Translating Concurrent Programs into Delay-Insensitive Circuits

Erik Brunvand*
Carnegie Mellon University

Robert F. Sproull
Sutherland, Sproull, and Associates

Abstract

Programs written in a subset of occam are automatically translated into delay-insensitive circuits using syntax-directed techniques. The resulting circuits are improved using semantics-preserving circuit-to-circuit transformations. Since each step of the translation process can be proven correct, the resulting circuit behavior is a faithful copy of the original program behavior. A compiler has been constructed that automatically performs the translation and transformation.

1 Introduction

As VLSI systems become larger and more complicated, 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 circuits from behavioral descriptions. This allows the designer to abstract away details of the low-level circuits and think of system behavior in terms of high level programs. In addition to making system design easier, formal proof techniques can be used to verify that the program meets the system specification. Since the generated circuits faithfully mimic the behavior of the program, the resulting circuits are correct by construction. 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.

This paper presents a method for automatically translating a concurrent program into an asynchronous circuit. The translation procedure 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. Because these steps can be proven to be correct, the programmer is guaranteed that any specification met by the program will also be met by the circuit. A system has been constructed to perform the translation automatically. This paper gives a very brief description of the method followed by two examples of programs translated into circuits. A more complete description can be found in [3].

*This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976. The principal author has also been supported by an IBM Graduate Fellowship. The second author has been supported by the Asynchronous Systems Study, sponsored by Apple Computer, Austek Microsystems, Digital Equipment Corp., Evans and Sutherland, Floating Point Systems, and Schlumberger.

2 Related Work

Other groups apply different techniques to achieve the same objective of translating programs into asynchronous circuits. Martin [4,10] at CalTech has had good success compiling a language based on CSP [8] into gate-level circuits. Initial programs are first decomposed into many smaller processes initiated by signals on new communication channels. These simpler processes are expanded to include details of the four-phase handshake used for control signals and then mapped into a set of production rules that define when the handshake signals are set and reset. These productions are strengthened to enforce sequencing, and then mapped to a library of gates to construct the circuit.

A group at Philips Labs [11,17] has also compiled a similar CSP based language into circuits by translating the initial program into an intermediate representation that includes details of the four-phase control signals, and then implementing the circuit with a library of simple circuits.

Our target language is not a circuit of individual transistors, or even gates, but rather a collection of data and control modules, wired appropriately. The modules are like standard cells, each of which may be implemented with a small number of transistors and which correspond closely to programming constructs. Modules of this type trace their ancestry to Macromodules [5,12], designed at Washington University in the late 1960's.

3 Translation Method

The source language for this translation is a subset of occam [9], a language based on CSP [8]. Occam describes computation as a set of concurrent processes that interact by communication over fixed links called channels. Control over concurrent and sequential aspects of communication is explicit. The syntax for occam code in this paper is very similar to that in [9] except that parenthesis are used for block structuring instead of indentation alone.

The target for the translation is an asynchronous circuit. In particular, the target circuits are delay-insensitive control circuits using two-phase transition signalling [14] combined with bundled data paths. 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. 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.

A bundled data path uses a single set of control wires to indicate the validity of a "bundle" of data wires. This requires that the data bundle and the control wires be constructed such that the value on the data bundle is stable at the receiver before a signal appears on

CH2805-0/89/0000/0262$01.00 © 1989 IEEE

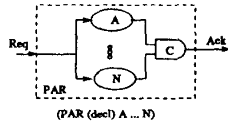

Figure 1: Initial Transformation of the Parallel Construct

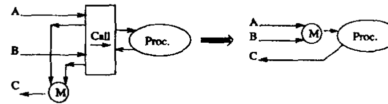

Figure 2: An Example Circuit Transformation

```
(PROC buffer ((CHAN A B))           ; Define a single-place buffer...
  (WHILE TRUE                       ; Repeat forever
    (SEQ ((VAR temp<8>))            ; Sequential composition
      (? A temp)                    ; store from channel A into temp
      (! B temp))))                 ; send from temp to channel B

(PAR ((CHAN input mid output))      ; Parallel composition
  (buffer input mid)                ; makes a two-place buffer
  (buffer mid output))
```

Figure 3: Code for a Two Place Fifo Buffer

the control wire. This condition is similar to, but weaker than, the equipotential constraint [14].

The main step in translating occam programs into circuits is the initial substitution of circuit structures for occam language constructs. The circuit modules used in the initial substitution are like those described in [2,16] and are implemented as a set of standard cells using the MOSIS scalable CMOS design rules. The modules used in the following examples include a *Merge* module that implements the "or" function for transitions, a *C-Element* that is the "and" function for transitions, a *Call* module that acts as a hardware subroutine call allowing multiple access to a shared subcircuit, a *Select* module that steers a transition to one of two outputs based on the value of a "select" signal, an *Enable* element that enables bundled data at the output in response to transition control signals, and a *Register* that latches bundled data in response to transition control signals.

Control modules are interconnected with signals that obey transition signalling conventions, in which a transition from low to high or from high to low signals an event [14]. Each module responds to a global *clear* signal by forcing its outputs to a known state and clearing any internal state. The environment issues a transition signal called *start* after a global clear to get things started. This is connected to the request input of process obtained by translating the single top-level occam process.

An example of a circuit substitution can be seen in Figure 1. An occam parallel (PAR) construct is implemented by sending the request signal to all of the component processes. The acknowledgments are combined in a C-element so that the PAR construct acknowledges after all the component processes acknowledge. Circuit substitutions for the other occam language constructs are shown in [3].

Once the initial translation from program constructs to circuit constructs has been carried out, a variety of interesting transformations, similar to peephole optimizations in software compilers, may be applied to the resulting circuit. A transformation is described as an initial circuit and a replacement circuit. The circuit is a graph made up of modules (nodes) and connections (arcs). This circuit graph is searched to find a match with the initial circuit graph of a transformation. When a circuit topology is matched with a template it is replaced with a new structure that retains the behavior of the original but improves its performance. The behavior of a circuit module can be described using trace theory [18], and automatic methods can be used to verify that the behavior of the replacement is an acceptable substitute for the original circuit [7]. An example of a transformation is shown in Figure 2. If the acknowledgments from all inputs of a call element are combined in a merge, then the circuit may be simplified by eliminating the call element as shown.

4 Examples

4.1 Fifo Buffer

The two-place fifo buffer shown in Figure 3 is the first example. A single place buffer that copies its input to its output is defined as a macro. This macro can be used to make a two-place fifo buffer by combining two single-place buffers operating concurrently. The initial translation of the occam program for the single-place buffer macro into a circuit is shown in Figure 4. First the WHILE construct is expanded. The component process of the WHILE construct is a sequential (SEQ) construct with two component processes: an input (?) and an output (!) statement. Notice that the single-place buffer macro begins operation upon receipt of a *start* signal and signals completion with an *ack* signal. There are two channels, one input and one output. The C-element that performs the synchronization at the channel is included as part of the circuit for the input channel. Two single-place buffers may be combined to operate concurrently by using the parallel construct shown in Figure 1.

Optimizing the circuit removes a great many components, to yield the final result shown in Figure 5. Most of the optimizations are readily apparent in Figure 4: a CALL module with only one client can be removed and replaced by wires; an ENABLE (EN) module whose outputs are the only drivers of a data bus can be removed; a MERGE (M) module with one input can be removed; a SELECT (SEL) module whose select condition is *true* can be removed. These steps, plus the removal of the *start* signal, yield the final circuit, which is the best circuit known for a FIFO using bundled data paths and transition signalling.

4.2 Torus Routing Chip

The last example is a switch for cut-through packet routing in a multiprocessor interconnection network similar to the Torus Routing Chip (TRC) described by Dally and Seitz [6]. Each processor in the system has an associated routing circuit. The processor communicates to the routing circuit through the *Pin* and *Pout* channels. Each routing circuit routes packets in two dimensions through the *Xin, Xout, Yin,* and *Yout* channels as shown in Figure 6. The circuit must accept packets on any of the three input ports and route them to the appropriate output port depending on address information in the packet header.

The code for the routing process is shown in Figure 7 with the top-level macro *mesh-element*. Like the TRC, this process uses byte-wide data paths on all channels, but this version uses an additional tag bit to indicate the end of a packet. Address information, contained in the first two bytes of a data packet, specifies relative addresses in the X

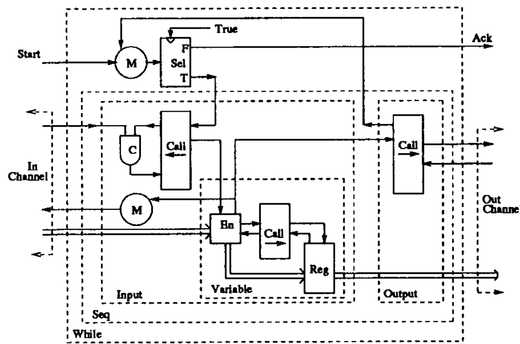

Figure 4: Initial Transformation of Single-Place Buffer Macro

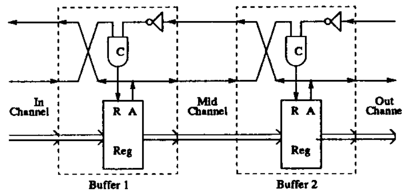

Figure 5: Optimized Two-Place Fifo Buffer

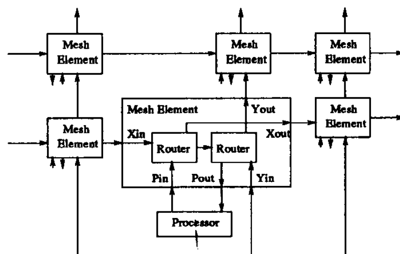

Figure 6: Routing Circuit Block Diagram

```
; send data from in to out while tag is non-zero
(PROC send-data ((CHAN In Out))
  (WHILE (not data<8>)                  ;while tag bit is not zero
    (! Out data)                        ;send data out
    (? In data))                        ;and get the next byte
  (! Out data))                         ;send last byte

; switch data from in to one of two outputs depending on first byte
(PROC switch-data ((CHAN In Aout Bout))
  (IF ((notzero data)                   ;if data is non-zero
    (send-data In Aout))                ;send it out on Aout
  (TRUE                                 ;otherwise
    (? In data)                         ;drop it, get new byte
    (send-data In Bout))))              ;and send out on Bout

; Take data from whichever input channel is ready and route it to
; either of the two output channels.
(PROC router ((CHAN Ain Bin Aout Bout))
  (SEQ ((VAR data<9>))                  ;define a local variable
    (WHILE TRUE                         ;do forever
      (FAIR-ALT                         ;choose fairly
        ((TRUE (? Ain data))            ;data on the A input
          (SET data (decr data))        ;decrement address
          (switch-data Ain Aout Bout))  ;send out data
        ((TRUE (? Bin data))            ;data on the B input
          (SET data (decr data))        ;decrement address
          (switch-data Bin Aout Bout))))) ;send it on out

; The router element. Route data streams first in the X and then
; in the Y direction based on information in header bytes
(PROC mesh-element ((CHAN Xin Yin Pin Xout Yout Pout))
  (PAR ((CHAN mid))
    (router Pin Xin Xout mid)           ;route in X direction
    (router mid Yin Yout Pout)))        ;route in Y direction
```

Figure 7: A Simple Routing Process

and Y directions. Our example does not include the virtual channels used in the TRC, although they are important in a real implementation to insure deadlock-free routing.

The mesh-element macro is actually made up of two *router* macros operating in parallel. One accepts packets from the processor and from the X direction and routes them along the X direction, or to the other router. The other router takes packets from the first router or from the Y direction, and routes them along the Y direction or to the attached processor. Note that a single mesh element can be routing a packet along the X direction and a different packet along the Y direction at the same time. Packets are routed first in X and then in Y. The first byte is used as the current address byte and decremented as the process reads it. If the result is non-zero, the packet is forwarded in the same direction it came from, if the result is zero, the byte is dropped and the packet changes direction. When the Y address is decremented to zero, the packet has arrived at its destination and is routed to the attached processor.

The initial translation of a complete mesh element uses 120 modules. Optimization reduces this number to 66, unfortunately too complex to illustrate in this paper. An version of this circuit with 5-bit data paths has been generated so that the circuit will fit in a standard 40-pin package and is currently in fabrication.

5 Conclusion

The methods of translating occam programs into asynchronous circuits and then optimizing the circuits have been embedded in a silicon

compiler. The examples in this paper, as well as many others, have been successfully compiled. The resulting circuits have been simulated, using a switch-level simulator, to demonstrate their correctness. Some of the examples have been laid out on chips, using the MOSIS FUSION place-and-route service and fabricated [1]. The fabricated examples have all been fully functional on the first fabrication run.

We have found that occam is an excellent way to describe asynchronous systems, especially pipelined data processors or micropipelines [15]. Almost all of the circuits we designed by hand using the standard control modules can be expressed simply in occam. The occam programs are clear, easy to manipulate, 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 we have been exploring in hardware. Because there is a clear relationship between the occam programs and the resulting circuits, the programmer or system designer has a good idea how modifications will affect the performance or size of the result.

There are several ways in which the current silicon compiler can be improved. The most important limitation on the performance of the circuits we obtain is the time required for data delivery, i.e., to enforce the bundling constraint by waiting for the outputs of a register to propagate through all combinatorial circuitry connected to the register. Techniques can be applied to the occam program or to the resulting circuit which can establish that some data-delivery delays are not required, or so that in so far as possible, data-delivery delay is overlapped by control delay. It may seem odious to deal with delays, since 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. In particular, no timing requirements need to propagate across a communication channel. 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. Rather than using the module library, an occam process could be implemented as a Q-module [13]. Control signalling might use four-phase rather than transition conventions. Fully self-timed data, rather than bundling conventions, could be used. And, of course, the translation could use a globally synchronous implementation rather than asynchronous techniques. These areas represent avenues for further research.

The silicon compiler we have built suggests that a principal weakness of asynchronous systems—the difficulty of their design—can be largely overcome by compilation from a clear 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.

References

[1] R. Ayres. *FUSION: A New MOSIS Service.* Technical Report ISI/TM-87-194, Information Sciences Institute, 1987.

[2] Erik Brunvand. *Parts-R-Us: A Chip Aparts ...* Technical Report CMU-CS-87-119, Carnegie Mellon University, 1987.

[3] Erik Brunvand and Bob Sproull. *Translating Concurrent Communicating Programs into Delay-Insensitive Circuits.* Technical Report CMU-CS-89-126, Carnegie Mellon University, April 1989.

[4] Steven Burns and Alain J Martin. *Synthesis of self-timed circuits by program transformation.* Technical Report 5253:TR:87, California Institute of Technology, 1987.

[5] Wesley A. Clark. Macromodular computer systems. In *Spring Joint Computer Conference*, AFIPS, April 1967.

[6] William J. Dally and Charles L. Seitz. The torus routing chip. *Distributed Computing*, 1:187–196, 1986.

[7] David Dill. *Trace theory for for the hierarchical verification of speed independent circuits.* PhD thesis, Carnegie Mellon University, 1987.

[8] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[9] *Occam Programming Manual.* Inmos, 1983.

[10] Alain J. Martin. Compiling communicating processes into delay insensitive circuits. *Distributed Computing*, 1(3), 1986.

[11] Cees Niessen, C.H. (Kees) van Berkel, Martin Rem, and Ronald W.J.J. Saeijs. Vlsi programming and silicon compilation; a novel approach from philips research. In *ICCD*, Rye Brook, NY, Oct 1988.

[12] S. M. Ornstein, M. J. Stucki, and W. A. Clark. A functional description of macromodules. In *Spring Joint Computer Conference*, AFIPS, 1967.

[13] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37(9), Sept 1988.

[14] C. L. Seitz. System timing. In *Mead and Conway, Introduction to VLSI Systems*, chapter 7, Addison-Wesley, 1980.

[15] Ivan Sutherland. Micropipelines. *CACM*, 32(6), 1989.

[16] Ivan E. Sutherland, Robert F. Sproull, and Ian Jones. *Standard Asynchronous Modules.* Technical Memo 4662, Sutherland, Sproull and Associates, 1986.

[17] C.H. (Kees) van Berkel and Ronald W.J.J. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *ICCD*, Rye Brook, NY, Oct 1988.

[18] J. van de Snepscheut. Trace theory and vlsi design. In *Lecture Notes in Computer Science 200*, Springer-Verlag, 1985.