Automated Compilation of Concurrent Programs into Self-timed Circuits

Steven M. Burns

Computer Science Department
California Institute of Technology

Caltech-CS-TR-88-2

Automated Compilation of Concurrent Programs into Self-timed Circuits

Thesis by
Steven M. Burns

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

Caltech-CS-TR-88-2

California Institute of Technology
Pasadena, California

December 1987

Contents

List of Figures

Introduction

While the size of circuits implemented on single VLSI chips has increased year to year, tools and abstractions for managing the complexity of these systems have not been developed at an equal pace. We propose a framework for developing provably correct digital systems from an abstract specification through a series of semantics preserving transformations.

$$\text{Specification} \Rightarrow \text{Concurrent Program} \rightarrow \text{Operator Set} \Rightarrow \text{VLSI Chip}$$

The system designer translates the system specification into a concurrent program in a high-level language with a simple semantics. It is in this form, and not the final form of the digital circuit, that the designer proves the system is an implementation of the specification. In this first transformation step, the designer may apply the tools of concurrent program development and verification.

The concurrent program is then transformed into a semantically equivalent self-timed circuit. Alain Martin[5,7] describes a general framework for performing these transformation. In this thesis, we describe a way of automating the compilation procedure.

The final task is the realization of the operator sets in a computing medium such as a VLSI chip. One approach is to implement each operator as a CMOS standard cell and use placement and routing techniques to position and interconnect the cells.

In this thesis, we are only concerned with automatically transforming concurrent programs into self-timed operator sets. The body of this thesis is divided into five logical sections:

- We define a concurrent programming language, a variant of CSP that includes syntax for interconnecting any number of sequential processes.
- We show how to systematically break up large processes into more manageable units based on the basic constructs of the language.
- We implement each basic construct as a collection of self-timed operators.
- We provide refinements to implement processes of arbitrary size by fixed-size constructs.
- We discuss optimization strategies to improve these circuits.

Chapter 1

The Programming Language

We chose to develop a new language for specifying concurrent programs. Our language is based on CSP[3] with the *probe*[8]. A few new constructs have been added to facilitate efficient implementation in VLSI. The new constructs reduce the number of explicit variables needed in some programs by introducing more control structures. The language is described in a bottom-up manner by incrementally extending BNF descriptions of the syntax as the new constructs are introduced. A complete description of the language syntax is given later in the chapter. The low level entities used as terminals in the BNF descriptions are tokens corresponding to punctuation symbols, names and integers. All punctuation symbols appear in the BNF as strings surrounded by single quotes. The tokens *NAME* and *INT* denote terminal symbols for names and integers.

1.1 Sequential Constructs

The sequential parts of our language differ from CSP in only a few ways. We allow only boolean valued variables. Assignments to **true** or **false** are denoted by the raising ($x\uparrow$) or lowering ($x\downarrow$) of a variable. The selection statement has been extended to allow repetition in a subset of cases. Syntactically, a '$*$' is appended to the end of the guarded command if after the guarded command is executed, the selection statement is to be repeated. For example, the guarded structure

$$\alpha; [G_1 \longrightarrow S_1; * [\!] G_2 \longrightarrow S_2]; \beta$$

has the following operational semantics: After the completion of the arbitrary program part α, both G_1 and G_2 are evaluated. If G_1 is true, the command S_1 is executed and the flow of control is returned to the beginning of the guarded structure. If G_2 is true, the command S_2 is executed but the flow of control continues with β. If both G_1 and G_2 are false, the program waits until $G_1 \vee G_2$ becomes true. In the case where both G_1 and G_2 are true, we may define the semantics in two ways. The programmer may choose the interpretation which is most efficient or convenient for a particular program by using different syntactic symbols

⟨seq process⟩ ::= {⟨decl⟩} ⟨sequence⟩
⟨decl⟩ ::= ⟨variable decl⟩
⟨variable decl⟩ ::= **boolean** ⟨boolean spec⟩ {‘,’⟨boolean spec⟩}
⟨boolean spec⟩ ::= *NAME* [‘=’ (**true** | **false**)]
⟨sequence⟩ ::= ⟨statement⟩ {‘;’⟨statement⟩}
⟨statement⟩ ::= **skip** | ⟨assignment⟩ | ⟨guarded structure⟩
⟨assignment⟩ ::= *NAME* (‘↑’ | ‘↓’ | ‘:=’⟨expr⟩)
⟨guarded structure⟩ ::= (‘[’ | ‘*[’) ⟨guarded command set⟩‘]’ |
‘*[’⟨sequence⟩‘]’ | ‘[’⟨expr⟩‘]’
⟨guarded command set⟩ ::= ⟨guarded command⟩ { (‘|’ | ‘▯’) ⟨guarded command⟩}
⟨guarded command⟩ ::= ⟨expr⟩‘⟶’⟨sequence⟩ [‘;’‘*’]
⟨expr⟩ ::= ⟨conjunct⟩ {‘∨’⟨conjunct⟩}
⟨conjunct⟩ ::= ⟨literal⟩ {‘∧’⟨literal⟩}
⟨literal⟩ ::= [‘¬’] ⟨primary⟩
⟨primary⟩ ::= **true** | **false** | *NAME* | ‘(’⟨expr⟩‘)’

Figure 1.1: Sequential Constructs

as separators between the guarded commands. By using the ‘▯’ (read thickbar) separator, the programmer specifies that an arbitrary but mutually exclusive choice is made between the commands S_1 and S_2. If the programmer chooses the ‘|’ (read thinbar) separator, he specifies that no choice will ever need to be made, that is, at most one guard will be true each time the selection statement is executed.

The semantics of a selection statement with the ‘|’ separator may be defined in terms of a selection statement with the ‘▯’ separator.

$$[G_0 \longrightarrow \alpha_0 | \ldots | G_{n-1} \longrightarrow \alpha_{n-1}]$$

is semantically equivalent to

$$[G_0 \longrightarrow \alpha_0 \talloblong \ldots \talloblong G_{n-1} \longrightarrow \alpha_{n-1} \talloblong \bigvee_{i \neq j} G_i \wedge G_j \longrightarrow \textbf{abort}\,]$$

If more than one guard is true when the guards are evaluated, the **abort** statement may be reached, and thus no post-condition of the selection statement may be guaranteed.

We introduce the following abbreviations:

- $[G \longrightarrow$ **skip** $]$ may be written more concisely as $[G]$. We call this construct a *wait*.
- Infinite repetition, $[$**true** $\longrightarrow S; *]$ may be rewritten as $*[S]$.
- The iteration construct,

$$*[G_0 \longrightarrow \alpha_0 \talloblong \ldots \talloblong G_{n-1} \longrightarrow \alpha_{n-1}]$$

is an abbreviation for

$$[G_0 \longrightarrow \alpha_0; * \| \ldots \| G_{n-1} \longrightarrow \alpha_{n-1}; * \| \neg \bigvee_{i=0}^{n-1} G_i \longrightarrow \textbf{skip}]$$

All occurences of '‖' may be replaced by the '|'.

- The assignment statement $v := E$ is an abbreviation for

$$[E \longrightarrow v\!\uparrow \mid \neg E \longrightarrow v\!\downarrow]$$

1.2 Procedures

⟨decl⟩ ::= ⟨procedure decl⟩
⟨procedure decl⟩ ::= **procedure** *NAME*⟨sequence⟩
⟨statement⟩ ::= ⟨procedure call⟩
⟨procedure call⟩ ::= *NAME*

Figure 1.2: Procedure Syntax

Procedures provide a means of sharing code in a sequential process. Procedure declarations of a process are mixed together with port and variable declarations. Statements in the procedure body may reference the ports and variables of the parent process. We specify procedure calls by referencing the procedure's name. Recursion is not allowed.

1.3 Concurrency

A collection of processes will operate concurrently if defined together in a parallel composition statement. A process is either a single sequential process or a (nested) collection of sequential processes.

Synchronization between two processes is accomplished by zero-slack communication actions across communication channels denoted by pairs of named ports. For any given channel, one process may determine whether the other process is waiting on this communication by evaluating a boolean condition called a probe. These probes may be used in arbitrary boolean expressions. Only one of the two processes connected by a channel may use the probe. The port which initiates the communication is called ***active***. The other port, which may use the probe, is called ***passive***. The decision of which port is active and which port is passive must be made at compile time. In our language, the programmer explicitly declares a channel as an active-passive pair of ports.

Concurrently operating processes may not share variables. However, processes may share data by sending values from a small finite set during a synchronization. As an example,

⟨process⟩ ::= ⟨seq process⟩ | ‘(’⟨process⟩ {‘||’⟨process⟩} ‘)’ {⟨channel decl⟩}
⟨statement⟩ ::= ⟨communication⟩
⟨communication⟩ ::= *NAME* [‘(’*INT*‘)’] [‘:’‘[’⟨response⟩ {‘|’⟨response⟩} ‘]’] |
NAME‘?’*NAME* | *NAME*‘!’⟨expr⟩
⟨response⟩ ::= *INT*‘⟶’⟨sequence⟩
⟨primary⟩ ::= $\overline{NAME}$ [‘(’*INT*‘)’]
⟨channel decl⟩ ::= **channel** ⟨channel spec⟩ {‘,’⟨channel spec⟩}
⟨channel spec⟩ ::= ‘(’*NAME*‘,’*NAME*‘)’
⟨decl⟩ ::= ⟨port decl⟩
⟨port decl⟩ ::= (**passive** | **active**) ⟨port spec⟩ {‘,’⟨port spec⟩}
⟨port spec⟩ ::= *NAME* [‘(’*INT*‘,’*INT*‘)’ | ‘?’ | ‘!’]

Figure 1.3: Concurrency Constructs

consider a pair of processes with an interconnecting channel. One process cyclicly sends the values $0, 1, 2$ across the channel (L, R) to a second process. The second process cyclicly performs a read communication and based on the value received executes the code segments α, β or γ. We may code such a process pair by

```
(active L(3,1)
   *[L(0); L(1); L(2)]
||passive R(1,3)
   *[R : [0 ⟶ α|1 ⟶ β|2 ⟶ γ]]
)channel (L, R)
```

We note now that the language construct $R : [\ldots]$ is *not* a selection statement. It denotes a communication action from the port R. Alternate executions occur based on the value received by the communication. In this example, the port L sends a value to port R. We specify the set of values that may be sent from L by a single integer, in this case 3, short for the set of values $\{0, 1, 2\}$. We use 1 to represent the set of values $\{0\}$ that may be sent from R to L. Note that the value set $\{0\}$ consists of a single value and thus no data is transferred from R to L during the synchronization. When a port is declared, both its send and receive sets are specified. Its name and its active or passive nature are also included in the declaration. There need be no association between the direction of information flow and the active or passive nature of the port. The port declarations **passive** $L(3, 1)$ and **active** $R(1, 3)$ are also perfectly valid.

As a natural generalization of the communication construct, data may be sent in both directions during a synchronization. When declaring a channel, one may specify both multiple input values as well as multiple output values. Also, the probe is generalized to test which value the other process is sending before actually performing the communication action. All communications will be defined in terms of this general form.

$$
\begin{array}{ll}
\{\mathbf{c}L = C \wedge 0 \leq i < m\}L(i): & [0 \longrightarrow \{\mathbf{c}L = C+1 \wedge j=0\}\alpha_0 \\
& |1 \longrightarrow \{\mathbf{c}L = C+1 \wedge j=1\}\alpha_1 \\
& \vdots \\
& |n-1 \longrightarrow \{\mathbf{c}L = C+1 \wedge j=n-1\}\alpha_{n-1} \\
&] \\
\{\mathbf{c}R = C \wedge 0 \leq j < n\}R(j): & [0 \longrightarrow \{\mathbf{c}R = C+1 \wedge i=0\}\beta_0 \\
& |1 \longrightarrow \{\mathbf{c}R = C+1 \wedge i=1\}\beta_1 \\
& \vdots \\
& |m-1 \longrightarrow \{\mathbf{c}R = C+1 \wedge i=m-1\}\beta_{m-1} \\
&]
\end{array}
$$

Figure 1.4: Hoare Triple Semantics of the General Communication Action

We give a more formal semantics of the general communication action by introducing the ghost variables $\mathbf{c}X$ and $\mathbf{q}X$ attached to the communication port X. The integer valued variable $\mathbf{c}X$ represents the number of completed communication action through the port X. The boolean $\mathbf{q}X$ states whether X is ready but not able to finish a communication action. The standard zero slack axioms apply[4].

$$
\begin{array}{c}
\neg\mathbf{q}L \vee \neg\mathbf{q}R \\
\mathbf{c}L = \mathbf{c}R
\end{array}
$$

A Hoare triple semantics of the data transmission mechanism is given in Figure 1.4. The communications L and R are performed by separate sequential processes. Information is exchanged during the communication. One of m values is transmitted from port L to port R. Simultaneously, one of n values is sent from R to L. The synchronization axioms require the communication action to complete simultaneously. After the exchange has completed, the control flow of each process continues with different sequences of commands based on the values received during the communication. The passive-active distinction is only needed when the probe is used. Let L be a passive port. Then, we define the meaning of the probes $\overline{L}(0), \ldots, \overline{L}(n-1)$ of L as

$$
\begin{array}{c}
\overline{L}(k) \Rightarrow (\mathbf{q}R \wedge k = j) \\
(\mathbf{q}R \wedge k = j) \Rightarrow \Diamond\overline{L}(k)
\end{array}
$$

The diamond is a formalism borrowed from temporal logic, meaning eventually the predicate following the diamond will hold. In this case, if R is suspended and $k = j$, the probe $\overline{L}(k)$ will eventually hold.

Various abbreviations apply to the communication syntax. Consider the L action in Figure 1.4.

- $L(i)$ means $L(i) : [0 \longrightarrow \mathbf{skip}]$ and is only applicable if L has only one input value.

- If L has only one output value, the abbreviation

$$L : [\alpha_1 \longrightarrow \beta_1 | \ldots | \alpha_n \longrightarrow \beta_n]$$

means

$$L(0) : [\alpha_1 \longrightarrow \beta_1 | \ldots | \alpha_n \longrightarrow \beta_n]$$

- L combines the two previous abbreviations.
- $\overline{L}$ means $(\overline{L}(0) \vee \ldots \vee \overline{L}(n-1))$.
- In port declarations, $L?$ and $L!$ are short for $L(1,2)$ and $L(2,1)$, respectively.
- $L?x$ abbreviates $L : [0 \longrightarrow x\downarrow | 1 \longrightarrow x\uparrow]$
- $L!E$ abbreviates $[E \longrightarrow L(1) | \neg E \longrightarrow L(0)]$

1.4 Definition and Instantiation

⟨program⟩ ::= {⟨process def⟩} ⟨program def⟩
⟨program def⟩ ::= ⟨process⟩ **program** *NAME* [⟨port list⟩]
⟨process def⟩ ::= ⟨process⟩ **process** *NAME*⟨port list⟩
⟨process⟩ ::= ⟨process inst⟩
⟨process inst⟩ ::= **instance** *NAME*⟨port list⟩
⟨port list⟩ ::= '('*NAME* {','*NAME*} ')'

Figure 1.5: Definition and Instantiation Constructs

A completely specified concurrent program consists of a set of processes and a list of ports to connect with the environment. The ⟨program⟩ nonterminal provides the necessary syntax. The ports connected to the environment are parameters to the program definition.

As an aid to the programmer for constructing large sets of processes, we allow processes to be separately defined and assigned a name. These definitions are placed before the program specification. All ports and variables not internal to the definition must be declared as parameters. No variables or ports may be inherited from the surrounding scope when these code segments are instantiated.

We illustrate the instantiation mechanism with a complete example (see Figure 1.6), a ring of four processes which insures mutually exclusive access to a single resource[7]. The process ring is created by instantiating three *priv0* processes and one *priv1* process. The L and R ports are connected via channels to form a ring of four elements. The four ports $U0, U1, U2, U3$ connect to the environment.

```
passive $U, L$
active $R$
boolean $b$ = false
procedure $P$   [$b \longrightarrow$ skip $|\neg b \longrightarrow R$]
*[[$\overline{U} \longrightarrow P; b\uparrow; U$
  $\|\overline{L} \longrightarrow P; b\downarrow; L$
 ]]
process $priv0(U, L, R)$

passive $U, L$
active $R$
boolean $b$ = true
procedure $P$   [$b \longrightarrow$ skip $|\neg b \longrightarrow R$]
*[[$\overline{U} \longrightarrow P; b\uparrow; U$
  $\|\overline{L} \longrightarrow P; b\downarrow; L$
 ]]
process $priv1(U, L, R)$

passive $U0, U1, U2, U3, L0, L1, L2, L3$
active $R0, R1, R2, R3$
(instance $priv1(U0, L0, R0)$
$\|$instance $priv0(U1, L1, R1)$
$\|$instance $priv0(U2, L2, R2)$
$\|$instance $priv0(U3, L3, R3)$
)channel $(L0, R3), (L1, R0), (L2, R1), (L3, R2)$
program $ring(U0, U1, U2, U3)$
```

Figure 1.6: Program for Mutual Exclusion Ring

1.5 Grammar Rules

Figure 1.7 displays a complete BNF description of the context-free syntax of our language. A few context-sensitive rules need to be added.

- All names must be declared before they are used.
- Each channel declaration must name one active and one passive port. The number of input symbols of each port must match the number of output symbols of the other.
- Only variable names may be used on the left hand side of an assignment statement.
- Port names must be used in communication actions. Furthermore, the integer symbols must fall within the range declared.
- When instantiating processes or statements, the number and types of parameters must match the number and types in the definition.
- Recursive procedure calls are not allowed.
- Only passive ports may be named in probes.
- The separators '$\|$' and '$|$' may not be used in the same selection or repetition statement.
- The repetition abbreviation may not be used if any of the guarded sequences use the trailing '$*$' construct.

1.6 New Constructs

Our language extends CSP in a number of ways. Most of the new constructs were added to reduce the number of explicit variables need to code common programs.

Repetitive selection — The following simple repetition is more difficult to code without the trailing '$*$' construct:

$$\ldots ; [\overline{X} \longrightarrow \alpha ; X ; * | \overline{Y} \longrightarrow Y] ; \ldots$$

Without this control construct, an explicit variable must be added, as in:

$$\ldots ; b\!\uparrow ; *[b \longrightarrow [\overline{X} \longrightarrow \alpha ; X | \overline{Y} \longrightarrow Y ; b\!\downarrow]] ; \ldots$$

Such iteration constructs are common in concurrent programs using the probe.

Branching Communication — The selection-like statement as a part of the communication action was introduced to provide a means of changing the flow of control after a read without the explicit use of a variable. For example, the code segment:

$$\ldots ; X : [0 \longrightarrow \alpha_0 | 1 \longrightarrow \alpha_1] ; \ldots$$

⟨program⟩ ::= {⟨process def⟩} ⟨program def⟩

⟨program def⟩ ::= ⟨process⟩ **program** *NAME* [⟨port list⟩]

⟨process def⟩ ::= ⟨process⟩ **process** *NAME*⟨port list⟩

⟨port list⟩ ::= '('*NAME* {','*NAME*} ')'

⟨process⟩ ::= ⟨process inst⟩ | ⟨seq process⟩ |

'('⟨process⟩ {'||'⟨process⟩} ')' {⟨channel decl⟩}

⟨process inst⟩ ::= **instance** *NAME*'('*NAME* {','*NAME*} ')'

⟨seq process⟩ ::= {⟨decl⟩} ⟨sequence⟩

⟨channel decl⟩ ::= **channel** ⟨channel spec⟩ {','⟨channel spec⟩}

⟨channel spec⟩ ::= '('⟨NAME⟩','⟨NAME⟩')'

⟨decl⟩ ::= (⟨port decl⟩ | ⟨procedure decl⟩ | ⟨variable decl⟩)

⟨port decl⟩ ::= (**passive** | **active**) ⟨port spec⟩ {','⟨port spec⟩}

⟨port spec⟩ ::= *NAME* [('('*INT*','*INT*')' | '?' | '!')]

⟨procedure decl⟩ ::= **procedure** *NAME*⟨sequence⟩

⟨variable decl⟩ ::= **boolean** ⟨boolean spec⟩ {','⟨boolean spec⟩}

⟨boolean spec⟩ ::= *NAME* ['=' (**false** | **true**)]

⟨sequence⟩ ::= ⟨statement⟩ {';'⟨statement⟩}

⟨statement⟩ ::= **skip** | ⟨assignment⟩ | ⟨guarded structure⟩ |

⟨communication⟩ | ⟨procedure call⟩

⟨assignment⟩ ::= *NAME* ('↑' | '↓' | ':='⟨expr⟩)

⟨procedure call⟩ ::= *NAME*

⟨guarded structure⟩ ::= ('[' | '*[') ⟨guarded command set⟩']' |

'*['⟨sequence⟩']' | '['⟨expr⟩']'

⟨guarded command set⟩ ::= ⟨guarded command⟩ { ('|' | '▮') ⟨guarded command⟩}

⟨guarded command⟩ ::= ⟨expr⟩'⟶'⟨sequence⟩ [';''*']

⟨expr⟩ ::= ⟨conjunct⟩ {'∨'⟨conjunct⟩}

⟨conjunct⟩ ::= ⟨literal⟩ {'∧'⟨literal⟩}

⟨literal⟩ ::= ['¬'] ⟨primary⟩

⟨primary⟩ ::= **true** | **false** | *NAME* | $\overline{NAME}$ ['('*INT*')'] | '('⟨expr⟩')'

⟨communication⟩ ::= *NAME* ['('*INT*')'] [':''['⟨response⟩ {'|'⟨response⟩} ']']

⟨response⟩ ::= *INT*'⟶'⟨sequence⟩

Figure 1.7: BNF Description of the Programming Language

requires an explicit variable without this construct. We must use the explicit variable x:

$$\ldots; X?x; [\neg x \longrightarrow \alpha_0 | x \longrightarrow \alpha_1]; \ldots$$

Different Selection Semantics — The distinction between the two different types of selection semantics is necessary on efficiency grounds. Extra circuitry is required to implement the mutually exclusive choice of the '$[\!]$' construct. It is needed in some programs, but when the programmer can prove exclusion of the guards, the arbitration circuitry may be avoided. We have considered allowing the compiler to decide when this optimization can be applied, but then the dynamic nature of the computation can not be considered. There would always be some programs for which mutual exclusion is not detected. We force the programmer to make this optimization decision.

Definitions — The definition facility is a compromise between nothing at all and a full-fledged language for constructing process connection graphs. Circuits for each definitions may be separately compiled and connected with the circuit for the main program at the operator level, allowing a concise operator description of programs which generate circuits with regular structures.

Chapter 2

Decomposition

Our goal is to produce a translation procedure that transforms an arbitrary program of the language introduced in the last chapter into a self-timed circuit. The method we demonstrate faithfully applies the low level techniques described in [7,5]. However, we choose to do most of the work at a higher level by first decomposing each process into a collection of smaller, more easily compiled processes. We use the syntax of the original program to guide the decomposition.

To ease discussion of the translation procedure, we somewhat arbitrarily divide it into several stages. In the actual compiler, more than one stage might be performed in the same pass through the source code.

Stage 1. Each process definition is compiled separately. All syntactic abbreviations are expanded to their most general forms. The resulting expanded form is a syntactically correct process definition.

Stage 2. New sub-processes are created to implement assignments to variables and procedures. All assignments to a particular variable in the original process are transformed into communications with a new sub-process designed explicitly to assign values to the variable. Procedure calls are transformed into communications with the new sub-process implementing the procedure body.

Stage 3. We decompose each sequential process into a collection of simple sub-processes, each sub-process implementing a basic construct of the language. Each simple sub-process is associated with a nonterminal ⟨statement⟩ of the source language.

Stage 4. Further non-syntax-directed decompositions are performed. One decomposition transforms the nonterminal ⟨sequence⟩ into a collection of simple sequencing sub-processes. Exclusive guard sets are transformed into simple sub-processes corresponding to conjunctions of possibly negated variables and probes. Sub-processes that continually poll the values of variables and probes are used as implementations of non-exclusive guard sets.

The resulting collection of processes and sub-processes semantically equivalent to the original program may now be implemented as a self-timed circuit. In the next chapter, we develop methods of implementing channels, as well as rules for compiling each of the remaining sub-processes into a self-timed circuit.

```
passive $U(1,1), L(1,1)$
active $R(1,1)$
boolean $b =$ false
procedure $P$
   $[b \longrightarrow$ skip $| \neg b \longrightarrow R(0) : [0 \longrightarrow$ skip $]]$
$[$true $\longrightarrow$   $[\overline{U}(0) \longrightarrow P; b\uparrow; U(0) : [0 \longrightarrow$ skip $]$
                 $|\overline{L}(0) \longrightarrow P; b\downarrow; L(0) : [0 \longrightarrow$ skip $]$
                 $]; *]$
process $priv0(U, L, R)$
```

Figure 2.1: Expanded Process *priv0*

2.1 Abbreviations

All abbreviations are replaced by their implementation in the general form of the construct. In particular, waits, infinite repetitions, iterations and assignments are replaced by the general forms involving only the selection construct with the optional tailing '*'. (see Page 3) Abbreviated communication actions are replaced by full communication actions by filling in both the output value sent and alternate execution paths based on the input received. All the abbreviations described on Page 6 are performed. The expansion of these abbreviations seemingly introduces inefficiencies because the most general construct is potentially more difficult to implement than the simplier constructs. In Chapter 5, we will define optimization procedures that remove the inefficiencies introduced here. The early expansion of abbreviations eases the discussion of the decompositions without compromising the *quality*, i.e. the performance in both space and time, of the implementation.

The code of the *ring* program (Figure 1.6) expands to the code shown in Figure 2.1 after these transformations are performed. Notice that the abbreviations for infinite repetition, communication, and probes have been expanded to their general forms.

2.2 Decomposition and Sub-Processes

When implementing a complex process, one may wish to divide the process code into pieces and compile the pieces separately. Process decomposition[5] provides a method of structuring the division of process code. A process may be decomposed by moving a statement or sequence of statements from the original process to a new concurrently executing process. Synchronization between the main process and the newly created *sub-process* is performed by a communication action over a new channel connecting the process and sub-process. For instance, the program part

$$\alpha; \beta; \gamma$$

may be decomposed into a sub-process implementing β, an interconnecting channel, and the original program part with β replaced by an active communication.

$$(\textbf{passive}\, D \quad *[[\overline{D} \longrightarrow \beta; D]] \| \textbf{active}\, D' \quad \alpha; D'; \gamma)\textbf{channel}\,(D', D)$$

By the structure of this decomposition, no command in the sub-process will execute concurrently with a command in the original process. This ***fundamental safety property*** is always obeyed by the decompositions that follow. We allow the process and sub-process created by process decomposition to share variables and channel ports. The language syntax is extended for the description of the decomposition to allow variables and ports to be shared between processes. This is done by adding the BNF rule:

$$\langle\text{process}\rangle ::= \{\langle\text{decl}\rangle\}\ \langle\text{process}\rangle$$

The channel introduced by process decomposition may be implemented more simply than the general communication actions because the signaling protocol of this channel may be interleaved (reshuffled) with other communications. We leave the syntax describing communications on these channels in the abbreviated form as a way of distinguishing them from the general communications specified in the original program.

We apply process decomposition to the assignment of variables. All assignments to a particular boolean variable are performed by a single sub-process owning the variable. For example, the sequence of statements

$$x\uparrow; \alpha; x\downarrow; \beta; x\uparrow$$

is decomposed by introducing a simple register sub-process and two channels. Each assignment to a variable is replaced by an active communication to this server process.

$$\begin{array}{l}(\textbf{passive}\, C, D \\ \quad *[[\overline{C} \longrightarrow x\uparrow; C | \overline{D} \longrightarrow x\downarrow; D]] \\ \| \textbf{active}\, C', D' \\ \quad C'; \alpha; D'; \beta; C' \\ )\textbf{channel}\,(C', C), (D', D)\end{array}$$

Procedures are implemented by directly applying the process decomposition. The procedure body is made into a sub-process and is surrounded by a probed passive communication action. Each procedure call becomes a simple active communication.

We apply these decompositions to the process *priv0* of our ring example in Figure 2.2. A sub-process for the variable b and a sub-process for the procedure P are created during this decomposition.

2.3 Syntactic Decomposition

The crux of our implementation strategy is a decomposition of each concurrent program based on its parse tree. As the parser recognizes larger constructs, these larger constructs

```
passive $U(1,1), L(1,1)$
active $R(1,1)$
boolean $b =$ false
(passive $P$
    $*[[\overline{P} \longrightarrow [b \longrightarrow$ skip $| \neg b \longrightarrow R(0) : [0 \longrightarrow$ skip $]]; P]]$
||passive $D, E$
    $*[[\overline{D} \longrightarrow b\uparrow; D | \overline{E} \longrightarrow b\downarrow; E]]$
||active $P', D', E'$
    [true $\longrightarrow$  $[\overline{U}(0) \longrightarrow P'; D'; U(0) : [0 \longrightarrow$ skip $]$
                     $\| \overline{L}(0) \longrightarrow P'; E'; L(0) : [0 \longrightarrow$ skip $]$
                     $]; *]$
)channel $(P', P), (D', D), (E', E)$
process $priv0(U, L, R)$
```

Figure 2.2: Register and Procedure Decomposition of *priv0*

are divided up, using process decomposition, into sub-processes which implement only a single construct. The splitting is repeated on all the generated sub-processes until the only remaining sub-process forms are those corresponding to a basic statements of the language, as shown in Figure 2.3. Expressions are not sub-divided until the next phase of the decomposition. In this figure, the port names C_i' and C_i represent active and passive ends of synchronization channels. The γ_i represent arbitrary boolean expressions and x represents an arbitrary variable.

Continuing on with our example, syntactic decomposition of the process *priv0* yields the collection of sub-processes shown in Figure 2.4. The first sub-process requires no further decomposition. The next sub-process ($*[[\overline{P} \longrightarrow \ldots]]$) is decomposed into sub-processes for the following statements: the guarded structure with b and $\neg b$ in the guards and the communication on R. For brevity, we do not show the expansion of the **skip** statements into separate sub-processes. The main body of the process is called by a single active communication action G'. Further decomposition yields a sub-process for infinite repetition (guarded by $\overline{G}$), one for selection between two boolean expressions ($\overline{H}$), two for sequencing three active communications ($\overline{K}_0$ and $\overline{K}_1$) and two for performing passive communications ($\overline{J}_0$ and $\overline{J}_1$).

2.4 Non-Syntactic Decompositions

2.4.1 Sequencing Sub-process

For simplicity and compatibility with other compilations, we decompose the sub-process

$$*[[\overline{D} \longrightarrow S_0'; S_1'; \ldots; S_{n-1}'; D]]$$

$$\begin{array}{lll}
\langle\text{process}\rangle & (C'_0\|C'_1\|\ldots\|C'_{n-1}) & \\
\langle\text{sequence}\rangle & *[[\overline{C} \longrightarrow C'_0;C'_1;\ldots;C'_{n-1};C]] & \\
\langle\text{statement}\rangle & *[[\overline{C} \longrightarrow \textbf{skip}\,;C]] & \\
\langle\text{statement}\rangle & *[[\overline{C}_1 \longrightarrow x\uparrow;C_1|\overline{C}_0 \longrightarrow x\downarrow;C_0]] & \\
\langle\text{statement}\rangle & *[[\overline{C} \longrightarrow & [\gamma_0 \longrightarrow C'_0;*|\ldots|\gamma_{k-1} \longrightarrow C'_{k-1};* \\
 & & |\gamma_k \longrightarrow C'_k|\ldots|\gamma_{n-1} \longrightarrow C'_{n-1} \\
 & &];C]] \\
\langle\text{statement}\rangle & *[[\overline{C} \longrightarrow & [\gamma_0 \longrightarrow C'_0;*\|\ldots\|\gamma_{k-1} \longrightarrow C'_{k-1};* \\
 & & \|\gamma_k \longrightarrow C'_k\|\ldots\|\gamma_{n-1} \longrightarrow C'_{n-1} \\
 & &];C]] \\
\langle\text{statement}\rangle & *[[\overline{C} \longrightarrow A(i) : [0 \longrightarrow C'_0|\ldots|n-1 \longrightarrow C'_{n-1}];C]] &
\end{array}$$

Figure 2.3: Resulting Process Forms After Syntactic Decomposition

into a number of simple sub-processes of the form

$$*[[\overline{C} \longrightarrow S';T';C]]$$

These simple sub-processes may be connected together in a variety of ways as we shall see in Chapter 5. For now, we assume a simple linear chain defined inductively by:

$$\begin{array}{l}
(\textbf{passive}\, C \\
\quad *[[\overline{C} \longrightarrow S'_1;\ldots;S'_{n-1};C]] \\
\|\textbf{active}\, C' \\
\quad *[[\overline{D} \longrightarrow S'_0;C';D]] \\
)\textbf{channel}\,(C',C)
\end{array}$$

2.4.2 Control Sub-Process

The repetition and selection sub-process is also decomposed before compilation. The sub-process,

$$*[[\overline{D} \longrightarrow [\gamma_0 \longrightarrow S'_0;*\|\ldots\|\gamma_k \longrightarrow S'_k\|\ldots];D]]$$

which is equivalent to

$$*[[\overline{D} \longrightarrow [\gamma_0 \longrightarrow S'_0\|\ldots\|\gamma_k \longrightarrow S'_k;D\|\ldots]]]$$

is split into two sub-processes with a wide interconnecting communication channel.

$$\begin{array}{l}
(\textbf{active}\, P'(1,n) \\
\quad *[[\overline{D} \longrightarrow P' : [0 \longrightarrow S'_0|\ldots|k \longrightarrow S'_k;D|\ldots]]] \\
\|\textbf{passive}\, P(n,1) \\
\quad *[[\overline{P}\wedge\gamma_0 \longrightarrow P(0)\|\ldots\|\overline{P}\wedge\gamma_{n-1} \longrightarrow P(n-1)]] \\
)\textbf{channel}\,(P',P)
\end{array}$$

```
passive $U(1,1), L(1,1)$
active $R(1,1)$
boolean $b =$ false
(passive $P$
    (passive $T$
        $*[[\overline{T} \longrightarrow R(0) : [0 \longrightarrow$ skip $]; T]]$
    ||active $T'$
        $*[[\overline{P} \longrightarrow [b \longrightarrow$ skip $| \neg b \longrightarrow T']; P]]$
    )channel $(T', T)$
||passive $D, E$
    $*[[\overline{D} \longrightarrow b\uparrow; D | \overline{E} \longrightarrow b\downarrow; E]]$
||active $P', D', E'$
    (passive $H$
        (passive $K_0$
            (passive $J_0$
                $*[[\overline{J}_0 \longrightarrow U(0) : [0 \longrightarrow$ skip $]; J_0]]$
            ||active $J'_0$
                $*[[\overline{K}_0 \longrightarrow P'; D'; J'_0; K_0]]$
            )channel $(J'_0, J_0)$
        ||passive $K_1$
            (passive $J_1$
                $*[[\overline{J}_1 \longrightarrow L(0) : [0 \longrightarrow$ skip $]; J_1]]$
            ||active $J'_1$
                $*[[\overline{K}_1 \longrightarrow P'; E'; J'_1; K_1]]$
            )channel $(J'_1, J_1)$
        ||active $K'_0, K'_1$
            $*[[\overline{H} \longrightarrow [\overline{U}(0) \longrightarrow K'_0 \| \overline{L}(0) \longrightarrow K'_1]; H]]$
        )channel $(K'_0, K_0), (K'_1, K_1)$
    ||active $H'$
        (passive $G$
            $*[[\overline{G} \longrightarrow [$true $\longrightarrow H'; *]; G]]$
        ||active $G'$
            $G'$
        )channel $(G', G)$
    )channel $(H', H)$
)channel $(P', P), (D', D), (E', E)$
process $priv0(U, L, R)$
```

Figure 2.4: Syntactic Decomposition of the *priv0* process

This decomposition allows the guard sub-process to completely evaluate the guards before the computation continues, providing a nice separation of concerns between evaluation and program flow. None of the actions $S_i^!$ may effect guard evaluation because their activities are sequenced. However, in some cases this separation introduces an unneeded state in the program flow. In Chapter 5, we discuss way of removing the inefficiencies introduced in this step.

This same decomposition is performed on the sub-processes with '|' separators. We now further decompose the guard set sub-processes. There are two types of decompositions, one for each type of separator.

2.4.3 Exclusive Guards

In the mutually exclusive guard set

$$*[[\overline{Q} \wedge \gamma_0 \longrightarrow Q(0)| \ldots |\overline{Q} \wedge \gamma_{n-1} \longrightarrow Q(n-1)]]$$

we know no two guards $\overline{Q} \wedge \gamma_i$ are true at the same time. We may split this sub-process into n sub-processes of the very simple structure

$$*[[\overline{Q} \wedge \gamma_i \longrightarrow Q(i)]]$$

Each sub-process is then further decomposed independently. This technique is not a legal decomposition if the guards are not mutually exclusive. In the case of non-exclusive guards, this decomposition yields several sub-processes that may run concurrently, violating the fundamental safety requirement.

Let F be the boolean expression represented by the syntax $\overline{Q} \wedge \gamma_i$. If we expand F into sum of products form, $F = \bigvee_{j=0}^{m-1} f_j$, we may expand the sub-process to have more guards, but each of a simple conjunctive form. We may now use the sub-process

$$*[[f_0 \longrightarrow Q(i)\| \ldots \| f_{m-1} \longrightarrow Q(i)]]$$

as an implementation of the above sub-process. However, if we want to repeat the decomposition into sub-processes with a single guard, we must insure mutual exclusion among the f_j. We give a simple algorithm for transforming a boolean expression F into a form suitable for implementation by this method.

Algorithm: Disjoint Disjunctive Form

```
Let $F = \bigvee_{f_i \in C} f_i$ where each $f_i \in C$ is a conjunction of literals.
while $\exists(f_i, f_j : C : f_i \wedge f_j$ is satisfiable)
  remove $f_i$ from $C$
  add each term of the disjunctive expansion of $f_i \wedge \neg f_j$ to $C$
```

If we use the sum of products form produced from this algorithm, all mutually exclusive guard sets may be decomposed into collections of sub-processes of this form:

$$*[[\overline{Q} \wedge x_0 \wedge \ldots \wedge x_{k-1} \wedge \neg x_k \wedge \ldots \wedge \neg x_{n-1} \longrightarrow Q(i)]]$$

2.4.4 Non-exclusive Guards

The decomposition technique described for exclusive guards suffers from several inadequacies. The resulting guards, while being simple conjuncts, are still of arbitrary size. Furthermore, the total size of the sub-processes implementing the guard set may be exponential in the number of variables named in the guard set. We may avoid these problems by evaluating the guards sequentially.

We consider, for the time being, guard set sub-processes with only variables. Probes will be added later as a natural extension. The guards $\gamma_0, \gamma_1, \ldots, \gamma_{n-1}$ of the sub-process

$$*[[\overline{Q} \longrightarrow [\gamma_0 \longrightarrow Q(0) \,\|\, \ldots \,\|\, \gamma_{n-1} \longrightarrow Q(n-1)]]]$$

are evaluated cyclically. This type of busy waiting may be implemented by

```
*[[$\overline{Q}$ $\longrightarrow$[$\gamma_0$ $\longrightarrow$  $Q(0)$
           |$\neg\gamma_0$ $\longrightarrow$[$\gamma_1$ $\longrightarrow$ $Q(1)$
                    |$\neg\gamma_1$ $\longrightarrow$ ... [$\gamma_{n-1}$ $\longrightarrow$ $Q(n-1)$
                                 |$\neg\gamma_{n-1}$ $\longrightarrow$ skip
 ]]        ]        ]            ]
```

If a guard evaluates to true, the communication on Q is performed and the sub-process waits again for the probe of Q. The guards are reevaluated in order if all the guards evaluate to false since $\overline{Q}$ still holds.

This guard sub-process may be decomposed syntactically in the same manner as any other process, creating a new sub-process for each guard γ_i of the form:

$$*[[\overline{V_i} \longrightarrow [\gamma_i \longrightarrow V_i(1) | \neg\gamma_i \longrightarrow V_i(0)]]]$$

and a control sub-process of the form:

```
*[[$\overline{Q}$ $\longrightarrow$ $V_0'$ :[1 $\longrightarrow$      $Q(0)$
              |0 $\longrightarrow$ $V_1'$ :[1 $\longrightarrow$ $Q(1)$
                        |0 $\longrightarrow$ ... $V_{n-1}'$ :[1 $\longrightarrow$ $Q(n-1)$
                                        |0 $\longrightarrow$ skip
 ]]           ]         ]               ]
```

These sub-processes communicate through the n channels

$$\textbf{active } V_i'(1,2) \textbf{ passive } V_i(2,1) \ \ldots \ \textbf{channel}\,(V_i', V_i)$$

We do not create sub-processes for evaluating both γ_i and $\neg\gamma_i$. Instead, we evaluate one to either true or false. If γ_i is a simple variable x, the sub-process is

$$*[[\overline{V} \longrightarrow [x \longrightarrow V(1) | \neg x \longrightarrow V(0)]]]$$

If γ_i is the negation of another expression δ, we perform the decomposition

$$\begin{aligned}
&(\textbf{passive}\, U(2,1)\\
&\quad *[[\overline{U} \longrightarrow [\delta \longrightarrow U(1)|\neg\delta \longrightarrow U(0)]]]\\
&\|\textbf{active}\, U'(1,2)\\
&\quad *[[\overline{V} \longrightarrow U' : [0 \longrightarrow V(1)|1 \longrightarrow V(0)]]]\\
&)\textbf{channel}\,(U', U)
\end{aligned}$$

If γ_i is the conjunction of two expressions δ_0 and δ_1, we use the following decomposition:

$$\begin{aligned}
&(\textbf{passive}\, U_0(2,1)\\
&\quad *[[\overline{U_0} \longrightarrow [\delta_0 \longrightarrow U_0(1)|\neg\delta_0 \longrightarrow U_0(0)]]]\\
&\|\textbf{passive}\, U_1(2,1)\\
&\quad *[[\overline{U_1} \longrightarrow [\delta_1 \longrightarrow U_1(1)|\neg\delta_1 \longrightarrow U_1(0)]]]\\
&\|\textbf{active}\, U_0'(1,2), U_1'(1,2)\\
&\quad *[[\overline{V} \longrightarrow U_0' : [0 \longrightarrow V(0)|1 \longrightarrow U_1' : [0 \longrightarrow V(0)|1 \longrightarrow V(1)]]]]\\
&)\textbf{channel}\,(U_0', U_0), (U_1', U_1)
\end{aligned}$$

By De Morgan's rule, disjunction is similar to conjunction.

By allowing probes in expressions, we can no longer be assured that the sequential evaluation of the expressions will result in a possible instantaneous evaluation of the expressions. For example, the expressions $\neg\overline{X} \wedge \overline{X}$ might evaluate to true if the probe's value changes between the first and second evaluation of the probe. We add extra circuitry to prevent this inconsistency. If any of the guards of the original guard set contain probes, we create a stable copy of the probe's value before evaluating any of the guards. Each probe is evaluated and a stable copy is assigned to a variable each time though the busy-waiting loop. This insures the probe's value is identical everywhere it is referenced in the guard set. Such an extreme procedure is not always required. We may save the extra variable used to store the stable value if the probe's value is used only once in the guarded command set. In either case, it is sufficient to use the sub-process

$$*[[\overline{V} \longrightarrow [\overline{X} \longrightarrow V(1)[\!]\neg\overline{X} \longrightarrow V(0)]]]$$

to evaluate a probe.

2.5 Example

We illustrate some of the non-syntactic decompositions in of the *priv0* process in Figure 2.5 The sub-process $*[[\overline{P} \longrightarrow \ldots]]$ is further decomposed into control and guard sub-processes. The guard sub-processes are created using the techniques for decomposing exclusive guard sets. The sub-process with the **true** expression is compiled similarly, except the control process forms an infinite repetition. The sub-process $*[[\overline{H} \longrightarrow \ldots]]$ is split into sub-processes using the technique for non-exclusive guard sets. The guard process busy–waits to arbitrate between the probes of *U*0 and *L*0. The other decompositions performed in this phase but not shown in the figure are the division of the sub-processes which sequence three actions into two sub-processes each sequencing two actions.

```
$*[[\overline{P} \longrightarrow [b \longrightarrow \textbf{skip} \mid \neg b \longrightarrow T']; P]] \rhd$
    $(\textbf{passive}\ B(2,1)$
        $*[[\overline{B} \wedge b \longrightarrow B(0)]] \parallel *[[\overline{B} \wedge \neg b \longrightarrow B(1)]]$
    $\parallel \textbf{active}\ B'(1,2)$
        $*[[\overline{P} \longrightarrow B' : [0 \longrightarrow \textbf{skip} \mid 1 \longrightarrow T']; P]]$
    $)\textbf{channel}\ (B', B)$

$*[[\overline{H} \longrightarrow [\overline{U}(0) \longrightarrow K'_0 [\!] \overline{L}(0) \longrightarrow K'_1]; H]] \rhd$
    $(\textbf{passive}\ A_0(2,1)$
        $*[[\overline{A_0} \wedge \overline{U}(0) \longrightarrow A_0(1) [\!] \overline{A_0} \wedge \neg \overline{U}(0) \longrightarrow A_0(0)]]$
    $\parallel \textbf{passive}\ A_1(2,1)$
        $*[[\overline{A_1} \wedge \overline{L}(0) \longrightarrow A_1(1) [\!] \overline{A_1} \wedge \neg \overline{L}(0) \longrightarrow A_1(0)]]$
    $\parallel \textbf{passive}\ C(2,1) \quad \textbf{active}\ A'_0(1,2), A'_1(1,2)$
        $*[[\overline{C} \longrightarrow A'_0 : [1 \longrightarrow C(0) \mid 0 \longrightarrow A'_1 : [1 \longrightarrow C(1) \mid 0 \longrightarrow \textbf{skip}\ ]]]]$
    $\parallel \textbf{active}\ C'(1,2)$
        $*[[\overline{H} \longrightarrow C' : [0 \longrightarrow K'_0 \mid 1 \longrightarrow K'_1]; H]]$
    $)\textbf{channel}\ (C', C), (A'_0, A_0), (A'_1, A_1)$

$*[[\overline{G} \longrightarrow [\textbf{true} \longrightarrow H'; *]; G]] \rhd$
    $(\textbf{passive}\ D(2,1)$
        $*[[\overline{D} \wedge \textbf{true} \longrightarrow D(1)]] \parallel *[[\textbf{false} \longrightarrow D(0)]]$
    $\parallel \textbf{active}\ D'(1,2)$
        $*[[\overline{G} \longrightarrow D' : [1 \longrightarrow H' \mid 0 \longrightarrow G]]]$
    $)\textbf{channel}\ (D', D)$
```

Figure 2.5: Decomposition of the guards in the *priv0* process

Chapter 3

Compilations

We have decomposed the original program into sub-processes which are now small enough to compile easily using the method described in [5,7]. We begin the description of the compilations by a brief review of this method and how it is applied to our problem.

3.1 Synopsis

The compilation method consists of a sequence of step-wise refinements. Each process is transformed through a series of semantically equivalent yet increasingly more concrete forms until an implementable circuit results. In the first transformation, communications are replaced by explicit four-phase handshaking. We call the resulting intermediate form the ***handshaking expansion*** of the original process. In the next transformation, all explicit sequencing in the handshaking expansion is removed and the computation is represented by a set of simple guarded commands called a ***production rule set***. By virtue of the way the set is constructed, certain invariant conditions hold for production rule sets, and thus they may be implemented in a race– and hazard–free manner as a collection of interconnected ***self-timed operators***. The resulting operator sets may be used as a specification for a standard-cell implementation in VLSI. We now discuss each stage of the compilation procedure in more detail.

3.1.1 Handshaking Expansions

The semantics of CSP requires that both processes involved in a communication complete the communication simultaneously. While we can not enforce "true" simultaneity in an asynchronous environment, it is possible define the simultaneous completion of a communication action in a consistent and implementable manner (see [5]). Consider the following implementation of a communication from the active port X to the passive port Y. Each port is implemented by an input variable (denoted by xi for the port X) and an output variable (xo). A channel introduces a causal dependency between the output and input variables of the connected ports. The variables yi and xi are delayed copies of xo and yo, respectively. The CSP communication

$$\begin{array}{ll|l} \alpha_x; & X; & \beta_x \\ \alpha_y; & Y; & \beta_y \end{array}$$

where the α's and β's represent arbitrary commands is replaced by the four-phase handshaking expansion

$$\begin{array}{ll|l|l|l|ll} \alpha_x; & xo\!\uparrow; & & [xi]; xo\!\downarrow; & & [\neg xi]; & \beta_x \\ \alpha_y; & & [yi]; yo\!\uparrow; & & [\neg yi]; yo\!\downarrow; & & \beta_y \end{array}$$

With this implementation, just as with the CSP communication, the events β_x and β_y can occur in either order, but neither β_x nor β_y can occur before α_x or α_y.

The four-phase protocol performs four one-sided synchronizations (denoted by the vertical bars) between the processes. Only the first two or the last two are strictly necessary for a synchronization. However, the others are used to reset the values of the handshaking variables allowing an identical implementation of each subsequent communication. In some cases, we may separate the synchronizations by moving part of the handshaking expansion to an earlier or a later point in the sequence. In general, this transformation, named ***reshuffling***, may introduce deadlock. However, when applied to channels introduced by process decomposition, no deadlock is introduced and a very efficient implementation results. The handshaking expansion for a channel introduced by process decomposition

$$\alpha; \beta; \gamma \ \rhd *[[yi]; \beta; yo\!\uparrow; [\neg yi]; yo\!\downarrow] \parallel \alpha; xo\!\uparrow; [xi]; xo\!\downarrow; [\neg xi]; \gamma$$

may be reshuffled in three ways:

$$*[[yi]; (\beta, yo\!\uparrow; [\neg yi]); yo\!\downarrow] \parallel \alpha; xo\!\uparrow; [xi]; xo\!\downarrow; [\neg xi]; \gamma$$

$$*[[yi]; \beta; yo\!\uparrow; [\neg yi]; yo\!\downarrow] \parallel \alpha; xo\!\uparrow; [xi]; (xo\!\downarrow; [\neg xi], \gamma)$$

$$*[[yi]; yo\!\uparrow; [\neg yi]; \beta; yo\!\downarrow] \parallel (\alpha, xo\!\uparrow; [xi]); xo\!\downarrow; [\neg xi]; \gamma$$

The notation S, T represents an arbitrary interleaving of the actions in the sequences S and T. (The ',' operator binds less tightly than the ';' operator.) Of these reshufflings, the first will be used most often in our compilations.

3.1.2 Production Rule Sets

A handshaking expansion is transformed into a set of production rules each of the form $G \longmapsto S$ where G is an arbitrary boolean expression and S is a simple assignment of true or false to a variable. A production rule set and its environment must obey the ***stability*** and ***exclusiveness*** conditions. The stability condition states that for each production rule, if the guard G becomes true, it must remain true until the assignment S is performed. The exclusiveness condition states that no two production rules with the same output variable may simultaneously assign opposite values. The conditions may be used as the definition

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

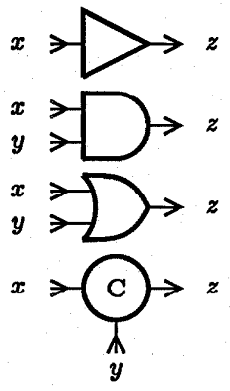

Figure 3.1: Production Rule Sets for Operators

of race– and hazard–free circuits. With this restriction, we may now define an execution of a production rule set as the sequence of states starting at some initial state and where each subsequent state differs from the previous state by the execution of one production rule firing. A production rule set implements a handshaking expansion if each possible execution corresponds to a sequence of actions allowed by the handshaking expansion. In order to enforce this correspondence, state variables may need to be introduced into the handshaking expansion, and guards may need to be added or strengthed to certain assignments.

3.1.3 Operator Sets

Operator sets are constructed by grouping together production rules with the same output variable and matching these rules with the definition of an operator. Variables may be arbitrarily inverted to facilitate the matching procedure. For our compilations, only the operators in Figure 3.1 are required. All references to an identical variable name must be implemented with the identical variable value. We call the implementation of this rule an *isochronic fork.*

3.1.4 Extensions for Non-stable and Non-exclusive Program Constructs

Production rule sets do not allow non-stable nor non-exclusive rules. However, the non-deterministic nature of the programming language introduces intrinsically non-stable or non-exclusive constructs. We introduce in Figure 3.2 two primitive processes which allow the implementation of non-determinacy. These primitive processes can not be represented in production rule set form, yet they both have practical implementations in VLSI. The primitive processes may still be interconnected with circuits constructed from production rules at the operator set level. (Isochronic forks may be required.)

$$(q,x,y)\mathbf{S}(u,v) \equiv \left\{ \begin{array}{l} *[[q \wedge x \longrightarrow u\uparrow; [\neg q]; u\downarrow \\ \ \ [\!]\, q \wedge y \longrightarrow v\uparrow; [\neg q]; v\downarrow \\ \ \ ]] \end{array} \right.$$

$$(x,y)\mathbf{me}(u,v) \equiv \left\{ \begin{array}{l} *[[x \longrightarrow u\uparrow; [\neg x]; u\downarrow \\ \ \ [\!]\, y \longrightarrow v\uparrow; [\neg y]; v\downarrow \\ \ \ ]] \end{array} \right.$$

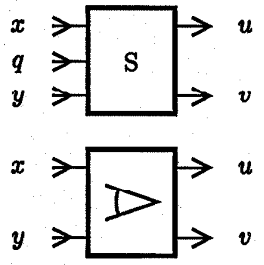

Figure 3.2: Primitive Process Definitions

3.1.5 Application to Our Problem

Decisions need to be made at each stage of the translation procedure. Since a major design goal is to produce efficient circuits which are constructed from a very limited set of operator types, we choose a compilation style which follows these heuristics

- We *reshuffle* communications when possible to result in implementations with the least number of state holding elements.
- When *state variables* are required, we introduce one for each state in the original program.
- We *symmetrize* production rules when possible to create symmetric combinational operators, such as the *AND*, *OR*, and *wire* operators.
- If state holding elements are required, we symmetrize the production rules to form *C* elements.

Such a compilation style can not always be applied top-down, that is by the blind application of rules at each stage of the compilation procedure. Some backtracking is required. However, the interaction between the compilation stages is justifiable in our case because the processes we wish to compile are small in size.

3.2 Communication Channels

We use a four phase handshaking protocol to implement the general communication action. For channels which send data as well as synchronization information, we use more than the minimum two wires. Consider the channel declaration

$$\textbf{passive}\, L(m,n) \quad \textbf{active}\, R(n,m) \quad \ldots \quad \textbf{channel}\, (L,R)$$

Such a channel is implemented by $m+n$ unidirectional wires using a unary encoding of the data. The n send wires of R correspond to the n receive wires of L. The active port

R initiates a communication action and thus these wires point from R to L. One of n values may be sent from R to L during the communication by raising exactly one of these wires. The m receive wires of R and the m send wires of L are also connected and similarly transmit one of m values from L to R. This communication channel may be described by the operator set:

$$\begin{array}{ll} (l(i)o)\mathrm{w}(r(i)i) & (0 \le i < m) \\ (r(j)o)\mathrm{w}(l(j)i) & (0 \le j < n) \end{array}$$

Notice the notation used to name the wires of a channel. In the wire name $l(i)o$, the root l represents the port name, the suffix o represents this wire is an output of the port and the parameter in parenthesis represents which value corresponds to this wire. A suffix i signifies a wire as an input to a port. We allow the parameter in parenthesis to be a free running variable in order to concisely specify constructs of arbitrary size. Since there can be no ambiguity between a parameter and a suffix, we may without confusion use i as a free running variable.

Using handshaking expansion notation[5], the general passive communication action (see Figure 1.4) may be written as

$$[l(0)i \longrightarrow l(i)o\uparrow; [\neg l(0)i]; l(i)o\downarrow; \alpha_0 | \ldots | l(n-1)i \longrightarrow l(i)o\uparrow; [\neg l(n-1)i]; l(i)o\downarrow; \alpha_{n-1}]$$

and the general active communication as

$$r(j)o\uparrow; [r(0)i \longrightarrow r(j)o\downarrow; [\neg r(0)i]; \alpha_0 | \ldots | r(m-1)i \longrightarrow r(j)o\downarrow; [\neg r(m-1)i]; \alpha_{m-1}]$$

(The parameters i and j represent the output values of the communication on ports L and R respectively.)

3.3 Probe and Variable Values

The value of the probe $\overline{L}(k)$ is implemented as the value on the single wire $l(k)i$. The values of boolean variables are also implemented as the value on a single wire. These values must be distributed to all the implementations of the sub-processes that use these probes and variables. We refer to these positions in the circuit as *usage points*. Some technique is needed to insure that, after one of these values changes, every point in the circuit that references it receives the same value before execution is allowed to continue. For now, we assume the usage points are connected together by *isochronic* forks. The value of a probe or variable is defined to be the same at all usage points. In Chapter 4, we will discuss various implementations of these isochronic forks.

3.4 Compilations of Syntactic Forms

Each of the sub-processes created in the last chapter is now compiled into self-timed circuit.

3.4.1 Parallel Composition

Circuits are intrinsically concurrent entities. A collection of simple active communications may be implemented by separate circuits which perform single active communications after a global variable g becomes true:

$$[g]; co\uparrow; [ci]; co\downarrow; [\neg ci]$$

The value of g signifies whether the entire computation is *going* or *reseting*. We do not want an infinite repetition of communications, so a state variable is introduced into the handshaking expansion to distinguish when the communication has finished:

$$[\neg x \wedge g]; co\uparrow; [ci]; x\uparrow; [x]; co\downarrow; [\neg ci]$$

The resulting operator set is

$$(ci, \textbf{true})\mathbf{C}(x)$$
$$(g, \neg x)\bigwedge(co)$$

When g is false, co is false and every sequential process is inactive. All state variables of the system may be reset at this time. In particular, x is set to the initial value **false** . Raising g starts every sequential process simultaneously.

3.4.2 Skip Statement

The **skip** statement has a very simple implementation. The process

$$*[[\overline{D} \longrightarrow \textbf{skip}\,; D]]$$

is equivalent to infinitely repeating the communication action D.

$$*[D] \equiv *[[di]; do\uparrow; [\neg di]; do\downarrow]$$

The wire operator $(di)\mathbf{w}(do)$ implements this handshaking expansion.

3.4.3 Assignment Statement

We must implement the sub-process

$$*[[\overline{D} \longrightarrow x\uparrow; D | \overline{F} \longrightarrow x\downarrow; F]]$$

Transforming into handshaking expansion form, we get

$$*[[di \longrightarrow x\uparrow; [x]; do\uparrow; [\neg di]; do\downarrow \,|fi \longrightarrow x\downarrow; [\neg x]; fo\uparrow; [\neg fi]; fo\downarrow]]$$

The derivation of the resulting production rule set, operator set and resulting circuit shown in figure 3.3 is straight–forward and given in [5]. We initialize the **C** element to the initializer declared in the program.

$$\begin{array}{lll}
di & \longmapsto & x\uparrow \\
di \wedge x & \longmapsto & do\uparrow \\
\neg di & \longmapsto & do\downarrow \\
fi & \longmapsto & x\downarrow \\
fi \wedge \neg x & \longmapsto & fo\uparrow \\
\neg fi & \longmapsto & fo\downarrow \\
(di, \neg fi)\mathbf{C}(x) & & \\
(di, x)\bigwedge(do) & & \\
(fi, \neg x)\bigwedge(fo) & &
\end{array}$$

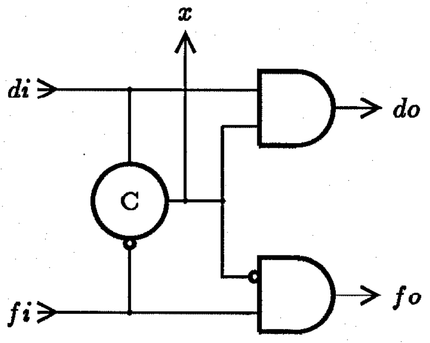

Figure 3.3: Compilation of Assignment Statement

3.4.4 Sequencing Sub-process

Recall the definition of the sequencing process:

$$*[[\overline{D} \longrightarrow S; T; D]]$$

In the handshaking expansion for this process

$$*[[di]; so\uparrow; [si]; so\downarrow; [\neg si]; to\uparrow; [ti]; to\downarrow; [\neg ti]; do\uparrow; [\neg di]; do\downarrow]$$

the sequence $do\uparrow; [\neg di]$ may be reshuffled with the S communication

$$*[[di]; so\uparrow; [si]; do\uparrow; [\neg di]; so\downarrow; [\neg si]; to\uparrow; [ti]; to\downarrow; [\neg ti]; do\downarrow]$$

Introducing a single state variable x,

$$*[[di]; so\uparrow; [si]; x\uparrow; [x]; do\uparrow; [\neg di]; so\downarrow; [\neg si]; to\uparrow; [ti]; x\downarrow; [\neg x]; to\downarrow; [\neg ti]; do\downarrow]$$

we may now expand the handshaking expansion into a production rule set and equivalent operator set (see figure 3.4).

We will use circuits similar to this in the implementation of guarded structures. We define the implementation of

$$*[[\overline{D} \longrightarrow \textbf{skip}\,; T; D]]$$

to be a new operator named a **D** element.

$$\begin{array}{rl}
(di, do)\mathbf{D}(to, ti) \equiv & (di, \neg ti)\mathbf{C}(x) \\
& (x, ti)\bigvee(do) \\
& (\neg di, x)\bigwedge(to)
\end{array}$$

$$\begin{array}{lll}
di & \longmapsto & so\uparrow \\
si \wedge \neg ti & \longmapsto & x\uparrow \\
x \vee ti & \longmapsto & do\uparrow \\
\neg di & \longmapsto & so\downarrow \\
\neg si \wedge x & \longmapsto & to\uparrow \\
ti \wedge \neg si & \longmapsto & x\downarrow \\
\neg x \vee si & \longmapsto & to\downarrow \\
\neg ti \wedge \neg x & \longmapsto & do\downarrow \\
\multicolumn{3}{l}{(di)\mathbf{w}(so)} \\
\multicolumn{3}{l}{(si, \neg ti)\mathrm{C}(x)} \\
\multicolumn{3}{l}{(x, ti)\bigvee(do)} \\
\multicolumn{3}{l}{(\neg si, x)\bigwedge(to)}
\end{array}$$

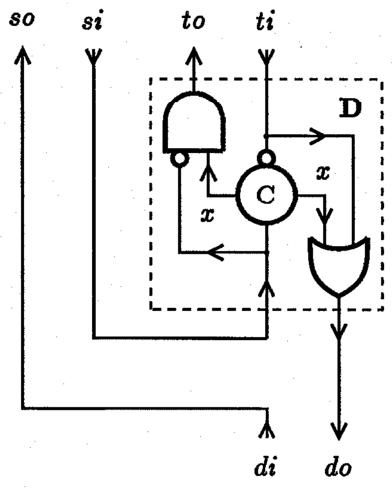

Figure 3.4: Compilation of Sequencing Process

3.4.5 Control Sub-process

By transforming the control process,

$$*[[\overline{D} \longrightarrow P : [0 \longrightarrow S_0 | \ldots | k \longrightarrow S_k; D | \ldots]]]$$

into a handshaking expansion, we get:

$$\begin{array}{ll}
*[[di]; po\uparrow; & [p(0)i \longrightarrow po\downarrow; [\neg p(0)i]; s(0)o\uparrow; [s(0)i]; s(0)o\downarrow; [\neg s(0)i] \\
 & \vdots \\
 & |p(k)i \longrightarrow po\downarrow; [\neg p(k)i]; s(k)o\uparrow; [s(k)i]; s(k)o\downarrow; [\neg s(k)i]; do\uparrow; [\neg di]; do\downarrow \\
 & \vdots \\
 &] \\
]
\end{array}$$

As in the sequencing case, we may reshuffle the middle two actions of each D handshake to directly after the arrow and introduce a new state variable for each path. In general we may not reshuffle the communication P with the communication S. Delaying the completion of P until after the initiation of a command which influences the guard set process connected to P potentially causes instability in the guard set process. The values of the state variables persist after the communication with P. We use these values to selected the

$$\begin{array}{lll}
\neg u(0) \wedge \ldots \wedge \neg u(k-1) & \longmapsto & v\downarrow \\
\neg u(k) \wedge \ldots \wedge \neg u(n-1) & \longmapsto & do\downarrow \\
\neg v \wedge di & \longmapsto & po\uparrow
\end{array}$$

$$\left.\begin{array}{lll}
p(i)i \wedge \neg s(i)i & \longmapsto & x(i)\uparrow \\
x(i) \vee s(i)i & \longmapsto & u(i)\uparrow \\
u(i) & \longmapsto & v\uparrow \\
v & \longmapsto & po\downarrow \\
\neg p(i)i \wedge x(i) & \longmapsto & s(i)o\uparrow \\
s(i)i \wedge \neg p(i)i & \longmapsto & x(i)\downarrow \\
\neg x(i) \vee p(i)i & \longmapsto & s(i)o\downarrow \\
\neg s(i)i \wedge \neg x(i) & \longmapsto & u(i)\downarrow \\
p(j)i \wedge \neg s(j)i & \longmapsto & x(j)\uparrow \\
x(j) \vee s(j)i & \longmapsto & u(j)\uparrow \\
u(j) & \longmapsto & do\uparrow \\
\neg di & \longmapsto & po\downarrow \\
\neg p(j)i \wedge x(j) & \longmapsto & s(j)o\uparrow \\
s(j)i \wedge \neg p(j)i & \longmapsto & x(j)\downarrow \\
\neg x(j) \vee p(j)i & \longmapsto & s(j)o\downarrow \\
\neg s(j)i \wedge \neg x(j) & \longmapsto & u(j)\downarrow
\end{array}\right\} \quad (0 \le i < k, k \le j < n)$$

Figure 3.5: Production Rule Set for the Control Process

proper command sequence to execute.

```
*[[$di$]; $po\uparrow$;  [$p(0)i \longrightarrow$  $x(0)\uparrow$; [$x(0)$]; $po\downarrow$; [$\neg p(0)i$]; $s(0)o\uparrow$; [$s(0)i$];
                                   $x(0)\downarrow$; [$\neg x(0)$]; $s(0)o\downarrow$; [$\neg s(0)i$]
              ⋮
              |$p(k)i \longrightarrow$  $x(k)\uparrow$; [$x(k)$]; $d(k)o\uparrow$; [$\neg di$]; $po\downarrow$; [$\neg p(k)i$]; $s(k)o\uparrow$; [$s(k)i$];
                                   $x(k)\downarrow$; [$\neg x(k)$]; $s(k)o\downarrow$; [$\neg s(k)i$]; $d(k)o\downarrow$
              ⋮
              ]
  ]
```

The production rule set for this handshaking expansion is shown in Figure 3.5. Making use of the previously defined **D** elements, we get the simple operator set

$$\begin{array}{ll}
(u(0),\ldots,u(k-1))\bigvee(v) & \\
(u(k),\ldots,u(n-1))\bigvee(do) & \\
(di,\neg v)\bigwedge(po) & \\
(p(i)i,u(i))\mathbf{D}(s(i)o,s(i)i) & (0 \le i < k) \\
(p(j)i,u(j))\mathbf{D}(s(j)o,s(j)i) & (k \le j < n)
\end{array}$$

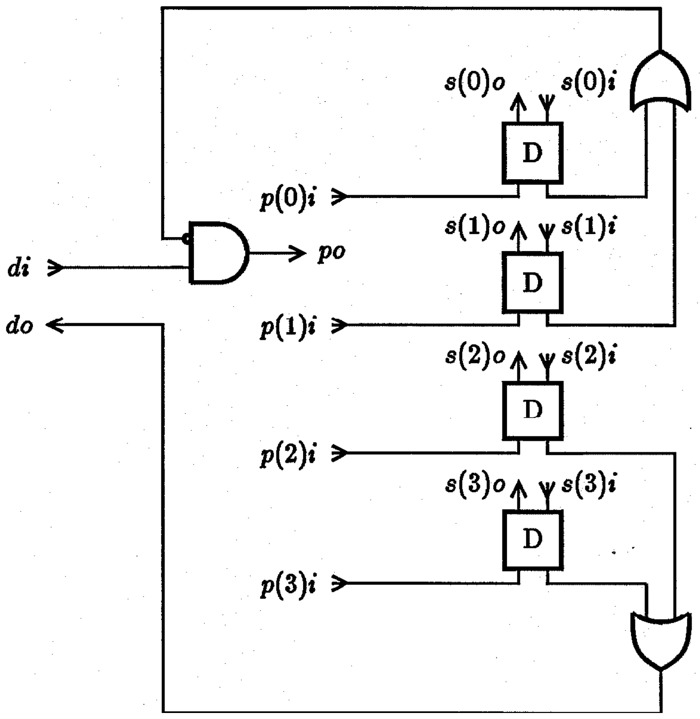

Figure 3.6: Circuit for Control Sub-process ($k = 2, n = 4$)

An example circuit is shown in Figure 3.6.

3.4.6 Communication Actions

We start the compilation of communication actions by considering the special case of only one usage per process.

Simple Active

The simple active communication

$$*[[\overline{D} \longrightarrow A(j) : [0 \longrightarrow S_0| \ldots |n-1 \longrightarrow S_{n-1}]; D]]$$

may be implemented with a specific instance of the control process in the last section. In this case, no branch repeats, yielding the following operator set:

$$\begin{array}{ll} (di)\mathbf{w}(a(j)o) & \\ (a(i)i, u(i))\mathbf{D}(s(i)o, s(i)i) & \quad (0 \leq i < n) \\ (u(0), \ldots, u(n-1))\bigvee(do) & \end{array}$$

Figure 3.7 shows an example circuit.

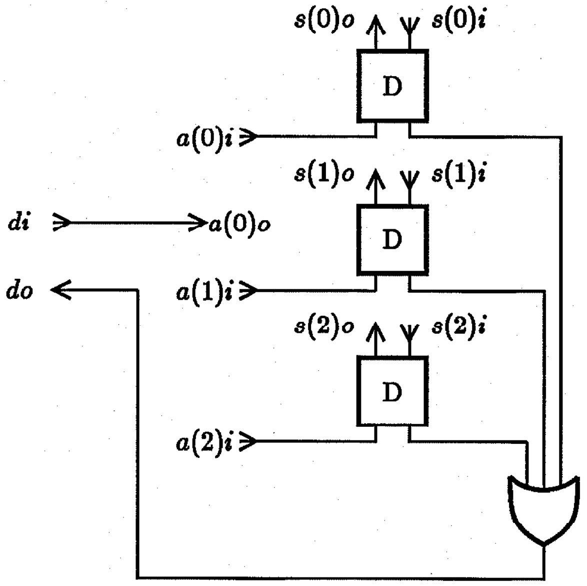

Figure 3.7: Circuit for Active Communication ($n = 3, j = 0$)

Simple Passive

The simple passive communication sub-process

$$*[[\overline{D} \longrightarrow P(i) : [\ldots|j \longrightarrow S_j|\ldots]; D]]$$

has the non-reshuffled handshaking expansion:

```
*[[di]; [
         ⋮
        |p(j)i ⟶  p(i)o↑; [¬p(j)i]; p(i)o↓;
                  s(j)o↑; [s(j)i]; s(j)o↓; [¬s(j)i]
         ⋮
        ]; do↑; [¬di]; do↓
]
```

$$\begin{array}{lll}
u(0) \vee \ldots \vee u(m-1) & \longmapsto & p(i)o\uparrow \\
\neg u(0) \wedge \ldots \wedge \neg u(m-1) & \longmapsto & p(i)o\downarrow \\
v(0) \vee \ldots \vee v(m-1) & \longmapsto & do\uparrow \\
\neg v(0) \wedge \ldots \wedge \neg v(m-1) & \longmapsto & do\downarrow
\end{array}$$

$$\left.\begin{array}{lll}
di \wedge p(j)i & \longmapsto & u(j)\uparrow \\
u(j) \wedge \neg s(j)i & \longmapsto & x(j)\uparrow \\
x(j) \vee s(j)i & \longmapsto & v(j)\uparrow \\
\neg di \wedge \neg p(j)i & \longmapsto & u(j)\downarrow \\
\neg u(j) \wedge x(j) & \longmapsto & s(j)o\uparrow \\
s(j)i \wedge \neg u(j) & \longmapsto & x(j)\downarrow \\
\neg x(j) \vee u(j) & \longmapsto & s(j)o\downarrow \\
\neg s(j)i \wedge \neg x(j) & \longmapsto & v(j)\downarrow
\end{array}\right\} \quad (0 \le j < m)$$

Figure 3.8: Production Rule Set for the Passive Communication

Again, we reshuffle the middle actions of D and introduce auxiliary variables.

```
*[[
⋮
$di \wedge p(j)i \longrightarrow$   $u(j)\uparrow; [u(j)];$
                    $p(i)o\uparrow, (x(j)\uparrow; [x(j)]; v(j)\uparrow; [v(j)]; do\uparrow);$
                    $[\neg di \wedge \neg p(j)i];$
                    $u(j)\downarrow; [\neg u(j)];$
                    $p(i)o\downarrow, (s(j)o\uparrow; [s(j)i]; x(j)\downarrow; [\neg x(j)];$
                    $s(j)o\downarrow; [\neg s(j)i]; v(j)\downarrow; [\neg v(j)]; do\uparrow)$
⋮
]]
```

The production rule expansion shown in Figure 3.8 produces this symmetrized operator set.

$$\left.\begin{array}{l}
(di, p(j)i)\mathrm{C}(u(j)) \\
(u(j), v(j))\mathrm{D}(s(j)o, s(j)i)
\end{array}\right\} \quad (0 \le j < m)$$

$$\begin{array}{l}
(u(0), \ldots, u(m-1))\bigvee(p(i)o) \\
(v(0), \ldots, v(m-1))\bigvee(do)
\end{array}$$

It is worthwhile to note that the C elements in these passive communication circuits provide the sole means of synchronizing the concurrent activities of the system. An example circuit is shown in Figure 3.9.

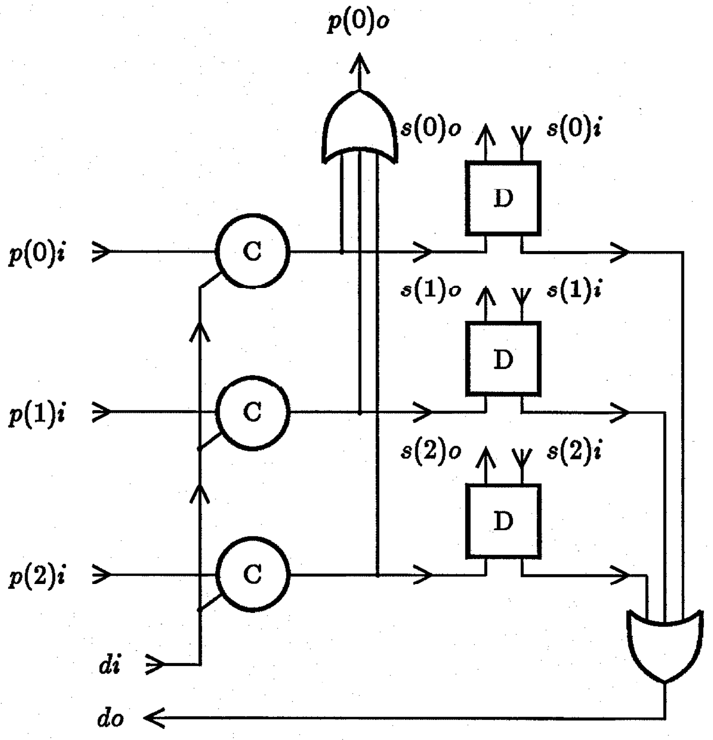

Figure 3.9: Circuit for Passive Communication ($n = 3, i = 0$)

Multiple Usages of Passive Ports

Since communications on a particular channel may be performed in several different sub-processes, we must produce a means sharing the use of these channel among the various sub-processes.

We expand the simple passive communication to allow several sub-processes to respond and complete a communication on the same port. We use a "passive to multiple active port" converter with no buffering capacity to implement this expansion. Let P be the single passive port and $A_0, A_1, \ldots, A_{k-1}$ be the k active ports of the converter. The active ports pair up with the passive ports of sub-processes. When one of the input wires of the passive port is raised, the corresponding output wire is raised in each of the k active ports. Because no two sub-processes operate concurrently, only one of the active ports $A(i)$ will respond to the communication by raising an input wire $a(i)(j)i$. (Notice the use of double parameters to name the wires of the multiple active ports.) The converter will then raise the corresponding output wire of the passive port $p(j)o$, resulting in the lowering of the initiating input wire of P and then the lowering of the corresponding output wires of each of the k active ports. We assume until a refinement is given in Chapter 4 that the corresponding output wires of the active ports are connected via isochronic forks. The communication action then continues when the single input wire $a(j)(i)i$ lowers and finally completes with the lowering of output wire $p(j)o$. We may use the operator set

$$\begin{array}{ll} p(\ell)i \equiv a(i)(\ell)o \qquad (0 \leq i < k, 0 \leq \ell < n) & \\ (a(0)(j)i, a(1)(j)i, \ldots, a(k-1)(j)i) \bigvee (p(j)o) & (0 \leq j < m) \end{array}$$

to implement this passive to multiple active converter. The $\equiv$ operator allows two different names to represent the same variable. An example circuit is shown in Figure 3.10.

Multiple Usages of Active Ports

In the case of several sub-processes calling a single active port, we use a multiple passive port to single active port converter called a *merger*. Let $P_0, P_1, \ldots, P_{k-1}$ represent the k passive ports and A the single active port of the converter. The passive ports of the converter are paired with the active ports of the sub-processes. A communication through an active port of a sub-process is transparently converted into a communication on the active port A. The converter must store which passive port initiated the communication in order to direct the acknowledgement of the communication back to its source. The following operator set specifies an implementation of such a converter:

$$\begin{array}{ll} (p(0)(j)i, p(1)(j)i, \ldots, p(k-1)(j)i) \bigvee (a(j)o) & (0 \leq j < m) \\ (p(i)(0)i, p(i)(1)i, \ldots, p(i)(m-1)i) \bigvee (u(i)) & (0 \leq i < k) \\ (u(i), a(j)i) \mathrm{C} (p(i)(j)o) \qquad (0 \leq i < k, 0 \leq j < n) & \end{array}$$

The double parameters specify port number and value number, respectively. An example circuit is shown in Figure 3.11.

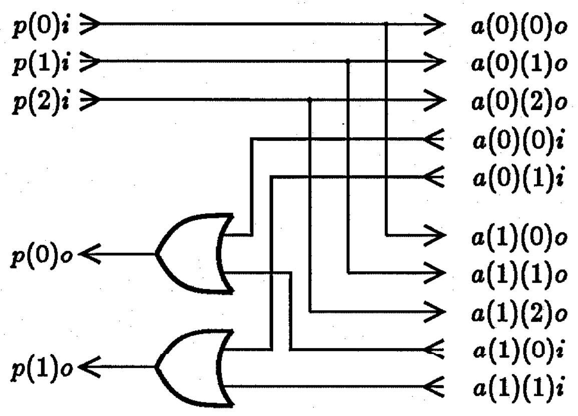

Figure 3.10: Passive $k-(m,n)$ Expansion Circuit $(k=2, m=2, n=3)$

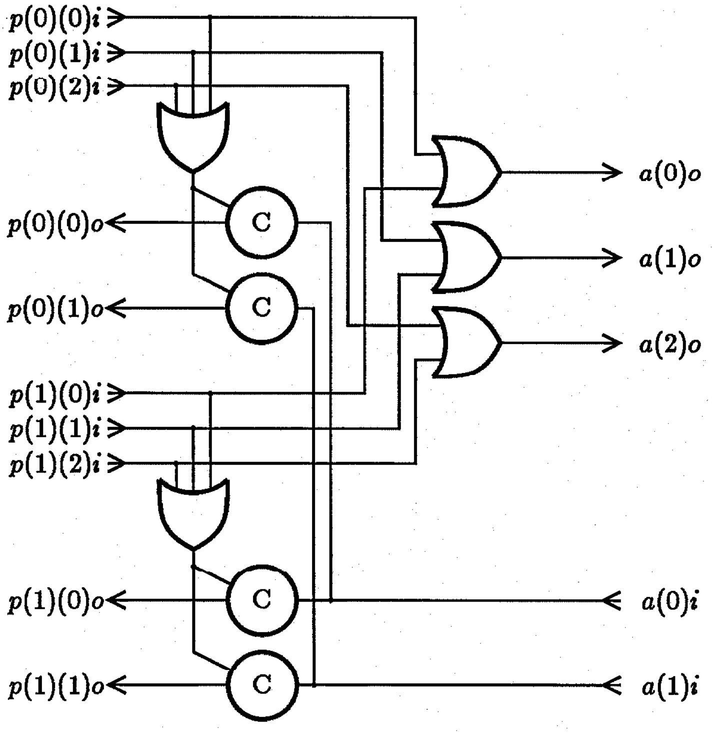

Figure 3.11: Active $k-(m,n)$ Merger Circuit $(k=2, m=3, n=2)$

3.4.7 Exclusive Guards

The sub-process

$$*[[\overline{Q} \wedge x_0 \wedge \ldots \wedge x_{k-1} \wedge \neg x_k \wedge \ldots \wedge \neg x_{n-1} \longrightarrow Q(i)]]$$

may be implemented simply with an $n+1$ input AND gate.

$$(qi, x_0, \ldots, x_{k-1}, \neg x_k, \ldots, \neg x_{n-1}) \bigwedge (q(i)o)$$

3.4.8 Non-exclusive Guards

The sub-process

$$*[[\overline{Q} \longrightarrow [x \longrightarrow Q(1) | \neg x \longrightarrow Q(0)]]]$$

may be implemented with two 2 input AND gates. If probes are used, we need to implement the process

$$*[[\overline{Q} \longrightarrow [\overline{X}(k) \longrightarrow Q(1) \| \neg\overline{X}(k) \longrightarrow Q(0)]]]$$

By expanding this process to its handshaking expansion,

$$\begin{array}{l} *[[qi \wedge x(k)i \longrightarrow q(1)o\uparrow; [\neg qi]; q(1)o\downarrow \\ \ \| qi \wedge \neg x(k)i \longrightarrow q(0)o\uparrow; [\neg qi]; q(0)o\downarrow \\ \ ]] \end{array}$$

we see that it matches the synchronizer process defined earlier if we use the instantiation

$$(qi, x(k)i, \neg x(k)i)\mathrm{S}(q1o, q0o)$$

The ***and***, ***or***, ***negation***, and ***busy-waiting*** sub-processes all have trivial implementations after reshuffling. Both the ***and*** and the ***or*** sub-processes are implemented with a single OR gate. The ***negation*** sub-process merely interchanges two wires. We use the ***busy-waiting*** sub-process as an example of the reshufflings performed on these sub-processes. Consider the simple sub-process

$$*[[\overline{Q} \longrightarrow U : [0 \longrightarrow V : [0 \longrightarrow \textbf{skip} \,|1 \longrightarrow Q(1)] \,|1 \longrightarrow Q(0)]]]$$

In handshaking expansion form,

$$\begin{array}{lll} *[[qi]; uo\uparrow; & [u(0)i \longrightarrow & uo\downarrow; [\neg u(0)i]; vo\uparrow; \\ & & [v(0)i \longrightarrow vo\downarrow; [\neg v(0)i] \\ & & |v(1)i \longrightarrow vo\downarrow; [\neg v(1)i]; q(1)o\uparrow; [\neg qi]; q(1)o\downarrow \\ & &] \\ & |u(1)i \longrightarrow uo\downarrow; [\neg u(0)i]; q(0)o\uparrow; [\neg qi]; q(0)o\downarrow & \\ &]] & \end{array}$$

The U and Q communications may be reshuffled together arbitrarily because they are both communications introduced by process decomposition.

```
*[ [qi]; uo↑; [u(0)i ⟶ vo↑; [v(0)i ⟶ uo↓; [¬u(0)i]; vo↓; [¬v(0)i]
                            |v(1)i ⟶  q(1)o↑; [¬qi]; uo↓;
                                      [¬u(0)i]; vo↓; [¬v(1)i]; q(1)o↓
                            ]
              |u(1)i ⟶ q(0)o↑; [¬qi]; uo↓; [¬u(0)i]; q(1)o↓
              ]]
```

The resulting operator set is

$$\begin{array}{l} (qi, \neg v(0)i)\bigwedge(uo) \\ (u(0)i)\mathbf{w}(vo) \\ (u(1)i)\mathbf{w}(q(0)o) \\ (v(1)i)\mathbf{w}(q(1)o) \end{array}$$

3.5 Example

Figure 3.18 shows the compiled circuit corresponding to the *priv0* example.

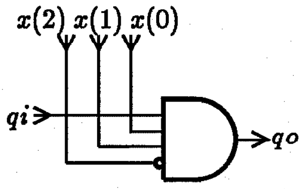

Figure 3.12: Exclusive Conjunction

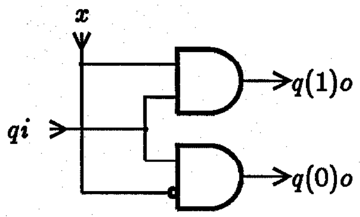

Figure 3.13: Simple Variable

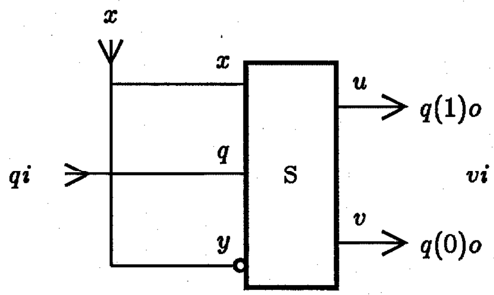

Figure 3.14: Simple Probe

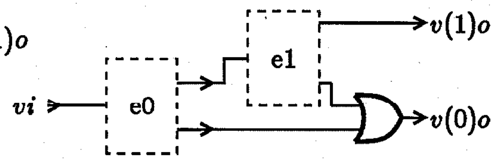

Figure 3.15: Conjunction

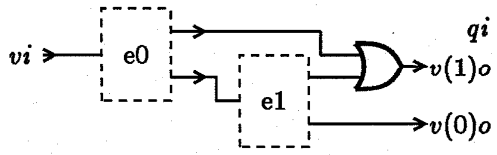

Figure 3.16: Disjunction

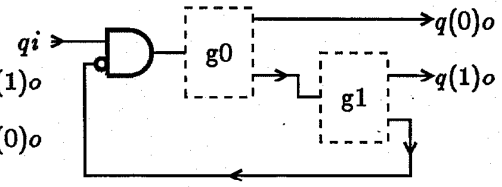

Figure 3.17: Busy Evaluation

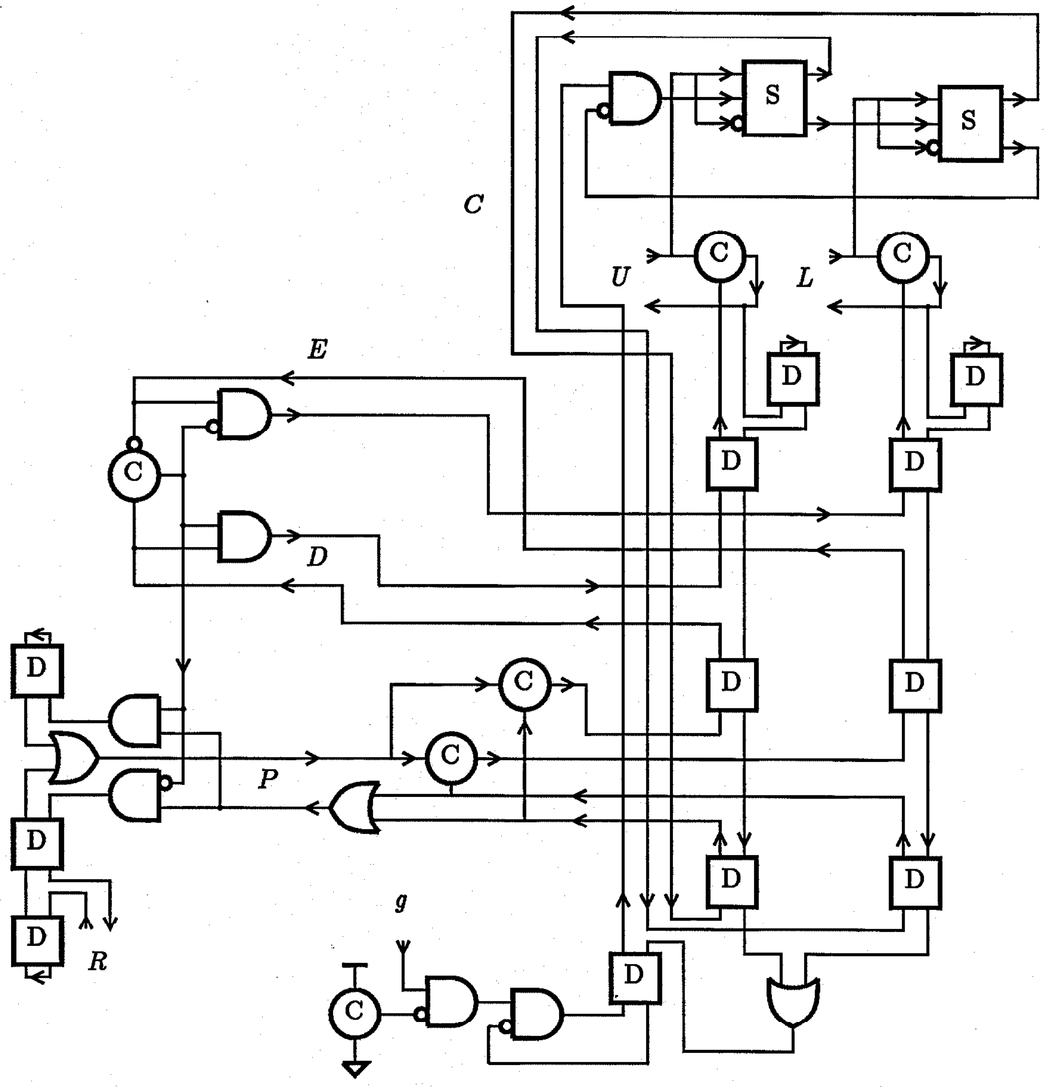

Figure 3.18: Complete Circuit for *priv0* Process

Chapter 4

Isochronic Forks

In the previous chapter, we implemented the use of probes and variables at different points throughout the circuit by postulating the existence of an isochronic fork which distributes the value of a probe or variable to all usage points instantaneously. This isochronicity assumption of a fork is valid if the usage points are close together. However, locality of usage points can not be guaranteed unless some bound is placed on the size of the sequential processes from which the circuits are generated. We place no such bound on the sequential processes defined in our source language. In this chapter, we discuss methods of realizing large isochronic forks as well as alternate methods of implementing certain constructs of the language in order to avoid the need for large isochronic forks.

4.1 Electrical Solution

We may implement isochronic forks local to a single chip by connecting the usage points together with a low resistance interconnect, typically either *metal1* or *metal2*. With present technologies, the delay in changing the potential of a usage point two chip widths away from its source is still significantly less than the delay in an ordinary operator as long as these long wires are driven with optimally sized inverter horns. However, in future technologies, interconnect delays will become more significant and the isochronic fork assumption may be invalid [1]. Fortunately, arbitrary-size isochronic forks are not intrinsically necessary for our compilation strategy. We may algorithmically reduce each large isochronic fork into an isochronic fork between two usage points.

4.2 Algorithmic Solution

The isochronic forks produced by the compilation methodology are local to a single sequential process. We modify each sequential process so that when it changes a (circuit) variable which must isochronically fork to several usage points, it suspends until all usage points have received the change. Only after the process detects that the variable's new value has

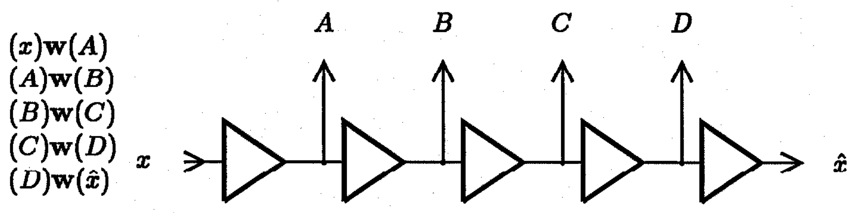

Figure 4.1: Distribution and Detection Using a Linear Ordering

reached all usage points can execution resume. We split the description of how to modify our existing implementation into two parts:

- We describe how to distribute a variable's value to all usage points in such a way that we can detect when all points have reach their new values. Furthermore, we use isochronic forks with only two outputs in this implementation.
- We describe the changes needed to the previous implementations of the language constructs to suspend the process until the fork has completed distributing the variable's value.

4.2.1 Variable Distribution and Completion Detection

We detect that a variable's value has been distributed to all usage points by observing the value of a new circuit variable called a ***completion signal.*** Assume we must distribute the circuit variable x to the usage points A, B, C and D. If we introduce the completion signal $\hat{x}$ and connect x to A, A to B, B to C, C to D and D to $\hat{x}$ with wire operators, we can be assured that when $\hat{x}$ changes, the change has already propagated through all the intermediate usage points. In so doing, we have reduced the large isochronic fork to several binary isochronic forks. Thus, we may use the circuit shown in Figure 4.1 to distribute and detect distribution of an isochronic fork.

Instead of linearly ordering the usage points, we may send separate wires to each point and construct the completion signal using a tree of C operators. (see Figure 4.2) The output of such a tree changes when all its inputs have changed and may be used in exactly the same fashion as the completion signal of the above linear ordering.

4.2.2 Process Suspension

The previous implementations of the active channel merger, the passive channel expansion, the passive communication and the assignment constructs are sources of isochronic forks

$(x)\mathbf{w}(A)$
$(x)\mathbf{w}(B)$
$(x)\mathbf{w}(C)$
$(x)\mathbf{w}(D)$
$(A)\mathbf{w}(\hat{x}_A)$
$(B)\mathbf{w}(\hat{x}_B)$
$(C)\mathbf{w}(\hat{x}_C)$
$(D)\mathbf{w}(\hat{x}_D)$
$(\hat{x}_A, \hat{x}_B)\mathrm{C}(\hat{x}_{AB})$
$(\hat{x}_C, \hat{x}_D)\mathrm{C}(\hat{x}_{CD})$
$(\hat{x}_{AB}, \hat{x}_{CD})\mathrm{C}(\hat{x})$

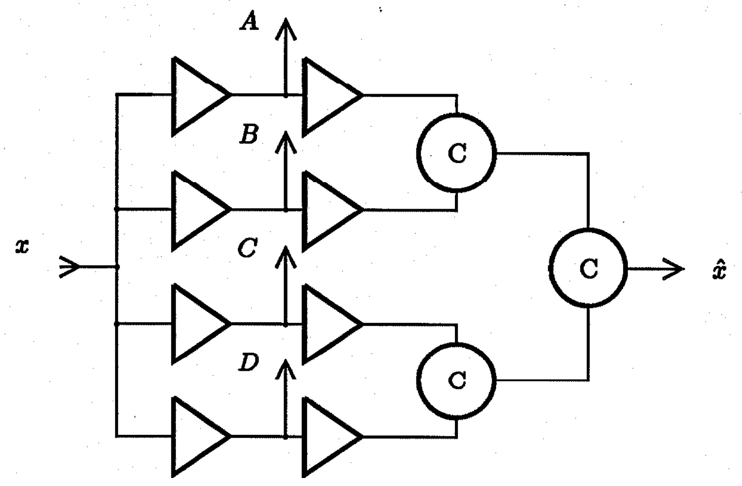

Figure 4.2: Concurrent Distribution and Detection

between (potentially) more than two usage points. We modify these implementation to wait for the completion signals of the forked variables.

The k–way isochronic forks needed in a k–way active channel merger may be reduced to 2–way isochronic forks by using trees of 2–way active channel mergers. The remaining large isochronic forks (only if the number of input values is greater than one) are eliminated by the circuit shown in Figure 4.3. The **M** block in the circuit merges together m one–output and one–input value channels. This block may be implemented without large isochronic forks. The completion signals for the forks $p(0)(*)i$ and $p(1)(*)i$ must change before an output value changes.

Passive communications and passive channel expansions are modified as shown in Figures 4.5 and 4.4. The completion signals for the input variables must change before the flow of control is transfered to the next statement. The synchronization is introduced before the calls to the statements S_i, ensuring the correct value of the input variables inside the implementations of these statements.

The modified assignment implementation (see Figure 4.6) uses the completion signal instead of the variable value to detect when the assignment has finished.

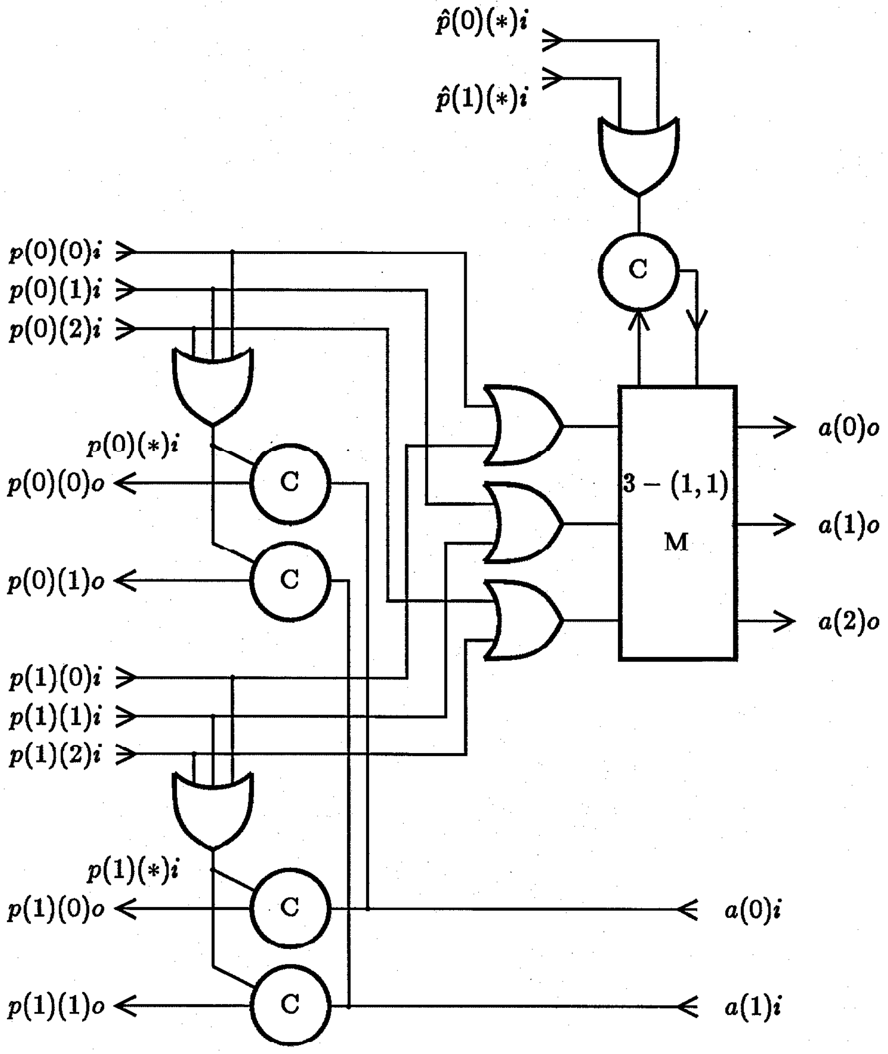

Figure 4.3: Modified $2-(3,2)$ Active Merger

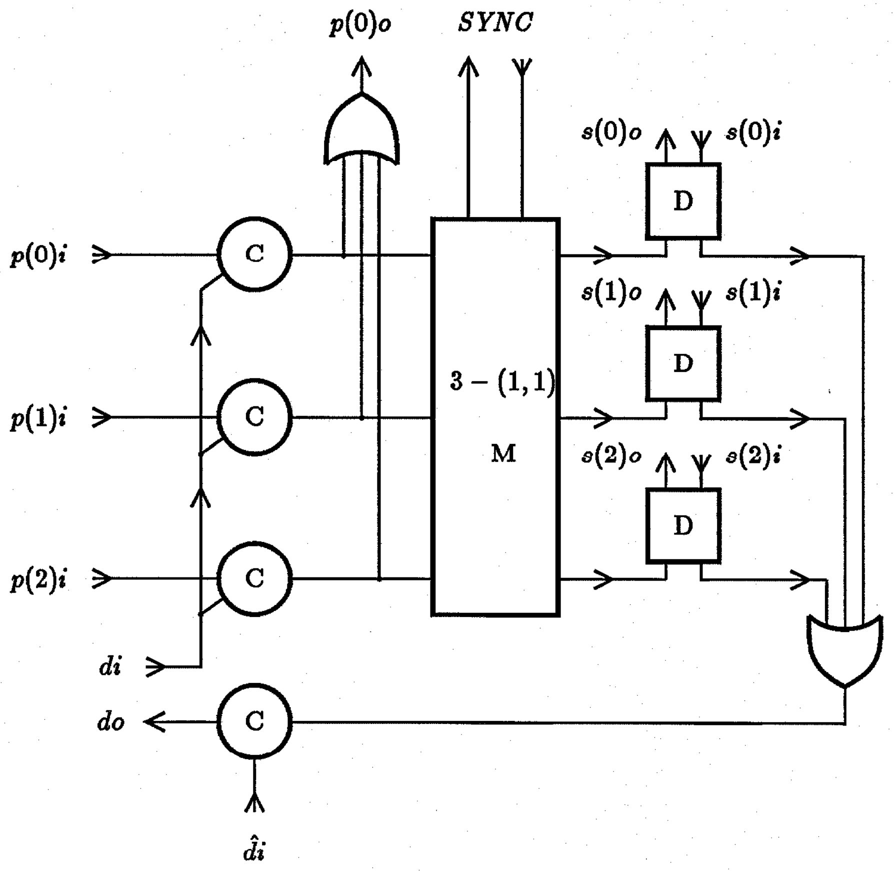

Figure 4.4: Modified Passive Communication

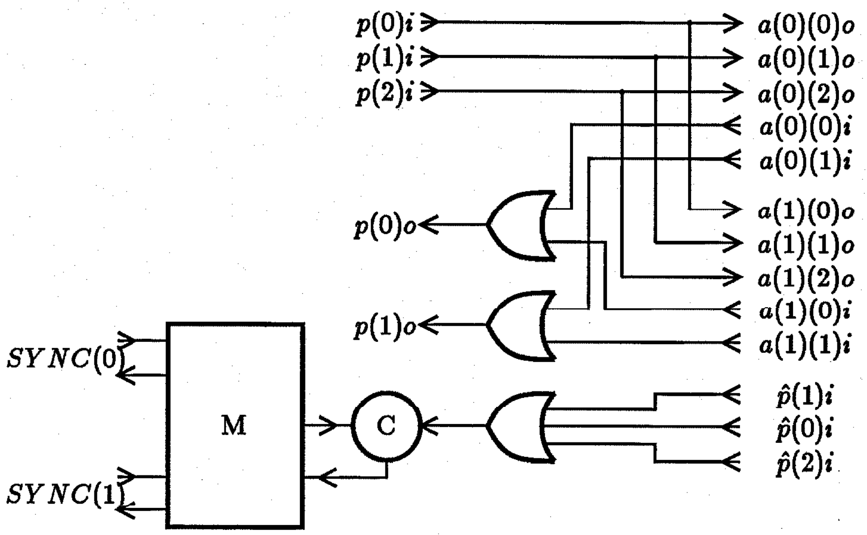

Figure 4.5: Modified 2 – (2, 3) Passive Expansion

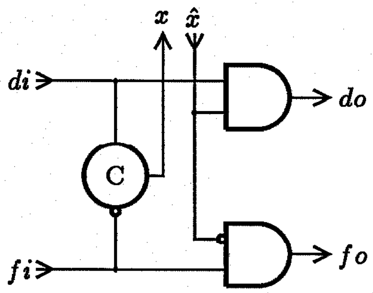

Figure 4.6: Modified Assignment

Chapter 5

Circuit Optimization

The circuits generated by the techniques of the last chapters are functionally correct but some inefficiencies have been introduced by the generality of the procedure. The composing rule for each language construct is strong enough to avoid interaction between arbitrary commands. Better circuits may be synthesized if these interactions are considered in more detail. In particular, when commands do not interact, a more simple implementation of the language construct may be used. We consider some of the important special cases.

5.1 Removing D-elements

The most useful optimization is the removal of unnecessary **D**-elements. The removal of sequencing operators corresponds directly to reshuffling the handshaking sequences of the adjacent commands. Replacing the operator $(si, so)\mathbf{D}(to, ti)$ with the two wire operators $(si)\mathbf{w}(to)$ and $(ti)\mathbf{w}(so)$ intertwines the handshaking expansion of the passive communication S and the active communication T yielding:

$$[si]; to\uparrow; [ti]; so\uparrow; [\neg si]; to\downarrow; [\neg ti]; so\downarrow$$

We may remove **D**-elements in cases where the above reshuffling does not destroy the original semantics of the program.

In the general case of the control structure process, **D**-elements are required between each guard and the sequence of statements that follows it. The **D**-element may be removed if the sequence of statements does not change the value of any guard in the guarded command set. A trivial example is a guard followed by the single statement **skip**. The statement **skip** can not change the value of any guard. Assignments are also easy to characterize. If a simple assignment to the variable x immediately follows a guard, the **D**-element may be removed if the assignment to x does not change any guard.

$$\begin{array}{lll}
\neg x \wedge di & \longmapsto & so\uparrow \\
di \wedge si & \longmapsto & x\uparrow \\
\neg di \vee x & \longmapsto & so\downarrow \\
x \wedge \neg si & \longmapsto & to\uparrow \\
ti & \longmapsto & do\uparrow \\
\neg di \wedge \neg si & \longmapsto & x\downarrow \\
si \vee \neg x & \longmapsto & to\downarrow \\
\neg ti & \longmapsto & do\downarrow \\
\multicolumn{3}{l}{(\neg x, di)\bigwedge(so)} \\
\multicolumn{3}{l}{(di, si)\mathbf{C}(x)} \\
\multicolumn{3}{l}{(\neg si, x)\bigwedge(to)} \\
\multicolumn{3}{l}{(ti)\mathbf{w}(do)}
\end{array}$$

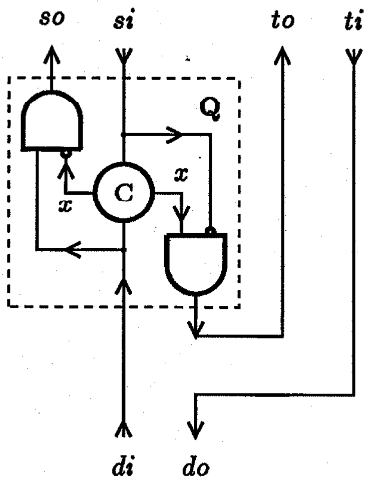

Figure 5.1: Another Sequencing Circuit

5.2 Sequencing Operators

By reshuffling the sequencing process

$$*[[\overline{D} \longrightarrow S; T; D]]$$

in a different way than in the last chapter (Page 28)

$$*[[di]; so\uparrow; [si]; so\downarrow; [\neg si]; to\uparrow; [ti]; do\uparrow; [\neg di]; to\downarrow; [\neg ti]; do\downarrow]$$

we get a different circuit made of wires and a so-called **Q** operator (Figure 5.1). We call the operators **D** and **Q** sequencing operators. Sequencing may be performed by arbitrary trees of sequencing operators, not just the linear chains described in Chapter 2. (The **Q** operator may *not* be used to replace the **D** operator in the control and communication sub-processes.) By manipulation of these sequencing trees, smaller implementations of the control flow are possible. Sequencing operators may be factored through active channel mergers if the sequencing operators are of the same kind and never active at the same time. If an action is common to the last action of two or more selection branches, the active merge process may be replaced by a simple OR gate. We do not need a full merger process because regardless of which branch initiated the action, control will continue with the same action.

5.3 Passive Channels

The special form

$$*[[\overline{D} \longrightarrow S; D]]$$

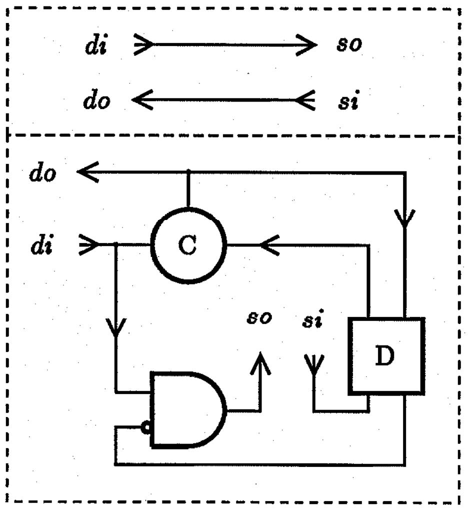

Figure 5.2: Comparison of Passive Communication Implementations

where S does not call the communication D, may be implemented simply as a pair of wires. This implementation is superior to the standard implementation requiring one **C**, one **D** and one $\bigwedge$ element (Figure 5.2). A user programmed passive communication may be implemented with wires if the compiler is able to prove that the passive communication is only performed as the very last action of an continuing repetition. Such passive communications do not introduce concurrency. This implementation involves a reshuffling of the handshaking expansion:

$$*[[di]; so\uparrow; [si]; so\downarrow; [\neg si]; do\uparrow; [\neg di]; do\downarrow]$$

The middle action and wait of the D handshaking protocol are moved forward.

$$*[[di]; so\uparrow; [si]; do\uparrow; [\neg di]; so\downarrow; [\neg si]; do\downarrow]$$

The process commits to finishing the D communication before S is finished. The communication S must not start a separate communication on port D. Otherwise, the two D communications would have non-exclusive access to the same channel. The reshuffling can only be performed if the communications on the active port connected to D are not reshuffled. In the method presented here, we never reshuffle user specified active communications.

The control structures of the more complicated guarded command

$$*[[\overline{D} \wedge G_0 \longrightarrow S_0; D| \ldots |\overline{D} \wedge G_{n-1} \longrightarrow S_{n-1}; D]]$$

where the G_i are conjunctions of possibly negated variables, may also be implemented directly as wires as long as D is not called in any of the S_i. To reshuffle the even more general form with different passive probes,

$$*[[\overline{D} \wedge G_0 \longrightarrow S; D|\overline{E} \wedge G_1 \longrightarrow T; E]]$$

the communication T must not raise the probe of D and S must not raise the probe of E.

Optimizations also exist for repetitive structures with guards the conjunct of positive probes. For instance,

$$*[[\overline{D} \wedge \overline{F} \longrightarrow S; D; T; F]]$$

may be reshuffled to

$$\begin{array}{l} *[[di \wedge fi]; so\uparrow; [si]; do\uparrow, fo\uparrow; \\ [\neg di \wedge \neg fi]; so\downarrow; [\neg si]; do\downarrow, (to\uparrow; [ti]; to\downarrow; [\neg ti]); fo\downarrow \\ ] \end{array}$$

This reshuffling has a simple implementation (Figure 5.3)

$$\begin{array}{l} (di, fi)\mathbf{C}(so) \\ (si)\mathbf{w}(do) \\ (si, fo)\mathbf{D}(to, ti) \end{array}$$

5.4 Special Circuits for Arbitration

In special cases, efficient arbitration circuits may be used to implement non-exclusive guards. Both

$$\begin{array}{l} \textbf{passive}\, U, L \quad \textbf{active}\, S, T \\ \quad *[[\overline{U} \longrightarrow S; U \| \overline{L} \longrightarrow T; L]] \\ \textbf{process}\ arbiter(U, L, S, T) \end{array}$$

and

$$\begin{array}{l} \textbf{passive}\, Q(2,1) \\ \quad *[[\overline{Q} \wedge \overline{U} \longrightarrow Q(0) \| \overline{Q} \wedge \overline{L} \longrightarrow Q(1)]] \\ \textbf{process}\ enabled\_arb(Q) \end{array}$$

may be implemented without busy waiting. In the first case we use the arbiter from [6].

$$\begin{array}{l} (ui, li)\mathbf{me}(u, l) \\ (u, \neg ti)\bigwedge(so) \\ (l, \neg si)\bigwedge(to) \\ (si)\mathbf{w}(uo) \\ (ti)\mathbf{w}(lo) \end{array}$$

The **me** operator is a primitive process which implements the handshaking expansion

$$(ui, vi)\mathbf{me}(uo, vo) \equiv *[[ui \longrightarrow uo\uparrow; [\neg ui]; uo\downarrow \| vi \longrightarrow vo\uparrow; [\neg vi]; vo\downarrow]]$$

as long as the four-phase protocol is obeyed on ui, uo and vi, vo.

The second case, after handshaking expansion, is an instantiation of the synchronizer primitive process

$$(qi, xi, li)\mathbf{S}(q(1)o, q(0)o)$$

When the enable signal qi is raised, this element behaves like a standard mutual exclusion operator. However, when qi is low, both outputs remain low.

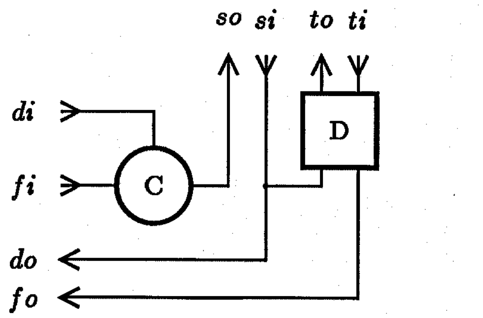

Figure 5.3: Reshuffled Circuit for Multiple Passive Communications

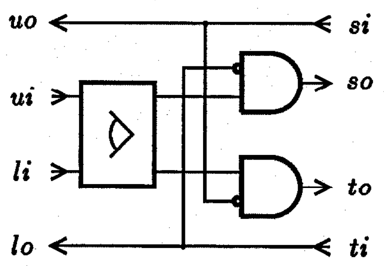

Figure 5.4: Non–busy-waiting Arbiter

5.5 Example

We apply these optimizations to the *priv0* circuit produced in the last chapter. The arbiter process is used instead of the busy-waiting implementation of the guard set in the main process. The redundant **D** operators sequencing **skip** statements are also removed. The communication R does not effect the guards b and $\neg b$ and thus the sequencing operators corresponding to the arrows in the procedure may be removed. No statements in the procedure influence the guards of the main process and thus no sequencing element is needed to implement the arrows of that selection statement either. The sole sequencing remaining separates the procedure call from assignment to the variable b. By manipulating the sequencing tree for both paths through the main selection statement, a sequencing operator may be factor though the merger for the procedure call. After these optimizations, only one sequencing element remains. The resulting circuit is shown in Figure 5.5. This circuit corresponds exactly to the best circuit produced by hand compilation.

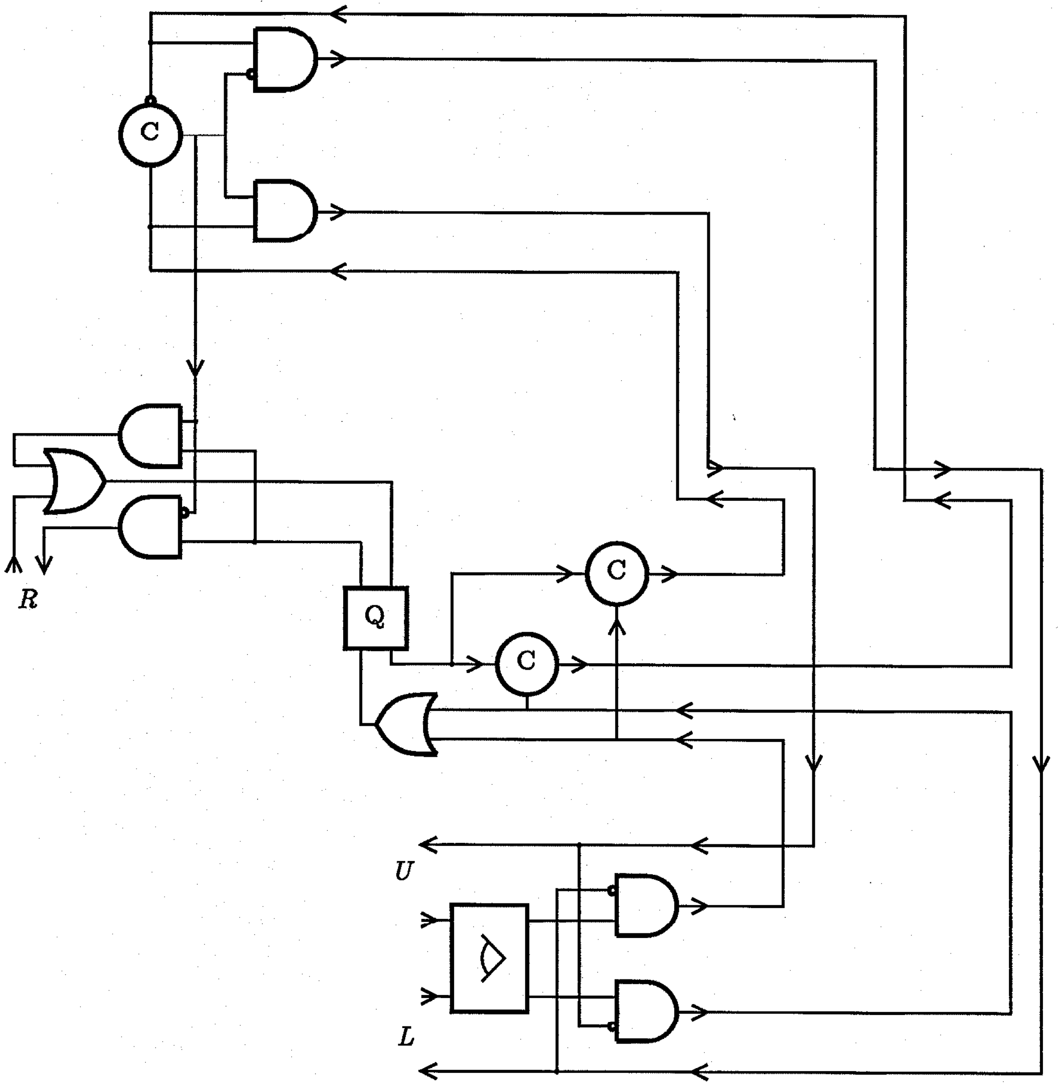

Figure 5.5: Optimized Circuit for the *priv0* Process

Conclusion

We have produced a working automatic compiler that implements a subset of the translation techniques described in this thesis. This compiler closely follows the structure of standard source–to–machine code compilers, in particular the Prolog implementations described in [10,11]. While it is beyond the scope of this thesis to describe the compiler in detail, we will make claims about its performance, and thus make claims about the quality of the circuits produced by the compilation technique as a whole. While many criteria make up the quality of a circuit, we will restrict our attention to the size of the resulting circuit, defined in terms of the number of gates or the number of transistors needed to implement it.

The circuits produced by the compiler are slightly larger than those produced by directly applying Martin's method. The compiler does not introduce state variables between arbitrary actions in the handshaking expansion of the process, nor does it always introduce the minimal number of these variables. The compiler is handicapped in this sense. However, the optimal introduction of state variables is a hard problem and solutions to it will be tractable only for small programs. The compiler produces good solutions, yet they are not always optimal.

We claim the solutions are good because we can prove that a circuit produced by the compiler is no larger than a constant times the size of its corresponding input description. The proof follows from that fact that each program is decomposed into a number of sub-processes proportional to the number of constructs in the program, and each resulting sub-process has a constant sized implementation. (For this statement to be true, we may not use the exclusive guard set technique described in Section 2.4.3 because of the possible exponential blow-up when representing an arbitrary boolean function in disjunctive normal form. However, the non-exclusive guard set technique of Section 2.4.4 may be used in its place.) We can not expect a better asymptotic growth rate for our compiler.

We have compared several medium sized designs produced by hand at Caltech with designs produced by the compiler.

- $3x+1$ Engine (Tony Lee)
- Systolic Multiplier (Pieter Hazewindus)
- Bit-serial Routing Automata (Steve Burns)

- Lazy Stack (Alain Martin and Steve Burns)
- Ring of Mutual Exclusion (Alain Martin and Andy Fyfe)

The results vary: the compiled designs being no more than three times as large as the hand designs and in some cases smaller than the hand designs. The variability is due mostly to difficulties in specifying efficiently the algorithms used in the hand designs in our source language.

In summary, the compilation procedure described in this thesis represents a complete, systematic method for translating an arbitrary concurrent program into a self-timed circuit. Furthermore, the circuits produced are of high quality. The size of a constructed circuit is no larger than a constant times the size of the input specification. Also, the translation mechanism mimics that used in standard source code to machine code compilers, and thus compiler theory can be applied to construct a working translator.

Bibliography

[1] M. Annaratone, *Digital CMOS Circuit Design*, Kluwer Academic, Boston, pp 134-5 (1986)

[2] E. W. Dijkstra, "Guarded commands, nondeterminacy, and formal derivation of programs", *Comm. ACM* 18, 8, pp 453-457 (August 1975)

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

[4] A. J. Martin, "An Axiomatic Definition of Synchronization Primitives", *Acta Informatica*, 16, pp 219-235 (1981)

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

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

[7] 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)

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

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

[10] L. Sterling and E. Shapiro, *The Art of Prolog*, The MIT Press, Cambridge MA (1986)

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