Syntax-directed Translation of Concurrent Programs into Self-timed Circuits[1]

Steven M. Burns and Alain J. Martin

Computer Science Department
California Institute of Technology
Pasadena, CA 91125

We present a method for the automatic compilation of concurrent programs into self-timed circuits. The compilation is directed by the syntax of the source language and produces linear-sized implementations of arbitrary concurrent programs. We have constructed a compiler which performs this translation.

An automatic method for synthesizing a digital circuit from an abstract specification represents an important step forward in managing the complexity of VLSI system design. By allowing the designer to think in terms of high-level programs instead of detailed circuits, systems can be designed in less time with fewer errors. In fact, if formal proofs are used to verify that the program implements the system specification, the circuits derived are correct by construction.

To take full advantage of the inherently concurrent nature of digital circuits, we have chosen a variant of Communicating Sequential Processes (CSP)[4] as the source language for our synthesis method. While self-timed circuits[10] offer several advantages over clocked circuits, we chose them as our synthesis target mainly because of the composition properties they possess. Self-timed sub-circuits may be composed together to form circuits of arbitrary size without the hazards associated with clock distribution in large synchronous designs.

In this paper, we present a mechanical method for transforming a concurrent program into a semantically equivalent self-timed circuit. This method is based on the manual circuit-compilation techniques of [6, 8], but the translation mechanism follows standard syntax-directed techniques. We describe the necessary translation rules and apply these to an example. This transformation method allows any program in the source language to be compiled into a self-timed circuit, and further-

[1]in *Advanced Research in VLSI: Proceedings of the 5th MIT Conference*, MIT Press, pp. 35–50 (1988)

```
⟨process⟩     ::=  ( ⟨process⟩ { || ⟨process⟩ } ) { ⟨channel⟩ }
                 | { ⟨port⟩ } { ⟨var⟩ } ⟨sequence⟩
⟨channel⟩     ::=  channel ( ⟨pNAME⟩ , ⟨pNAME⟩ )
⟨port⟩        ::=  ( passive | active ) ⟨pNAME⟩ ( ⟨rINT⟩ , ⟨rINT⟩ )
⟨var⟩         ::=  boolean ⟨vNAME⟩ = ( true | false )
⟨sequence⟩    ::=  ⟨statement⟩ { ; ⟨statement⟩ }
⟨statement⟩   ::=  skip | ⟨vNAME⟩ ( up | down ) | [ ⟨gcs⟩ ] | *[ ⟨gcs⟩ ]
                 | ⟨pNAME⟩ ( ⟨vINT⟩ ) : [ ⟨responses⟩ ]
⟨gcs⟩         ::=  ⟨gc⟩ { | ⟨gc⟩ }
⟨gc⟩          ::=  ⟨expr⟩ --> ⟨sequence⟩
⟨responses⟩   ::=  ⟨response⟩ { | ⟨response⟩ }
⟨response⟩    ::=  ⟨vINT⟩ --> ⟨sequence⟩
⟨expr⟩        ::=  ⟨conjunct⟩ { or ⟨conjunct⟩ }
⟨conjunct⟩    ::=  ⟨primary⟩ { and ⟨primary⟩ }
⟨primary⟩     ::=  ⟨vNAME⟩ | probe ⟨pNAME⟩ | true | false
                 | not ⟨primary⟩ | ( ⟨expr⟩ )
```

Figure 1: BNF for Source Language

more constructively proves that the size of the resulting circuit is linearly related to the size of the source program. The method has been automated and we compare the designs produced by the compiler with designs produced by manual methods.

1 Source Language

The source language is based on CSP[4] with the addition of the *probe*[9] and a new communication construct. A complete description of the language syntax is given in Figure 1. A program in this language consists of a set of sequential processes with interconnecting channels. Associated with each sequential process is a set of ports, a set of private variables, and a list of statements to be executed sequentially. Ports that do not connect to another process connect to the environment.

Only boolean variables are allowed. Variables are changed by assignment to true (`x up`) or to false (`x down`). The selection (`[`⟨gcs⟩`]`) and repetition (`*[`⟨gcs⟩`]`) constructs are based on guarded commands. We use `*[`⟨sequence⟩`]` as an abbreviation for `*[true-->`⟨sequence⟩`]`.

Synchronization between two processes is accomplished by zero-slack communication actions across channels denoted by pairs of ports. Of the two ports which make up a channel, one is declared *active* and the other is declared *passive*. The process which owns the *passive* port may determine whether the other process is waiting for a communica-

tion on this channel by evaluating a boolean condition called a probe. Probes may be used in arbitrary boolean expressions.

Though concurrently operating processes may not share variables, processes may communicate data by exchanging values from small sets during a synchronization action. The communication construct provides a syntax allowing differing sequences of commands to be executed based on the value received during a communication. When declaring a port, we specify both the send and receive sets of values—each set being represented by a single integer. For example,

```
passive L(3,2)
```

declares a *passive* port `L` with send set $\{0,1,2\}$ and receive set $\{0,1\}$. The communication action (on the same port, `L`)

```
L(1):[ 0 --> x down | 1 --> x up ]
```

sends the value `1` while simultaneously receiving either a `1` or a `0`. If a `0` is received, x is set to false; if a `1` is received, x is set to true.

The output value specification of the communication action may be suppressed if the port has only one send value. Similarly, the receive value selection need not be specified if the port has only one receive value.

We illustrate the programming language and, later, the compilation procedure with the following example.

Routing Automata

By interconnecting processes of two different types, we can construct worm-hole routing systems[3] that allow processors, connected in a variety of different network structures, to communicate. The *switch* process (diamond), together with an *arbiter* process (circle), are sufficient to implement deterministic routing systems for hypercubes, tori, meshes and other networks in which the path of the message can be determined before the message is sent. Figure 2 shows a building block configuration for sending messages up and right in a two-dimensional mesh. Each processor injects messages (strings of bits) into the system through port P and extracts messages through port Q. The processor sending a message prefixes the message with a string of bits specifying the path the message will travel through the network. A special token is used to mark the end of the message. The packet—the combination of the header, the message, and the trailing token—cuts a path through the mesh, first to the right and then up, to its destination. The *arbiter* transmits an entire packet contiguously from an input port to the output port. Packets from different input ports are not interleaved. The

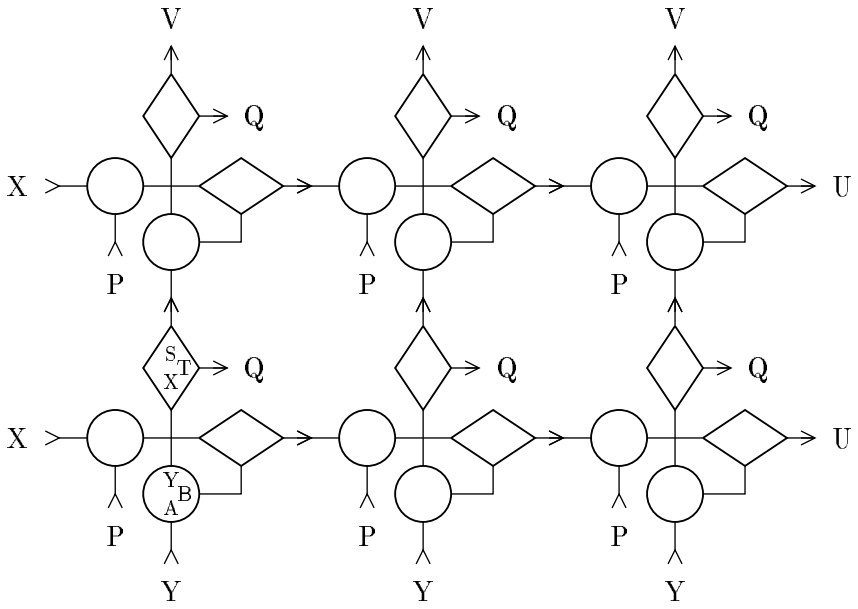

Figure 2: Building Block Interconnections for a One-way Mesh

switch consumes the first bit of the packet, and, based on whether the bit is one or zero, passes the remainder of the message out through either port S or port T.

Figure 3 shows programs for these two sequential processes. The *switch* process first communicates through port `X`, and raises one of two boolean flags, `s` or `t`. If `s` was raised, the process acts as a one-place buffer, sending the value received by port `X` out through port `S`, until the special trailing value `2` is received; in which case, flag `s` is lowered, and the repetition terminates. If `t` was raised, the process behaves in a similar manner, except `T` is used as the output port. The *arbiter* process performs fair arbitration[7] between the two input streams by first checking if an input is pending on port `A` and, if it is, then acting as a one-place buffer with output port `Y` until the trailing value is received. The port `B` will be chosen next if an input is pending, ensuring that each port is given the opportunity to transmit its packet.

2 Target Language

The target of the compilation is a self-timed circuit—a set of circuit variables (nodes) interconnected by a set of operators (gates). These circuits are designed to function correctly regardless of the internal delays of the operators. The target operator types include the combina-

```
passive X(1,3) active S(3,1) active T(3,1)
boolean s = false boolean t = false
*[ X:[ 1--> s up | 0--> t up ];
   *[ s --> X:[ 0--> S(0) | 1--> S(1) | 2--> S(2); s down ]
    | t --> X:[ 0--> T(0) | 1--> T(1) | 2--> T(2); t down ]
 ]  ]
process switch(X,S,T)

passive A(1,3) passive B(1,3) active Y(3,1)
boolean a = false boolean b = false
*[  [ probe A --> a up | not probe A --> skip ];
   *[ a --> A:[ 0--> Y(0) | 1--> Y(1) | 2--> Y(2); a down ]];
    [ probe B --> b up | not probe B --> skip ];
   *[ b --> B:[ 0--> Y(0) | 1--> Y(1) | 2--> Y(2); b down ]]
 ]
process arbiter(A,B,Y)
```

Figure 3: Programs for the Switch and Arbiter Processes

tional operators WIRE, AND, and OR; and the state-holding operators shown in Figure 4. Each operator is defined in terms of a set of production rules[6, 8]. A production rule is a simple transition rule of the form $G \longmapsto S$; where G is a boolean expression and S is an assignment to true or false. All references to a circuit variable are assumed to have the same value (isochronic forks)[2, 6]. A synchronizer, which cannot be represented in terms of production rules, is included to allow the implementation of programs with negated probes[7]. The synchronizer, as well as the other operators, have been implemented as CMOS standard cells.

Self-timed circuit implementations of concurrent programs are generated by implementing each sequential process as a separate sub-circuit. The sub-circuits are connected (by wire operators) only to implement communication actions. The simultaneity required in the zero-slack communications is implemented using a four-phase handshaking protocol. In order to implement the exchange of data, the usual request/acknowledge pair of wires is replaced by one wire for each send value and one wire for each receive value. For example, using this unary encoding scheme, a channel, connecting the active port `Y` of the *arbiter* process and the passive port `X` of the *switch* process, is implemented using four wire operators: three pointing from `Y` to `X` and

$$(x,y)\,\mathbf{C}\,(z) \equiv \begin{cases} x \wedge y & \longmapsto z\uparrow \\ \neg x \wedge \neg y & \longmapsto z\downarrow \end{cases}$$

$$(\underline{x},\underline{y})\,\mathbf{F}\,(z) \equiv \begin{cases} \bigvee x_i & \longmapsto z\uparrow \\ \bigvee \neg y_i & \longmapsto z\downarrow \end{cases}$$

$$(x,b)\,\mathbf{S}\,(u,v) \equiv \begin{cases} *[[x \wedge b & \longrightarrow u\uparrow;[\neg x];u\downarrow \\ \;\;|x \wedge \neg b & \longrightarrow v\uparrow;[\neg x];v\downarrow \\ ]] \end{cases}$$

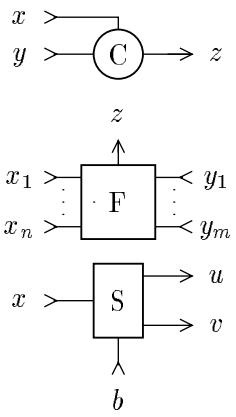

Figure 4: State-holding Operators

one pointing from `X` to `Y` (Figure 5).

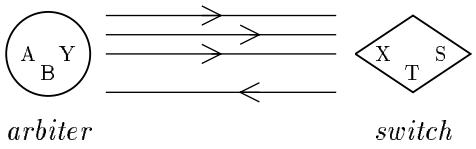

Figure 5: Implementation of a CSP Communication

The delay-insensitive nature of interprocess communication allows concurrent programs with any number of processes to be implemented. In implementing individual processes, we shall exploit this property, through a technique known as process decomposition.

3 Syntax-directed Translation

The translation from a source program into a target circuit is performed using standard syntax-directed techniques. We derive rules for generating sub-circuits that correspond to each syntactic construct of the source language, and for composing these sub-circuits to generate circuits for arbitrary programs. These translation rules are derived by applying the process decomposition transformation[6].

3.1 Process Decomposition

An arbitrary program statement β can be replaced by a single active communication C' and a new sub-process implementing β.

$$\alpha;\beta;\gamma \ \triangleright\ (\alpha;C';\gamma \parallel *[[\overline{C} \longrightarrow \beta;C]])$$

The probe $\overline{C}$, of the connected passive port, is used to guard the execution of the statement β. While process decomposition introduces a new sub-process, it does not add concurrency to the implementation; the active communication C' cannot finish until its corresponding passive communication C does, and thus the strict sequencing $\alpha;\beta;\gamma$ is maintained. The original process and the new sub-process may now share variables and ports, since processes and sub-processes are never active concurrently. The variables and ports of a process are implemented by circuit variables, which must be distributed to many sub-circuits. We assume these circuit variables are distributed by isochronic forks. This assumption may be relaxed by suspending a sequential process until distribution of the variable to every reference point is detected. This refinement, described in [2], reduces all isochronic forks to size two.

Communications across channels introduced by process decomposition have simple implementations because the handshaking actions of the passive communication may be interleaved with the execution of β. Wire operators are typically sufficient to implement this special form of the passive communication action.

We apply process decomposition to generate translation rules for the BNF rules naming ⟨statement⟩. Every occurrence of ⟨statement⟩ on the right-hand side of a BNF rule is replaced by an active communication. For every BNF rule with ⟨statement⟩ on its left-hand side, a new sub-process, protected by a passive communication, is generated to implement the right-hand side.

We generate the translation rule for the sequencing constructby the following series of transformations:

$$\begin{array}{ll}
 & *[[\overline{C} \longrightarrow \langle\text{statement}\rangle;C]] \\
\triangleright & \qquad\qquad \{\textit{BNF rule}\} \\
 & *[[\overline{C} \longrightarrow \langle\text{statement}\rangle_1;\langle\text{statement}\rangle_2;C]] \\
\triangleright & \qquad\qquad \{\textit{Process decomposition}\} \\
 & (\quad *[[\overline{C} \longrightarrow C'_1;C'_2;C]] \\
 & \parallel \quad *[[\overline{C_1} \longrightarrow \langle\text{statement}\rangle_1;C_1]] \\
 & \parallel \quad *[[\overline{C_2} \longrightarrow \langle\text{statement}\rangle_2;C_2]] \\
 &)
\end{array}$$

The first sub-process is of fixed size and may be compiled into a self-timed circuit using the techniques in [6, 8]. Circuits for the other

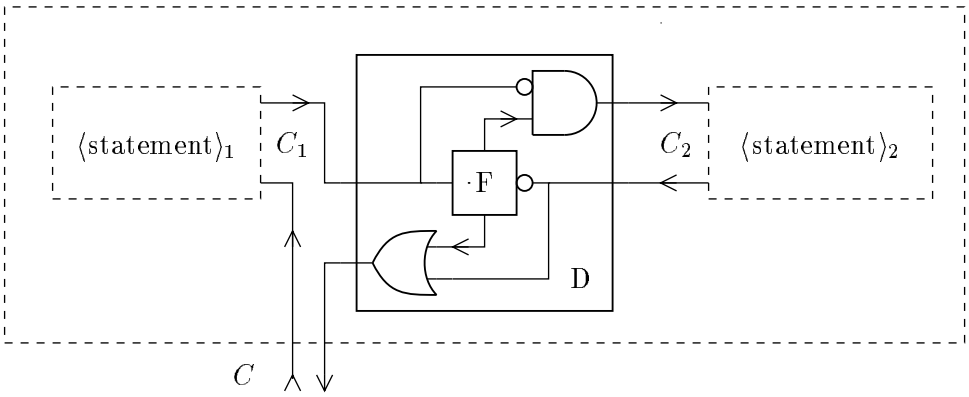

Figure 6: Translation Rule For Sequencing

two sub-processes are generated recursively by applying the translation rules. Figure 6 displays the translation rule in circuit form. The sub-circuit enclosed in box **D** is used in the implementation of several other constructs. In the following, we shall refer to instances of this circuit as instances of a **D**-element.

The other translation rules that use ⟨statement⟩ can be derived similarly. Complete derivations are described in [2]. Figure 7 shows the circuit representations of these rules. The leaf sub-processes are interconnected by several circuit variables. The named variables within the dashed boxes are connected to form the complete circuit. Each assignment sub-circuit adds a new input to the flip-flop implementing the variable. The output of the flip-flop is distributed to all sub-circuits using the variable. Each port usage adds a new input to an **OR** operator merging its output transitions. A circuit variable is generated to directly implement each probe by merging together, with an **OR** operator, each input wire of a *passive* port. The translation rule for ⟨process⟩ uses a global reset variable g. When g is false, no process is active, and thus all state-holding operators can be reset to an initial state.

Figure 8 shows the first few translation rules applied to the *switch* process. The complete circuit is shown in Figure 9.

3.2 Guarded Command Evaluation

Data channels can be used in process decomposition, allowing a subprocess to communicate the result of an evaluation to its parent process. We apply this general form of decomposition to derive translation rules for guarded command evaluation.

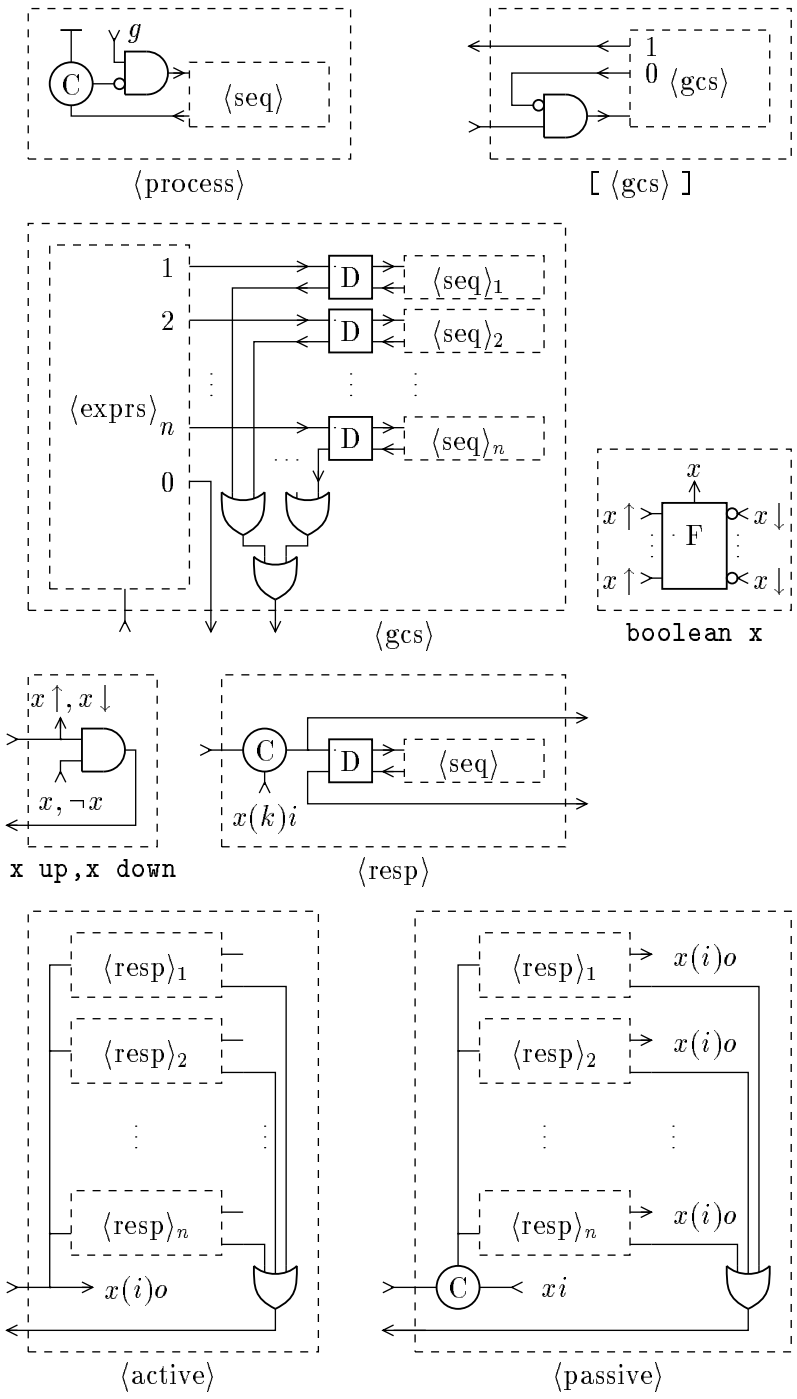

Figure 7: Circuits Implementing Statements

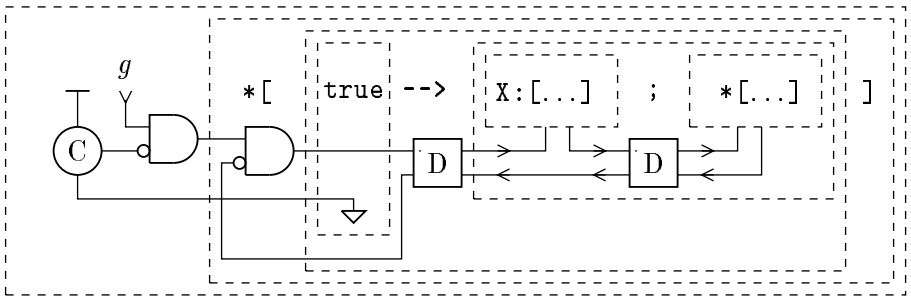

Figure 8: Translation of the *switch* Process

The implementation of a guarded command set (g.c.s.) returns one of two values: 1 if a guard evaluated to true and its corresponding command was executed, and 0 if no guard evaluated to true. The repetition statement reevaluates the g.c.s. if 1 is returned. Using busy-waiting, the selection statement reevaluates the g.c.s. if 0 is returned.

The g.c.s. process is decomposed into a process for guard set evaluation; and a control process which sequences guard evaluation and the associated command execution. The guard set process selects at most one true guard, returning a value corresponding to the selected guard. This process also detects when no guard evaluates to true, and returns the value 0. The control process explicitly introduces the state variables necessary to distinguish between the program state just before guard evaluation and the program states just after a guard has been selected, thus allowing guard evaluation to complete before subsequent statements change variables and probes used in the guards.

We concentrate now on evaluating an arbitrary guard set. The semantics of the language does not specify the order in which the guards are to be evaluated. They may be evaluated in any order or all simultaneously. We introduce three schemes for implementing guards: *sequential*, *concurrent-all*, and *concurrent-one*. The *sequential* scheme imposes a total order on the guards and generates a circuit which conditionally evaluates each guard and each connective with a guard. Evaluation of the guard set terminates when a guard evaluates to true. In the *concurrent-all* scheme all guards are evaluated concurrently. Evaluation of the guard set finishes after *all* guards have been evaluated. In the *concurrent-one* scheme all guards are evaluated concurrently, but the guards and the sub-expressions within a guard have been strengthened so that only *one* path through the evaluation circuit will become active. For all three schemes, we must ensure mutual exclusion among the guards and generate the all-false value.

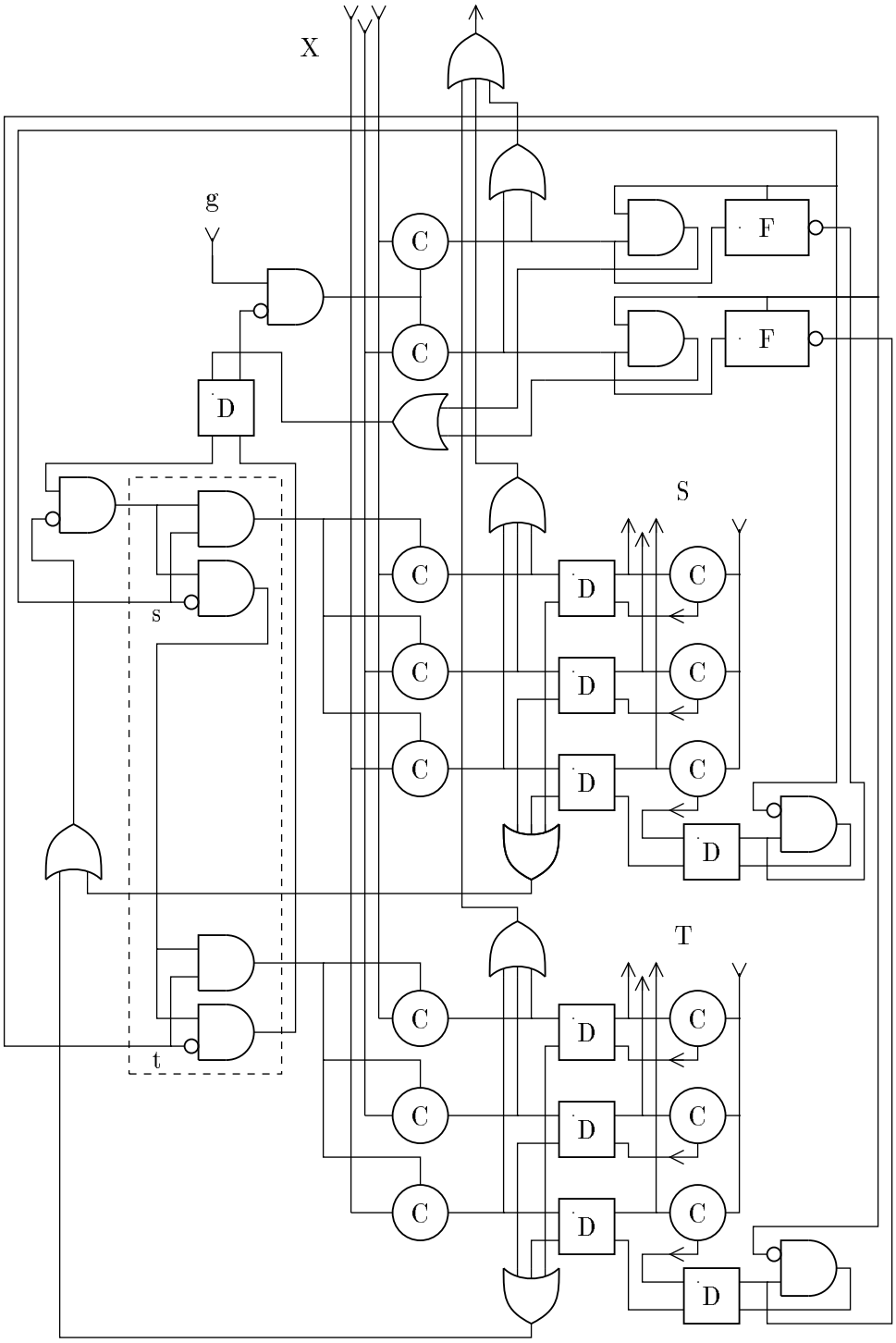

Figure 9: Optimized Circuit Implementing the *switch* Process

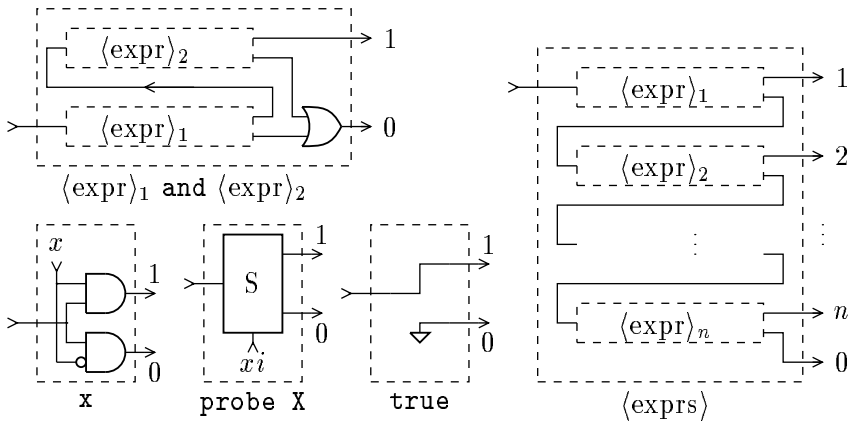

Figure 10: Circuits Implementing Sequential Guard Evaluation

3.2.1 Sequential Guard Evaluation

The guards are evaluated one by one until either one evaluates to true or the last guard evaluates to false. Mutual exclusion among the guards is accomplished implicitly by this construction.

Implementations of complex expressions are built up from implementations of simple ones. Primitive circuits exist for evaluating variables and probes. We use conditional evaluation to implement the logical connectives. Negation is accomplished by exchanging the meanings of the false and true values. We show the translation rules for sequential guard evaluation in Figure 10.

If probes are used, a further transformation is required. Because the value of a probe may change from false to true while an expression is being evaluated, all probes used more than once in a guard set are evaluated and assigned to a variable before the guard set itself is evaluated. Probes within these expressions are changed to refer to the corresponding stable variable.

The sequential scheme is applied to implement the repetition construct of the *switch* process (see outlined sub-circuit in Figure 9). The variable `s` is evaluated, and, if false, `t` is evaluated. If both are false, an all-false signal is returned.

3.2.2 Concurrent-all Guard Evaluation

All guards can be evaluated concurrently if each guard evaluation completes before the result for the entire guard set is returned. If the circuit does not wait for all the evaluations to complete, subsequent evalua-

tions may fail because internal variables of the implementation have not been reset to a required state.

In the concurrent-all scheme, all variable and probe evaluations for the entire guard set are started simultaneously. Logical connectives are evaluated by first evaluating the two sub-expressions, and then applying the logical operation to the results. Sub-expressions may be shared among the guards, producing more efficient implementations. The number of probe references per guard set can be reduced to one, thereby eliminating the need to store each probe in a variable.

The exclusion transformation must be performed explicitly to use the concurrent-all evaluation scheme. Each guard E_i is strengthened with $\bigwedge_{j=1}^{i-1} \neg E_j$, ensuring that no two guards will evaluate to true. An all-false guard $\bigwedge_{j=1}^{n} \neg E_j$ is also created. By factoring common sub-expressions, the $O(n^2)$ increase in guard size can be reduced to adding $O(n)$ new `and` connectives and $O(n)$ nesting levels. The transformation also can be performed by a parallel prefix network [5], leading to $O(n \log n)$ new `and` connectives, but with $O(\log n)$ nesting levels, resulting in the asymptotically fastest evaluation of the three schemes.

3.2.3 Concurrent-one Guard Evaluation

As in the concurrent-all scheme, all guards are evaluated concurrently, but instead of adding circuitry to wait for all the guards to evaluate, the guards are transformed so that only one path through the evaluation circuit is active at a time. The guard set finishes evaluation when the single guard becomes true.

Each guard is transformed into AND-OR form. The AND terms are further strengthened until exclusive evaluation of each term is ensured. While exponential blow-up may occur when transforming pathological expressions, this method produces the smallest and fastest implementations of most small expressions.

This scheme cannot be used to implement all guard sets. Negated probes cannot be used, and if many variables are used in the guard set, AND-operators with several inputs are required. However, selection statements with only positive probes may be implemented without the busy-waiting iteration of the previous schemes.

4 Optimizations

Simple optimizations greatly improve the compiled circuits. *Peephole* optimization is applied to the target circuits by removing operators that can be shown redundant by a local analysis of the circuit. Other

Name	Operators per process	
	Manual	Compiled
$3x+1$ iterator	95	175
routing switch	37	44
routing arbiter	36	49
lazy stack	50	86
token ring	13	27
systolic multiplier	38	59

Figure 11: Comparison of Manual and Compiled Designs

optimizations are applied at the program level. They take the form of proving that the program satisfies some invariant condition, and then using this invariant to simplify the implementation of a construct.

As an example, the explicit sequencing (**D**-element) between guard evaluation and the execution of the first statement of a corresponding command can be removed if the first statement of a command sequence does not change the value of any guard. This is always the case if the guard set consists of only constants or if the first statement is i) a `skip`, ii) an assignment to a variable not named in the guard set, or iii) a communication and no probes are named in the guard set.

After applying optimizations of these forms, the *switch* process compiles into the circuit shown in Figure 9. Notice the removal, from the non-optimized compilation started in Figure 8, of the initialization operators and the **D**-element protecting the constant guard `true`.

5 Compiler and Performance

We have implemented the translation and optimization rules described in this paper. The compiler consists of approximately 800 PROLOG clauses and is based on the program-to-machine-code compiler described in [11]. PROLOG provides an excellent environment for developing compilers and, in our case, its unification procedure provides a simple means of representing and composing circuits.

The compiled circuits are typically no more than twice the size of those derived by hand. In Figure 11, a rough comparison between the number of operators is shown for a variety of fabricated hand designs. The extra operators needed in the compiled circuits can be explained by two interrelated factors: The current source language cannot represent

the interleaving of statements within a sequential process (reshuffling of handshaking expansions), and the current program level optimizer does not detect all the cases where explicit sequencing can be removed. We expect to narrow this gap in the future.

Using 3μm SCMOS, we have fabricated several circuits derived manually from concurrent programs. We have used either a standard-cell place-and-route preprocessor for the MAGIC design tool, or the MOSIS FUSION service[1], to generate layout from an electrically optimized operator-level description of the circuit. Rudimentary electrical optimization is performed individually on each circuit variable by inserting appropriately sized drivers to minimize the transition time of each operator. Such simple techniques have met with good results; in an early implementation of a mesh router (Figure 2), the transition time of a typical operator was 2.3ns, resulting in a pad-to-pad propagation delay of 45ns for an *arbiter* and a *switch* process connected in series. We expect similar electrical performance from chips generated by the compiler.

The translation method produces correct, self-timed implementations of arbitrarily large concurrent programs, and because each translation rule is of fixed size, the size of an implementation is no worse than linearly related to the size of the source program. The translation method and the compiler provide a constructive proof that a design methodology based on programs instead of circuits is not inherently inefficient, and thus represents a practical approach to the design of VLSI systems.

Acknowledgments

We wish to thank Pieter Hazewindus for his comments on early versions of this manuscript. This research is sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

References

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

[2] S.M. Burns, *Automated Compilation of Concurrent Programs into Self-timed Circuits*, M.S. Thesis, California Institute of Technology, Caltech-CS-TR-88-2 (1988)

[3] W.J. Dally and C.L. Seitz, "The Torus Routing Chip", *Distributed Computing*, 1, pp 187-196 (1986)

[4] C.A.R. Hoare, "Communicating Sequential Processes", *C. ACM* 21, 8, pp 666-677 (August 1978)

[5] R.E. Ladner and M.J. Fischer, "Parallel Prefix Computation", *J. ACM*, 27, pp. 831–838 (1980)

[6] A.J. Martin, "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits", *Distributed Computing*, 1, pp 226-234 (1986)

[7] A.J. Martin, *A Delay-Insensitive Fair Arbiter*, Caltech Computer Science Technical Report, 5193:TR:85 (1985)

[8] A.J. Martin, "The Design of a Self-Timed Circuit for Distributed Mutual Exclusion," *Proc. 1985 Chapel Hill Conf. VLSI*, ed. Henry Fuchs, pp 247-260 (1985)

[9] A.J. Martin, "The Probe: an Addition to Communication Primitives," *Information Processing Letters*, 20, pp 125-130 (1985)

[10] C.L. Seitz, "System Timing," Chapter 7 in Mead and Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980)

[11] D. Warren, "Logic Programming and Compiler Writing," *Software—Practice and Experience*, 10, 2 (1980)