HIGH LEVEL OPTIMIZATIONS IN COMPILING PROCESS DESCRIPTIONS TO ASYNCHRONOUS CIRCUITS

GANESH GOPALAKRISHNAN[1]

ganesh@cs.utah.edu

VENKATESH AKELLA[2]

akella@cs.utah.edu

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

UUCS-92-019-a

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 some of the high level communication abstractions available in hopCP. We illustrate how these abstractions are realized in the asynchronous circuits generated by SHILPA. We then present a series of examples that present many of the high level optimization strategies used by SHILPA. Some of these optimizations aim to speed up the generated circuits by avoiding un-necessary waiting. Others synthesize components that are much easier to realize in a variety of technologies. We also discuss some of the tradeoffs possible between optimizations and timing constraints.

A version of this paper has been submitted to the Journal of VLSI and Signal Processing: Special Issue on Asynchronous Design

[1]Supported in part by NSF Award MIP-8902558
[2]Supported in part by a Graduate Fellowship from the University of Utah

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

HIGH LEVEL OPTIMIZATIONS IN COMPILING PROCESS DESCRIPTIONS TO ASYNCHRONOUS CIRCUITS

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

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

Keywords: Asynchronous VLSI Design, Self-timed Systems, High Level Synthesis, Concurrent Programming, Formal Methods in Design

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 some of the high level communication abstractions available in hopCP. We illustrate how these abstractions are realized in the asynchronous circuits generated by SHILPA. We then present a series of examples that present many of the high level optimization strategies used by SHILPA. Some of these optimizations aim to speed up the generated circuits by avoiding un-necessary waiting. Others synthesize components that are much easier to realize in a variety of technologies. We also discuss some of the tradeoffs possible between optimizations and timing constraints.

1 Introduction

Recently, there has been a revival of interest in asynchronous digital circuits. There are many compelling reasons for seriously considering asynchronous circuits as alternatives for synchronous circuits. From the point of view of designing large VLSI chips, it is becoming increasingly hard to distribute high frequency clocks that have low rise- and fall-times [1]. Since all the enabled gates in a synchronous circuit switch nearly simultaneously on every clock edge, synchronous circuits have higher peak power requirements, and so they place a higher burden on power and ground lines. Asynchronous circuits do not suffer from this problem. Due to the increasing levels of integration, asynchronous interfaces (that existed *outside* the chip in the days of lower integration) have started moving inside single chips [2]. Since asynchronous circuits are almost always incrementally expandable due to their use of explicit completion signals, they have shorter design (and *re*-design) times. From the point

*Supported in part by NSF Award MIP 8902558
†Supported in part by a University of Utah Graduate Fellowship

of view of performance, in many applications asynchronous circuits have shown the ability to run at speeds close to the combinational propagation times [3].

We refer the reader to [4] and [5, Chapter 7] for a lucid account of the issues in asynchronous circuit design, and why asynchronous circuits matter. For a survey of recent works on asynchronous design, see [6] and [7]. In the former, several recent design techniques are illustrated on one design example. In the latter, a thorough account of the fundamentals of gate-style and switch-style asynchronous circuits is presented, followed by a survey of recent works.

Approaches for Specifying Asynchronous Circuits

There are two prevalent approaches for specifying asynchronous computations: the *concurrent programming* approach, and the *state machine* approach. Many designers view asynchronous circuit design as *concurrent programming*, where the computation to be implemented is written in a high level concurrent HDL that is then compiled into circuits. This approach is followed by [8], [9, 10], [11], and also by us. Others specify the computation to be implemented using various automata, or in various *trace models.* Examples in this category include classical works [12, 13], as well as more recent works [14, 15, 16, 17].

Generally speaking, the concurrent programming approach is well suited for *high level synthesis* while the state machine approach is well suited for *low level* and *asynchronous state machine* synthesis.

Approaches for Realizing Asynchronous Circuits

There are also two classes of approaches for *realizing* asynchronous circuits in hardware. The basic problem in asynchronous circuit design is to translate high level *problem descriptions* onto *networks of transistors.* Since a transistor exhibits so many useful circuit properties (switching, ratioing, charge storage, threshold voltage drop to help sense the presence of metastability, to name a few) [5, 18], it is very difficult to directly compile high level problem descriptions into efficient transistor networks. Presently, therefore, designers only attempt to compile high level problem descriptions into something akin to "intermediate code". This intermediate code takes the form of either *Boolean gates* or *macromodules.*

Boolean gates implement Boolean functions in the usual manner. Macromodules, on the other hand, implement complex combinational or sequential functions. Macromodules come *both* in the synchronous and in the *asynchronous* variety. Some examples of synchronous macromodules are barrel-shifters, precharged ALU circuits, shift-registers with processing, stacks, etc. [5, 18]. Asynchronous macromodules implement functions such as rendezvous, arbitration, procedure call and return, and control merging.

Both gates and macromodules throw away much of the power that exists in "raw" transistors. Boolean gate based approaches translate the design problem into equivalent state transition specifications, perform state assignment and minimization, and finally realize the

circuit using Boolean gates. 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 macromodule programs (which are a priori given). This approach is generally used for high level synthesis. There are also many efforts in which macromodules are used directly for realizing state machines (*i.e.* for low level synthesis). Some examples are [14, 16]. 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.

Very little published work exists that makes a fair comparison of these techniques. This is promising area for future research. In our system, we employ macromodules as targets, at present.

The Case for Mixed Style Designs

In the long run, it appears that mixed synchronous/asynchronous systems will get employed increasingly. Classical solutions to this problem involve the use of *synchronizers*; however, synchronizers only *minimize* the chances for failure due to metastability; they do not totally avoid the chances of failure due to metastability. Another class of solutions that totally avoids failure due to metastability involves the use of *stoppable clocks* [5, 19, 20, Chapter 7] or Q-modules [21] that use a special flip-flop called the Q-flop. These solutions require the use of special components whose designs are not widely known or widely available, yet.

From the point of view of making *evolutionary* changes to the way in which digital designers design their circuits, it seems very important that this avenue of research be pursued.

About the Rest of the Paper: High Level Optimizations

There are many issues to be addressed before asynchronous circuits are widely adopted. (There are also many good reasons why they should be widely adopted!) Space restrictions prevent us from surveying this area any further. In this paper, we address our own recent efforts in this area, hoping to illustrate *some* of these issues more clearly.

As said before, we follow the programming view of asynchronous circuit design. For specifying asynchronous computations, we have developed a concurrent HDL called hopCP, which is a process + functional language. We specify system descriptions in hopCP into macromodules (which are currently realized in Field Programmable Gate Arrays) using our circuit compiler SHILPA[1]. The emphasis of this paper will be the *high level optimizations* employed in SHILPA.

By the term "high level optimizations", we mean the ability to write succinct and expressive specifications, and the ability to compile these specifications directly into efficient asynchronous circuits. High level optimizations are important for several reasons. Directly

[1] "System for the HIgh Level synthesis of Process descriptions to Asynchronous circuits."

expressing concurrency in asynchronous system descriptions is difficult and error prone. It would be much more preferable to write relatively more sequential code, and employ powerful high level abstractions, and let the system automatically discover concurrent evaluation possibilities and generate efficient circuit implementations. This is what are trying to do in our approach.

In section 2, we provide an overview of the suite of tools that constitute the hopCP system: a parser that compiles descriptions in the process + functional language hopCP into intermediate form; a process composition tool that infers the behavior of a network of asynchronous processes; a flow-analyzer that detects sharing opportunities; a compiled code simulator that can efficiently simulate hopCP descriptions; and a circuit compiler that compiles the intermediate form into asynchronous circuits.

We illustrate SHILPA on two examples. In the first (section 3), we illustrate the compilation of synchronization barriers and multicast channels. Also discussed are resource sharing, flow analysis based optimization, and efficient compilation of mutually exclusive communication-only guards. In section 4, we present an example of transforming a purely functional description into a pipelined process + functional description in hopCP. This allows us to derive pipelined circuits from hopCP. Concluding remarks are provided in section 5.

2 hopCP System Overview

Control intensive ICs pose several significant challenges to designers who wish to describe them at multiple levels of abstraction, simulate their descriptions efficiently, and synthesize circuits using sound semantics preserving transformation techniques starting from their descriptions. To help meet these challenges, we have developed a simple multi-paradigm HDL called hopCP, an efficient compiled code functional simulator called CFSIM, and an asynchronous circuit compiler SHILPA that takes hopCP descriptions and generates circuit netlists that can, at present, be technology mapped onto Actel Field Programmable Gate Arrays.

hopCP is a CSP-style language that provides constructs for expressing hardware behavior clearly and succinctly. It can also serve as a language for specifying existing synchronous and asynchronous off-the-shelf components. It supports a variety of communication primitives. For system-level description of behavior, it provides the features of barrier synchronization and multicast. Often subsystems in a hardware system communicate through "wire assignments". Wire assignments interactions cannot be directly specified in CSP-like languages. A precise notion of wire assignments is included in hopCP. For details on hopCP, see [22, 23].

A procedure has been developed to conduct flow analysis on hopCP descriptions. Basically this procedure determines if two actions a and b are always guaranteed to be serially ordered or are potentially concurrent. This information is valuable in compiling guards, as illustrated in section 3. Also this information helps in determining whether wire assignments and

the usage of the assigned wire values (which happen without explicit synchronization) are, indeed, race free (by virtue of being indirectly ordered by other synchronous communication actions used in the specification). The same set of flow-analysis procedures also help in detecting opportunities for resource sharing. For details, see [24].

A formal verification methodology for hopCP is yet to be developed. Currently, hopCP descriptions are validated through the simulation tool CFSIM, by compiling them into a concurrent derivative of the Standard ML programming language [25], called Concurrent ML (CML) language [26] (mainly because CML provides support for implementing many of the constructs of hopCP) and running these CML descriptions. CFSIM generates fairly efficient simulators: we have successfully simulated an Intel 8251 USART (a process with six concurrent sub-processes with more than 160 high-level control states and countless data states—see technical report enclosed) over 32,000 synchronizations and 60,000 function calls (for word packing, unpacking, etc.) in about a minute, including garbage collections. The CFSIM environment also allows one to write *tester processes* that serve as an environment specification, and help exercise various modes of behaviors of the system under test with great flexibility.

The SHILPA high level synthesis system takes behavioral descriptions in hopCP and produces a netlist for the Actel FPGA, supported by the VIEWlogic tools. Except for commands and options that guide the synthesis in various directions, SHILPA produces circuits automatically. hopCP descriptions are initially translated into an intermediate-form based on hypergraphs called HFG. SHILPA then applies action refinement, which is a technique for transforming HFGs into asynchronous hardware by a series of transformation rules. Action refinement is characterized by *incremental resource allocation* and *control decomposition.*

The major contributions of SHILPA when compared to the published works of Brunvand [9, 10], van Berkel [11], and Martin [8] (which are the most closely related works to ours) are the following. First of all, the source language hopCP is equipped with shared variables, broadcast channels, and barrier synchronization. Consequently, it is more expressive than their input languages (CSP/Occam like languages). SHILPA incorporates numerous high level optimizations. Flow-analysis based optimizations in SHILPA can analyze a given process in the context of its environment, and detects those actions that are serially ordered in the process, and those which are potentially concurrent. This information can be used to detect concurrent read/write accesses to asynchronous ports (which are erroneous), and also help share resources among serial actions. In this sense, SHILPA is closely related to traditional synchronous high level synthesis tools designed along the lines of Camposano, Parker, Ku and De Micheli, for example [27].

SHILPA supports the compilation of barrier synchronization and multicast. It supports the *speculative* style of evaluating *mutually exclusive* guards. For mutually exclusive guards, this results in circuits that are more area efficient and potentially faster. More importantly, it does not require the use of components such as the Q-SELECT as in [9] or ARBITRATION

```
module Guardex
...type and channel declarations...
behavior
  (P <=  (a? -> b? -> P)
        |(b? -> a? -> P))
  ||
  (Q <=  (c? -> a! -> Q)
        |(d? -> b! -> Q))
end
```

Figure 1: Example Illustrating Shared Channels with Speculative Evaluation

WITH TEST AND SET as in [28] that are very hard to realize in the FPGA technology, as they involve the use of analog circuitry such as the *interlock* [29]. (However, not all guards are mutually exclusive, and for those that aren't, we use the exact same circuit used in [9].) These will be discussed in section 3.

SHILPA has the ability to handle circuits involving both computations and communication. Since hopCP is a process + functional language, it allows tail recursive computations to be elegantly expressed. We show that tail recursive specifications can be transformed to process descriptions that evaluate tail recursive loops in a pipelined fashion ("software pipelining"). We illustrate these ideas in section 4 on the design of a series-parallel multiplier.

The SHILPA system comes with a macro-module library of self-timed parts built using Actel macros (an extension of the library developed by Brunvand [30]). All the circuits shown in this paper are automatically generated from the hopCP compilation system which is interfaced to the *Viewlogic*™ CAD tools.

3 Design Example 1: Speculative Guard Evaluation with Sharing

In this section, we present a simple example (Figure 1) that illustrates the kinds of issues one faces in compiling concurrent process descriptions into hardware. Though simple, this example illustrates several inter-related issues: the implementation of communication channels; the implementation of *guarded commands* (in the sense of the CSP language [31]); the effect of resource sharing; and *speculative evaluation.*

Two concurrent processes `P` and `Q` are described in the syntax of hopCP. Process `P` employs the channels `a?` and `b?` while `Q` employs `a!`, `b!`, `c?`, and `d?`. Communcations through these channels follows the *rendezvous* discipline—the sender and the receiver wait for each other,

finish the communication, and then proceed.

Our implementation of the channels follows the two-phase transition style signaling convention [4]: a request transition (zero to one or one to zero) is acknowledged by an acknowledge transition. The *active* [8] channels (whose names end with a `!`) are implemented by generating the request and awaiting the acknowledge, while passive [8] channels are implemented by awaiting request and generating acknowledge. Notice the difference between *channels* and *communication actions* on channels: "channels" denote physical resources through which communication takes place, while communication actions are particular usages of channels for effecting a certain communication. In the fragment

```
P <=   a? -> b? -> P
     | b? -> a? -> P
```

the two `a?` are two distinct occurrences of passive communication on the same channel hardware dedicated for channel `a`, and similarly the two `b?` denote distinct occurrences of passive communication on channel `b`.

`P`'s behavior is as follows. It awaits a communication on channel `a?` or channel `b?` (the "or" is indicated by `|`). (The above is an example of a guarded command.) Depending on which request of the guarded command arrives first, `P` commits to that choice, and then proceeds to perform another communication coming sequentially afterwards. Sequencing is indicated by `->`. For example, if `a?` arrives first in the guard, `P` commits to it, and then awaits the `b?` coming sequentially after the `a?`, and finally recurses back to state `P`.

`Q`'s behavior may be similarly understood, except that after selecting between one of `c?` and `d?`, it generates an `a!` or a `b!`, which match the `a?` and the `b?` awaited by `P`. In other words, `Q` is the *environment* for `P`.

We present the compilation of the example in Figure 1 through a series of steps. We first show how simple control flow constructs (sequencing and iteration) are compiled. We then show how communication channels are compiled. Next, we show how general *communication only* guarded commands can be compiled. We then show how *flow analysis* can be conducted to detect *mutually exclusive guards* (called *mutex* guards, for short). We present a circuit for mutex guards. Finally, we show how sharing can be handled. We then put together all the above ideas to get the circuit corresponding to Figure 1. We then present an analysis of the circuit obtained, comparing it against alternative ways of compiling this circuit.

3.1 Compilation of Simple Control Flow Constructs

In the hopCP system, a set of *actions* is compiled into a circuit called an *action block*. An action block has an explicit initiate signal and an explicit complete signal. Data (if any are

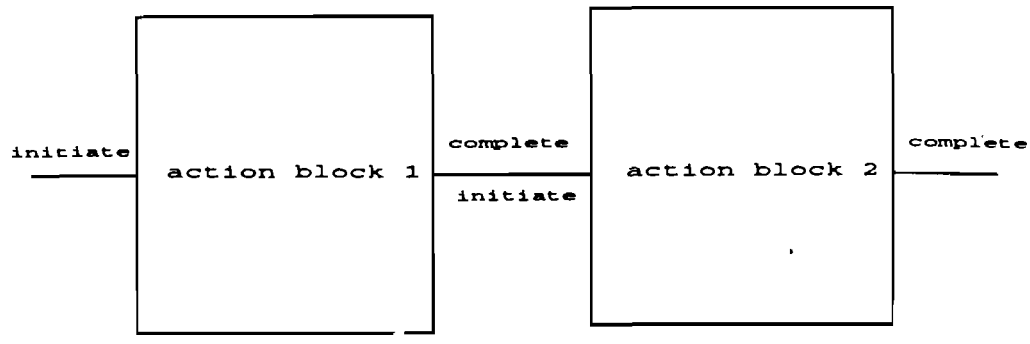

Figure 2: Compilation of Sequencing

involved) must be presented to action blocks bundled [4] with control signals. Compilation of the sequencing operator -> is shown in Figure 2.

Tail recursive computations are compiled by using the completion signal of the body to re-initialize the body, using a MERGE element (as is shown in figure 3). In general, a MERGE element is employed whenever control can enter a state from more than one place.

3.2 Compilation of Communication Channels

SHILPA, the high level synthesis sub-system of hopCP, first analyzes the usage pattern of every channel. A hopCP specification consists of several *sequential* processes that run in parallel. For example, in module `Guardex`, `P` and `Q` are sequential processes, and `||` denotes that they can run in parallel.

Every communication channel used in a hopCP specification is used in the *active* mode by exactly one sequential process; all other sequential processes can use this channel only in the *passive* mode. (Other sequential processes are, of course, not required to use this channel.) If a channel is associated with data, then the data is supplied by the active user of the channel. Those channels that are used passively by exactly one process are implemented by allocating a C-ELEMENT that implements a two-way rendezvous. Channels that are used passively by more than one process are implemented by using either *multicast* or *barrier synchronization*, depending upon whether the channel is associated with data (multicast) or not (barrier synchronization). These two ways of compiling channels will now be illustrated.

For the sake of definiteness, consider the following hopCP specification (only the `behavior`

section is shown):

```
module barriersync
...type and channel declarations...
behavior
  (P <=   a! -> P)
  ||
  (Q <=   a? -> Q)
  ||
  (R <=   a? -> R)
end
```

In this fragment, channel a is used actively by P, and passively by Q and R. All of P, Q, and R are sequential processes. In other words, they have no "internal parallelism". They are described using only *sequencing* and *tail recursion.* Channel a is implemented using barrier synchronization, with the following execution semantics. Both Q and R await P to send a request on a!. When the request arrives, Q and R send their own acknowledge signals. All these acknowledges are synchronized using a *completion tree* before being fed back as the acknowledge of P; this very signal also forms the initiate signal for the computation in Q and R that come after a?. In this way, *time alignment* between P, Q, and R is achieved. The circuit in figure 3 shows the circuit automatically generated from the above hopCP fragment. Barrier synchronization is a handy paradigm for modeling and implementing many parallel algorithms [32].

Now, consider the following hopCP fragment

```
module multicast
...type and channel declarations...
behavior
  (P[x1] <= a!x1 -> P[x1])
  ||
  (Q[x2] <= a?y  -> Q[y])
  ||
  (R[x3] <= a?z  -> R[z])
end
```

The automatically generated circuit corresponding to this program is shown in figure 4. In this example, process P sends the value of the variable x1 on channel a, which is received by Q and R. We allow the processes Q and R to latch the values sent as soon as they arrive at the synchronization point. Process P is blocked at its synchronization point till all the receivers (in this case Q and R) have latched the value sent by P. However, the circuit in Figure 4 is still sub-optimal because Q and R are not allowed to proceed as soon as they have latched

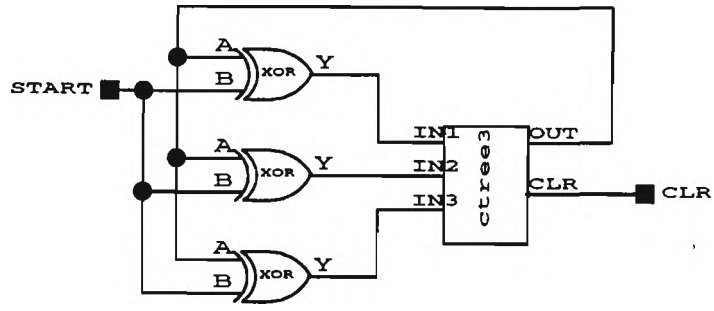

Figure 3: Compilation of Barrier Synchronization

the value sent by P. An optimization of this circuit is shown in Figure 5. (This circuit will be obtained from that shown in Figure 4 with the help of a circuit optimizer that is currently being written.) We next turn our attention to the compilation of communication guards in hopCP.

3.3 Compilation of Communication-only Guards

Communication-only guards are employed in guarded commands such as shown in process Q, below:

```
module Guardex
...type and channel declarations...
behavior
  (P <=  (a? -> P)
        |(b? -> P))
  ||
  (Q <=  (c? -> a! -> Q)
        |(d? -> b! -> Q))
end
```

In this example, communications on channels c? and d? may arrive in any order. If both arrive, exactly one of these guards is picked by Q non-deterministically. The implementation of this guard uses a *ring style* arbiter, as shown in Figure 6. This circuit is exactly the one

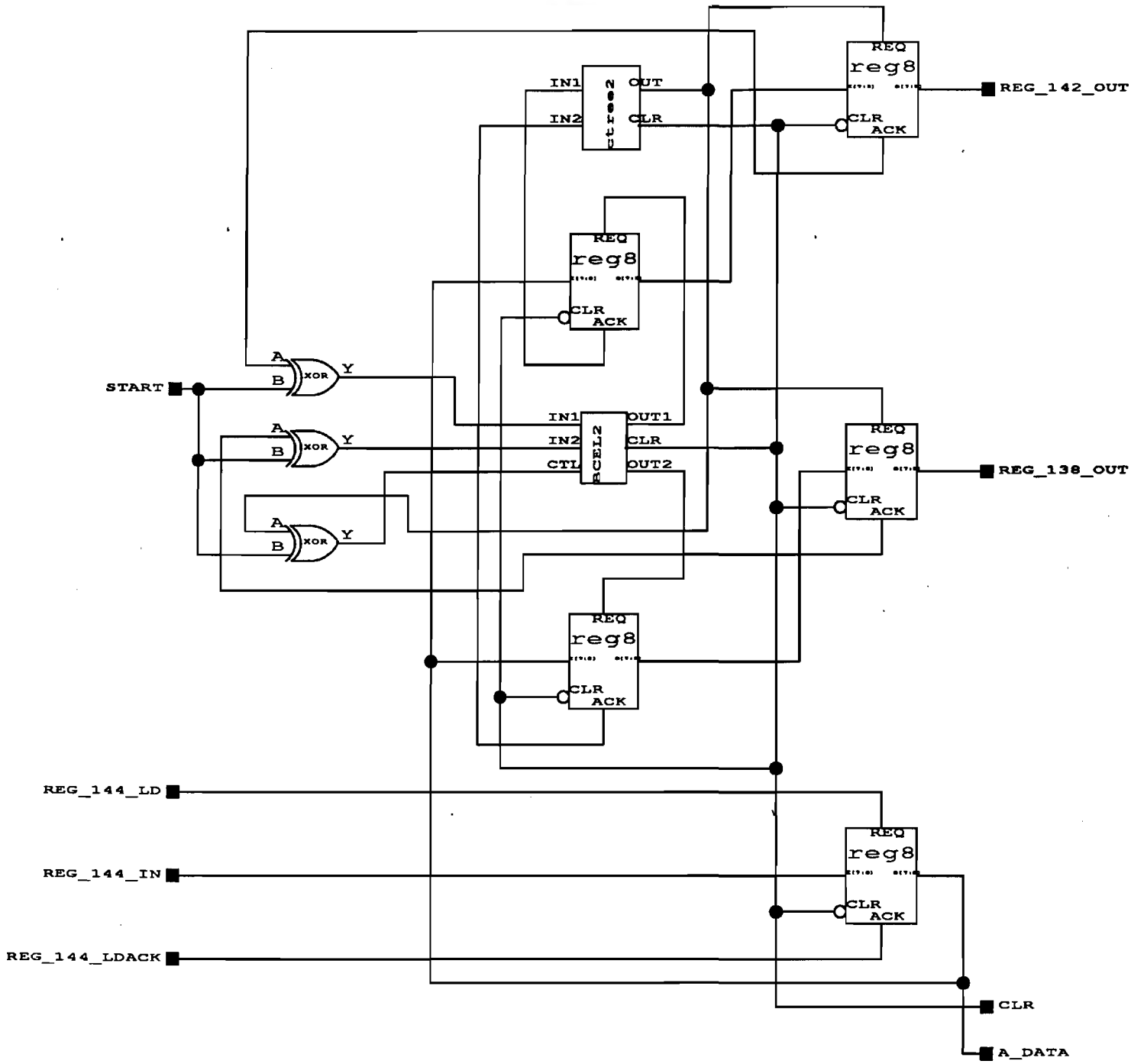

Figure 4: The Partially Optimized Compilation for Multicast

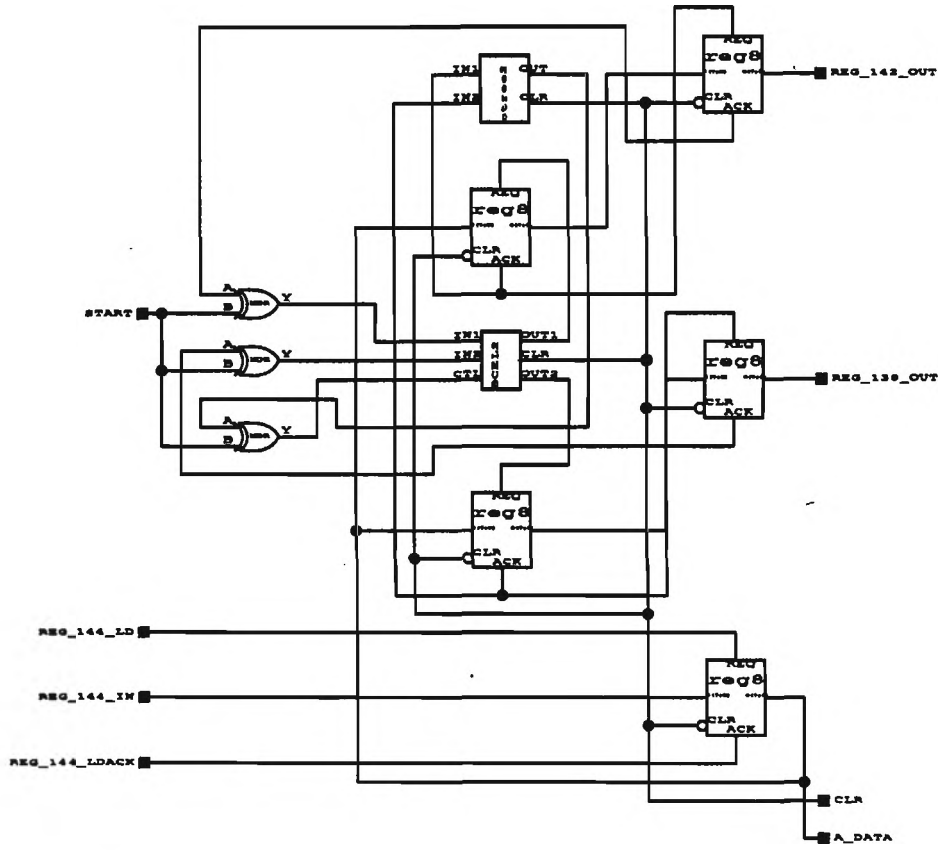

Figure 5: The Optimized Multicast Circuit

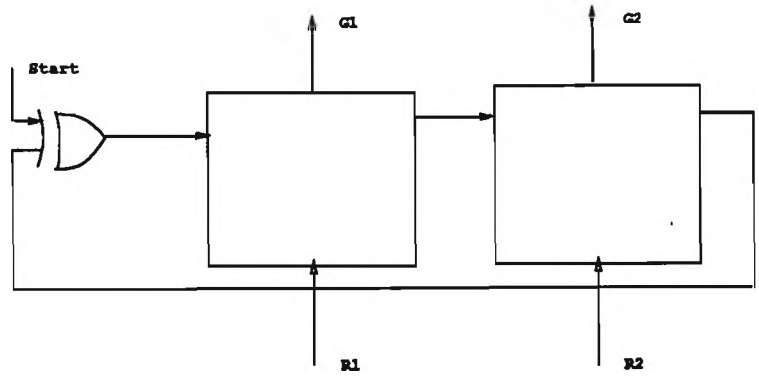

Figure 6: The Implementation of General Communication Guards

used in [30]. It requires the use of `Q-select` modules. A `Q-select` module is capable of steering an incoming token to either its `true` output or its `false` output depending on the voltage level (high for `1` or low for `0`) of a third input signal. In Figure 6, a token is injected into the ring through a transition on `start`. Depending on which of `R1` and `R2` comes first (these corresponds to the guards), a `G1` or a `G2` will be generated (these correspond to the computations that follow the guards). After that, the ring is re-initialized.

Such a circuit is typically realized in full custom VLSI using arbitration circuits. Prototyping these circuits using available programmable devices is virtually impossible, because of the fact that most arbitration circuits require the use of low level transistor circuits [5, Chapter 7].

We propose a solution to this problem, in case the guards are *mutex*. An example of a mutex guard is the guard used in process `P`. It is guaranteed that only one of `a?` or `b?` will arrive at `P`'s guarded statement precisely because the corresponding `a!` and `b!` are generated by `Q` in the two "arms" of its own guarded command; since only *one* among `c?` and `d?` will be non-deterministically picked, both `a!` and `b!` cannot arrive at P's guarded statement. Therefore, by detecting mutex guards, we can compile a better circuit for implementing guarded commands. This topic will be discussed in sections 3.4 and 3.5.

3.4 Detecting Mutex Guards Through Flow Analysis

The guard evaluation circuitry shown in figure 6 is used for implementing general communication only guards because it is not possible to tell a priori which of the communication

actions in the guard(s) will arrive. However, if we know that the guards are mutex, a much simpler circuit can be synthesized. The question now is "how straight-forward is it to detect mutex guards"?

In [24], we have proposed a flow analysis procedure that detects mutex guards. Although this procedure is inherently exponential time (it is based on the reachability analysis procedure on Petri nets) and does not yet consider dependencies of control flow on data, we employ numerous heuristics that make this procedure perform well on realistic examples. For example, we have applied the flow analysis procedure on the specification of an Intel 8251 USART—a fairly involved serial communications adapter—and obtained good performance results [24]. In that example, several communication-only guards proved to be mutex. Hence, it appears that the technique we are going to propose in section 3.5 can be widely applied.

3.5 Compilation of Mutex Guards

Consider the example from section 3.3, which is repeated below:

```
module Guardex
...type and channel declarations...
behavior
  (P <=  (a? -> P)
        |(b? -> P))
  ||
  (Q <=  (c? -> a! -> Q)
        |(d? -> b! -> Q))
end
```

The SHILPA system determines which guards are mutex (the guards of P in our example) and which are not (the guards of Q in the present example), and produces the circuit shown in Figure 7. In this circuit, a transition on **start** produces a transition on input A of MERGE 2 as well as input A of MERGE 4. These, in turn, produce transitions on the input A of the C-ELEMENTs 3 and 5. Thus, C-ELEMENTs 3 and 5 are "armed" to look, in parallel, for a communication on either a? or on b?. In other words, we *speculate* that both these communications have equal likelihood of arrival; however, only one of the communications can arrive (because the guard is *mutex*) and so we must also be prepared to "undo" the effect of the unsuccessful guard. The circuit in figure 7 has this capability.

Now, a transition on the B inputs of these C-ELEMENTs comes from the outputs of the C-ELEMENTs 9 and 10 which implement c? and d? respectively. Only one of these outputs will be chosen. Let us say that c? arrives, and hence the output of C-ELEMENT 9 produces a transition. This causes a transition on the B input of C-ELEMENT 3. This, in effect, allows a? to succeed. It also causes a transition on the B input of MERGE 4 which causes

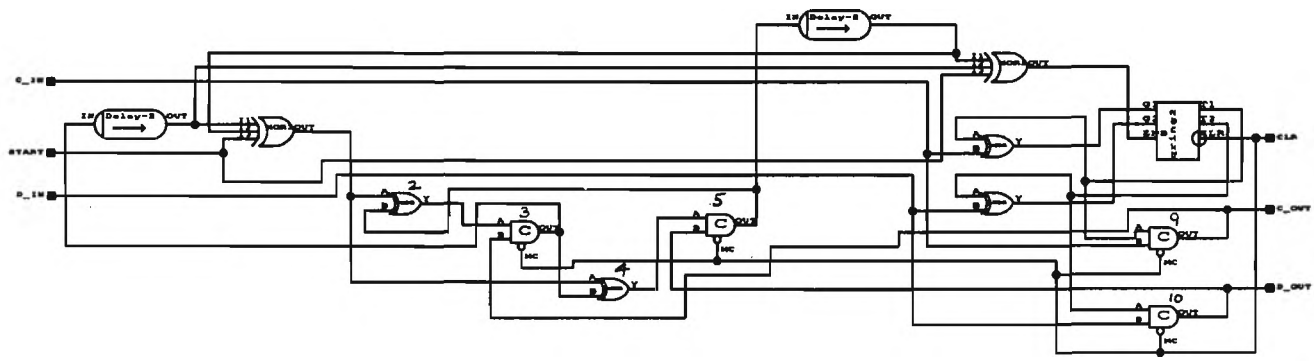

Figure 7: The Implementation of Mutually Exclusive and Non-Exclusive Guards

another transition on the A input of C-ELEMENT 5, thereby "undoing" the effect of the initial transition on this very input.

The circuit that has been automatically synthesized is precisely the CAL component of [16]. This is not surprising, considering that both [16] and [33], in their compilation approaches, use the CAL component to implement guards. However, for our circuits to work correctly, they must be *foam-wrapper packaged*, as explained in the next section. This explains the role of the DELAY elements, *which are also automatically generated by SHILPA.*

3.6 Foam-wrapper Packaging for Delay Insensitivity

The class of purely delay insensitive gate circuits is very limited [34]. By this, it is meant that there are very few Boolean gate based sequential circuits whose behavior is invariant over wire and gate delays. For example, a C-ELEMENT commonly realized using the state equation `c' = ab + bc + ac` (where `c'` is the next value of `c`) is not delay insensitive [35]. The usual way to make this circuit delay insensitive is by making sure that the internal feedback has occurred before the external `c` transition is produced. One specific way to do this is by padding the outgoing `c` wire with a DELAY element whose value exceeds the time it takes the internal feedback paths to stabilize. For lack of a standard term, we call this *foam-wrapper* packaging, based on Molnar's use of the term "foam-wrapper principle" for describing delay insensitive circuits. The role played by the delay elements is precisely to *foam-wrapper package* the generated CAL component.

Therefore, modulo the one-sided constraint introduced by the DELAY element, the guard evaluation circuitry for the guard of P in Figure 7 only employs standard gate style circuits. It is also smaller in size and could be faster in operation, especially if there are many communication guards in a guarded statement. In this example, we can see the example of a design where: (i) the initially generated circuit is not delay insensitive; (ii) by foam-wrapper packaging certain sub-circuits, we again obtain a *delay insensitive* circuit.

3.7 Handling Sharing

Resource sharing is handled in the SHILPA system by using CALL elements, as shown in [4]. Recall that channels are also shareable resources. Therefore, two occurrences of communication actions using the same channel name share the same physical channel resource. Now consider the following hopCP specification

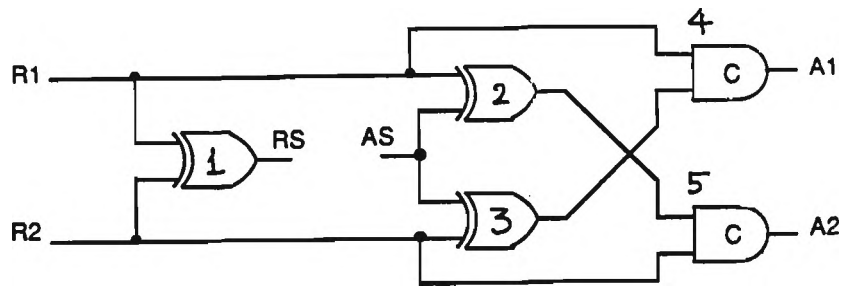

Figure 8: The Implementation of a Call Element

```
module Guardex
...type and channel declarations...
behavior
  (P <=  (a? -> b? -> P)
       |(b? -> a? -> P))
  ||
  (Q <= (c? -> a! -> Q)
      |(d? -> b! -> Q))
end
```

In this specification, P uses channels a and b *twice*, once in a guard situation, and once outside. Both these occurrences of a? as well as b? share the same physical channel resources. We would like to employ CALL elements to share the channel hardware for channels a and b.

The question now is: can we employ the optimized circuitry for implementing the mutex guard of P, despite the presence of channel sharing?

3.8 Putting It All Together

This problem defied a simple solution until we realized that typical implementations of a CALL element [36] support many more behaviors than are normally exploited. In the TRANSITION CALL element shown in Figure 8, if a transition on R1 is applied, a request RS is made on the output. In addition, internally to the CALL element, MERGE element 2 is subject to an input transition, **c-element** 4 is subject to an input transition on its

top input, and C-ELEMENT 5 is subject to a transition on its top input. When AS arrives, internally to the CALL element, MERGE elements 2 and 3 are subject to input transitions. The transition applied to MERGE 2 triggers the top input of C-ELEMENT 5 again, thereby effectively "reset"ing this `c-element`'s input. However, the transition applied to MERGE 3 also causes a transition on the bottom input of C-ELEMENT 4. This C-ELEMENT fires, producing a transition on `A1`. The call/call-return with respect to `R2/A2` works similarly.

If we apply the sequence `R1; R1` to this CALL element, a sequence `RS; RS` is generated, and if we wait for sufficient time for the internal elements of CALL to settle, it ends up in the same state as it was before `R1; R1` was applied. This additional capability of the CALL element is exploited in the circuit compiled in the next subsection.

3.9 Analysis of the Circuit for Example 1

The specification shown in module **Guardex** generates the circuit shown in Figure 9. The main difference between the circuits in Figure 7 and Figure 9 is the following: the latter circuit uses a CALL element to share the channels. Since the sequence `R1; R1` applied to a CALL element causes a sequence `RS; RS` at its output, we can still perform speculative evaluation of the shared channels, with the "resetting" of the C-ELEMENT being performed through the CALL element.

Again, the DELAY elements are automatically generated by SHILPA. What we have implemented is, in effect, a *foam-wrapper* packaged version of a shareable CAL component.

3.10 Discussion

The above examples illustrate many of the design options that designers have at their disposal in compiling concurrent programs into asynchronous circuits. By making the HDL more expressive, we place a greater burden on the circuit compiler writer. However, the circuits generated by us are area efficient (compared to the results of related works). In some cases, especially with mutex guards involving many communication guards, the circuits can be more time efficient also, by not delaying actions un-necessarily (as demonstrated in the example involving multicast).

It is possible to simulate the effects of higher level communication abstractions in virtually every concurrent HDL available. However, this greatly burdens the specification writer, and can be error prone. Already, concurrent programming is hard, and we should try not to make it harder.

Trade-offs such as the use of complex building blocks (*e.g.* Q-select modules) *vs. unusual usages* of *standard* building blocks (*e.g.* the synthesized CAL components) is also worthy of further study.

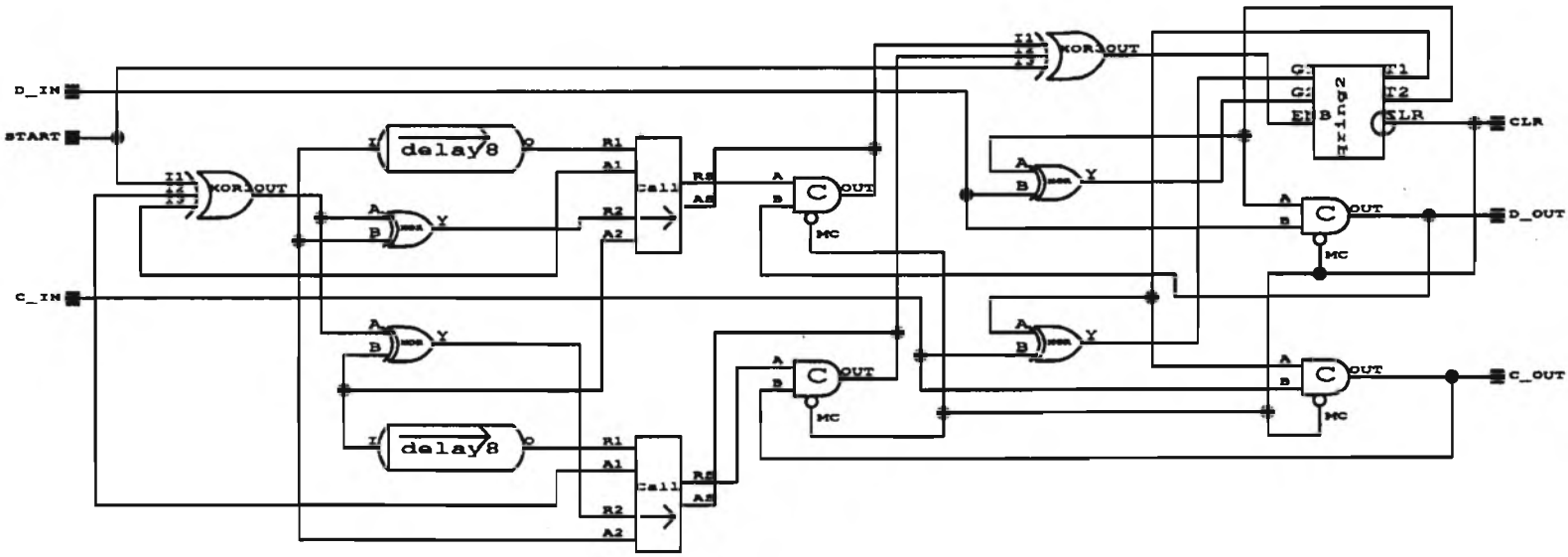

Figure 9: Mutex Guards with Sharing

4 Design Example 2: Transforming Tail Recursion to Pipelining

In this section, we demonstrate that the use of a process+functional language can greatly facilitate the systematic compilation of pipelined circuits for implementing iterative computations. Functional languages have a simple and well understood semantics, and are very expressive [37, 38]. There are many efforts which have shown that significant pieces of hardware can be specified in a functional notation and compiled into *synchronous* hardware [39, 40, 41]. Here, we hope to show that (i) the operational details of evaluating functional expressions, including whether to evaluate the actual parameters before or after the function call is made [42], can be captured in a process notation; (ii) in doing so, we can elaborate the control aspects for pipelining a tail-recursive functional loop; and (iii) pipelined asynchronous hardware can be compiled from such transformed specifications. The basic intuition is very simple: the repeated evaluation of a functional expression of the form $f(E)$ is equivalent to the execution effects of the following hopCP program:

```
behavior
 Process_f <= channel?x -> deliver!f(x) -> Process_f
||
 Process_E <= channel!E -> Process_E
```

In other words, the implicit *rendezvous* between the evaluation of E and the application of f can be elaborated quite simply.

As an example of the transformations involved, consider the specification of a series-parallel multiplier written in a purely functional style

```
MULTFN(x,y,z)  =  if (isZero y) then z
                  else if (odd y)
                        then MULTFN(x, y-1, z+x)
                        else MULTFN(lshift x, rshift y, z)
                     endif
                  endif
```

It is possible to capture the evaluation of a call to MULT in hopCP, as shown below:

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

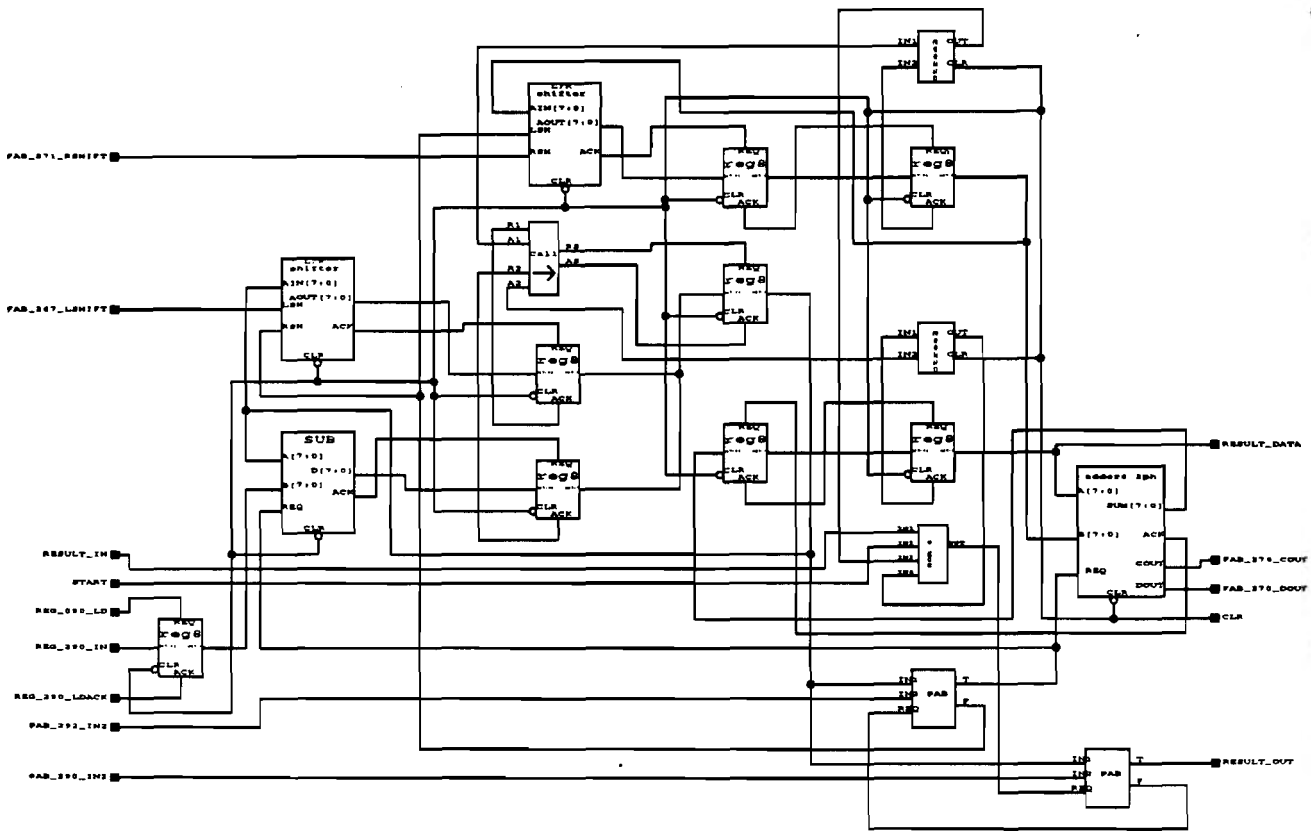

Figure 10: The Nonpipelined Multiplier

A circuit compiled from this recursive hopCP specification is shown in figure 10. This circuit uses two *predicate-action* blocks (to test for *zero* and *odd*), two shifters, one adder, one subtractor, eight registers (to hold variables x, y, and z, and intermediate values), a four-input MERGE, two two-input C-ELEMENTs, and a CALL element (to share the y register). (Please note that all data bus connections are shown as "logical connections" *i.e.* our compiler does not, at the time of writing, break down a data bus connection into its constituent bits; such a feature will be available very soon. Until that happens, two data output wires that are shorted together actually represents a place where a multiplexor has to be introduced.)

4.1 Obtaining a Pipelined Implementation

The conventional way of evaluating a call to MULTFN, or MULTP involves the evaluation of all the actual argument expressions followed by a "jump" to MULT (this is always possible to do when recursive calls are outermost—*i.e.* not nested inside other function calls [39]. However, notice that the value of the actual expression z+x is not needed until z is used in the body of MULTFN or MULTP. Thus, holding up the recursive call of MULTP can be wasteful.

Fortunately, because MULTP is written in a process notation, we can transform it to MULTPIPE shown next:

```
(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]))
```

The variable z has been factored out of MULTPIPE and made a local variable of process PZ. PZ's role is to treat variable z as an abstract data type, allowing it to be accessed only through its external operations. In the above specification, we provide two operations on PZ: operation sz that stands for "send z", and operation azx that stands for "add z to x". These operations are implemented through the rendezvous communications sz? and azx?x respectively. Notice that the second rendezvous communication involves data x that is sent from MULTPIPE 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

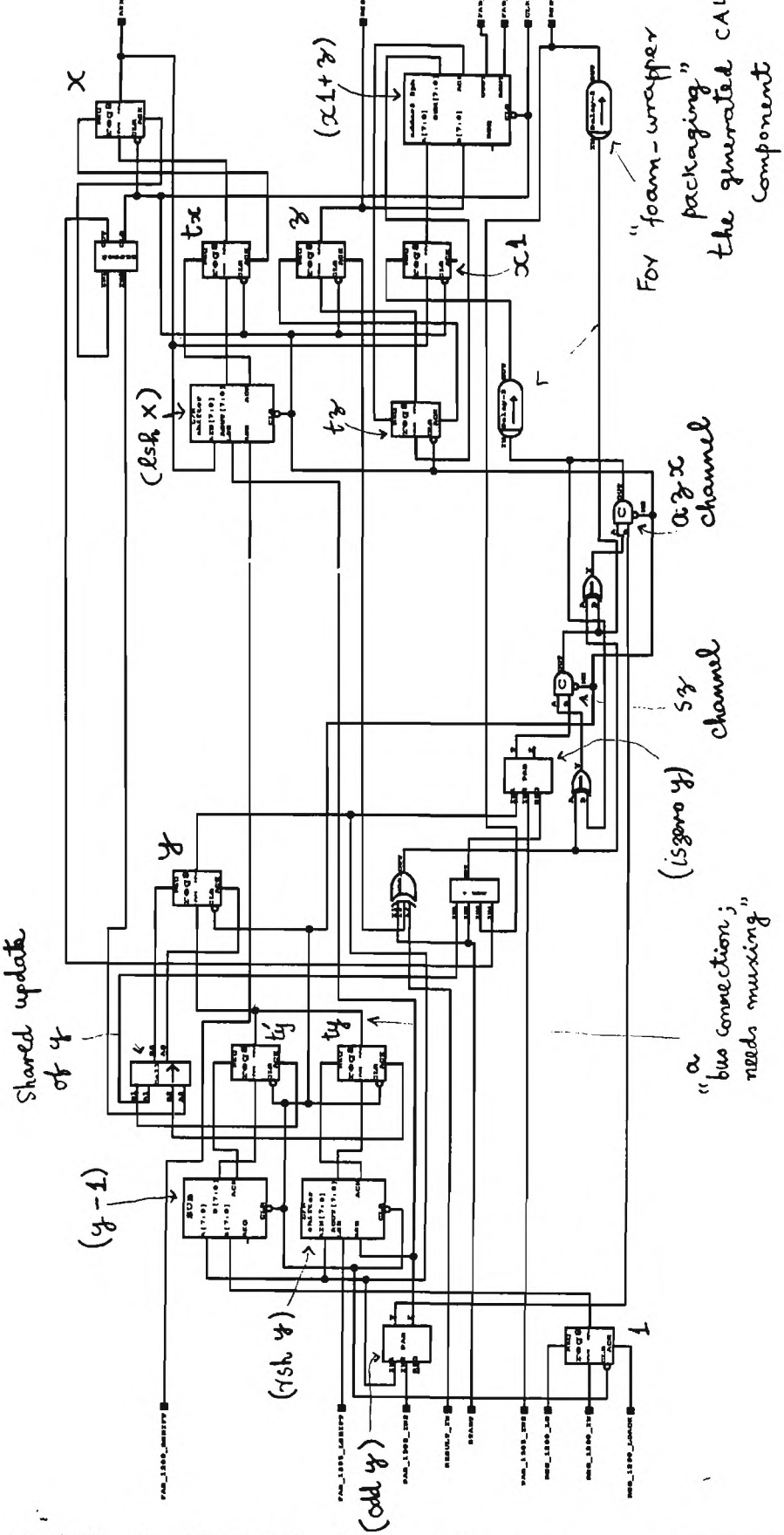

Figure 11: A Pipelined Multiplier Obtained Through Program Transformation

of software pipelining, and results in the circuit shown in figure 11. This circuit uses the following *additional* components over the circuit in Figure 10: one register, a three-input MERGE, and a TWO-INPUT CAL. A TWO-INPUT CAL, in turn, is generated precisely as explained in section 3.5 using two two-input C-ELEMENTs, two two-input MERGEs, and two delay modules.

Thus, for a very small overhead, we are able to obtain (in our estimate) a circuit that can perform potentially much faster. (Detailed performance analysis studies are pending.)

We are studying the process of automating this kind of program transformation in SHILPA. Until this happens, SHILPA still offers a convenient way to derive pipelined circuits from non-pipelined iterative computations by allowing the specification writer to hand-transform tail recursive functional specifications into process specifications. In our estimate, automating this would not be hard.

5 Concluding Remarks

We have argued that it is possible to view asynchronous VLSI design as *concurrent programming.* We have shown how SHILPA synthesizes asynchronous circuits from concurrent programs written in hopCP through a series of transformations. The target circuits are realized using *macromodules.*

It is well known that writing and debugging concurrent programs is hard. The hopCP notation avoids many possible pitfalls in writing concurrent descriptions by offering the facilities of barrier synchronization and multicast. To ease design debugging, the hopCP system offers CFSIM, a compiled code functional simulator. Finally, hopCP also allows low level hardware features such as "wire assignments" that can be checked for safe usage.

Our example in section 4 suggests that data related aspects can be handled by first writing iterative computations in a functional notation, and transforming it into process descriptions which can be pipelined. These, and other examples suggest that the area of high level optimizations is very promising for future investigations into investigating area and time efficient circuits.

We also have touched upon the tradeoffs possible between circuit optimizations and timing constraints. These issues surface when one tries to perform relatively lower level optimizations, such as shown in section 3.

In fact, we believe that one of the most fundamental problems that asynchronous designers are left to answer is the following: given a certain silicon area A and a sequential function F to be implemented, how does one select the "right" set of building blocks (in terms of their sizes and functionality) and populate the area A using the blocks in such a way that F is realized *efficiently* as well as *reliably.* This question cannot have a universal answer; various asynchronous design methodologies that are being proposed can be viewed as trying

to locate a point in the possible design space and justifying why that point is better for a given application.

At present, we have synthesized many small circuits using the SHILPA system. We are now in the process of synthesizing a large communications chip. Through this example, we hope to gain insights into optimization techniques to be incorporated into future versions of SHILPA.

References

1. Teresa H. Meng. *Synchronization Design for Digital Systems.* Kluwer Academic Publishers, Boston, 1991.

2. Gordon M. Jacobs. *Self-timed Integrated Circuits for Digital Signal Processing.* PhD thesis, Electronic Research Laboratory, University of California, Berkeley, November 1989.

3. 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.

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

5. C. A. Mead and L. Conway. *An Introduction to VLSI Systems.* Addison Wesley, 1980.

6. 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. *Being revised based on comments from the acm Computing Surveys.*

7. 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.

11. 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.

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

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

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

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

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

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

18. Lance A. Glasser and D. W. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Addison-Wesley, Reading, MA., 1985.

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

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

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. Venkatesh Akella and Ganesh Gopalakrishnan. hopcp: A concurrent hardware description language. Technical Report UUCS-TR-91-021, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991.

23. 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. *Submitted to the IEEE Transactions on Software Engineering.*

24. 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. *To appear in TAU '92: 1992 Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, Princeton, NJ, March 18–20, 1992.*

25. Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991. ISBN 0-521-39022-2.

26. John H. Reppy. CML: A Higher-order Concurrent Language. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.

27. Michael C. McFarland, Alice C. Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

28. Geoffrey Brown. Towards truly delay insensitive realization of asynchronous circuits in process algebras. In *Workshop on Designing Correct Circuits, Oxford, 26–28.* Springer Verlag, September 1990. *Proceedings of the DCC Workshop, Oxford, September, 1990, published in Springer's new series 'Workshops in Computing'.*

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

30. Erik Brunvand. Implementing self-timed systems with fpgas. In *International Workshop on Field Programmable Logic and Applications, Oxford*, September 1991.

31. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. Original article on CSP.

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

33. Jan Tijmen Udding and Mark B. Josephs. The design of a delay-insensitive stack. In *Workshop on Designing Correct Circuits, Oxford, 26–28.* Springer Verlag, September 1990. *Proceedings of the DCC Workshop, Oxford, September, 1990, published in Springer's new series 'Workshops in Computing'.*

34. John Brzozowski and Jo C. Ebergen. On the Delay-Sensitivity of Gate Networks. Technical Report CSN90/X, Eindhoven University of Technology, 1990.

35. John Brzozowski and Jo C. Ebergen. Recent developments in the design of asynchronous circuits. Technical Report CS-89-18, Department of Computer Science, University of Waterloo, May 1989.

36. Erik Brunvand. Parts-r-us. *a chip aparts(s)....* Technical Report CMU-CS-87-119, Carnegie Mellon University, May 1987.

37. Paul Hudak. Conception, evolution, and application of functional programming languages. *acm Computing Surveys*, (3):359–411, September 1989.

38. John Hughes. Why functional programming matters. Technical Report 16, Programming Methodology Group, University of Goteborg and Chalmers Institute of Technology, Sweden, November 1984.

39. Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* The MIT Press, 1984. An ACM Distinguished Dissertation-1983.

40. Stephen Johnson, B. Bose, and C. Boyer. A tactical framework for hardware design. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

41. Robert M. Wehrmeister. Derivation of an SECD machine: Experience with a transformational approach to synthesis. Technical report, Indiana University, Bloomington, 1990.

42. Zohar Manna. *Mathematical Theory of Computation.* New York: McGraw-Hill, 1974.