Chapter 7

Evaluation and Comparison

The immediate goal of the work presented in this dissertation is to describe VLSI systems in terms of concurrent programs and automatically translate those descriptions into asynchronous circuits with the same behavior. With such a general goal in mind there are obviously a huge number of different choices to make on how best to achieve that goal. Different choices of source language, special language features, translation method, target circuit structure, circuit signalling protocol, data delivery details, optimization technique, and degree of asynchrony of the resulting circuit, just to name a few, can make dramatic differences to the completed system. This chapter will explore some of the choices made in the system described here, and attempt to provide some post-implementation perspective on those choices. It will also compare the system presented here with other compilers that have the same general goal [68, 12]. In particular, the chapter will make a comparison with one of the most successful of the competing systems: the compiler built by Burns [12] that implements the synthesis technique of Burns and Martin [13, 57].

7.1 Choice of Source Language

Although there are many programming languages that allow the programmer to express concurrency, and a variety of methods used to actually express that concurrency, all of the silicon compilers of which I am aware and that produce asynchronous circuits use a language based on CSP as their source language. CSP and its descendants such as OCCAM share certain properties that make them especially suitable for describing systems that are intended for eventual hardware implementation and for asynchronous hardware in particular. These properties include the following:

- **Description of Concurrency** Circuits are inherently concurrent. A source language must be able to describe the concurrent activities desired in the implementation. CSP-based languages were designed specifically to express concurrency.

- **Control of Concurrency** Concurrency in a circuit requires replicated hardware. The explicit control over the concurrent and sequential aspects of the system offered by these languages allows the programmer to trade increased processing power of concurrent execution against the added cost of replicated hardware.

- **Message Passing Communication Model** Rather than assume a shared memory system, CSP processes communicate by passing messages along specific communication channels. This model of communication maps well to circuits, and especially well to asynchronous circuits that pass messages to synchronize local processes. It also avoids the complications of imposing noninterference or disjointness requirements on memory references.

- **Control of Communication** Communication is a primitive building block for message-based languages like CSP. Communication is also an expensive resource in a hardware implementation. Direct control of communication allows the programmer to control the communication costs in the eventual circuit by modifying the source program.

- **Unbuffered Communication** CSP assumes no inherent buffering of the communication channels. Because these channels will eventually be implemented as wires, this matches the behavior of the eventual circuit. Any desired buffering on the channels may easily be expressed using a buffer process connected to that channel.

- **Static Connection Topology** Communication channels in CSP connect exactly two processes and are fixed at compile time. This is a good match for the hardware system where those channels are implemented with physical wires connecting different parts of the system. It can also be a limitation if a more general communication structure, such as a bus, is desired.

- **Simplicity** CSP is a very simple language and its descendants have retained this simplicity. Simple language constructs have a more direct mapping to the circuit implementation.

- **Well Defined Semantics** CSP-based languages have been used extensively for theoretical investigations into concurrency. This allows system designers to reason about the properties of their systems before committing the system to hardware.

7.1.1 OCCAM as a Hardware Description Language

The translation method described in this dissertation uses a subset of OCCAM as its source language. OCCAM is a language directly descended descended from CSP and designed to run on Inmos Transputers [43]. OCCAM offers some extensions to CSP and a slightly different syntax. As such, the choice between OCCAM and CSP as a source language is largely a matter of personal taste. One reason that OCCAM is used rather than CSP is simply the difference in syntax. OCCAM programs look much like standard programs written in a conventional language with the addition of a method for expressing concurrent activity. OCCAM also uses no special characters beyond those found on a standard keyboard. CSP syntax includes a variety of special characters that, while easy to write, are not as easy to type. OCCAM is also a "real" language in the sense that there are a variety of compilers that support OCCAM, and hardware, most notably the Transputer, specifically designed to run OCCAM programs.

As OCCAM shares the properties of CSP-based languages, it appears that it is an excellent way to describe asynchronous systems, especially pipelined data processors. The circuits that were designed by hand using the asynchronous modules previous to building this compiler can be expressed simply in OCCAM. The OCCAM programs are clear, easy to manipulate at the source level, and easy to simulate by compiling and executing with a conventional OCCAM compiler. The OCCAM style of programming with collections of small concurrent processes mirrors exactly the style of asynchronous system design I have been exploring in hardware and the programs are translated into circuits that, after optimization, are remarkably similar to the circuits designed by hand to implement the same functions. In many cases the resulting circuits are actually identical to the best circuit designed by hand.

The OCCAM representation of a system provides many opportunities for manipulation. Large processes may be split into smaller processes, or multiple smaller processes may be combined together as the situation dictates. Multiple channels can be multiplexed into one with the aid of an OCCAM process to do the multiplexing, and then demultiplexed with another process. If the original channels link processes that will end up on different chips, the multiplexed channel may be desirable because it will reduce the chip's pin count. An internal channel might, on the other hand, be implemented as a single wide channel rather than a narrower channel used multiple times. One of the great strengths of this method is that different system design alternatives can be explored by modifying the OCCAM source code.

While any program written in this dialect of OCCAM can be translated into a circuit, different versions of the code will, of course, result in different circuits. These circuits may be optimized for speed, or area, or some other metric of quality by modifying the program as long as the relationship between the program and circuit is well understood. The translation process must be sufficiently transparent that the programmer has some idea of how the resulting circuit is produced. With a little experience there is a clear relationship between the OCCAM programs and the circuits resulting from this compiler, thus the programmer or system designer can get a good idea how program modifications will affect the performance or size of the resulting circuit. As with software compilers, correct programs will result in working systems, but programs written carefully with the target implementation in mind will result in better system implementations.

7.1.2 Limitations of OCCAM

While OCCAM is an excellent description language for this type of circuit, this is not the application for which the language was originally designed. Parts of the semantics of OCCAM limit the type of circuits that can be described easily. OCCAM channels, for example, are an extremely convenient way to describe the type of synchronized message passing that is implemented in pipelined self-timed circuits but may limit other types of communication because of their behavior. OCCAM semantics require that if data are passed along a channel, a register must be used to store the value received as input from that channel. This is a natural circuit to use to implement a pipelined system. For example, the branching and merging FIFOs shown in Figures 4.30 and 4.31 are pipelined structures expressed naturally in OCCAM. However, because of the requirement that a register be used, a FIFO merge that contains no pipeline stage cannot be expressed as an OCCAM program. A circuit is easily imagined that merges two FIFO streams by choosing an input channel and then passing the data on to the output channel before acknowledging that input channel. In this case no extra register is used at the expense of losing a pipeline stage; the input channel is blocked until the output channel acknowledges. Because OCCAM has no provision for an input channel that does not store the data received in a register, this behavior cannot be expressed in OCCAM.

Another problem related to the communication semantics of OCCAM involves an input statement used in the guard of an alternative command. In this case, if the guard is chosen the input is completed before the body of the alternative is executed. Because the system presented in this thesis uses this definition of an *Alt* statement, it is not possible to schedule any action between when the guard is chosen and the input is completed. For example, a desired behavior might be that an alternative is chosen based on which input channel has data available, but those data are not accepted from the input channel until some other action has been taken. This behavior requires a signal that indicates the availability of the data but that takes no action regarding that communication. Martin calls this signal a *probe* [56] of the input channel. OCCAM semantics do not provide for this separation of a guard and the associated communication action.

Note that this is a limitation of the semantics of OCCAM and not the circuits that can be built. There is no reason why the circuits generated by the method presented in this thesis could not implement the behavior of a *probe*. In fact, the generated circuits could behave in this manner with no change to the circuit elements that are generated. A small wiring change is all that would be required. Because the semantics of OCCAM are followed, this behavior is not supported directly by an *Alt* statement. However, this behavior can be expressed, albeit somewhat awkwardly, by using a separate channel to indicate the availability of data on the input channel. This can be followed by the alternative body, and finally, by an input from that channel to receive the data.

One of the reasons that OCCAM programs map so well to asynchronous hardware is that communication is implemented by passing messages along dedicated channels. The point-to-point connection of OCCAM communication channels is a good match for the generated circuits. Unfortunately, many VLSI systems are more easily expressed using a richer form of communication. One-to-many communication where a single sender broadcasts data to many receivers, or many-to-many communication where multiple senders compete for the same communication channel, or bus, are possible behaviors that are not directly supported by OCCAM. These behaviors may be implemented by writing an OCCAM process to distribute the message to multiple receivers or to arbitrate between multiple senders, but bus-style communication is not supported directly.

These examples point out some of the opportunities for extending the current version of the compiler. None of the limitations are fundamental to the method itself. While this version of the compiler uses standard OCCAM semantics, additions to the semantics to allow other behaviors could be implemented and are one avenue of extension for this work. Some of these extensions are discussed in the future work section of the next chapter.

7.2 Choice of Signalling Protocol

The main choice to be made for a circuit signalling protocol in a self-timed system is between two-phase and four-phase signalling. Each protocol has advantages in certain situations. Descriptions of these protocols can be found in Chapter 4.

- Two-phase protocol advantages
  - Two-phase signalling can be faster than four-phase because it uses half the signal transitions for a given transaction. The return-to-zero phase of the four-phase handshake usually carries no information and simply returns the wires to their initial state before the following transaction.
  - Control matches programmer intuition. Two-phase transactions can be thought of as passing a token of control from one software process to the other.
  - Two-phase control is also easy to follow on a circuit schematic. The "token of control" is easily followed on its path through the circuit because no return information, such as the return-to-zero phase of the handshake, is sent until the token is passed back.
  - Two-phase implements some constructs, like *Seq*, much more efficiently. Four-phase circuits require active circuitry to implement a sequential composition. If optimizations are done on the four-phase handshake, a choice is required of how to interleave the return-to-zero transitions with the next transaction. The choice of interleaving strategy is particularly hard to make automatically and may require human intervention.

  - Optimizations are simpler because they also do not involve reshuffling the handshake. Four-phase optimizations are complicated by possible reshuffling of the four-phase handshake and by possible interleaving of the return-to-zero phase with the next transaction.

- Four-phase advantages

  - Four-phase circuits tend to use AND and OR gates where two-phase circuits use XOR gates. If, as is often the case, the complexity of XOR is greater than other gates in a particular implementation technology then four-phase circuits may use fewer transistors.
  - Four-phase signalling fits more naturally with some logic families. Precharged logic styles such as CVSL [33], or the carry completion sensing adder from Chapter 4, for example, can use the return-to-zero phase to precharge the circuit for the next evaluation.
  - Four-phase modules may be simpler circuits in some cases because two-phase modules may need to store state to allow actions on both rising and falling edges of control signals.
  - In many cases four phase signalling is as fast as two-phase because the troublesome return-to-zero phase of the handshake may be completely overlapped with the following transaction and not require extra time.
  - Four-phase circuits mesh well with traditional four-phase dual-rail data delivery techniques.

7.2.1 Examples

There is no reason why the target circuit library presented in Chapter 4 could not be implemented using a four-phase signalling scheme. The main reason for selecting two-phase signalling for my work is its simplicity. In general there is a lot of simplicity in the two-phase world and quite a bit of complexity hiding in the four-phase world if optimization of the four-phase circuits is desired. Most of the complexity of four-phase signalling is directly related to optimizing the return-to-zero phase. Two-phase circuit modules are, in some cases, more difficult to design, but once they are designed they may be connected into circuits in a much simpler way than the corresponding four-phase modules. The following are simple examples of two circuit modules implemented using both two-phase and four-phase signalling protocols.

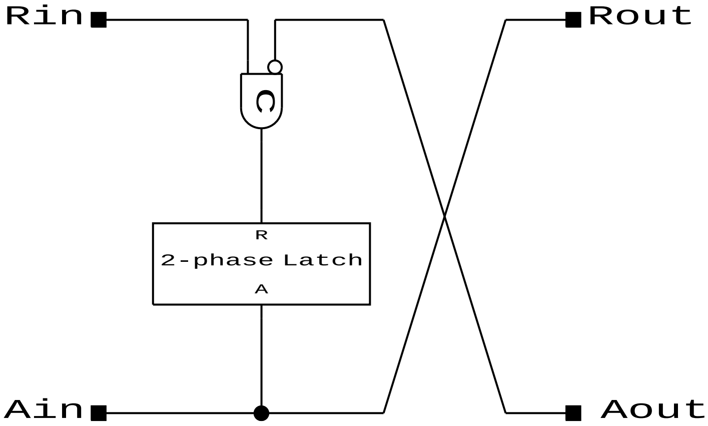

Figure 7.1: Two-Phase FIFO Circuit

FIFO Control Circuit

The first example is the control circuitry for a simple FIFO. The two-phase circuit is shown earlier in this dissertation and repeated in Figure 7.1. In this circuit a single C-element is used to control the sequencing between the FIFO stages. The generic process shown in the figure may be a simple latch to implement a FIFO buffer, or some more complicated action. A pipelined arithmetic unit, for example, might execute part of the computation in each process stage of the FIFO [94]. Whatever the process is, the interface is also uses two-phase signalling. The latch of a simple two-phase FIFO, for example, is a transition latch that stores a new value on each transition of the control signals.

The equivalent four-phase FIFO circuit uses a four-phase latch. This is a standard gated latch that is transparent when the gate signal is high, and latches that signal when the gate signal goes low. The four-phase control circuit that surrounds this latch depends on which four-phase protocol is used for passing data. Recall from Chapter 4 that there are two standard four-phase protocols: narrow and broad. These protocols define the portion of the four-phase handshake in which data are required to be stable. These protocols are shown in Chapter 4 and repeated here as Figure 7.2.

A circuit for a four-phase FIFO using a gated latch and broad data passing conventions is shown in Figure 7.3 [67, 29]. Note that two C-elements are now required to implement the four-phase handshake between the FIFO stages. This is not too surprising if a four-phase handshake is viewed as two two-phase handshakes executed in sequence. There is also an AND gate to ensure that new data are not allowed to pass through the latch until the final falling of the output acknowledge *Aout*.

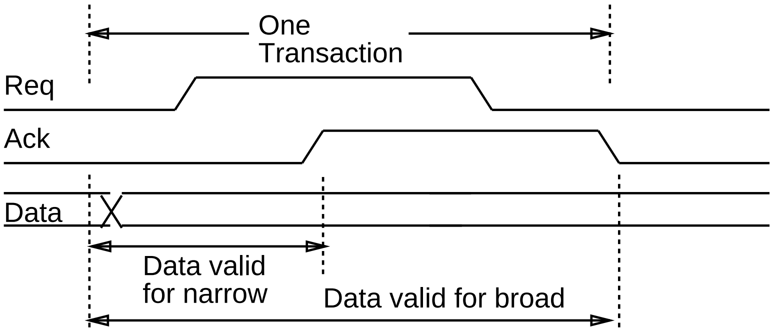

Figure 7.2: Bundled Four-Phase Signaling

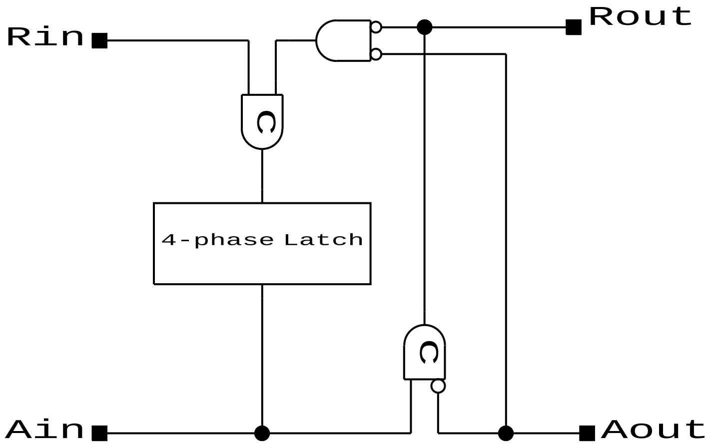

Figure 7.3: Four-Phase FIFO Circuit

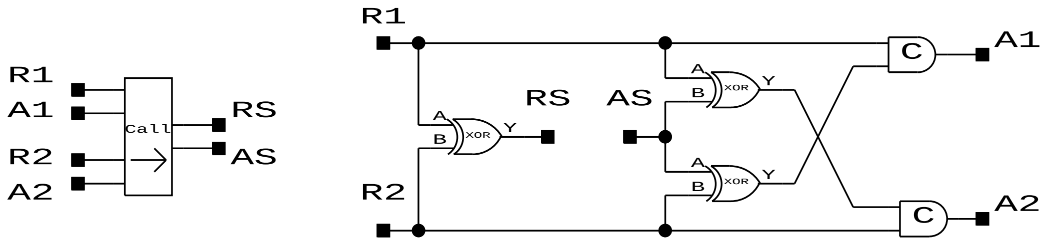

Figure 7.4: A Two-Phase Call Module

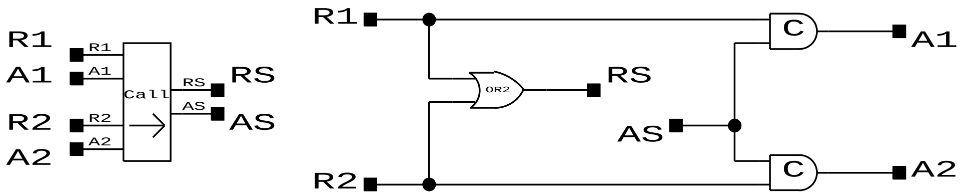

Figure 7.5: A Four-Phase Call Module

A Call Module

As another example consider the Call module. The two-phase version can be implemented as shown in the Appendix as Figure A.12 and repeated here as Figure 7.4. A Call module remembers who made a request by using the storage inherent in the C-elements. When a request is made on *R1* or *R2*, a transition is generated on the *Rs* subroutine request, and one input of the appropriate *A1* or *A2* C-element is changed. When the subroutine acknowledge *As* occurs, it changes the other input of the C-element to match, thus causing a transition on the correct acknowledge signal. The subroutine acknowledge also resets the other C-element to the state it was in before the transaction. The Call module is now ready to accept another request on either *R1* or *R2*.

In this case the four-phase version shown in Figure 7.5 is actually simpler than the two-phase implementation. The four-phase version can use an OR gate to generate the *Rs* signal rather than an XOR because the four-phase protocol requires *Rs* to return to zero before the next transaction. Likewise the extra XOR gates in the two-phase circuit that reset the C-elements are not needed because all the signals return to zero at the end of the four-phase cycle.

Note, however, that this version of a four-phase Call module essentially follows the broad convention defined in Chapter 4. That is, the module is still busy as long as either signal in an input control pair is high. Like the two-phase version, the input requests to this module may not be active at the same time, but in the four-phase version a possible

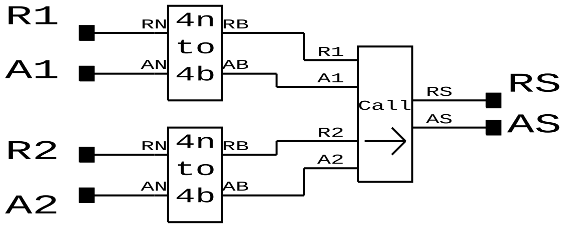

Figure 7.6: A Four-Phase Narrow Call Module

optimization is to overlap the return-to-zero phase of one client with the request phase of the other. In this version of the four-phase Call module neither the request phase nor the return-to-zero phase of one client may be overlapped with those of the other client. In particular, if client 1 has made a request, you cannot raise *R2* until *A1* is low.

This is unfortunate because it means that you cannot simply connect the acknowledge of one of the client interfaces to drive the request of the other. This is a connection that is used frequently with the two-phase module to call the subroutine twice in succession. For example, imagine that *A1* is connected to *R2* of the two-phase Call. After a request transition on *R1*, the subroutine finishes and produces a transition on *A1*. This causes the subroutine to execute again and subsequently produce a transition on *A2*. Because of the extra transitions of the four-phase handshake, this connection is not possible using the four-phase Call element.

A Call module that uses a narrow convention for the clients would solve this problem. One solution is to use a circuit that converts from the narrow convention that is desired into the broad convention used by the four-phase Call module. Appending these modules to the inputs of the four-phase Call module, as shown in Figure 7.6, allows the call module to be called using narrow conventions. This will, among other things, allow the Call module to be used sequentially to perform the subroutine multiple times in succession.

One circuit that implements this conversion is shown in Figure 7.7. The Inet for this circuit was shown in Chapter 4 as Figure 4.20. Its purpose is to embed an entire four-phase transaction, including the return-to-zero, within the active part of a narrow cycle and thus converts the narrow convention on the left to a broad convention on the right. This circuit is also exactly the same as the module called a “Q-element” by Burns [12]. This element, and others like it, are used by Burns to implement sequencing of four-phase elements. This is one way of viewing the purpose of the narrow to broad converter here: a means of sequencing actions to the Call module. There are a number of variants of these sequencing elements used in four-phase circuits depending on the desired degree of overlap between the two handshakes. In each case, active circuitry is required to implement simple sequencing of four-phase processes. When these modules are included in the implementation of the Call module, the four-phase version becomes considerably larger than the two-phase Call module of Figure 7.4.

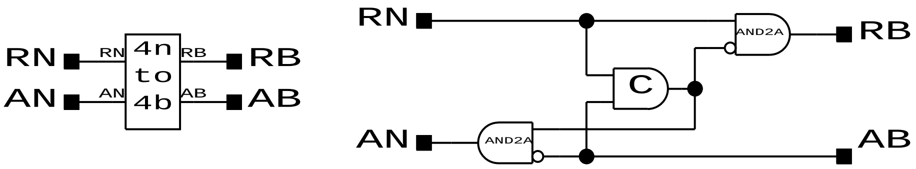

Figure 7.7: Four-Phase Narrow to Four-Phase Broad Converter

7.3 Choice of Data Delivery Method

As with the signalling protocols, there are a variety of possible schemes that can be used to deliver data from one process to the next. The main choice is whether to use delay-insensitive techniques such as dual-rail signalling or to use a simpler scheme such as bundling that requires reasoning about delays in signal paths. The properties of the main choices are summarized below:

- Delay-Insensitive Data Delivery
  - The safest methods because they do not require reasoning about delays in signal or data wires.
  - Expensive in terms of wires. Data bits are encoded on multiple wires. Standard schemes range from four wires per bit to two wires per bit plus one for the acknowledge.
  - Requires either that new data-manipulation modules, such as adders, be built that use data encoded in this form, or that data are converted back and forth to a bundled form.
  - Different schemes may use either two-phase or four-phase signalling.
- Bundled Data Delivery
  - Requires reasoning about relative delays in the data path and control path of the bundle. The bundling constraint is similar to, but weaker than, equipotential and isochronic constraints.
  - Economical in terms of wires. Bundling uses only a single wire per data bit plus two wires for control.
  - Can use standard data-manipulation modules such as adders and combinational logic.

– Different schemes may use either two-phase or four-phase signalling for the control wires associated with the data bundle.

As with the choice of signalling protocol, my choice of data delivery method is guided by simplicity. Using a bundled data protocol allows the use of standard data manipulation modules such as adders, multipliers, and logic generators because the data bundle is a collection of standard data bits. Standard tools may be used to generate combinational logic blocks as long as the maximum delay through the block can be computed. The cost of wiring is reduced dramatically because a single wire is used to encode each data bit rather than the multiple wires required in delay-insensitive protocols.

The main drawback to bundling is the necessity of modeling delays to guarantee that the bundling constraint is met. Recall that the bundling constraint requires the value on the data bundle to be valid at the receiver before the request transition is seen by the receiver. This requires that the longest path in the data bundle be no greater than the path of the control signal. In practice this is a relatively simple constraint to meet because the control signal almost always traverses more logic than the data wires and thus has longer delay. In cases where this cannot be guaranteed it is a simple matter to delay the control signal sufficiently to meet the constraint.

It may seem odious to deal with delays, because one of the reasons for using self-timed techniques is to avoid timing constraints. However, 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.

Even though the delay constraints are local, the current version of the compiler does not automatically model the bundling delay of the data paths in the circuit. This is another avenue for improvement of the current system. Instead, the compiler counts on the fact that the control paths are slower than the data paths in the current library of cells. By building the circuit only out of predefined library cells, certain assumptions can be made about the behavior at the interfaces of the cells. It is also the case that the control path usually includes some active circuitry, whereas the data path is often simply a metal wire which can propagate a signal very quickly. Many aspects of the physical circuits combine to make it a rare case where extra delay must be explicitly added to a control path to satisfy the bundling constraint. Nevertheless, extended versions of the compiler should include more accurate models of data delivery delay, and of the delay in the control paths of the circuit. This should also take physical layout into account.

7.4 Comparison with Related Work

Burns [12] has built and reported on a compiler that has exactly the same goal as the work presented in this thesis: to automatically translate a concurrent program into an asynchronous circuit. However, the details of his system, and the translation and optimization methods that he employs, are quite different from those presented here. His compiler is based on a circuit synthesis technique due to Martin and Burns [13, 57, 55]. The main differences between the two compilers are summarized below:

- **Source Language** Rather than OCCAM, Burns' compiler uses as its source the same CSP-like language used by Martin. To this base language they have added several extensions such as the *probe* signal to detect activity on a channel, and declarations to define which process initiates communication on a particular channel. Other than simple syntactic differences however, and the addition of the *probe* signal, the choice of CPS versus OCCAM is largely a matter of taste. The two languages are extremely similar. OCCAM may have a slight advantage in its ability to deal with data types other than Boolean variables, and the availability of a few existing tools to help the programmer.

- **Data Handling** Data in Burns' source language are restricted to Boolean variables that may be set and reset. My system allows variables of arbitrary width.

- **Communication** Burns' communication channels are asymmetric because they must be declared *passive* or *active* depending on which process initiates the communication. Information is potentially passed in both directions during a communication so the channel declaration does not effect the direction of information flow, only control flow.

  Channel control is symmetric in the method presented in this dissertation. The use of an input or output process simply determines the direction of information flow. If no data are passed, input and output reduce to synchronization and are symmetric. The exception to this symmetry is in guards. The version of OCCAM used here requires that channels used in guards must be input channels.

- **Communicated Values** Processes compiled by Burns' compiler may share data by sending values from a small finite set during a synchronization. These values are passed using unary encoding in the eventual circuit with a separate wire for each possible value. This can be a disadvantage if many values are passed, but can be quite compact in some cases. In contrast, values in the system described here are transmitted across channels as data bundles of arbitrary width.

- **Synthesis technique** The translation from program to circuit presented in this dissertation is completely syntax directed. The translation method used in Burns' compiler, while mostly syntax directed, is quite different. Rather than directly replace syntactic units of the source program with circuit replacements, Burns' and Martin's method proceeds in a number of steps.

- **Process Decomposition** The original CSP program is decomposed and expanded into a simple standard form where larger constructs are divided up into a series of sub-processes that each implement only a single primitive construct. These sub-processes are activated by communication over new channels that are inserted between the old process and the new sub-process.
- **Handshaking Expansion** After the original code is expanded and decomposed into the new form, communication actions on channels are expanded into the series of transitions that are required by the four-phase handshake. The wires that carry the handshake transitions are assigned to variables and it is the setting and resetting of these variables that actually implement the four-phase handshake.
- **Handshake Reshuffling** For added efficiency in some cases, the transitions of the four-phase handshake are moved to an earlier or later point in the handshake sequence. This optimization may, in general, introduce deadlock, but is safe for handshakes on channels introduced through process decomposition.
- **Production Rule Sets** After the handshake expansion and reshuffling, the original program can be expressed in terms of set and reset conditions for the variables that implement the communication handshakes. A production rule set is generated that extracts the conditions under which each variable may be set and reset. This may require the addition of extra state variables.
- **Operator Expansion** The production rule set is implemented by matching the left hand side conditions of the production rules with the available operator set. These operators include simple gates, C-elements, an arbiter and a synchronizer. These circuits set and reset the latches that implement the state variables of the system.

• **Data Delivery in the Circuit** In Burns' compiler, after the program has been translated into circuit form, the Boolean variables are delivered to receivers through the use of "isochronic forks" which assume negligible delay between sender and receiver. Timing constraints are still local, and this is a similar condition to the bundling constraint but slightly stronger.

Note that the use of isochronic forks is not limited to data delivery. The Q-element described in Section 7.2.1, for example, uses an isochronic fork to pass control information inside the Q-element. If the Q-element is defined as a self contained module then the isochronic constraint can be met easily. If, instead, the Q-element is constructed from simple gates which may end up in different parts of the circuit, problems due to timing of the fork can cause problems.

Many similar situations are seen in the modules described in Appendix A. Because the modules are designed as individual modules that are not broken up during layout, timing constraints inside the modules can be controlled carefully.

- **Signalling Protocol** Burns' compiler produces circuits that use four-phase signalling. My compiler uses two-phase signalling for control paths.

- **Optimizations** Burns' optimizations are complicated by the need to reshuffle handshake signals and fold the return-to-zero phase of the handshake into the following transaction. This means that the optimization procedure cannot be completely syntax directed. A number of different coupling circuits are used to implement sequencing with a varying amount of overlap. The optimization process presented in Chapter 5 of this dissertation is a simple pattern-directed replacement from one circuit topology to another.

- **Target Module Set** The granularity of the pieces used to construct the circuit is smaller than in my method. Both Martin and Burns map to simple gates and C-elements rather than to modules with a higher level of abstraction and can therefore obtain somewhat smaller circuits.

- **Degree of Automation** Burns' compiler implements a subset of the transformation and optimization procedure described in his thesis. It is not clear which parts of the process are done by the compiler and which parts are done manually. In contrast, the prototype compiler presented in Chapter 6 automatically performs all of the translation and optimization steps described in this dissertation.

7.4.1 Example: Distributed Mutual Exclusion

The distributed mutual exclusion process used as an example in Chapter 5 is also used as an extended example in Burns' thesis. This example allows us to compare the operation and results of the two compilers. Of course, this is only one example, so results on different circuits might vary in either compiler's favor. Recall from Chapter 5 the definition of one server process from the distributed mutual exclusion system. One version of the OCCAM code for the server is repeated here as Figure 7.8. This version of the server operates in the exactly the same way as the server described by Martin [55] and shown here in Figure 7.9. The code for Burns' version, which uses different syntax and a slightly different version of the server algorithm, is shown in Figure 7.10. Note that the CSP-style code is much denser than the OCCAM code, but that the process descriptions are very similar. Burns' version sets and resets the privilege bit *b* before further communication rather than after

```
(Process server1 ((Chan Req Grant Done    ; Macro to implement the server
          Req-Priv-Left Ack-Priv-Left Req-Priv-Right Ack-Priv-Right))
  (Seq ((Var priv))                       ; Define local variable to hold privilege.
    (While True                           ; Repeat forever.
      (Alt                                ; choose an alternative
        ((True (? Req))                   ; Request from master.
          (get-privilege)                 ; Call macro to get privilege
          (? Grant)                       ; Grant access to master.
          (? Done)                        ; Master is finished.
          (Set priv True))                ; Remember privilege.
        ((True (? Req-Priv-Left))         ; Request for privilege.
          (get-privilege)                 ; Call macro to get privilege
          (? Ack-Priv-Left)               ; Pass privilege left.
          (Set priv False))))))           ; Forget privilege.

(Process get-privilege ()                 ; Macro to get privilege from the right
  (If ((not priv)                         ; Only if not privileged.
    (! Req-Priv-Right)                    ; Ask for the privilege.
    (! Ack-Priv-Right))))                 ; Received the privilege.
```

Figure 7.8: OCCAM Code for Distributed Mutual Exclusion Server (See also Figure 5.39)

the communication as in the previous versions, and contains only a single synchronization to the master process over the U channel. Burns' version of the server assumes that an optimization has been performed already that suppresses one of the two consecutive U synchronizations in Martin's code. This optimization splits the four-phase handshake into two parts and uses the high-going phase to signal the first U communication and the return-to-zero phase to signal the second U communication. Essentially two two-phase transactions on U have been merged into a single four-phase communication in the source code. This optimization is not a part of the automatic compilation and is not directly expressible in CSP.

The main difference between the OCCAM code and the two CSP versions is that the use of probed communication channels in this dialect of CSP allows the specification to be written using fewer communication channels. A probe of a channel is indicated in CSP by a bar over the channel name. This is essentially a condition which evaluates to *true* if there is a communication pending at that channel. The process can sense this condition, do some other processing, and then execute that communication later. The OCCAM version, which does not use probed communication, defines separate channels to initiate and complete a

$*[[\overline{U} \rightarrow [b \rightarrow \text{skip} \mid \neg b \rightarrow R] \; ; \; U \; ; \; U \; ; \; b \uparrow$	— If U, get privilege, comm. on U twice, set b
$\square \overline{L} \rightarrow [b \rightarrow \text{skip} \mid \neg b \rightarrow R] \; ; \; L \; ; \; b \downarrow$	— If L, get privilege, communicate on L, reset b
$]]$	

Figure 7.9: Martin's code for Distributed Mutual Exclusion Server [55]

passive U, L	— Declare U (master) and L (left) channels passive
active R	— Declare R (right) channel active
boolean b = false	— Declare privilege bit, initially false
procedure P $[b \rightarrow \text{skip} \mid \neg b \rightarrow R]$	— Procedure P communicates on R to get privilege
$*[[\overline{U} \rightarrow P \; ; \; b \uparrow \; ; \; U$	— If U, get privilege, set b, communicate on U
$\square \overline{L} \rightarrow P \; ; \; b \downarrow \; ; \; L$	— If L, get privilege, reset b, communicate on L
$]]$	
process *priv0 (U, L, R)*	— Declare process *priv0*

Figure 7.10: Burns' code for Distributed Mutual Exclusion Server [12]

two-part communication.

For example, the server may be requested to pass the privilege to the server on the left. In the CSP code of Figure 7.10 this action is initiated when the probe of the *L* channel is *true*. In this case the server calls the *P* routine to get the privilege, resets its own privilege bit to indicate that it is no longer holding the privilege, and finally synchronizes on the *L* channel to pass the privilege to the left. In the OCCAM code, on the other hand, the guarded communication action on the left channel must be completed before the body of the alternative is executed. Two channels are defined to implement this behavior. Communication on *Req-Priv-Left* indicates that the server on the left desires the privilege. Once the *get-privilege* process is called to get the privilege, the server communicates on another channel called *Ack-Priv-Left* to pass the privilege to the other server. This corresponds to the completion of the *L* action in the CSP code.

The resulting circuits from the two compilers are shown in Figures 7.11 and 7.12. The two-phase circuit, described in more detail earlier in this dissertation, uses a Call module to differentiate the two different parts of the code that communicate on the right channels. The selector steers the transition depending on whether the privilege bit is already set or not. This circuit is compiled completely automatically from the OCCAM source into the circuit shown in the figure.

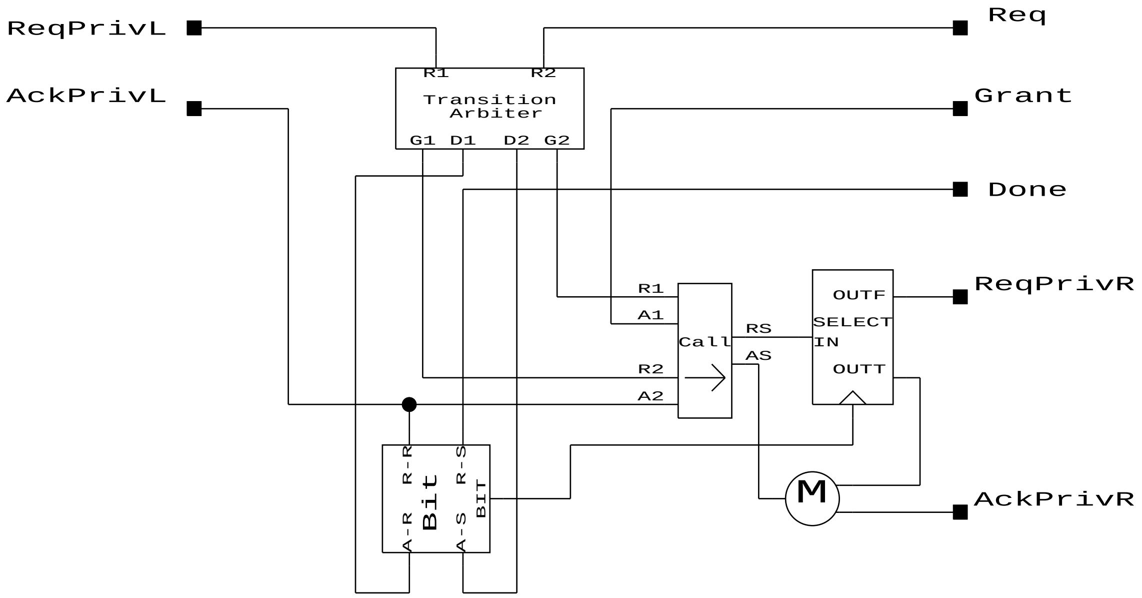

Figure 7.11: Two-Phase Server Circuit (Same as Figure 5.41)

As explained in Chapter 5, some of the optimizations do not become available until the server circuit is connected to other servers and to the master circuit, but when this is done, and the optimizations applied by the compiler, this circuit is the result. The remarkable thing about this example is the way that the optimizations reduce the extra overhead of the multiple channels used to simulate the *probe*. Because the OCCAM program used extra channels to simulate the *probe* the initial circuit contained extra wires and circuitry. This overhead was eliminated through the optimizing transformations leaving only the single wires shown in the figure.

The circuit resulting from Burns' compiler is shown in Figure 7.12. The example is carried out in great detail in Burns' thesis but, because his compiler implements only a subset of the translation and optimization process, it is not clear whether this circuit is the direct result of the compiler or requires extra modification. Like the two-phase version, Burns' circuit uses a set-reset bit to hold the value of the privilege, and an arbiter used to decide which communication request to participate in. The box marked with a *Q* in Figure 7.12 is a "Q-element" used to sequence the actions on the right channel with other actions in the circuit.

Although these two circuits are generated from different source programs, compiled using different techniques, and are built using a different style of circuit implementation, the sizes of the resulting circuits are remarkably similar. The implementations of the

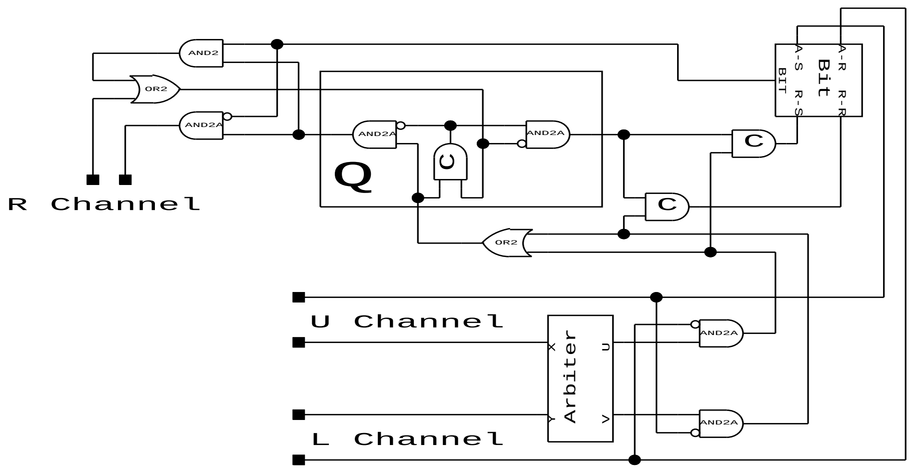

Figure 7.12: Four-Phase Server Circuit due to Burns [12]

	Brunvand	Burns
Arbiters	1 Transition Arbiter	1 Mutex Element
Gates	6 XOR	8 AND/OR
S-R Bits	1	1
C-elements/latches	4	3

Figure 7.13: Distributed Mutual Exclusion Circuit Comparison

set-reset bits and arbiters will, of course, be different because of the different signalling strategies. The two-phase transition arbiter, for example, is more complicated than the simple mutual exclusion circuit used as an arbiter in the four-phase circuit, but the gate counts of the surrounding circuits are almost the same. The main difference seems to be that the two-phase circuit uses XOR gates where the four-phase circuit uses AND and OR gates. Gate and module counts for the two circuits are summarized in Figure 7.13. The gate counts for the two-phase circuit include the building block gates for the Call and Select modules. Example implementations for these modules are shown in Appendix A.

7.5 Strengths of the Current System

The translation and optimization method presented in this dissertation has used simplicity as a guiding principle, and many of the strengths of this method are a direct result of its simplicity. The choice of OCCAM as a source language means that there is already a well understood semantics to guide system description. There are also compilers and interpreters

```
(Par ((Chan A)(Var x))          ; Input and output in parallel
   (! A exp)
   (? A x))

(Set x exp)                      ; Should behave like simple assignment
```

Figure 7.14: OCCAM Code for Parallel Input and Output, and for Simple Assignment

that allow the system descriptions to be run as ordinary programs prior to translation into hardware. The initial translation from OCCAM into an asynchronous circuit is a simple syntax-directed replacement of OCCAM language constructs with pieces of circuit. These circuits are then optimized by another syntax-directed replacement of pieces of the circuit graph with different circuit structures. The greatest strength of the method is the power that results from these simple processes.

Consider, for example, the code fragments shown in Figure 7.14. According to transformation rules for OCCAM programs [78], the semantics of two communication actions in parallel using a common channel should be the same as an assignment statement. This is a fairly subtle semantic point that should be reflected in the implementation if the optimization process is powerful enough.

The result of the initial transformation of the parallel communication commands from Figure 7.14 into a circuit is shown in Figure 7.15. The structure of the circuit is visible in the figure. The output command enables the expression *exp* onto channel *A* and the input command reads information from channel *A* and stores it in the variable *X*. Because they are executed in parallel, both communication processes are started using the same request signal. When both communication commands have completed, the C-element in the top right will report an acknowledge for the parallel composition.

The optimizations immediately remove most of the circuit elements by eliminating vacuous modules. The single client Call modules and single client Enable modules are replaced with wires, and the single input Merge module is also replaced with a simple wire. These are pattern-directed circuit replacements that do not require any reasoning about the operation of the circuit. The result of applying these optimizations is shown in Figure 7.16. The C-element on the *Req* line is the remaining C-element from the input statement that synchronizes the actions of the communication processes. The C-element on the *Ack* line is the C-element that detects when the components of the parallel composition have both completed. These C-elements are also removable with the application of another optimization and the final result is shown in Figure 7.17. Although the translation and optimization processes are individually simple, their application to the parallel communication code has resulted in exactly the same circuit as the assignment statement.

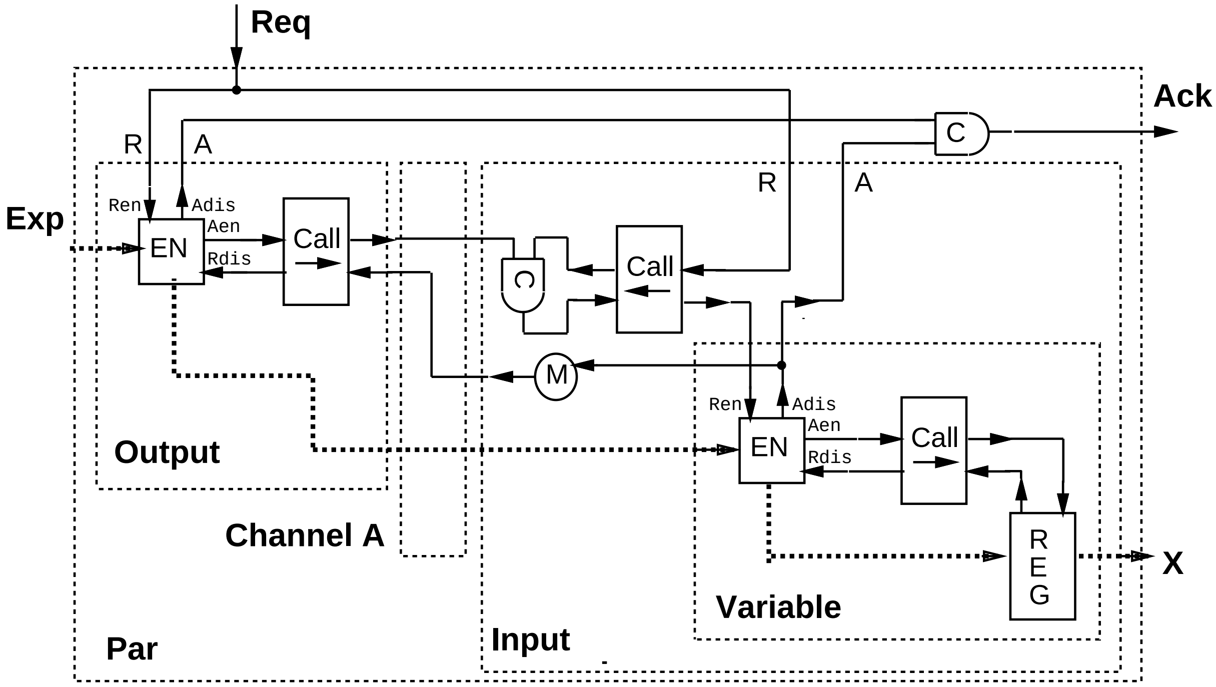

Figure 7.15: Circuit Result from Parallel Input and Output Code

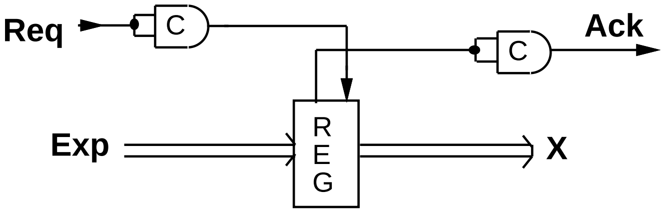

Figure 7.16: Parallel Input-Output Circuit After Initial Transformations

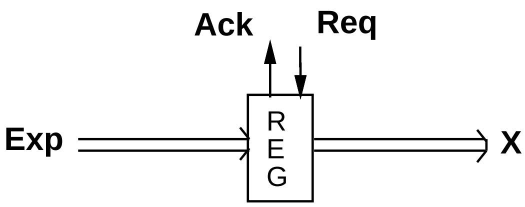

Figure 7.17: Final Parallel Input-Output Circuit

The power of this method is further demonstrated by the fact that the small OCCAM programs compiled using this method are either exactly the same, or substantially similar, to the best circuits designed by hand using the same technology. Bigger programs that were not attempted by hand also appear to have reasonable circuit implementations. With a little experience, the simplicity and transparency of the method allows a clear relationship between the OCCAM source and the resulting circuit. This allows the designer to influence the eventual circuit by modifying the source code.

Martin's method, in contrast, tries to explode the source program during the compilation process in such a way that novel circuits not envisioned by the designer might result. While this is certainly an interesting result when it occurs, I believe it is more important that the designer be able to easily understand the circuit that results from the compiler.

Although the compilation technique presented here is quite different from Martin's and Burns', the resulting circuits are not much different in size for this example. This is further evidence of the power of simplicity. Even without the extra effort involved in searching for novel designs, the circuit sizes are very similar.

Simplicity also allows my system to completely automate the compilation procedure. The initial translation and all the optimizations presented in the dissertation are implemented in the prototype compiler.

The main weakness of the present system, and of most systems of the same type, involve the conversion from circuit graph to VLSI layout. The compiler produces a netlist of the circuit modules and their connections that may be simulated using a circuit simulator, but must be processed by place and route software to prepare for fabrication. Placement and routing of circuit modules is a complex problem in its own right and beyond the scope of this dissertation. There are many available programs, but none produce circuits as dense as careful, time-intensive, human design.

One major problem with current place and route software is the inability to influence the routing of individual wires in the circuit. Placement and routing has a large impact on circuit performance. For example, there is no way to ensure that one wire in a data bundle will not be routed in a completely different way from the other wires in the bundle, or that that wire is not routed in such a circuitous path that its delay is longer than the delay in the associated control signal. A major advantage of asynchronous circuits is that for the most part these timing issues have no effect on the functionality of the circuit, only on the performance. However, in the case of bundled data paths there are timing constraints to be met, and current place and route software does not address these issues.

This is one argument for using a set of standard modules to assemble the circuits. These modules will always be implemented as an indivisible unit which allows careful control of local signal timing inside the module. The modules used here and described in Appendix A, for example, are not constructed internally from delay insensitive compositions of gates, but can present a delay insensitive interface to the external world if they are designed carefully. This helps the timing problem considerably by reducing the number of timingcritical signals that the router deals with, but, because data paths are bundled, there are still signals that require careful control.

Other weaknesses involve the limitations discussed earlier on the circuits that be described using OCCAM as a source language, or that can be implemented using two-phase signalling. The simplification of using bundled data paths also leads to limitations in the actual implementation of the circuits. These and other shortcomings of the current system are addressed in the future work section of the following chapter.