Abstract

As VLSI technology improves, the number of devices that can be built on a chip, and the speed of those devices continue to increase. These improvements allow much more complicated systems to be considered than were possible a short time ago. Along with these improvements, however, come many challenges directly associated with the speed and scale of the new circuits. This thesis presents a method for taming the complexity of large and fast VLSI systems.

As chips get larger, and delays in signal propagation even inside a single chip become more significant, systems that are designed around a global synchronizing signal such as a clock become more difficult to design. One alternative is to design the system hierarchically as a set of subsystems each operating at its own rate and cooperating through communication. These subsystems can be built using *asynchronous* or *self-timed* circuit techniques which allow the circuits to run at their own speed without synchronizing to a global clock signal. The act of communicating synchronizes the processes involved in the communication and allows data to be shared between processes.

Another problem of large systems is related to the very complexity of the system. One method for taming this complexity is to use automatic methods for generating circuits from behavioral descriptions. Such a system is usually called a *silicon compiler*. This allows the designer to abstract away details of the low-level circuits and think of system behavior in terms of high level programs. Because the generated circuits faithfully mimic the behavior of the program, the resulting circuits are correct by construction. In order to design efficient systems in this way there must be a way for the programmer to reason about the resulting circuit based on the program text. The translation process must be sufficiently transparent to give the programmer some idea of how different program alternatives will affect the compiled circuit.

Combining these ideas, I present a method for designing a VLSI system as a concurrent program written in a subset of OCCAM and automatically translating that system description into an asynchronous circuit. The translation procedure is syntax-directed, and the resulting circuits are improved using correctness-preserving circuit-to-circuit transformations. A compiler has been constructed that automatically performs the translation and transformation.

Acknowledgements

Many thanks to the members of my committee, Bob Sproull, Randy Bryant, Steve Brookes, and Ivan Sutherland, for their thoughtful comments and suggestions regarding this thesis. I would especially like to thank Bob Sproull for his friendship and guidance throughout this project. I am continually amazed with his technical knowledge and the speed with which he considers new ideas. His extremely careful reading of this thesis greatly improved the document. Steve Brookes and Randy Bryant also made helpful comments at the right times which helped me focus on the important issues.

I would like to send a special thanks to Ivan Sutherland. He is a constant source of fascinating ideas. His ideas about how self-timed circuits should be organized have had a major impact on my work.

Carnegie Mellon is a wonderful place to learn about computer science, and about research in general. For a community this large to have such consistently impressive individuals, and still be friendly and relaxed is rare indeed. In particular I would like to thank my office-mates over the years: Dave Anderson, Alex Hauptmann, Bruce Horn, Tom Rodeheffer, Ellen Siegel, and Marco Zagha. Your friendship has been a valued part of my years at CMU. I hope that you have learned at least a little from me in return for all that I have learned from you. Special thanks to Dave and Ann Detlefs and Dan and Gayle Kuokka for your friendship and support over the years. Also to Don Stone who, along with Dan and Dave, dragged me off to the golf course occasionally to replace the frustrations of thesis work with new frustrations. Other people who deserve special thanks include Bernd Bruegge, Claire Bono, Sid and Sarmita Chatterjee, Craig Knoblock, Doug and Carol Reece, Angelika Zobel and a long list of others. I'm sure I've left some names off and I apologize for that. A big thanks to you all. I would also like to thank Sharon Burks for her optimism and encouragement, as well as help with the administrative details of being a graduate student and for her patience in the long last stretches of finishing things up.

I owe a special thanks to my parents for instilling in me the curiosity and love of learning that have led me to where I am today. Finally, a huge thank you to my wife Allison who has shared the experience of graduate school through good and bad, and been a constant source of encouragement, support, companionship, and optimism throughout.

Contents

List of Figures

Chapter 1

Introduction

In the quest to increase system performance, VLSI technology and concurrency emerge as two of the main research areas. The trend in VLSI technology is towards smaller, denser, and faster circuitry with circuit size shrinking and circuit density increasing at an almost constant rate. Along with these improvements in speed and circuit size, however, come many problems directly associated with the speed and scale of the new circuits. A major challenge to realizing the benefits of these technological improvements is how to reduce the design time and energy required to produce correct circuits given the new possibilities.

As technological improvements continue to make faster and denser circuits available more economically, the possibility of applying concurrency becomes more realistic. Because VLSI circuits are inherently concurrent at a low level, and VLSI technology is often used to implement systems that are concurrent at a higher level, an attempt to improve the ability to design circuits should include explicit control of the concurrent aspects of the system being designed. Concurrent operation of multiple subsystems holds great promise for increasing system performance, but only if the results of the concurrency are understood. Because concurrent systems are inherently nondeterministic, they are much harder to understand than traditional sequential systems. Reducing the time and effort required of the designer to produce correctly operating concurrent systems is extremely important.

1.1 Translation from Programs to Circuits

As VLSI systems become larger, more complicated, and more highly concurrent, managing their very complexity becomes a major part of the design process. One method for taming their complexity is to use automatic methods for generating the circuits that comprise complex systems from some higher level description. This allows the designer to abstract away details of the low-level circuits and think of system behavior in higher level terms. This higher level description might be a set of Boolean expressions, a logic schematic, a behavioral description in terms of a special hardware description language (HDL), a behavioral description in terms of a standard high level programming language, or simply an algorithm in some specialized domain like signal processing. The main benefit of using any of these descriptions is to reduce the number of components that the designer must deal with, possibly by several orders of magnitude. Rather than wrestling with millions of transistors, or the colored polygons that represent those transistors in an integrated circuit layout, the designer may have to deal with only a few tens of larger architectural or behavioral elements.

Although there are benefits to using automatic translation methods from each of the different levels of abstraction, the highest leverage is achieved by using a behavioral description in terms of a programming language. Whether this language is designed to describe hardware systems, or based on some standard programming language, using a program as a description allows the system to be designed without detailed knowledge of the implementation technology by the designer. This allows complex hardware to be designed by the person who best understands what the system's function should be. This person is oftentimes a programmer and not a hardware designer.

If the translation process is correct, then the resulting circuit will be free from structural errors and will satisfy all the technology dependent design rules. The resulting circuits will faithfully mimic the behavior of the program and will be correct by construction. The design errors introduced by the designer on higher, more abstract, levels are much easier to detect and correct at the program level than at the circuit level. At the same time, if a programmer is to design efficient systems in this way, then there must be a way for the programmer to reason about the resulting circuit based on the program text. The translation process must be sufficiently transparent to give the programmer some idea of how different program alternatives will affect the compiled circuit.

In addition to making system design easier, a variety of formal proof techniques can be used to verify that the program meets the system specification. These proofs can be carried out at the level of the program description, before the design is translated into hardware. If the translation process preserves the semantics of the program description, the circuits are correct by construction and the resulting hardware will also meet the system specification.

Another advantage of translating programs into circuits is that programs written to be translated into a circuit may just as easily be compiled into object code and executed on ordinary computers. The system can then be simulated by running the compiled program like any other program. This software version of the system is easily instrumented with performance monitors that would be very difficult to include in the hardware simulation. The behavior of the software simulation can also be compared with the simulation of the resulting circuit at the switch or device level. Because the behavior of each representation of the system is guaranteed to be the same, the results of the different simulations should also be the same but can provide different types of information about the system. Whichever simulation method best supplies the desired information can be used.

The main drawbacks to designing hardware systems by writing program descriptions are in the inefficiencies of the translation process. Automatically generated circuit layouts will not be as dense as a custom circuit designed carefully by hand, and the performance of the generated circuit will most likely not be as high as the carefully tuned custom circuit. These drawbacks must be weighed against the main strengths of this design method, which include abstracting away large numbers of details, allowing programmers to design hardware systems, and the ability to generate new systems rapidly. Using program descriptions, many different designs for the same system can be evaluated quickly without casting the system into hardware. The overall design cycle is shortened so that, if the generated design is acceptable in terms of size and speed, the new design is completed in substantially less time.

1.2 Asynchronous Systems

The design of asynchronous or self-timed [82] circuits and systems has long been considered too difficult by all but a few researchers and engineers. One reason for this perception is that a long history of designing synchronous circuits has resulted in a great deal of knowledge and large variety of tools to aid in that style of circuit design. The same level of support for asynchronous design does not exist at present. Because there is no great demand for suitable circuits, semiconductor manufacturers do not offer commercially available parts that are suitable for asynchronous design. This only reinforces the tendency to design traditional synchronous systems.

In addition to being difficult to design, individual self-timed circuit modules are usually larger than their synchronous counterparts. Historically, because space on an integrated circuit is limited, the overhead involved in implementing self-timed circuits has been considered too great to consider that form of implementation.

Asynchronous design is, however, currently attracting renewed interest as a method for coping with some of the problems associated with improved VLSI technology. These problems arise from the increasing scale of modern VLSI, whether that scale measures the size of a die, the number of available transistors, or the speed of those transistors. The lack of commercially available parts can be largely overcome simply by designing the needed parts directly and the necessary structure of the circuits and the design process to ease self-timed design is becoming more clear. While the individual circuit modules may still be more difficult to design than their synchronous counterparts, the more serious problem of system integration can be made much easier.

One main reason that asynchronous design is attractive is that it separates timing from functionality. Timing has always been a major concern in traditional clocked synchronous circuits. As transistor sizes decrease, and system sizes increase, the problems involved in distributing a global clock signal to the entire circuit become severe. Self-timed circuits localize the timing information to individual circuits rather than synchronizing the entire system to a global clock and thereby avoid the standard problems of clock distribution and clock skew. Also, in a complicated pipelined system such as a processor, the necessity of balancing the worst case path through all the pipe stages requires analyzing literally thousands of paths and iterating a large number of times.

The greatest benefit of designing with self-timed parts may be composability. Systems may be constructed by connecting components and assembling subsystems based only on their functionality rather than having to consider their timing characteristics. Timing is now a separate issue related only to performance. Increasing the speed of the individual modules can improve performance without affecting correctness. Because timing is separated from functionality, large self-timed systems can be built from subsystems that operate at widely different speeds, or that will continue to operate over a wide range of voltage or temperature conditions. Increased composability of subsystems can also greatly decrease the time required to assemble a new system. Circuit modules can be thought of as a sort of "self-timed Lego" that can be snapped together easily to make whatever system is desired.

A benefit related to composability is incremental improvement. In a properly designed asynchronous system it is possible to improve the performance or functionality of a system by improving or replacing individual subsystems incrementally without changing or retiming the whole system. Incremental improvement of large synchronous systems is usually difficult, if it is possible at all. This notion of incremental improvement also extends to improvements in the process in which the circuit is fabricated. A large synchronous design is not likely to be able to take advantage of incremental improvements in the manufacturing process because these improvements may upset the timing relationships necessary for the synchronous system to operate correctly. A self-timed circuit, on the other hand, should not require any drastic changes to take advantage of improvements in the underlying implementation process.

Performance itself is another issue where asynchronous designs may be beneficial. Traditional synchronous systems usually exhibit worst case behavior while asynchronous systems tend to reflect the average case. Because the results from a piece of computation must be available at the next clock tick, the time between clock ticks in a synchronous system must reflect the worst case delay in any subcircuit. A self-timed system, on the other hand, starts computation when data are available, and signals that the result has been computed however long or short a time this takes. In systems where the latency through parts of the system depends on the data values, this can allow the system to run much faster than the worst case in most instances. The subsystem will slow down to the worst case speed only in the rare case where that delay is actually necessary.

Another possible benefit of self-timed systems that bears further investigation is lower power consumption. This is especially critical in portable and battery powered systems where power consumption is a limiting factor in their design. In current large synchronous designs, especially in systems designed for high performance, the clock distribution network

```
PROC buffer (CHAN A B)          – Define a simple buffer
  WHILE TRUE                    – Repeat forever
    VAR temp:                   – Define a temporary variable
    SEQ                         – Sequential composition
      A ? temp                  – Get value from Channel A, store in temp
      B ! temp                  – Send value in temp to channel B
```

Figure 1.1: OCCAM Code for a Simple One-Place Buffer

is beginning to require a significant portion of the power budget for the system. The power required to run a clock across a large system is substantial, and the power is used whenever the system is running, even if no actual work is being done. In a self-timed system there is no global clock. Timing is local and parts of the systems synchronize for communication only when needed. Parts of the system that are not active will not consume dynamic power.

Finally, as it becomes possible to integrate more and more onto a single integrated circuit, it becomes more attractive to think of the system as a collection of independent, cooperating systems rather than one large single system. This allows the designer to deal with large systems hierarchically. Each subsystem can be designed and tested independently with confidence that, because a self-timed communication protocol is adhered to, they will operate correctly when assembled into a larger system.

1.3 Translation from Programs to Asynchronous Circuits

This thesis presents a method for automatically translating a program into a circuit. The source language is a subset of OCCAM [42], a language based on CSP [37]. OCCAM describes computation as a set of concurrent processes that interact by communication over fixed links called channels. For example, an OCCAM process that implements a simple one place buffer is shown in Figure 1.1. A process like this might be combined in parallel with other buffer processes to implement a deeper or more complex buffer. This example will be examined in more detail in later chapters. The goal of the translation system is to take a description like this and automatically generate a circuit that implements the same behavior.

The target for the translation is an asynchronous circuit. In particular, the target circuits are delay-insensitive control circuits using two-phase transition signalling and bundled data paths. Translating the buffer example results in the simple circuit shown in Figure 1.2. The steps and circuits involved in translating this example will be shown in more detail in later chapters.

The circuit in Figure 1.2 is small enough for its operation to be described easily. Recall from the program in Figure 1.1 that the operation of this circuit is to pass data from the *A* channel to the *B* channel using one stage of buffering. In the circuit, the synchronization

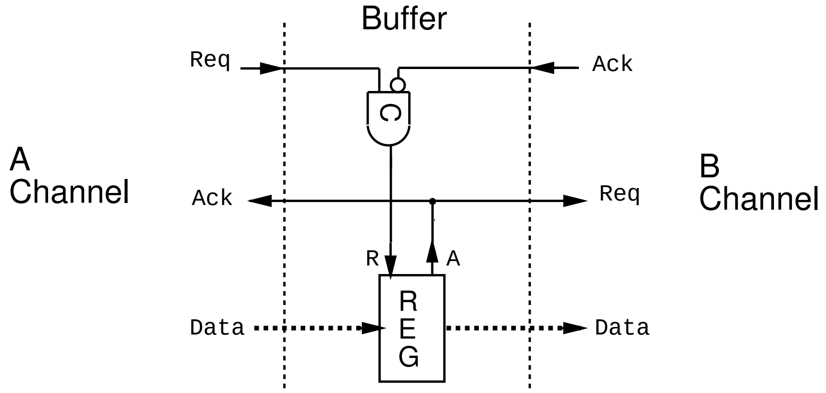

Figure 1.2: A Circuit for a Simple One-Place Buffer

of the input action and the output action is accomplished using a C-element. A C-element is a gate whose output is low until both of the inputs are driven high. At this point, the output of the C-element is driven high and remains high until both of the inputs return to a low voltage. The particular form of a C-element used in Figure 1.2 has an inversion on the right input which is connected to the *Ack* signal of the *B* channel.

After initializing all the wires in the circuit to a low state, the first action of the circuit is to accept data from the *A* channel. To signal this action the circuit connected to the *A* channel places data on the data wires, and makes a transition on the *A* channel *Req* wire. Because all the wires were initialized low, and there is an inversion on the right input of the C-element, this forces the output of the C-element to transition from low to high. This transition signals the register to latch the data present at its input. When the register responds with an acknowledge transition signifying that the data are latched, the *A* channel is acknowledged. Using the same signal the *B* channel is notified, through the *B* channel's *Req* wire, that the data are now available on the *B* channel. At this point the circuit connected to the *A* channel is free to send more data by changing the values on the data wires and making another transition on the *A* channel *Req* wire. The *B* channel may also accept the data currently held in the register by signalling, with a transition on the *B* channel *Ack* wire, that it is no longer necessary to hold the data in the register. Only when both of these events take place, involving transitions on both inputs of the C-element, will the C-element produce another transition to the register to latch more data from the *A* channel. This process repeats forever and implements the single stage of buffering described in the program of Figure 1.1.

A circuit is delay-insensitive if the correct operation of the circuit does not depend on any assumptions about delay in wires or operators of the circuit [100]. This property guarantees that any correctly functioning subcircuits may be composed together and continue to operate correctly. It also cleanly separates performance-related timing issues from functionality issues. Changes in relative delays of the system can affect performance, but not functionality.

This circuit, and others described in this thesis, use bundled data paths. The bundling constraint for the data paths is a condition similar to, but weaker than, the equipotential constraint [82]. A "bundle" of data wires is associated with a pair of control wires that indicate the validity of the value on the data bundle. The bundling constraint requires that the data bundle and the control wires be constructed such that the transitions on the data bundle are observed at the receiver before a signal appears on the control wire. Note that a set of wires with non-zero delay will not meet the equipotential constraint, but can meet the bundling constraint if the delay of the control wire is guaranteed to be longer than the longest delay in the data bundle. The use of bundled data paths in the target circuits is a convenience that allows these circuits to use existing sets of standard cells for implementation [95, 10]. These modules also lead to more compact circuits than if completely self-timed data paths were used. Self-timed data paths require multiple wires to represent each data bit, and, in general, require more circuitry to implement the same logic functions. The circuit in Figure 1.2, for example, uses two-phase transition signalling on the *Req* and *Ack* wires, and a bundled data path, shown as a dashed line, to the register. The bundling constraint in this case is that the data must be valid at the input to the register before the request signal arrives at the register.

In addition to translating programs into circuits that are delay-insensitive, using OCCAM as the system description language distinguishes this system from most other silicon compilers in yet another way: the user describes the behavior rather than the structure of the system. Most description languages for silicon compilers let the user describe, in high level terms, the circuit that is desired. Using the procedure described in this thesis the user describes the behavior that the desired system should exhibit, including any desired concurrency. Any circuit that behaves in the same way is acceptable and the system is free to transform the circuit in any way that preserves correctness. The system will not attempt to "extract" concurrency from non-concurrent code. The programmer is responsible for specifying any and all concurrency that the system should exhibit, but the system should be able to produce any circuit that has the specified behavior. The OCCAM code in Figure 1.1, for example, does not contain any information about how the process should be implemented. There are no geometric constraints and no explicit references to circuit modules. This code fragment shows the way an OCCAM programmer would implement buffering on a channel and not how a hardware designer would implement such a buffer.

The translation procedure presented here involves a simple syntax-directed translation from program constructs into initial asynchronous circuits. The resulting circuits are improved with correctness-preserving circuit-to-circuit transformations similar to peephole optimization in conventional compilers. The programmer is guaranteed that any specification met by the OCCAM program will also be met by the circuit. A system has been constructed to perform the translation automatically.

1.4 Overview of the Thesis

This thesis is organized with a chapter devoted to each of the main components of the translation system, followed by a description of an automatic compiler that implements the translation process. Each chapter includes a set of examples to illustrate the concepts described in that chapter.

Chapter 2 offers a review of previous work both in translating programs into circuits, and in asynchronous circuits themselves.

Chapter 3 describes the source language for the translation in detail. Specific omissions from the official version of OCCAM are noted and a new syntax for this version of OCCAM is presented along with examples of small processes written in this version of OCCAM.

Chapter 4 presents the target circuit modules used in assembling the translated system. A variety of different options for asynchronous modules are explored and the specific modules used by the prototype compiler are described in detail. Examples of circuits assembled from these modules are given at the end of the chapter. Circuit schematics for each of the target modules are described in Appendix A.

Chapter 5 describes the translation scheme that takes concurrent OCCAM programs and converts them into circuits. The initial circuit replacement for each of the language constructs is described followed by a description of the optimizing transformations applied to the resulting circuit. Two examples of applying this compilation procedure to the OCCAM examples from Chapter 3 are included to explain the process further.

Chapter 6 presents a prototype compiler that automatically applies the procedure presented in Chapter 5. Extensions of the source language for use with the compiler are presented along with examples of some larger circuits that have been compiled, simulated, and fabricated.

Chapter 7 attempts to evaluate some of the choices made in the implementation of the prototype compiler, and compares this work to the most similar of the related work.

Finally, Chapter 8 has a summary of the thesis and some suggestions for future work in this area.