Chapter 8

Conclusions

8.1 Summary

The goal of this thesis is to present and explore ideas to ease the complex task of designing VLSI systems. As VLSI technology continues to improve, the size and complexity of the systems being built will increase. The scale and speed of systems built using these technologies will require new ways of thinking about system design in order to reap the promised benefits. The system design method proposed here uses three main ideas as its foundation: that asynchronous circuits have many advantages over traditional synchronous circuits for building VLSI systems; that a VLSI system, in particular a system composed of self-timed modules, can be specified naturally as a concurrent program in an appropriate concurrent programming language; and that the program description can be compiled automatically into a reasonable circuit.

Asynchronous circuits are a natural solution to many of the problems facing VLSI system designers. Self-timed modules compose easily into larger self-timed structures and are convenient building blocks for VLSI system design. Systems built from composable parts can be improved incrementally, both by replacing components or by improving the underlying implementation technology. Because timing is separated from functionality, performance improvement does not require major retiming effort. With more and more of the power budget for large chips being used for clock distribution, a circuit style that does not use a global synchronizing clock can be very attractive.

Specifying the system as a concurrent program allows the designer to abstract away details of the low-level circuits and think of system behavior in higher level terms. In addition to drastically reducing the number of components that a designer must deal with, this allows the designer to focus on the problem instead of the implementation and to try out different design alternatives quickly. Because the specification is a concurrent program, formal proof techniques may be used to verify that the program meets the system specification, and the system may be simulated simply by running the program before the translation to a circuit has been applied. In this case, the language used to write concurrent programs is a dialect of OCCAM.

Asynchronous circuits are a natural match to the OCCAM style of expressing concurrent programs. OCCAM programs express concurrent and sequential aspects of computation explicitly and cooperate by communicating. This communication is purely in terms of message passing with no data shared amongst the concurrent processes. Communication also synchronizes the two participating processes. The processes and the pattern of communication between them are fixed at compile time and will not change as the programs run. This model of independent fixed processes operating at their own pace and synchronizing through communication fits exactly with the model of asynchronous modules cooperating through synchronizing handshake signals. In addition, OCCAM encourages the programmer to think hierarchically and incrementally. If submodules are to be designed independently and are expected to work together, then asynchronous circuits are again a good model for the implementation. Modules designed separately but with an agreed upon self-timed interface will work together without the timing effort common to synchronous assemblies.

Once the system is specified as a program to the satisfaction of the designer, the tedious process of translating that program into a circuit may proceed automatically. If the translation process is implemented correctly, the resulting circuit will faithfully mimic the behavior of the program and be correct by construction. Automating this procedure greatly reduces the chances that a simple clerical error on the part of the translator could cripple the resulting circuit. Furthermore, the correctness of the circuit translation should be provable. I believe that the construction of a semantics for OCCAM that is comparable to the semantics of the generated circuit should be possible. This would allow a proof that the translation process correctly preserves the semantics of the OCCAM program. To my knowledge, a proof of this nature has not been shown for any current circuit translation system, but should be possible.

Combining these ideas, a method has been presented in this thesis for translating a concurrent, communicating program written a dialect of OCCAM into an asynchronous circuit. The translation procedure involves a syntax-directed translation of the source code constructs into initial circuits followed by optimization of those circuits. This translation method has been embedded in a silicon compiler. The examples in this thesis, as well as many others, have been successfully compiled, and the resulting circuits have been simulated, using a switch-level simulator, to demonstrate their functionality. Some of the examples have been laid out on chips, using the MOSIS Fusion place-and-route service, and fabricated. The translation from source program all the way to returned chip is completely automated. Hand modification of the circuit is required only in a few cases where non-standard channel interfaces are required, as with the I/O data path used in the memory controller from Chapter 6 which conforms to the specification of the memory chips rather than OCCAM.

The silicon compiler described here suggests that a principal weakness of asynchronous systems, the difficulty of their design, can be largely overcome by compilation from a specification such as an OCCAM program. Such techniques may make asynchronous techniques more accessible and let designers exploit some of the benefits of the techniques, such as composition of components that operate at different speeds or incremental improvement of existing designs.

8.2 Future Work

There are many directions that future work based on this system could take. I believe that asynchronous circuits are an excellent match for large VLSI systems and the improved integrated circuit technology underlying those systems. As circuits get faster and chips get larger, structuring the desired system as cooperating subsystems running at their own rates is an attractive alternative to globally synchronous systems. The method presented here of capturing the behavior of the system in a concurrent program and translating that program into a circuit seems to be a natural and expressive way to design these sorts of systems. There are several ways in which the current silicon compiler can be improved, however.

8.2.1 Extending the Translation System

One way that his work could be extended is by focusing on the OCCAM source level. In particular, the current compiler applies optimizations only at the circuit level. There are substantial opportunities for optimization at the OCCAM source code level as well. Optimizations on the source code could be applied in much the same way as source-to-source optimization is applied to other programming languages but the transformations, such as channel splitting, would be different than those applied in normal compilers. There are a well defined set of algebraic laws for OCCAM [78] that could be used as a guide for source-to-source transformations of the OCCAM code.

Extending the translation system in a different direction could allow other description methods to use OCCAM as an intermediate language on the path to circuit implementation. OCCAM is used in this dissertation as essentially a hardware description language for a particular style of hardware. One of its strengths in this capacity is its simplicity. However, descriptions of large systems might require a higher level of abstraction. For example, OCCAM has inadequate provisions for talking about sharing system resources. Sharing of hardware resources must be detailed explicitly by the programmer. If multiple processes share some module like a multiplier for example, the details of how that multiplier is accessed by each of the processes must be specified by the OCCAM program. A system description at a higher level might simply describe the desired algorithm in terms of the operations done at each step. The Design Automation Assistant from CMU [48, 47] is an example of a system that uses this sort of algorithmic description. A compiler for a description like this might translate the algorithm into the appropriate OCCAM code as an intermediate step before circuit implementation. This would allow the algorithm compiler to make decisions about resource sharing and process decomposition in the OCCAM program.

Another avenue for extending the translation system is to integrate additional tools into the compiler's environment. The current compiler simply produces a netlist of modules and their connections. This netlist description may be used as the input to other tools, but they are not integrated into a unified environment. The design environment might include an interpreter for the OCCAM code, and a simulator for the synthesized circuit. These different forms of simulation could share a common driving program so that the same input could be applied to different versions of the system.

Integrating the place and route system to the environment would also be helpful. This could help the designer reason about timing in the parts of the system sensitive to delays. The current version of the compiler does nothing but flag the data paths so that the designer can check for violations of the bundling constraint. Closer ties with the place and route software could allow the system to influence the routing of timing-critical paths automatically. Even if this were not possible, information about delays after placement and routing is essential and should be available from the place and route software. The place and route software should be able to back-annotate timing information to the program or the circuit simulation.

8.2.2 Extending the Formal Basis

Another direction that future work could take is to put the translation and optimization procedure on a firm formal foundation. The semantics of OCCAM are well known and expressed in a variety of ways [14, 77, 78]. The semantics of the dialect of OCCAM used in this dissertation match those of the standard language with a only a few additions. For example, a suitable definition of the *Lex-Alt* should be added to whichever semantic description is used. A well understood semantics allows the programmer to use formal methods to prove properties of the program before it is committed to hardware.

Another formal property that could be addressed is the correctness of the translation method itself. If the translation process is proven to be correct then the resulting circuit will be correct by construction. What is required is that the circuit pieces be formally shown to have the same semantics as the OCCAM language constructs that they replace. One method for doing this would be to express the operational semantics of the language constructs in a way that they are comparable to the semantics of the circuit pieces. Circuits are naturally thought of in operational terms so the comparison could be relatively straightforward. Operational semantics exist for both CSP [75] and OCCAM [14] that could be used as starting points to this sort of investigation.

Denotational semantics provide a different way of reasoning about OCCAM program semantics based on the trace sets of OCCAM language constructs. There are well defined denotational semantics for both CSP [8, 7] and OCCAM [77]. The trace sets that these models use to reason about programs are similar to the trace sets used to describe target circuit modules in Chapter 4, but capture information about the system at a slightly higher level of abstraction. The symbols in the trace sets of the denotational models indicate atomic communication actions of the process with its environment. The elements of the trace sets used to model circuits are further broken down into the individual circuit events required to implement this communication.

One way of thinking about an OCCAM process is as a black box that performs some action and then terminates. A hardware implementation of an OCCAM process initiates activity on receipt of some start signal, performs the appropriate action, and then signals completion with a termination signal. Sequential composition of primitive processes, for example, is easily be implemented by wiring the termination signal of one component into the start signal of the next sequential process. A possible problem with comparing the mathematical model of OCCAM as defined by Roscoe [77] with this notion of a circuit is the implicit assumption in the denotational semantics that an OCCAM process is always ready to act. That is, if communication is the action of the process, then that process will always be ready to engage in an offered communication. In fact, the hardware model will be ready to communicate only after receipt of some initialization signal. Before that signal, the process will ignore the offered communication. This signal is represented as the *Req* signal to the hardware implementation of the communication action. After the communication has occurred, the hardware produces an *Ack* signal. Neither the *Req* nor the *Ack* appear in the trace set of the OCCAM source program.

One way to use this idea at the source code level is to imagine an OCCAM program where each program statement starts executing upon receipt of an initialization signal, and indicates that it is complete by producing some termination signal. This could be encoded in the trace set of the program by including special symbols to represent these control signals. For example, in addition to the symbols that represent communications events, an extra symbol is already used in the traces of the denotational model. This symbol, $\surd$, pronounced "tick", represents successful termination of a process. This symbol is not a part of the communication alphabet of the process and is used in trace sets to indicate that a process cannot refuse to terminate. As a possible extension of Roscoe's model I suggest that there are actually two kinds of events in the traces of processes: communication events and control events. The communication alphabet of a process is the set of communication events in which that process can participate. The control events enable the process to start participating in communications, and signal the termination of the process. These events might be noted $\surd_R$ for a request event, and $\surd_A$ to acknowledge termination of the process. These events can show up in the trace of a process, but are not a part of the communication alphabet. They also correspond directly to the start signal and termination signal of the hardware implementation of the process.

An interesting distinction between control and communication events is that while communication events require the cooperation of a pair of process running in parallel, control events do not require cooperation. Instead, control events require restraint on the part of the process producing the event. A $\surd_R$ request event is an input signal to the process that is produced by the environment and cannot be refused by the process. This is the same idea as the idea of *receptiveness* as defined by Dill [24]. Because the process may not refuse the $\surd_R$, it is the responsibility of the environment producing the $\surd_R$ to make sure that it produces only a single signal to start the process.

A $\surd_A$ event is produced by the process and sensed by the environment. Again, this does not require the cooperation of the process and the environment as a communication event would. The environment cannot refuse the $\surd_A$ and the process must generate only a single $\surd_A$ in any execution trace. Of course, many $\surd_A$ events may be produced given repeated invocation of a particular process, but only one $\surd_A$ may be produced from any particular instance of the process. It might be an interesting extension of the denotational model for OCCAM presented in [77] to add the explicit control events $\surd_R$ and $\surd_A$, and restrict the semantics to the subset of OCCAM used as a source language to translate into hardware.

Another method for reasoning formally about the correspondence between the OCCAM source code and the initial circuit translation would to model the control events directly in the source code as communications. These communications would then appear in the trace set of the program and be more directly comparable with the hardware implementation. The source code could be written to account for these events directly, or an existing program could be modified using source-to-source transformations. Each basic OCCAM process in a program could be expanded at the source code level into a process of its own. This process would be extended with two channels used to communicate the control events. For example, an input process in a program might be augmented with a *Req* channel and an *Ack* channel as shown in Figure 8.1. This source translation is syntax directed and will not change the overall behavior of the program. Its purpose is to elevate the control events to the status of communication events that show up in the trace sets of the program.

These ideas are merely sketches of possible extensions to the formal model and are not to be taken too literally. The reason for both of these proposed extensions to the model of the program is to allow a more formal characterization of the initial translation from the OCCAM program into a circuit. The idea is that the new model will still be useful for the application of this class of formal methods to designing and verifying programs, but will be more comparable to the hardware implementation of the OCCAM program.

8.2.3 Source Language Extensions

The version of OCCAM used in the system described in this dissertation could be extended in a number of ways. The simplest enhancements would extend the language to handle commonly used data constructs such as vectors and arrays. OCCAM also uses a syntax for replicating processes such as *Par* that is not supported in the current compiler.

Beyond these simple syntactic additions, there are a variety of additional behaviors that

```
(              ⋮
  (? Channel-A Variable-B)                  ; a primitive process
               ⋮)

;; This primitive process can be translated into the following:
(Block ((Chan Req Ack))                     ; define some new channels
  (While True                               ; repeat forever
    (? Req)                                 ; wait for a Req event
    (? Channel-A Variable-B)                ; execute the primitive process
    (! Ack)))                               ; produce an acknowledge
```

Figure 8.1: Method for Modeling Hardware Control Events in the Source Code

could be useful extensions to the current language. The *probe* signal described in Chapter 7 is a natural choice. As mentioned in that chapter, adding this behavior would be particularly simple because a slightly different wiring of the same modules is the only change that would be required of the generated circuits.

Another simple extension that would require very little change to the generated circuits would be to allow the use of output statements in guards as well as input statements. Currently OCCAM allows only input statements in guards to *Alt* constructs. There is no reason that output guards could not be implemented just as easily as input guards at the circuit level. Problems arise only if input and output guards are used on the same channel. Either type of guard may be used safely, but not both at the same time. While the addition of output guards would make it easier for the programmer to design a system that deadlocks, they can allow simpler descriptions of some situations if used carefully.

An extension that would require more serious changes but could be very useful is be to add the concept of a bus as a type of communication channel. Many systems use a shared bus of some sort to pass data from a single sender to any of multiple receivers. OCCAM channels, however, are defined to be point-to-point channels between only two processes. Extending a channel to a bus entails thinking about how to arbitrate for use of that bus so that only one process has control of the bus at any time. Currently, a process to mimic the operation of a bus could be written that accepts input from a number of processes, and sends that information to whichever process it is intended for, but this is a somewhat clumsy approach and would result in a rather large circuit.

Burns [12] suggests a number of extensions in his language that, because of the fundamental similarity of his source language to OCCAM, also make sense for my system. His suggestion that data can actually flow in both directions during a synchronization could be very useful to some circuits. It points out that the communication and synchronization actions of the channel need not be as tightly coupled as they are in OCCAM. Some synchronizations require no additional communication, and others can pass data in addition to synchronizing. The *probe* can be viewed as another way of separating the synchronization from the communication. It can report that a synchronization is pending without executing the associated communication. In Burns' compiler a *probe* is extended to test the data value being transmitted in addition to the readiness of a channel.

The unary data encoding that Burns uses for communication channels, while sometimes inefficient, is very useful in special cases. It might be possible, for example, for the OCCAM compiler to discover that a channel is being used for this type of communication and generate the appropriate circuits. It could also be supported through special syntax in the OCCAM code.

Burns also differentiates between *Alt* constructs that require arbitration, and those that do not. If the user can guarantee that the guards will not be changing during the time that the *Alt* circuit is checking them, the expensive arbitration circuit need not be generated. This is another very useful language feature that could be supported with a simple addition to the version of OCCAM used here.

8.2.4 Target Language Extensions

Peering down to the lowest level of the generated circuits, perhaps the major shortcoming of this whole system as it exists today is in the layout that is produced for fabrication. The current system produces a netlist which is then converted to layout with a place and route tool like the MOSIS Fusion system. This results in a correctly connected layout, but is not comparable in quality to careful hand layout even using the same cell set. One problem is that although the OCCAM program uses hierarchy to partition the system design, this hierarchy is lost when the netlist is produced. Retaining the hierarchical nature of the source program through to the placement of components is one way to increase the quality of the overall layout.

In addition to keeping information about the hierarchy in the circuit, information about the timing constraints of the bundled data paths should also be passed to the place and route software through the netlist. The software should use these constraints to guide the placement and routing process. The current system simply flags the bundled data paths to alert the designer to look for potential timing problems but passes no information to the place and route software about particular timing constraints of the circuit. Timing information should also flow the other direction in the form of back-annotation of projected delays from the place and route software into the design software.

Another shortcoming related to layout is seen in the switch level simulation of the generated circuit. The delay model used in the switch level simulation does not currently take the layout into account. This is especially critical when bundled data paths are used.

A natural addition to this system would be to back-annotate the netlist with delay values after placement and routing of the physical cells. This would also allow a timing verifier to perform a static check that the bundling constraints are met after layout.

Moving back to the level of the netlist itself, the most important limitation on the performance of the generated circuits is the time required for data delivery, i.e., to establish that the outputs of a register have propagated through all combinatorial circuitry connected to that register. Flow analysis techniques applied to the OCCAM program or to the resulting circuit can establish that some data-delivery delays are not required. For example, if two registers feed values to an adder, and both registers are changed, it is not necessary to wait for the propagation of each register's data separately, but only for the last data changes to become valid. The optimizer can also insert delays in the control paths, so that in so far as possible, data-delivery delay is overlapped by control delay. A good example of this technique is seen in micropipelines [94].

A mentioned in the Chapter 7, improving the data-delivery performance requires reasoning about delays only in local regions: each OCCAM process can be compiled independently of others. So the requirements of one part of a design do not constrain other parts of the design beyond its immediate neighbors. In particular, timing constraints at one end of a communication channel can be independent of those at the other end. No matter what approach is used to satisfy local timing constraint, the silicon compiler can produce estimates of the resulting performance.

A number of variations are possible in the target language for the translation of OCCAM programs. For example, rather than using the module library, an OCCAM process could be implemented as a Q-module [79]. A Q-module implements a state machine in much the same way as a traditional clocked state machine. However, rather than a standard clocked register, the state-holding register is built using Q-flops. These Q-flops sample their inputs on request and report when the value at their output is stable with an acknowledge. A simple delay between the request and acknowledge which models the delay through the next-state logic is sufficient to generate a local "clock" for the Q-module. This local clock will stretch to account for possible metastable states in the Q-flops caused by asynchronous inputs to the state machine. This is an attractive alternative circuit choice because of the similarity with traditional synchronous state machines. There are a large variety of tools available to help in the design of such state machines that could be used to help generate such an implementation. This circuit choice could also be more natural for many hardware designers to think about because it is closer to the circuits that they are used to designing.

Control signalling might use four-phase rather than transition conventions. One advantage is that four-phase data manipulation circuits sometimes have simpler implementations than their two-phase counterparts. These circuits could be used at the expense of the extra handshaking transitions. These extra return-to-zero transitions can often be overlapped with the following computation and thus not add any extra time to the overall system operation. The conditions under which the extra transitions may be overlapped, and how to rearrange the handshaking signals is quite complicated and must be done carefully so as not to introduce errors into the transformed circuit. Determining where the different signalling schemes are best used is a subject for future study. It might make sense, for example, to use smaller four-phase circuits internal to a chip where transition times are relatively fast, and use two-phase signalling at the chip boundary where the necessity of driving large capacitive loads slows down individual transitions.

Fully self-timed data, rather than bundling conventions, could be used. Using bundled data in the current module library was a simplification that allowed the use of standard datapath modules such as adders and logic generators. Dual-rail or other self-timed data delivery techniques could be used to eliminate the messy details of delay associated with bundled data paths. The trade off in this case is that self-timed data protocols use multiple wires per bit. Again, determining where the different schemes make the best sense should be investigated. Dual-rail data delivery might make sense internal to the chip at the expense of extra wires. These data might be converted to a bundled form at the chip boundary where input/output pins are a scarce commodity.

8.3 Final Words

Despite their many advantages, asynchronous and self-timed systems are not widely used at present. Synchronous systems can make use of a huge number of design tools, and can build on a large base of knowledge to ease the design process. These are currently not present for asynchronous designs. However, I believe the the improvement of implementation technology along with the continued quest for higher and higher system performance will eventually make asynchronous and self-timed systems more attractive to system designers.

Each time advances in technology threaten to make large synchronous systems too difficult to build, clever solutions are found to continue the development of this standard form of system design. Even with these clever solutions, however, the fine tuning of large synchronous systems is beginning to dominate the time to complete a large system. Clock distribution is requiring more and more of the design time and power budget of the system, and the process of balancing the worst case delay through all the parts of a large system requires extremely complicated analysis and simulation. As designers move to increasingly exotic high-performance technologies like sub micron CMOS, GaAs MESFET and Hetrojunction Bipolar Transistor (HBT) integrated circuits this problem promises to become even worse. The problems continue to worsen as systems are built using Multichip Modules (MCMs) to increase packing density and decrease interconnect delays in further efforts to improve system performance. Dealing with clock skew across a single integrated circuit is difficult. Dealing with clock skew across a Multichip Module is even more difficult.

Already, in the area of clock distribution, we are seeing complex schemes such as local phase-locked loops being used, with more exotic ideas such as optical clock distribution and superconducting interconnects being active research areas. In contrast, self-timed systems will let designers localize these low-level timing issues such that they are easier to deal with. This type of design does *not* ignore performance issues. On the contrary, it becomes much easier for the designer to consider performance, since the design may be changed to gain speed without the concern that it may cause the circuit to fail. In a synchronous design, any change to increase performance means analyzing possibly thousands of paths in order to ensure that the circuit will still function correctly. In a self-timed system this problem does not exist since functionality is separated from timing.

The circuits described in this dissertation are self-timed at a very low level. Although I think this is an interesting type of circuit to study, it is not clear that this is the best level for asynchrony in the design of large systems. Individual asynchronous circuits may be larger or more complicated than their synchronous counterparts and asynchrony at this low level may overwhelm a large system. An interesting area for further investigation is to study the overhead and the benefits to help define the level at which asynchrony is best applied in a large system. I believe that what *is* clear is that the future of large high-performance systems will definitely involve asynchrony at some level. The work presented here is one step along the path to discover the techniques that will be used in the design of these systems.