdam, 1989. *CWI Tract 56*.

10. David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, 1989. *An ACM Distinguished Dissertation*.

11. Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, UT Year of Programming Institute on Concurrent Programming. Addison-Wesley, 1989.

12. Erik L. Brunvand. The NSR Processor. In T.N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1*, pages 428–436, January 1993.

13. Kees van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993. International Series on Parallel Computation.

14. Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

15. Venkatesh Akella and Ganesh Gopalakrishnan. SHILPA: A High-Level Synthesis System for Self-Timed Circuits. In *International Conference on Computer-aided Design, ICCAD 92*, pages 587–591, November 1992.

16. Al Davis, Bill Coates, and Ken Stevens. The Post Office Experience: Designing a Large Asynchronous Chip. In T.N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1*, pages 409–418, January 1993.

17. Tam-Anh Chu. Synthesis of hazard-free control circuits from asynchronous finite state machine specifications. *TAU '92: 1992 Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, Princeton, NJ, March 18–20, 1992.*

18. Steven M. Nowick, Kenneth Y. Yun, and David L. Dill. Practical Asynchronous Controller Design. In *Proceedings of the International Conference on Computer Design*, pages 341–345, October 1992.

19. Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Department of Computer Science, Stanford University, October 1984.

20. M.J.Stucki and J.R.Cox, Jr. Synchronization strategies. In *Proceedings of the Caltech Conference on VLSI*, pages 375–393, January 1979.

21. Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pein Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37(9):1005–1018, September 1988.

22. David Ku. *Constrained Synthesis and Optimization of Digital Integrated Circuits from Behavioral Specifications.* PhD thesis, Department of Computer Science, Stanford University, June 1991.

23. Stephen H. Unger. *Asynchronous Sequential Switching Circuits.* John-Wiley, 1969.

24. Arthur D. Friedman. *Fundamentals of Logic Design and Switching Theory.* Computer Science Press, 1986.

25. Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, (1):197–204, 1986.

26. Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.

27. C. van Berkel, C. Niessen, M.Rem, and R.Saeijs. Vlsi programming and silicon compilation: a novel approach from phillips research. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, 1988.

28. Steven M. Burns and Alain J. Martin. Synthesis of self-timed circuits by program transformation. In *Proc. 1988 IFIP WG 10.2 International Working Conference on "The Fusion of Hardware Design and Verification", Univ. of Strathclyde, Glasgow, Scotland*, pages 97–114, July 1988.

29. Venkatesh Akella and Ganesh Gopalakrishnan. Static analysis techniques for the synthesis of efficient asynchronous circuits. Technical Report UUCS-91-018, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991. *TAU '92: 1992 Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, Princeton, NJ, March 18–20, 1992.*

30. Erik Brunvand. A cell set for self-timed design using Actel FPGAs. Technical Report 91-013, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991.

31. Venkatesh Akella. *An Integrated Framework for High-Level Synthesis of Self-timed Circuits.* PhD thesis, Department of Computer Science, University of Utah, 1992.

32. Venkatesh Akella and Ganesh Gopalakrishnan. Specification and validation of control intensive ics in hopcp. Technical Report UUCS-92-001, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991. *Accept subject to revisions by the IEEE Transactions on Software Engineering.*

33. Arthur Charlesworth. The Multiway Rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(3):350–366, July 1987.

34. Ganesh Gopalakrishnan and Venkatesh Akella. A Transformational Approach to Asynchronous High-level Synthesis. In *VLSI-93*. IFIP, September 1993. *To Appear.*