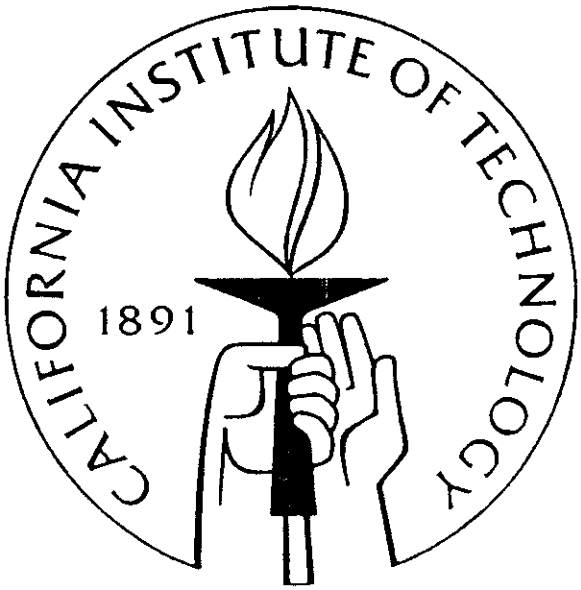

Synthesis of Asynchronous VLSI Circuits

Alain J. Martin

Computer Science Department
California Institute of Technology

Caltech-CS-TR-93-28

Erratum: Synthesis of Asynchronous VLSI Circuits

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA

March 22, 2000

This document is old (1991) and in several respects doesn't describe Caltech's current approach to asynchronous VLSI design, especially concerning design for high throughput. However, everything in the document is valid and relevant to today's design, except for one error.

On pp. 38 through 41, arbiters and synchronizers are described and a circuit implementation is given for the arbiter. This implementation, which has been used successfully in many circuits, is correct, and, as far as I know, it is the best implementation of an arbiter.

However, the document suggests that the synchronizer can be implemented in the same way as the arbiter, although it does not actually give such an implementation. An implementation of the synchronizer similar to the one of the arbiter would be incorrect, as the circuit could deadlock in some pathological cases.

We are currently writing a paper describing a correct implementation of the synchronizer. Anybody interested in getting a copy should send an email to alain@cs.caltech.edu.

Synthesis of Asynchronous VLSI Circuits

Alain J. Martin[1]
Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA

August 9, 1991

[1]The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

Contents

Chapter 1

Introduction

Delays have dangerous ends.
William Shakespeare

With chip size reaching one million transistors, the complexity of VLSI algorithms—i.e., algorithms implemented as a digital VLSI circuit—is approaching that of software algorithms—i.e., algorithms implemented as code for a stored-program computer. Yet design methods for VLSI algorithms lag far behind the potential of the technology.

Since a digital circuit is the implementation of a concurrent algorithm, we propose a concurrent programming approach to digital VLSI design. The circuit to be designed is first implemented as a concurrent program that fulfills the logical specification of the circuit. The program is then compiled—manually or automatically—into a circuit by applying semantic-preserving program transformations. Hence, the circuit obtained is correct by construction.

The main obstacle to such a method is finding an interface that provides a good separation of the physical and algorithmic concerns. Among the physical parameters of the implementation, *timing* is the most difficult to isolate from the logical design, because the timing properties of a circuit are essential not

only to its real-time behavior, but also to its logical correctness if the usual synchronous techniques are used to implement sequencing.

For this reason, *delay-insensitive* techniques are particularly attractive for VLSI synthesis. A circuit is delay-insensitive when its correct operation is independent of any assumption on delays in operators and wires except that the delays be finite[21] . Such circuits do not use a clock signal or knowledge about delays.

Let us clarify a matter of definitions right away: It has been proved in [14] that the class of entirely delay-insensitive circuits is very limited. Different asynchronous techniques distinguish themselves in the choice of the compromises to delay-insensitivity.

Speed-independent techniques assume that delays in gates are arbitrary, but there are no delays in wires[17]. *Self-timed* techniques assume that a circuit can be decomposed into *equipotential* regions inside which wire delays are negligible[20]. In our method, certain local 'forks' are introduced to distribute a variable as inputs of several operators. We assume that the differences in delays between the branches of the fork are shorter than the delays in the operators to which the fork is an input. We call such forks *isochronic*.

Although we initially chose delay-insensitive techniques for reasons of methodology, those techniques present other important advantages in terms of efficiency and robustness:

- The clock rate of a synchronous design has to be slowed to account for the worst-case clock skews in the circuit, and for the slowest step in a sequence of actions. Since delay-insensitive circuits do not use clocks, they are potentially faster than their synchronous equivalent.
- Since the logical correctness of the circuits is independent of the values of the physical parameters, delay-insensitive circuits are very robust to variations of these parameters caused by scaling or fabrication, or by some non-deterministic behavior such as the metastability of arbiters. For instance, all the chips we have designed have been found to be functional in a range of voltage values (for the constant voltage level encoding the high logical value) from above 10V to below 1V.
- Delay-insensitive circuit design can be modular: A part of a circuit can be replaced by a logically equivalent one and safely incorporated into the design without changes of interfaces.
- Because an operator of a delay-insensitive circuit is "fired" only when its firing contributes to the next step of the computation, the power consumption of such circuit can be much lower than that of its synchronous equivalent.
- Since the correctness of the circuits is independent of propagation delays in wires and, thus, of the length of the wires, the layout of chips is facilitated.

The method indeed produces correct and efficient circuits. It has been applied, both with "hand compilation" and automatic compilation, to a series

of difficult design problems, such as distributed mutual exclusion, fair arbitration, routing automata, stack, and serial multiplier. All fabricated chips have been found to be correct on "first silicon". Although our CMOS implementation of the basic operators has been overly cautious, and the electrical optimization techniques have been rather tame, the performance of the chips has been found to be at least equal to that of synchronous implementations. We have just completed the design of a general-purpose microprocessor, and its performances are very encouraging: in $1.6\mu m$ SCMOS, it runs at 18 million instructions per second. (See later for more detail.)

The main reason for the efficiency of the method is that, rather than going in one step from program to circuit, the designer applies a series of transformations to the original program. At each stage, powerful algebraic manipulations can be performed leading to important optimizations in terms of speed or area.

The most encouraging aspect of the method is that it is really a synthesis technique: it allows a designer to construct solutions that he would never have found had he not applied the method. We shall observe that different applications of the transformations lead to different circuits for the same program. Although all circuits are semantically equivalent, they may exhibit different behaviors in terms of speed or size (number of operators used). The method therefore includes the trade-offs between simplicity and efficiency that should be available to the VLSI designer.

Using concurrency to implement a sequential computation may seem wasteful at first sight. But VLSI is essentially a concurrent medium: concurrency is implemented at no cost by mere juxtaposition of the concurrent parts. On the other hand, implementing sequencing requires synchronization and is, in general, more expensive. We shall therefore implement sequencing as restricted concurrency. Once a process has been transformed into a semantically equivalent set, the problem of implementing sequencing has disappeared!

This technique entails one of the main novelties of the method. Other techniques implement sequencing by transforming the computation into a finite-state machine, and realizing each state with a state-holding element. In our technique, some state-holding elements may be needed: we shall see that in the transformation from sequences to PR set we may have to introduce so-called state variables which correspond to state-holding elements. But the number of those elements is drastically less than in techniques using finite-state machines.

We first introduce the "source code" notation, called *Communicating Hardware Processes*, or CHP, which is a concurrent programming notation inspired by C.A.R. Hoare's CSP[5]: A program is a set of concurrent processes communicating by input and output commands on channels. Second, we describe the object code notation, called *production rule set*, which is an entirely concur-

rent programming paradigm: All enabled commands can be fired concurrently at any time. This notation is one of the main innovations of the method and is an interesting notation for digital VLSI all by itself.

Next, we describe the four main steps of the compilation (process decomposition, handshaking expansion, production rule expansion, operator reduction) and illustrate them with a number of examples. In particular, we present the different algebraic transformations that can be applied at different stages of the compilation, and which give the method its flexibility and efficiency.

Chapter 2

Communicating Hardware Processes

The source notation is a program notation and not a hardware description language. It is inspired by C.A.R. Hoare's CSP[5] and E.W. Dijkstra's guarded commands[3], and is based on assignment and process communication by message-passing.

2.1 Data Types and Assignment

The only basic data type is the boolean. The other types—integer and floating point—are collections of booleans that we represent in the PASCAL record notation.

For b boolean, the command $b :=$ **true**, also denoted $b\uparrow$, is the assignment of the value **true** to b. Similarly, the command $b :=$ **false**, also denoted $b\downarrow$, is the assignment of the value **false** to b.

An integer of "length" n is a predefined record type consisting of n boolean components ("fields" in the PASCAL jargon). For instance, if x is declared as an integer of length 8, then x is a collection of the 8 boolean variables: $x.0$, $x.1$, $x.2$,..., $x.7$.

The existence of this predefined record type for integers does not preclude the programmer from introducing other records to structure the data. For instance, in the program of the microprocessor, which we will introduce later, the integer variable i represents (contains) the currently executed instruction. This value is declared as a record of several types depending on the type of the instruction. For ALU instructions and ordinary memory instructions, the

type is:

$$\begin{array}{lll} alu & = & \textbf{record} \\ & & op : alu.15..alu.12 \\ & & x : alu.11..alu.8 \\ & & y : alu.7..alu.4 \\ & & z : alu.3..alu.0 \\ & & \textbf{end}, \end{array}$$

where the field *op* contains the "opcode" of the instruction, the fields x and y contain the indices of the registers to be used as parameters of the instruction, and z contains the index of the register in which the result of the instruction execution is to be stored.

Since operations on boolean variables are the only primitive operations, any operation on other data types appearing in a program must be understood to be a shorthand notation or function call for the sequence of operations on boolean variables that will implement it.

For instance, given two integers x and y of the same length n, the assignment

$$y := x$$

is a shorthand notation for the multiple assignment

$$y.0,\ y.1,\ \ldots,\ y.(n-1) := x.0,\ x.1,\ \ldots,\ x.(n-1).$$

The multiple assignment of n expressions to n variables is different from the concurrent composition (which we will introduce shortly) of the n elementary assignments. In the multiple assignment, the n expressions are all evaluated before the results are assigned to the corresponding variables.

For the sake of clarity, we will use the usual integer arithmetic operators (for instance, $y := x + 1$ in the program of the microprocessor) in the first description of an algorithm. However, since these operators are not primitive constructs of the language, they are subsequently replaced with calls to functions that implement the operators in terms of boolean operations.

2.2 Arrays

The array mechanism is an address-calculation mechanism, and is used when the identity of the element in a set of variables that is to be used for some action will be determined during the computation. For example, the processor uses three arrays: the instruction memory array, *imem*, whose index is the program counter, *pc*; the data memory array, *dmem*; and the array of general-purpose registers, *reg*. Hence, the execution of a *load* instruction, i, is described by the assignment:

$$reg[i.z] := dmem\big[reg[i.x] + reg[i.y]\big].$$

In this example, $reg[i.z]$ represents the register whose location (address in the array) is the current value of the field z of the current instruction i. And similarly for $reg[i.x]$ and $reg[i.y]$. The assignment assigns to $reg[i.z]$ the value of the element in the array *dmem* at a location which is the sum of the contents of registers $reg[i.x]$ and $reg[i.y]$.

2.3 Composition Operators

There are three composition operators (also called "constructors"): the sequential operator, represented by the semicolon; the concurrent, or parallel, operator represented by the parallel bar, ||; and the coincident operator, represented by the bullet.

The semantics of the sequential composition $S1; S2$ are well known: "First execute $S1$ and then execute $S2$." The semicolon is associative, but of course not commutative.

We will assume that the semantics of the parallel composition are also well known, although we are aware of how difficult it is to define these semantics formally and simply: $S1 \parallel S2$ denotes the parallel, or concurrent, execution of $S1$ and $S2$.

We postulate that the parallel composition is *weakly fair: If at a certain point of the computation of* $S1 \parallel S2$, x *is the next atomic action of* $S1$, *then* x *will be executed after a finite number of atomic actions of* $S2$.

Parallel composition is associative and commutative.

The bullet operator is used solely to compose communication commands. (Communication commands will be introduced later.) Furthermore, the coincident composition of two communication commands is defined only if the two commands are *non-interfering: Two programs are non-interfering if a variable modified by one program is not used by the other program.*

For $S1$ and $S2$ non-interfering communication commands, if the executions of both $S1$ and $S2$ in a certain state of the computation terminate, then the execution of $S1 \bullet S2$ in that state terminates. Furthermore, the completion of $S1$ coincides with the completion of $S2$; i.e., $S1$ and $S2$ are completed in the same state of the computation. (We will return to this definition later when we define the notion of completion of a non-atomic action.)

The bullet operator is associative and commutative.

If $S1$ and $S2$ are non-interfering communication commands, the execution of $S1 \parallel S2$ is equivalent to the execution of either $S1; S2$ or $S2; S1$ or $S1 \bullet S2$.

The bullet has the highest priority, followed by the semicolon, followed by the parallel bar:

$$S0 \bullet S1; S2 \parallel S3 \equiv ((S0 \bullet S1); S2) \parallel S3.$$

2.4 Control Structures

The two control structures are the *selection* and the *repetition* of Dijkstra's guarded commands. However, the VLSI programmer and the software programmer adopt opposite attitudes towards non-determinism. Whereas the latter is encouraged to maximize non-determinism as a way to avoid unnecessary choices, the former is requested to minimize non-determinism to reduce the high cost of arbitration in a direct VLSI implementation of a set of guarded commands.

It is very difficult, if at all possible, to determine at "compile-time" which selections require arbitration. We therefore introduce two sets of control structures, a deterministic set and a non-deterministic set, and let the programmer explicitly indicate where arbitration is needed.

2.4.1 Selection

The execution of the deterministic selection command

$$[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n],$$

where G_1 through G_n are boolean expressions, S_1 through S_n are program parts, (G_i is called a "guard," and $G_i \rightarrow S_i$ is called a "guarded command") amounts to the execution of the arbitrary S_i for which G_i holds. At any time at most one guard holds. If none of the guards is **true** , the execution of the command is suspended until one guard is **true** .

The non-deterministic selection command

$$[G_1 \rightarrow S_1 | \ldots | G_n \rightarrow S_n]$$

is identical to the previous one, except that several guards may be **true** at the same time. In such a case, an arbitrary true guard is selected.

2.4.2 Repetition

The execution of the deterministic repetition command

$$*[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n],$$

where G_1 through G_n are boolean expressions, and S_1 through S_n are program parts, amounts to repeatedly selecting the arbitrary S_i for which G_i holds, and executing S_i. At any time, at most one guard holds. If none of the guards is **true** , the repetition terminates.

The non-deterministic repetition command

$$*[G_1 \rightarrow S_1 | \ldots | G_n \rightarrow S_n]$$

is identical to the previous one, except that several guards may be **true** at the same time. In such a case, an arbitrary true guard is selected.

$[G]$, where G is a boolean expression, stands for $[G \rightarrow \mathbf{skip}]$, and thus for "wait until G holds." (Hence, "$[G]$; S" and $[G \rightarrow S]$ are equivalent.)

$*[S]$ stands for $*[\mathbf{true} \rightarrow S]$ and, thus, for "repeat S forever."

2.4.3 Reactive Process Structure

From the preceding definitions, the operational description of the statement

$$*[[G_1 \rightarrow S_1] \| \ldots \| G_n \rightarrow S_n]]$$

is "repeat forever: Wait until some G_i holds; execute an S_i for which G_i holds." This structure, which we call "reactive," is used very frequently. For instance, the *server* processes in the distributed mutual exclusion example are reactive processes.

2.5 The Replication Construct

Both because of the restriction of basic operations to booleans and because of the high degree of concurrency of VLSI algorithms, such algorithms are characterized by an extensive use of *replication.* A typical example is that some action has to be performed (sequentially or concurrently) on all the boolean variables that represent an integer. Another example is that of an n-place buffer constructed as the concurrent composition of n identical one-place buffers.

The notation therefore contains a syntactic operator, called the replication construct, which makes it possible to "clone" any program part into a number of instances.

The replication mechanism is used to represent a fixed, finite, and non-empty list of syntactic objects. Operationally, we can say that the replication mechanism is used to generate a list of objects at compile-time. An element of the list is any program part. The concatenation operator of the list is any constructor or separator of the language. The constructors are the semicolon for sequential composition, and the comma and the parallel bar for parallel composition. The separators are the bar for guarded commands, and the blank and the comma for lists of declarations.

Recursion is the basic mechanism for creating such a list. Since it is often convenient to "unroll" the simplest form of tail recursion as an iteration mechanism, both iteration and recursion are available.

The construct for replication by iteration is defined as follows: If

- **op** is any constructor or separator,

- i is an integer variable, called the *running index*,
- the *range*, defined by $n..m$, where n and m are integer constants, is not empty, i.e., $n \leq m$,
- $S(i)$ is any program part in which i appears free,

then,

$$\langle \mathbf{op} i : n..m : S(i) \rangle \stackrel{\text{def}}{=} \begin{cases} S(n), & \text{if } n = m \\ S(n) \ \mathbf{op} \ \langle \mathbf{op} \ i : n+1..m : S(i) \rangle, & \text{if } n < m \end{cases}$$

For $n < m$, the definition is ambiguous if **op** is not associative. In this case the definition is taken to be equivalent to

$$S(n) \ \mathbf{op} \ (\langle \ \mathbf{op} \ i : n+1..m : S(i) \rangle).$$

The bracket notation for replication is borrowed from Chandy and Misra[2], who use it for defining so-called quantified expressions. Observe that a replication command is not a quantified expression.

For example, the construct $[\langle [] i : 0..3 : G(i) \rightarrow S(i) \rangle]$ expands to

$$\begin{array}{l} [G(0) \rightarrow S(0) \\ []G(1) \rightarrow S(1) \\ []G(2) \rightarrow S(2) \\ []G(3) \rightarrow S(3) \\ ]. \end{array}$$

The construct

$$\langle ; i : 0..2 : x.i := y.((i+1) \mathbf{mod} 3) \rangle$$

expands to

$$x.0 := y.1; \ x.1 := y.2; \ x.2 := y.0.$$

Replication constructs can be nested as in the following example:

$$\langle , i : 0..9 : \langle , j : 0..i : x(i,j) = 0) \rangle \rangle.$$

2.6 Procedures and Functions

Procedures are used with a simple parameter mechanism: A parameter is either *input* or *output*. For procedure p, declared as

$$\mathbf{procedure} \ p(x : \mathit{input}; y : \mathit{output}); S$$

the call $p(a, b)$ is equivalent to the program part

$$x := a; S; b := y.$$

A parameter of a function is always an input parameter. For function y, declared as

$$\textbf{function } y(x); S$$

where S is the same program part as in procedure p, a statement Q containing the function call $y(a)$ is equivalent to the program part

$$p(a, b); Q_b^{f(a)},$$

where b is a "fresh" variable.

Tail recursion is allowed but not general recursion, since general recursion requires the construction, at execution time, of a stack whose size may vary with the parameters of the computation.

2.7 Concurrent Processes

The main building block for the construction of concurrent computations is the *process*. In the design of the microprocessor for instance, each stage of the pipeline is a process. Concurrent composition of processes is also the main source of concurrency, although we allow the concurrent composition of statements inside processes. In strict communicating-process design style, a variable is local to a process, and communication among processes is uniquely by way of message exchanges. In the design of the processor, we have violated this rule and allowed processes to share variables in a restricted way: A variable of one process may be inspected by another process. (Whether this relaxation of the locality rule is a useful extension or a weakness of the flesh is not clear at the moment. More experimentation is necessary.)

Hence, the most common structure for the body of a concurrent computation is the parallel construct:

$$p1 \parallel p2 \parallel \ldots \parallel pn$$

where $p1$ through pn are the names of processes that have been declared beforehand. A process is used very much as a procedure is used: It is first declared in a declaration statement and then called by using its name in a statement. Several instances of the same process type can be called by assigning different names. But, unlike procedures, each (instance of a) process can be called only once.

2.7.1 Communication Commands, Ports, and Channels

Processes communicate with each other by using communication commands on *ports*. A port of a process is paired with a port of another process to

form a *channel*. For the time being, we assume that a channel is shared by exactly two processes; later, we will generalize the definition to more than two processes. For instance, the microprocessor uses one-to-one, one-to-many, and many-to-many (buses) channels.

A process is either *elementary* or *composite*. The ports of an elementary process are *external*: Each is to be connected by a channel to a port of another process to form a composite process. The external ports of a process are declared in the heading of the process, like the parameters of a procedure:

$$p \equiv \textbf{process}(R, L)$$

(Later on, we will add some type information to the declaration.) A composite process, p, is the parallel composition of several processes. The ports of a component process that are connected by a channel to ports of another component process are internal to p. The ports of the components that are left unconnected (dangling) are the external ports of p. The internal ports and the channels are defined by channel declaration in the process body.

We use two equivalent naming mechanisms for ports and channels. The first one gives local names to ports and pairs the two ports of a channel. For example, let two processes, $p1$ and $p2$, share a channel with port X in $p1$ and port Y in $p2$. The declaration is as follows:

$$\begin{aligned} p1 \equiv\ & \textbf{process}(X) \ldots \textbf{end} \\ p2 \equiv\ & \textbf{process}(Y) \ldots \textbf{end} \\ & p1 \parallel p2 \\ & \textbf{chan}(p1.X, p2.Y) \end{aligned}$$

The second mechanism gives global names to channels, and uses the channel names for all ports of the same channel. For instance, the same two processes would be described as:

$$\begin{aligned} p1 \equiv\ & \textbf{process}(C) \ldots \textbf{end} \\ p2 \equiv\ & \textbf{process}(C) \ldots \textbf{end} \\ & p1 \parallel p2 \\ & \textbf{chan}\ C \end{aligned}$$

We prefer local names for ports when the processes involved are identical (as in the case of the server processes in the distributed mutual-exclusion example); we prefer global names when the processes are different because this reduces the nomenclature. (We have used global names in the description of the processor.)

If the channel is used only for synchronization between the processes, the name of the port is sufficient for identifying a communication on this port. For instance, in the program for distributed mutual exclusion, the channel between a "master" process and its "server" process is identified with port D in the master and port U in the server, and is used for synchronization only.

2.7.2 Semantics of Synchronization

Since a message cannot be received before it has been sent, communications actions on the two ports of a same channel have to be synchronized. The weakest form of synchronization between the send actions on one port of a channel and the receive actions on the other port of the same channel is that at any moment the number cR of completed receive actions is at most equal to the number cS of completed send actions:

$$cR \leq cS$$

The difference $cS - cR$ is the number of messages sent that have not yet been received. These messages have to be buffered somewhere "in the channel." Allowing message buffering in the channels obviously implies that channels be implemented as complex storage devices. In view of our intention to use communication as an elementary sequencing and synchronization mechanism, we want to opt for as simple an implementation of channels as possible. Clearly, the simplest implementation is one in which *no* buffering of messages is required. In turn, this choice implies that the synchronization between send and receive actions on a channel be such that at any time $cR = cS$. Hence the following definition of the synchronization property of communication primitives.

If two processes, $p1$ and $p2$, share a channel with port X in $p1$ and port Y in $p2$, then, at any time, the number of completed X-actions in $p1$ will equal the number of completed Y-actions in $p2$; in other words, the completion of the n-th X-action "coincides" with the completion of the n-th Y-action.

If, for example, $p1$ reaches the n-th X-action before $p2$ reaches the n-th Y-action, the completion of X is suspended until $p2$ reaches Y. The X-action is then said to be *pending*. When, thereafter, $p2$ reaches Y, both X and Y are completed. The predicate "X is pending" is denoted as $\mathbf{q}X$.

If, for an arbitrary command A, $\mathbf{c}A$ denotes the number of completed A-actions, the semantics of a pair (X, Y) of communication commands is expressed by the two axioms:

$$\begin{array}{lcl} \mathbf{c}X & = & \mathbf{c}Y \\ \neg\mathbf{q}X & \vee & \neg\mathbf{q}Y \end{array}$$

2.7.3 Probe

Instead of the usual selection mechanism by which a set of pending communication actions can be selected for execution, we provide a general boolean command on channels, called the *probe*. The definition of the probe[6] states that the probe command $\overline{X}$ in process $p1$ has the same value as $\mathbf{q}Y$, and,

symmetrically, the probe command $\overline{Y}$ in process $p2$ has the same value as $\mathbf{q}X$.

Hence, in the guarded command $\overline{X} \rightarrow X$, the X-action is not suspended since $\mathbf{q}Y$ holds as a precondition of X.

Remark: In view of our declared intention to implement processes in a distributed and delay-insensitive way, our choice of definitions for communication may already puzzle some readers: The definition of $A1$ relies on the *simultaneous* completion of two actions in two different processes, and the value of the probe in one process is supposed to be *identical* to a suspended state of another process. A short explanation is that we have chosen definitions of completion and suspension that are unorthodox but valid! □

2.7.4 Example

Process *sel* repeatedly performs communication action X or communication action Y, whichever can be completed; *sel* is blocked if and only if neither X nor Y can be completed. The program body of *sel* is:

$$*[[\overline{X} \rightarrow\ X[]\overline{Y} \rightarrow Y]]$$

Obviously, process *sel* is not fair, because of the non-deterministic choice of a guard when both guards are true. Negated probes make it possible to transform *sel* into a fair version, *fsel*, whose body is:

$$\begin{array}{l} *[\,[\overline{X} \rightarrow X;\ [\overline{Y} \rightarrow Y[]\neg\overline{Y} \rightarrow \mathbf{skip}] \\ \quad []\overline{Y} \rightarrow Y;\ [\overline{X} \rightarrow X[]\neg\overline{X} \rightarrow \mathbf{skip}] \\ \,]\,] \,. \end{array}$$

This example illustrates the fact that negated probes are necessary for implementing fairness.

2.7.5 Communication

Matching communication actions are also used to implement a form of distributed assignment statement, to "pass messages" as it is often said. In that case, the pair of commands is specified to consist of an input command and an output command by adjoining to them the symbols "?" and "!", respectively. For example, $X?$ is an input command and then X is an input port, and $Y!$ is an output command, and then Y is an output port.

Communication axiom. *Let $X?u$ and $Y!v$ be matching, where u is a process variable, and v is an expression of the same type as u. The communication implements the assignment $u := v$. In other words, if $v = V$ before the communication, $u = V$ and $v = V$ after the communication.*

2.8 Examples

In this section, we illustrate the notation with a number of typical examples. The programs are given with a brief informal explanation. All proofs of correctnes are omitted.

2.8.1 Stream Merge

A process has two input ports X and Y, and an output port Z. The process outputs on port Z a stream of messages which is an arbitrary merge of the stream of messages received on X and the stream of messages received on Y. (The type of the messages is irrelevant. For the completeness of the declarations, let us assume they are integer of size 8.) The streams received on X and Y can each be either empty, or finite, or infinite. Because of the possibility that no message will be received on an input port in a current state of the system, an input port has to be probed before each input communication on the port in order to avoid deadlock. The solution is:

$$
\begin{array}{l}
MERGE \equiv \ \textbf{process}(X?int(8), Y?int(8), Z!int(8)) \\
\quad u : int(8) \\
\quad *[\ [\overline{X} \rightarrow X?u; Z!u] \\
\quad\quad \| \overline{Y} \rightarrow Y?u; Z!u] \\
\quad\quad] \\
\quad]\textbf{end}
\end{array}
$$

A number of remarks are in order. First, observe that the process has the typical "reactive process" structure mentioned in Subsection 2.4.3. Second, in absence of any other specification, we have to assume that both probes may be true at the same time if there are pending communications on both input ports at the same time. We therefore had to use the nondeterministic version of the selection statement. Third, the above solution requires an internal variable u of the same type as the messages to buffer the last message received and not yet sent. But such a buffering is expensive and, in this case at least, unnecessary. We can directly output on Z the message being received on X or Y. Instead of $X?u; Z!u$, we can write $Z!(X?)$. And similarly for the other guarded command.

2.8.2 Buffers

Next, we construct a one-place buffer. The process inputs 8-bit integer messages on the input port L, and outputs them in the same order on the output port R.

$$
\begin{aligned}
BUF1 \equiv\ & \textbf{process}(L?int(8), R!int(8)) \\
& x : int(8) \\
& *[L?x; R!x] \\
& \textbf{end}
\end{aligned}
$$

(Like the previous example, the above process can be implemented without introducing the internal variable x.)

A buffer of size n can be constructed as the linear composition of n one-place buffers.

$$
\begin{aligned}
1 \quad & BUF(n) \equiv \textbf{process}(L?int(8), R!int(8)) \\
2 \quad & p(i : 0..n-1) : BUF1 \\
3 \quad & \langle \| i : 0..n-1 : p(i) \rangle \\
4 \quad & \textbf{chan}\langle i : 0..n-2 : (p(i).R, p(i+1).L) \rangle \\
5 \quad & p(0).L = BUF(n).L \\
6 \quad & p(n-1).R = BUF(n).R \\
7 \quad & \textbf{end}
\end{aligned}
$$

Let us briefly explain the different commands of this declaration. Line 1 is the usual heading which contains the declaration of the external ports of the process, here L and R both of the type "integer of size 8." Line 2 is the declaration of n internal processes $p(0)$ through $p(n-1)$ of the type one-place buffer ($BUF1$). Line 3 is the body of the process which consists of the parallel composition of the n one-place buffers previously declared.

Line 4 describes how the internal ports are connected to form internal channels. Line 5 and line 6 are the identification of the external ports with two ports of the internal processes.

2.8.3 A Lazy Stack

We implement a stack S of size n, $n > 0$, as a string of n communicating processes defined as follows:

$$
S = \begin{cases} h, & \text{if } n = 1, \\ (h \| T), & \text{if } n > 1, \end{cases}
$$

where h, the head of the stack, is a process, and T, the tail of the stack, is a stack of size $n-1$. Process h communicates with the environment of the stack by the communication actions $in?x$ and $out!x$, and with T by the communication actions $put!x$ and $get?x$. Hence, $h.put$ matches $T.in$, and $h.get$ matches $T.out$. (We assume that no attempt is ever made to add a portion to a full stack, or to remove a portion from an empty stack.)

Each stack element is either empty and behaves as procedure E, or is full and behaves as procedure F. The epithet "lazy" is attributed to this

stack because no reshuffling of portions takes place after a portion has been removed from a full stack element. Hence, the full portions in the stack are not necessarily contiguous.

$$
\begin{array}{rl}
E \equiv & \textbf{procedure} \\
& [\overline{in} \rightarrow in?x;\ F \\
& \|\overline{out} \rightarrow get?x;\ out!x;\ E \\
&]\textbf{end} \\
F \equiv & \textbf{procedure} \\
& [\overline{out} \rightarrow out!x;\ E \\
& \|\overline{in} \rightarrow put!x;\ in?x;\ F \\
&]\textbf{end}\ .
\end{array}
$$

If we assume that a stack element is initially empty, such an element is described by the following process:

$$
\begin{array}{rl}
stack - element \equiv & \textbf{process}(in?int(8), out!int(8), get?int(8), put!int(8)) \\
& x : int(8) \\
& E \\
& \textbf{end}
\end{array}
$$

The following alternative coding of the body of the stack element process, due to Peter Hofstee, illustrates the advantages of the probe construct:

$$
\begin{array}{rl}
E \equiv & *[\,[\overline{in} \rightarrow in?x \\
& \|\overline{out} \rightarrow get?x \\
&]; \\
& [\overline{out} \rightarrow out!x \\
& \|\overline{in} \rightarrow put!x \\
&]\,]\ .
\end{array}
$$

2.8.4 Palindrome Recognizer

A palindrome is a finite sequence of characters (word, sentence) that reads the same backward and forward. Discounting the difference between uppercase and lowercase, the sentence "Able was I ere I saw Elba" is a palindrome. In other words, the sequence $S(i : 0..n-1)$ is palindrome if and only if:

$$\forall i : 0..\lfloor \frac{n}{2} \rfloor - 1 : S(i) = S(n-1-i)$$

We want to design a process that determines which prefixes of a given sequence of at most m characters are palindromes.

More precisely, the environment behaves as the process (body)

$$*[put!x; get?b]$$

where x is a character and b is the boolean whose value is equal to the predicate "the sequence of characters transmitted on port *put* so far is a palindrome."

The palindrome recognizer *pal* communicates with the environment through the input port *in* and the output port *out*: For each character received on *in*, the boolean answer is output on *out* whose value is "the sequence of characters received on port *in* so far is a palindrome."

The process *pal* is a linear array of M elementary processes $p(i : 0..M-1)$ of type *cell*, with $M = \lceil \frac{m}{2} \rceil$.

$$
\begin{aligned}
cell \equiv\ & \textbf{process}(in?char, out!boolean, put!char, get?boolean) \\
& \quad \textbf{var} x, y : char, z : boolean \\
& \quad in?x; out!\textbf{true}; z := \textbf{true}; \\
& \quad *[in?y; out!((x = y) \wedge z); put!y; get?z] \\
& \quad \textbf{end}
\end{aligned}
$$

$$
\begin{aligned}
pal \equiv\ & \textbf{process}(in?char, out!boolean) \\
& \quad p(i : 0..M-1) : cell \\
& \quad \langle \| i : 0..M-1 : p(i) \rangle \\
& \quad \textbf{chan}\langle i : 0..M-2 : (p(i).put, p(i+1).in) \rangle \\
& \quad \textbf{chan}\langle i : 0..M-2 : (p(i).get, p(i+1).out) \rangle \\
& \quad p(0).in = pal.in \\
& \quad p(0).out = pal.out \\
& \quad \textbf{end}
\end{aligned}
$$

The structure of the *cell* process can be simplified for the "bottom" process $p(M-1)$. The *cell* process can also be improved by introducing concurrency between communications.

2.8.5 Distributed Mutual Exclusion on a Ring of Processes

An arbitrary number (> 1) of cyclic automata, called "masters," make independent requests for exclusive access to a shared resource. The circuit should handle the requests from the masters in such a way that

1. Any request is eventually granted, and
2. there is at most one master using the shared resource at any time.

The masters are independent of each other: They do not communicate with each other, and the activity of a master not using the resource should not influence the activity of other masters.

A master, M, communicates with its private server, m. When M wants to use the shared resource (M is said to be a *candidate*), it issues a request to

m. When the request is accepted, M uses that resource (for a finite period of time), and then informs m that the resource is free again.

The servers are connected in a ring. At any time, exactly one (arbitrary) server holds a "privilege." Only the "privileged server" may grant the resource to its master and thereby guarantee mutual exclusion on the access to the resource. A non-privileged server transmits a request from its master (or from its left-hand neighbor) to its right-hand neighbor. A request circulates to the right (clockwise) until it reaches a server whose master is a candidate (this server ignores the request until it has served its master) or reaches the privileged server. The privileged server reflects the privilege to the left (counter-clockwise) until it reaches the server that generated the request. This server then becomes privileged, and may grant the resource to its master. The strategy of passing requests clockwise and reflecting the privilege counterclockwise has two important advantages: First, no boolean message need actually be transmitted; second, no message need be reflected, as the completion of a pending request is interpreted as passing the privilege.

$$
\begin{array}{rcl}
\mathit{master} & \equiv & *[\ldots D; CS; D] \\
\mathit{server} & \equiv & *[\,[\overline{U} \rightarrow [b \rightarrow \mathit{skip} \| \neg b \rightarrow R]; U; U; b\uparrow \\
 & & \;\;[\overline{L} \rightarrow [b \rightarrow \mathit{skip} \| \neg b \rightarrow R]; L; b\downarrow \\
 & & \;]\,].
\end{array}
$$

The boolean b is used to encode the presence of the privilege. The nondeterministic bar indicates that both guards may be true at the same time, and therefore arbitration has to take place. We can describe a system in which n servers are connected in a ring by first defining a process *pair* consisting of a master and a server, and then connecting n pairs in a ring:

$$
\begin{array}{rl}
\mathit{pair} \equiv & \mathbf{process}(L, R) \\
 & m : \mathit{server} \\
 & M : \mathit{master} \\
 & (m \| M) \\
 & \mathbf{chan}(m.U, M.D) \\
 & \mathbf{end} \\
\mathit{ring} \equiv & \mathbf{process} \\
 & p(i : 0..n-1) : \mathit{pair} \\
 & \langle \| i : 0..n-1 : p(i) \rangle \\
 & \mathbf{chan}\langle i : 0..n-1 : (p(i).R, p((i+1)\mathbf{mod}n).L) \rangle \\
 & \mathbf{end}
\end{array}
$$

(For a complete description and proof of correctness, see [7].)

2.8.6 An Asynchronous Microprocessor

We will describe the design of an asynchronous microprocessor in Chapter 9. In this section, we briefly explain how the concurrent program for the processor was derived from a sequential version by semantics-preserving transformations. We do not show the complete derivation but only the first few steps.

The processor is first described as a sequential program, which is then transformed into a set of concurrent processes so as to increase the concurrency in the execution of a sequence of instructions by pipelining. The sequential program is a non-terminating loop, each step of which is a *FETCH* phase followed by an *EXECUTE* phase.

$$
\begin{array}{l}
*[\ FETCH : i, pc := imem[pc], pc + 1; \\
\qquad\qquad [off(i) \rightarrow offset, pc := imem[pc], pc + 1; \\
\qquad\qquad [\!]\neg off(i) \rightarrow skip \\
\qquad\qquad]; \\
\quad EXECUTE : [alu(i) \rightarrow (reg[i.z], f) := \\
\qquad\qquad\qquad\qquad\qquad\qquad aluf(reg[i.x], reg[i.y], i.op, f) \\
\qquad\qquad\quad [\!]ld(i) \rightarrow reg[i.z] := dmem[reg[i.x] + reg[i.y]] \\
\qquad\qquad\quad [\!]st(i) \rightarrow dmem[reg[i.x] + reg[i.y]] := reg[i.z] \\
\qquad\qquad\quad [\!]ldx(i) \rightarrow reg[i.z] := dmem[offset + reg[i.y]] \\
\qquad\qquad\quad [\!]stx(i) \rightarrow dmem[offset + reg[i.y]] := reg[i.z] \\
\qquad\qquad\quad [\!]lda(i) \rightarrow reg[i.z] := offset + reg[i.y] \\
\qquad\qquad\quad [\!]stpc(i) \rightarrow reg[i.z] := pc \\
\qquad\qquad\quad [\!]jmp(i) \rightarrow pc := reg[i.y] \\
\qquad\qquad\quad [\!]brch(i) \rightarrow [cond(f, i.cc) \rightarrow pc := pc + offset \\
\qquad\qquad\qquad\qquad\qquad\quad [\!]\neg cond(f, i.cc) \rightarrow skip \\
\qquad\qquad\qquad\qquad\qquad\quad] \\
\qquad\qquad\quad] \\
]\ .
\end{array}
$$

The variables of the program are the following: As we already mentioned, variable i contains the instruction currently being executed. All instructions contain an *op* field describing the *opcode*. The parameter fields depend on the types of the instructions. The most common ones, those for ALU, load, and store instructions, consist of the three parameters x, y, and z. Variable cc contains the condition code field of the branch instruction, and f contains the *flags* generated by the execution of an *alu* instruction.

The two memories are described as the arrays *imem* and *dmem*. The index to *imem* is the program counter variable *pc*. Variable *offset* contains the offset field that extends certain instructions to the following word. The

general-purpose registers are described as the array $reg[0\ldots15]$. Register $reg[0]$ is special: It always contains the value zero.

The function evaluation $(z, f) := aluf(x, y, op, f)$ evaluates an *alu* instruction with the opcode, op; parameters x and y; and the current value of the flags, f. The result is an integer, z, and a new value of the flags, f. The function, *aluf*, is not described in the program. The boolean functions used in guards all determine certain properties of the current instruction i and are assumed to be self-explanatory.

2.8.7 First Decomposition into Concurrent Processes

The first step of the decomposition consists in replacing the previous program with the program:

$$*[FETCH; E1!i; E2] \parallel *[E1?i; EXECUTE; E2]$$

We leave it as an exercise to the readers to convince themselves that this decomposition does not introduce concurrency. The concurrent program is strictly equivalent to the sequential one.

Concurrent activity between the two processes will be introduced by moving $E2$ forward in the code of *EXECUTE* so that the $n+1$rst iteration of *FETCH* can start before the nth iteration of *EXECUTE* is finished. This refinement, and the further decomposition of *EXECUTE* into several processes is not discussed here. The resulting program can be found in Chapter9.

The rest of the exercise will concentrate on the further decomposition of *FETCH* . The practical way to exploit concurrency in *FETCH* is through the implementation of the multiple assignments. We introduce a process for the instruction memory which communicates the next instruction at address pc by a communication action on channel ID. Observe that variable pc is shared by the two processes. We get the following program:

$$\begin{array}{rcl}
IMEM & \equiv & *[ID!imem[pc]] \\
FETCH & \equiv & *[\,(ID?i \parallel y := pc+1); pc := y; \\
 & & \quad [off(i) \rightarrow (ID?offset \parallel y := pc+1); pc := y \\
 & & \quad \|\neg off(i) \rightarrow skip \\
 & & \quad]; E1!i; E2 \\
 & &] \\
EXEC & \equiv & *[E1?i; EXECUTE; E2]
\end{array}$$

Next, we delegate the execution of the assignments $y := pc+1; pc := y$ to a

separate process as follows:

$$
\begin{array}{lcl}
FETCH & \equiv & *[\,PCI1; ID?i; PCI2; \\
 & & \quad [\mathit{off}(i) \rightarrow PCI1; ID?\mathit{offset}; PCI2 \\
 & & \quad [\!]\neg \mathit{off}(i) \rightarrow \mathit{skip} \\
 & & \quad]; E1!i; E2 \\
 & &] \\
PCADD & \equiv & *[PCI1; y := pc + 1; PCI2; pc := y]
\end{array}
$$

(The reader worrying about the cost of these extra communications has to realize that the two pairs of communications $E1$ and $E2$, and $PC1$ and $PC2$ are each implemented as the two halves of the same communication action.)

Chapter 3

The Object Code, Production Rules

3.1 Introduction

Carrying the discrete model of computation down to the transistor level requires that the MOS transistor be idealized as an on/off switch. Unfortunately, the simple semantics of the switch ignore too many electrical phenomena that play an important role in the functioning of the circuit. A crucial innovation of the method is that the transistor need not be viewed as a discrete switch; voltages can change in a continuous way from one stable level to the other one, provided that the changes are monotonic.

The notation for the object code provides the weakest possible form of control structure and the smallest number of program constructs. In fact, it contains exactly one construct, the *production rule* (PR), and one control structure, the *production rule set.*

We consider the production rule notation to be the canonical representation of a digital circuit. This representation can be decomposed into several equivalent networks of digital operators, depending on the set of building blocks used, or even depending on the technology (e.g., CMOS or GaAs) used, but the production-rule set represents the circuit independently of the chosen physical implementation.

3.1.1 Definitions

Production Rule. *A production rule (PR) is a construct of the form* $G \mapsto S$, *where* S *is either a simple assignment or an unordered list "*$s1$, $s2$, $s3$, ... *" of simple assignments, and* G *is a boolean expression called the guard of the*

PR.

Example:

$$x \wedge y \mapsto z\uparrow$$
$$\neg x \mapsto u\uparrow, v\downarrow$$

The semantics of a PR are defined only if the PR is *stable*:

Stability. *A PR $G \mapsto S$ is said to be stable in a given computation, if, at any point of the computation, G either is* **false** *or remains invariantly* **true** *until the completion of S.*

Stability is not guaranteed by the implementation. It has to be enforced by the compilation procedure.

Execution of a PR. *An execution of the stable PR $G \mapsto S$ is an unbounded sequence of firings. A firing of $G \mapsto S$ with G* **true** *amounts to the execution of S. A firing of $G \mapsto S$ with G* **false** *amounts to a* **skip**.

If S is a list of several simple assignments, the execution of S is the concurrent execution of all asssignments of the list.

Production Rule Set. *A PR set is the concurrent composition of all PRs of the set.*

For example, a directed wire with input x and output y is represented by, or, perhaps more precisely, is the implementation of the production rule set

$$x \mapsto y\uparrow$$
$$\neg x \mapsto y\downarrow$$

The only composition operation on two PR sets is the set union.

Theorem. *The implementation of two concurrent processes is the set union of the two PR sets implementing the processes and of the PR sets implementing the channels between the processes, if any.*

The proof follows from the associativity of the concurrent composition operator. The other operations on the PRs of a set are those allowed by the following properties:

- Multiple occurrences of the same PR are equivalent to one as a consequence of the idempotence of the concurrent composition.
- The two rules $G \mapsto S1$ and $G \mapsto S2$ are equivalent to the single rule $G \mapsto S1, S2$.
- The two rules $G1 \mapsto S$ and $G2 \mapsto S$ are equivalent to the single rule $G1 \vee G2 \mapsto S$.

PRs are *complementary* when they are of the type $G1 \mapsto x\uparrow$ and $G2 \mapsto x\downarrow$. We require that complementary PRs be *non-interfering.*

Non-Interference. *Two complementary PRs are non-interfering when* $\neg G1 \vee \neg G2$ *holds invariantly.*

It can be proven that, under the stability of each PR and non-interference among complementary PRs, the concurrent execution of the PRs of a set is equivalent to the following sequential execution:

$$*[\textit{select a PR with a true guard; fire the PR}]$$

where the selection is weakly fair (each PR is selected infinitely often). From now on, we ignore the firings of a PR with a **false** guard; a firing will mean a firing of a PR with a **true** guard.

Hence, any valid execution of a production-rule set in which non-interference and stability are fulfilled is equivalent to a non-deterministic sequential execution of the production-rule set. This equivalence facilitates the analysis of production-rule sets.

Until we return to these issues, we shall assume that the stability and non-interference requirements are fulfilled.

3.2 VLSI implementation of PRs

Stability and non-interference are the two properties that make the VLSI implementation of PRs (almost) straightforward. As an example, we describe a simple implementation of PRs in CMOS technology.

3.2.1 The CMOS transistors

A CMOS circuit is a network of "nodes"—variables—interconnected by transistors. Certain nodes are also connected to the input-output "pads", which provide the interface with the environment—we will ignore the pads in this presentation. Other nodes are directly connected to the *power* node, providing the constant high-voltage value—called *VDD*—which represents the logical constant **true** or 1. Yet other nodes are directly connected to the *ground* node—called *GND*—providing the constant low-voltage value which represents the logical constant **false** or 0.

A node takes the continuous range of voltage values between the high voltage and the low voltage. Above a certain voltage $v1$ the value is interpreted as 1. Below another voltage $v0$, the value is interpreted as 0. Thanks to the stability property, the precise values of $v1$ and $v0$, which vary from node to node, are irrelevant provided that $v0 < v1$ and the voltage changes are *monotonic.*

(Strict monotonicity is not necessary, and is actually impossible to achieve because of noise, but we will not enter into these details here.)

A CMOS transistor is either of n-type or p-type. A transistor relates three nodes in the following way. Let g, standing for "gate", and x and y be the three nodes. When g is **false** for an n-transistor, and **true** for a p-transistor, no current passes through the the region between x and y, called the *channel*[1]; thus x and y are left unchanged. When g is set to **true** for an n-transistor, or **false** for a p-transistor, the channel becomes conducting. In this case, x and y either have the same voltages and are left unchanged, or a current is established in the channel until x and y reach the same voltage. The common value reached by x and y depends on electrical properties of x and y that are determined by the physical sizes (capacitances) of the nodes implementing x and y and by their interactions with the rest of the circuit. (Differences in node capacitances may cause charges to flow through the channel of a transistor in a way that results in unintended values of the nodes. This phenomenon, called *charge sharing*, may make it quite difficult to predict the final voltage value reached by x and y.)

In order to define the net-effect of a PR independently of the physical parameters of its implementation, we are going to restrict the use of transistors. (In particular, the restriction will eliminate most occurrences of charge sharing.)

We impose the condition that a transistor used in isolation connect only two variables of the circuit: the gate g and one of the other two nodes, say z. The third node of the transistor is either the power or the ground. With this restriction, the behavior of a single n-transistor is

$$g \mapsto z\uparrow \quad \text{or} \quad g \mapsto z\downarrow .$$

The behavior of a single p-transistor is

$$\neg g \mapsto z\uparrow \quad \text{or} \quad \neg g \mapsto z\downarrow .$$

3.2.2 Threshold voltages

The current in the channel of a transistor is a function of the so-called gate-to-source voltage, V_{gs}, defined as $V(g) - min(V(x), V(y))$ for an n-transistor, and as $V(g) - max(V(x), V(y))$ for a p-transistor. In first approximation, the current is assumed to be zero when

$$V_{gs} \leq V_{tn}$$

for an n-transistor, and

$$V_{gs} \geq V_{tp}$$

[1] This notion of channel is unrelated to the one we introduced for communication among processes.

for a p-transistor. V_{tn} and V_{tp} are called the *threshold voltages.* (Typically, $V_{tn} \approx 1V$ and $V_{tp} \approx -1V$.)

Because of the existence of threshold voltages, if an n-transistor is used to implement $g \mapsto z\uparrow$, the final value of z is not a "strong" 1, since the channel will stop conducting as soon as the voltage of z is within V_{tn} of the gate voltage. And symmetrically, a p-transistor used to implement $\neg g \mapsto z\downarrow$ does not produce a "strong" zero as final value of z. Since the voltage drops caused by the threshold voltages accumulate as we compose operators, it is important to produce strong signals in order to be able to compose an arbitrary number of operators. We shall therefore restrict our use of n-transistors to PRs of the form

$$g \mapsto z\downarrow \tag{3.1}$$

and p-transistors to production rules of the form

$$\neg g \mapsto z\uparrow . \tag{3.2}$$

With these restrictions, all implementations produce strong signals.

Threshold voltages are difficult to adjust in CMOS technology. Actually, they tend to become more variable as the feature size decreases. (They may also vary during the activity of the circuit because of some electrical interaction with the substrate, called *body effect.*) For constant node capacitance, variations in thresholds are accountable for most of the discrepancies in propagation delays on a CMOS chip. In particular, these variations exclude the possibility that the ordering in space of a set of variables along a common wire be used to infer an ordering in time of a set of transitions of these variables.

3.2.3 Switching circuits

Consider the canonical (stable) PR

$$b \mapsto z\downarrow$$

where b is a boolean expression in terms of a set of variables. These variables are used as gates of transistors implementing a switching circuit s corresponding to b: s is a series-parallel switching circuit between the ground node (also called *GND*) and z. *GND* has the constant value **false** . The other constant node, the power-node *VDD*, has the constant value **true** .

The switches are n-transistors whose gates are the variables of b, possibly negated. Furthermore, we have:

$$b \equiv \text{"}\textit{there is a path from ground to } z \textit{ in } s\text{"}$$

By construction of s, if b holds and remains stable, z is eventually set to **false** . (For this reason, s is called a *pull-down circuit.*) Hence, s is exactly the implementation of the production rule $b \mapsto z\downarrow$.

Using a symmetrical argument, we can show that the same series-parallel circuit as s, but with *VDD* and z connected, and whose switches are p-transistors, implements the production-rule:

$$bneg \mapsto z\uparrow ,$$

where *bneg* is derived from b by negating all variables. (This circuit is called a *pull-up circuit.*)

3.3 Operators

The two PRs that set and reset the same variable, like

$$\begin{array}{l} b1 \mapsto z\uparrow \\ b2 \mapsto z\downarrow , \end{array} \tag{3.3}$$

are implemented as one operator.

Let $s1$ be the pull-up circuit corresponding to $b1$, and let $s2$ be the pull-down circuit corresponding to $b2$. The two circuits are connected through the common node z. Since non-interference has been enforced, $\neg b1 \vee \neg b2$ holds at any time. This guarantees the absence of a conducting path between power and ground when the operator is not firing. (A path may exist for a short time when the operator is firing.)

Definition. *The operator implementing the two rules is called "combinational" if* $b1 \vee b2$ *holds at any time, and "state-holding" otherwise.*

By definition, if the operator is combinational, there is always a conducting path between either *VDD* or *GND* and the output z. Hence, the value of the output is always a strong **false** value or a strong **true** value, and therefore the circuit corresponding to the composition of $s1$ and $s2$ is a valid implementation of the operator.

For example, PRs 3.1 and 3.2 together implement an inverter. The circuit of Figure 3.2 implements the *nand*-operator defined by the PRs

$$\begin{array}{r} a \wedge b \mapsto z\downarrow \\ \neg a \vee \neg b \mapsto z\uparrow \end{array}$$

If 3.3 is a state-holding operator, $\neg b1 \wedge \neg b2$ may hold in a certain state. In such a state, node z is isolated; there is no path between z and either *VDD* or *GND*. In MOS technology, an isolated node does not retain its value forever; eventually the charges leak away through the substrate and also through the

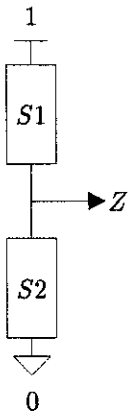

Figure 3.1: CMOS implementation of a combinational operator

transistors of the pull-up and pull-down circuits. If the PRs of the operator are fired frequently enough to prevent leakage, the implementation of Figure 3.1 can be used for a state-holding operator. Such an implementation is called *dynamic*.

Otherwise, it is necessary to add a storage element to the output node of a state-holding operator. Such an implementation is called *static*. In the sequel, we assume that only static implementations are used for state-holding operators.

A standard CMOS implementation of such a storage element consists of two cross-coupled inverters (see Figure 3.3). This implementation inverts the value of z.

The "weak" inverter, marked with a letter w on the figure, connects z to either *VDD* or *GND* through a high resistance, so as to maintain z at its intended voltage value [22].

The implementation of a static state-holding operator is slightly more costly than that of a combinational operator because of the need for a storage device. Hence, given a pair of PRs that are not combinational, we may first try to modify the guards—under the invariance of the semantics—so as to make them combinational.

3.3.1 The Standard Operators

All operators of one or two inputs are used, and are therefore viewed as the standard operators.

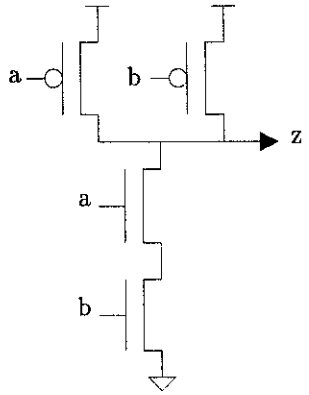

Figure 3.2: CMOS implementation of a NAND-operator

One-Input Operators

The two operators with one input and one output are the *wire*:

$$x \underline{w} y \equiv \begin{array}{l} x \mapsto y\uparrow \\ \neg x \mapsto y\downarrow , \end{array}$$

and the *inverter*:

$$\neg x \underline{w} y \equiv \begin{array}{l} \neg x \mapsto y\uparrow \\ x \mapsto y\downarrow . \end{array}$$

Most operators we use have more inputs than outputs. But, in general, the components we design have as many outputs as inputs. Hence, we need to reset the balance by introducing at least one operator, the *fork*, with more outputs than inputs. A fork with two outputs is defined as:

$$x \underline{f} (y,z) \equiv \begin{array}{l} x \mapsto y\uparrow, z\uparrow \\ \neg x \mapsto y\downarrow, z\downarrow . \end{array}$$

The wire and the fork are the only two operators that are not implemented as a pull-up/pull-down circuit—called a *restoring* circuit—but as a simple conducting interconnection between input and outputs.

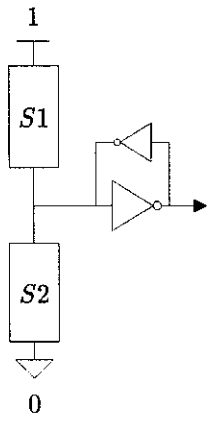

Figure 3.3: A static implementation of a state-holding operator

The Wire as a Renaming Operator

Because the implementation of a wire is the same as that of a node, the wire behaves as a renaming operator when composed with another operator: The composition of an arbitrary operator O with output variable x with the wire $x \underline{w} y$ is equivalent to O in which x is renamed y. The composition of operator O with input variable x with the wire $y \underline{w} x$ is equivalent to O in which x is renamed y. (Observe that O can even be a wire.)

Unfortunately, the fork is not a renaming operator since the concurrent assignments to the different outputs of the fork are not completed simultaneously. In order to use a fork as a renaming operator, we will later have to make the timing assumption that such a fork is *isochronic*.

Combinational Operators with Two Inputs

We construct all functions B of two variables x and y such that

$$\begin{aligned} B &\mapsto z\uparrow \\ \neg B &\mapsto z\downarrow \ . \end{aligned}$$

We get for B: $x \wedge y$, $x \vee y$, and $x = y$. We will not list the functions obtained by inverting inputs of B. (On the figures, a negated input or output is represented by a small circle on the corresponding line.) This gives the following set.
The *and*, with the infix notation $(x, y) \underline{\wedge} z$, is defined as:

$$\begin{aligned} x \wedge y &\mapsto z\uparrow \\ \neg x \vee \neg y &\mapsto z\downarrow . \end{aligned}$$

The *or*, with the infix notation $(x, y) \underline{\vee} z$, is defined as:

$$\begin{aligned} x \vee y &\mapsto z\uparrow \\ \neg x \wedge \neg y &\mapsto z\downarrow . \end{aligned}$$

The *equality*, with the infix notation $(x, y)\ \underline{eq}\ z$, is defined as:

$$\begin{aligned} x = y &\mapsto z\uparrow \\ x \neq y &\mapsto z\downarrow . \end{aligned}$$

State-Holding Operators with Two Inputs

Next, we construct all different two-input-one-output operators of the form

$$\begin{aligned} b1 &\mapsto z\uparrow \\ b2 &\mapsto z\downarrow \end{aligned}$$

such that $\neg b1 \vee \neg b2$ holds at any time, but $b1 \neq \neg b2$. We select for $b1$ either $x \wedge y$, or $x \vee y$, or $x = y$. For each choice of $b1$, we construct $b2$ as any of the effective strengthenings of $\neg b1$.

For $b1 \equiv (x \wedge y)$, we get for $b2$: $\neg x \wedge \neg y$, $\neg x \wedge y$, $\neg x$, and $x \neq y$. The first three choices of $b2$ lead to the following state-holding operators:
The *C-element*

$$\begin{aligned} (x, y)\ \underline{C}\ z \equiv \quad & x \wedge y \mapsto z\uparrow \\ & \neg x \wedge \neg y \mapsto z\downarrow . \end{aligned}$$

(The C-element was introduced by David Muller, and described in [17].)
The *switch*

$$\begin{aligned} (x, y)\ \underline{sw}\ z \equiv \quad & x \wedge y \mapsto z\uparrow \\ & \neg x \wedge y \mapsto z\downarrow . \end{aligned}$$

The *asymmetric C-element*

$$\begin{aligned} (x, y)\ \underline{aC}\ z \equiv \quad & x \wedge y \mapsto z\uparrow \\ & \neg x \mapsto z\downarrow . \end{aligned}$$

For $b2 \equiv (x \neq y)$, we get the operator

$$\begin{aligned} x \wedge y &\mapsto z\uparrow \\ x \neq y &\mapsto z\downarrow . \end{aligned}$$

But, if the stability condition is fulfilled, this operator is not state-holding. Because of the stability requirement, the state in which $\neg x \wedge \neg y$ holds—the "storage state"—can only be reached from states $x \wedge \neg y$ and $\neg x \wedge y$. In both states, $\neg z$ holds, and, therefore, $\neg z$ holds in the storage state. Hence, we can weaken the guard of the second PR as $(x \neq y) \vee (\neg x \wedge \neg y)$, i.e., $\neg x \vee \neg y$. Hence, the operator is equivalent to the *and*-operator $(x, y) \underline{\wedge} z$.

For $b1 \equiv (x \vee y)$, no effective strengthening of $\neg b1$ is possible.

For $b1 \equiv (x = y)$, we get the operator:

$$\begin{aligned} x = y &\mapsto z\uparrow \\ x \wedge \neg y &\mapsto z\downarrow \ . \end{aligned}$$

But if the stability condition is fulfilled, this operator is not state-holding for the same reasons that the operator with $b1 \equiv x \wedge y$ and $b2 \equiv (x \neq y)$ is not.

Flip-Flop

The canonical form we choose for the *flip-flop* is :

$$(x, y)\ \underline{ff}\ z \equiv \begin{array}{l} x \mapsto z\uparrow \\ \neg y \mapsto z\downarrow , \end{array}$$

which requires the invariance of $\neg x \vee y$ to satisfy non-interference. Observe that the flip-flop $(x, y)\underline{ff}\, z$ can always be replaced with the C-element $(x, y)\underline{C}z$ but not vice versa.

3.3.2 Multi-Input Operators

We use n-input *and*, *or*, *C-element*, whose definitions are straightforward. We use a *multi-input flip-flop* defined as:

$$(x_1, \ldots, x_k, y_1, \ldots, y_l)\ \underline{mff}\ z \equiv \begin{array}{l} \bigvee i : x_i \mapsto z\uparrow \\ \bigvee i : \neg y_i \mapsto z\downarrow \end{array}$$

where $(\forall i : \neg x_i) \vee \ (\forall i : y_i)$.

We also use the combinational *if*-operator—sometimes called *multiplexer*—defined as:

$$(x, y, z)\ \underline{if}\ u \equiv \begin{array}{l} (x \wedge y) \vee (\neg x \wedge z) \mapsto u\uparrow \\ (x \wedge \neg y) \vee (\neg x \wedge \neg z) \mapsto u\downarrow \ . \end{array}$$

The most general and most often used operator is the *generalized C-element*, of which all other forms of C-elements are a special case. It implements a pair of PRs

$$\begin{aligned} B1 &\mapsto x\uparrow \\ B2 &\mapsto x\downarrow \end{aligned}$$

in which $B1$ and $B2$ are arbitrary conjunctions of elementary terms. (As usual, the two guards have to be mutually exclusive.) For example:

$$\begin{aligned} a \wedge b \wedge \neg c &\mapsto x\uparrow \\ \neg a \wedge d &\mapsto x\downarrow \end{aligned}$$

can be directly implemented with a generalized C-element. Observe that the limiting factor for the size of the guards is not the number of inputs, but the number of terms in a conjunction.

3.4 Arbiter and Synchronizer

So far, we have considered only PR sets in which all guards are stable and non-interfering. But we shall have to implement sets of guarded commands—selections or repetitions—in which the guards are *not* mutually exclusive, as in the probe selection example. Therefore, we need at least one operator that provides a non-deterministic choice between two **true** guards.

3.4.1 Arbiter

The simplest selection between non-exclusive guards is of the form

$$\begin{aligned} *[\,&[x \rightarrow \cdots \\ &[\!]\, y \rightarrow \cdots \\ &]\,] \end{aligned}$$

where x and y are simple boolean variables, and the two guards are stable. In order to distinguish among the three basic states of the system—i.e., neither x nor y is selected, x is selected, or y is selected—we need to introduce two outputs, say, u and v, as follows:

$$\begin{aligned} *[\,&[x \rightarrow u\uparrow;\ \cdots \\ &[\!]\, y \rightarrow v\uparrow;\ \cdots \\ &]\,]\,. \end{aligned}$$

Initially, $\neg u \wedge \neg v$ holds as coding of the state "no selection made". Hence, when the selection is considered completed, which is just a matter of definition, u and v should be set back to **false**. We get

$$\begin{aligned} *[\,&[x \rightarrow u\uparrow;\ [\neg x];\ u\downarrow \\ &[\!]\, y \rightarrow v\uparrow;\ [\neg y];\ v\downarrow \\ &]\,]\,. \end{aligned} \tag{3.4}$$

If $\neg u \wedge \neg v$ holds initially, $\neg u \vee \neg v$ holds at any time.

The above program is a description of the operator known as the "basic arbiter" or "mutual exclusion element," denoted as $(x, y)\ \underline{arb}\ (u, v)$. Observe that the choice between the two guards is not fair.

3.4.2 Synchronizer

When negated probes are used, for instance to implement fairness, we have to implement selection commands with unstable guards . The synchronizer is the only operator that accepts non-stable guards. It is defined as

$$\begin{array}{l} *[\,[b \wedge z \rightarrow u\uparrow;\ [\neg z];\ u\downarrow \\ \ \ [\!]\neg b \wedge z \rightarrow v\uparrow;\ [\neg z];\ v\downarrow \\ \ ]\,] \ . \end{array} \tag{3.5}$$

Variable b may change at any time from **false** to **true** . But both b and z remain **true** until u or v has changed. Hence, the guard $\neg b \wedge z$ is unstable, whereas the guard $b \wedge z$ is stable. As in the arbiter case, if $\neg u \wedge \neg v$ holds initially, $\neg u \vee \neg v$ holds at any time. (The synchronizer operator was introduced in [9].)

3.4.3 Implementation and Metastability

Let us first consider the PR sets for 3.4 and 3.5 that contain unstable rules. The PR set for the "unstable arbiter" is

$$\begin{array}{l} x \wedge \neg v \mapsto u\uparrow \\ y \wedge \neg u \mapsto v\uparrow \\ \neg x \vee v \mapsto u\downarrow \\ \neg y \vee u \mapsto v\downarrow \ . \end{array}$$

The PR set for the "unstable synchronizer" is

$$\begin{array}{r} b \wedge z \wedge \neg v \mapsto u\uparrow \\ \neg b \wedge z \wedge \neg u \mapsto v\uparrow \\ \neg z \vee v \mapsto u\downarrow \\ \neg z \vee u \mapsto v\downarrow \ . \end{array}$$

The first two PRs of the arbiter are unstable and can fire concurrently. The same holds for the first two production rules of the synchronizer: since b can change from **false** to **true** at any time, both guards may evaluate to **true** .

Let us analyze the PR set implementation of the arbiter. The synchronizer case is very similar. The state $x \wedge y \wedge (u = v)$ of the arbiter is called *metastable.* When started in the metastable state, with $\neg u \wedge \neg v$, the set of PRs specifying the arbiter may produce the unbounded sequence of firings:

$$*[(u\uparrow, v\uparrow); (u\downarrow, v\downarrow)]$$

In the implementation, nodes u and v may stabilize to a common intermediate voltage value for an unbounded period of time. Eventually, the inherent asymmetry of the physical realization (impurities, fabrication flaws, thermal

noise, etc.) will force the system into one of the two stable states where $u \neq v$. But there is no upper bound on the time the metastable state will last, which means that it is impossible to include an arbitration device into a clocked system with absolute certainty that a timing failure cannot occur.

The spurious values of u and v produced during the metastable state must be eliminated since they are not stable and violate the requirement $\neg u \vee \neg v$. Hence, we compose the "bare" arbiter with a "filter" taking u and v as input and producing uf and vf as "filtered outputs". The net-effect of the filter is:

$$uf, vf := (u \wedge \neg v), (v \wedge \neg u)$$

(In the CMOS construction of the filter shown in Figure 5, we use the threshold voltages to our advantage: The channel of transistor $t1$ is conducting only when $(u \wedge \neg v)$ holds, and the channel of transistor $t2$ is conducting only when $(v \wedge \neg u)$ holds.)

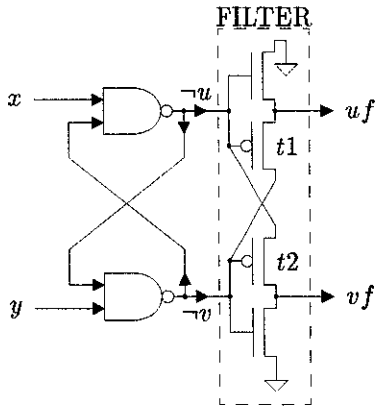

Figure 3.4: An implementation of the basic arbiter

In delay-insensitive design, the correct functioning of a circuit containing an arbiter or a synchronizer is independent of the duration of the metastable state; therefore, relatively simple implementations of arbiters and synchronizers can be used. In synchronous design, however, the implementations have

to meet the additional constraint that the probability of the metastable state lasting longer than the clock period should be negligible.

Chapter 4

The Compilation Method

4.1 Introduction

This chapter briefly introduces the main steps of the compilation procedure: process decomposition, handshaking expansion, and production rule expansion.

4.2 Process Decomposition

The first step of the compilation, called *process decomposition,* consists in replacing one process with several processes by application of the following **Decomposition rule**: *A process, P, containing an arbitrary program part, S, is semantically equivalent to two processes, $P1$ and $P2$, where $P1$ is derived from P by replacing S with a communication action, C, on the newly introduced channel (C, D) between $P1$ and $P2$, and $P2$ is the process* $*[[\overline{D} \rightarrow S; D]]$.

The structure of $P2$ will be used so frequently that we introduce an operator to denote it: the *call* operator. We denote it by (D/S), and we say that D *calls* (or *activates*) S. (We will later generalize the implementation of the call operator so that the implementation mentioned above in the defintion of the decomposition rule is just a particular case of eth general implementation.)

Observe that process decomposition does not introduce concurrency. Although $P1$ and $P2$ are potentially concurrent, they are never active concurrently; $P2$ is activated from $P1$, much as a procedure or a coroutine would be. The newly created subprocesses may share variables; but, since the subprocesses are never active concurrently, there is no conflicting access to the shared variables. The subprocesses may also share channels; this will require a special implementation for such channels. Decomposition is applied for each

construct of the language. For construct S, the corresponding process $P2$ can be simplified as follows:

- If S is the selection $[B_1 \rightarrow S_1 [] B_2 \rightarrow S_2]$, $P2$ is simplified as

$$\begin{array}{l} *[\,[\overline{D} \wedge B_1 \rightarrow S_1; D \\ \quad [] \overline{D} \wedge B_2 \rightarrow S_2; D \\ \,]\,] \,. \end{array} \tag{4.1}$$

- If S is the repetition $*[B_1 \rightarrow S_1 [] B_2 \rightarrow S_2]$, $P2$ is simplified as

$$\begin{array}{l} *[\,[\overline{D} \wedge B_1 \rightarrow S_1 \\ \quad [] \overline{D} \wedge B_2 \rightarrow S_2 \\ \quad [] \overline{D} \wedge \neg B_1 \wedge \neg B_2 \rightarrow D \\ \,]\,] \,. \end{array} \tag{4.2}$$

- The assignment $x := B$, where B is an arbitrary boolean expression, is implemented as the selection $[B \rightarrow x\uparrow [] \neg B \rightarrow x\downarrow]$, which gives for $P2$

$$\begin{array}{l} *[\,[\overline{D} \wedge B \rightarrow x\uparrow; D \\ \quad [] \overline{D} \wedge \neg B \rightarrow x\downarrow; D \\ \,]\,] \,. \end{array} \tag{4.3}$$

The generalizations to the cases of an arbitrary number of guarded commands in selection and repetition are obvious. All assignments to the same variable are also grouped in the same process. Process decomposition is applied repeatedly until the right-hand side of each guarded command is a straight-line program.

Process decomposition makes it possible to reduce a process with an arbitrary control structure to a set of subprocesses of only two different types: either a (finite or infinite) sequence of communication actions, or a repetition of type 4.1, 4.2, or 4.3.

4.3 Handshaking Expansion

The next step of the transformation, the *handshaking expansion,* replaces each communication action in a program with its implementation in terms of elementary actions, and each channel with a pair of wire-operators. We shall first ignore the issue of message transmission and implement only the synchronization property of communication primitives.

Channel (X, Y) is implemented by the two wires (xo $\underline{w}$ yi) and (yo $\underline{w}$ xi). If X belongs to process $P1$ and Y to process $P2$, then xo and xi belong to $P1$, and yo and yi to $P2$. Initially, xo, xi, yo, and yi—which we will call the "handshaking variables of (X, Y)"—are **false**. Assume that the program has

been proven to be deadlock-free and that we can identify a pair of matching actions X and Y in $P1$ and $P2$, respectively. We replace X and Y by the sequences U_x and U_y, respectively, with:

$$\begin{array}{l} U_x \equiv xo\uparrow;\ [xi] \\ U_y \equiv [y_i];\ yo\uparrow\ . \end{array} \tag{4.4}$$

Also:

$$\begin{array}{r} xo \mapsto yi\uparrow \\ \neg xo \mapsto yi\downarrow \\ yo \mapsto xi\uparrow \\ \neg yo \mapsto xi\downarrow, \end{array} \tag{4.5}$$

by definition of the wires. By 4.4 and 4.5, any concurrent execution of $P1$ and $P2$ contains the sequence of assignments:

$$xo\uparrow;\ yi\uparrow;\ yo\uparrow;\ xi\uparrow\ .$$

4.3.1 Simultaneous Completion of Non-Atomic Actions

We introduce a definition of *completion* of a non-atomic action which makes it possible to use the notion of simultaneous completion of two non-atomic actions.

By definition, the execution of an atomic action is considered instantaneous, and thus the simultaneous completion of two atomic actions does not make sense. (Atomic actions are simple assignments $x\uparrow$ and $x\downarrow$, and evaluation of simple guards, i.e., guards containing one variable. A wait action of the form $[ai]$ is a non-atomic action that may be treated as the repetition $*[\neg ai \rightarrow skip]$.)

A non-atomic action is *initiated* when its first atomic action is executed. A non-atomic action is *terminated* when its last atomic action is executed.

For non-atomic actions, the notion of completion does not coincide with that of termination. A non-atomic action might be considered completed even if it has not terminated, i.e., even if some atomic actions that are part of the action have not been executed. The definition of suspension is derived from that of completion.

Definition. *A non-atomic action X is completed when it is initiated and it is guaranteed to terminate, i.e., when all possible continuations of the computation contain the complete sequence of atomic actions of X.*

The above definition can be further explained as follows: Consider a prefix $t1$ of an arbitrary *trace* of a computation. (A trace is a sequence of atomic actions corresponding to a possible execution of the program.) The completion of X is identified with the point in the computation where $t1$ has been

completed, if 1) X is initiated in $t1$, and 2) all possible sequences $t2$, such that $t1$ extended with $t2$ is a valid trace of the computation, contain the remaining atomic actions of X. *Hence, the completions of two non-atomic actions coincide if their completion points coincide.*

(Observe that there may be several points in a trace that can act as completion point, which makes it easier to align the two completion points of two overlapping sequences so as to implement the bullet operator.)

Definition. *Between initiation and completion, an action is suspended.*

These definitions of completion and suspension are valid because they satisfy the three semantic properties of completion and suspension that are used in correctness arguments, namely:

- $\{\mathbf{c}X = x\}\ X\ \{\mathbf{c}X = x + 1\}$,
- $\mathbf{q}X \Rightarrow pre(X)$, where $pre(X)$ is any precondition of X in terms of the program variables and auxiliary program variables,
- If X is completed, eventually X is terminated.

These definitions will be used to implement the bullet operator and the communication primitives as defined by axioms $A1$ and $A2$. Consider the interleaving of U_x and U_y. At the first semicolon, i.e., after $xo\uparrow$, U_x has been initiated, but cannot be considered completed since the valid continuation that does not contain U_y does not contain the rest of U_x. At the second semicolon, both U_x and U_y have been initiated, and thus, all continuations contain the rest of the interleaving of U_x and U_y. Hence, U_x and U_y are guaranteed to terminate when they are both initiated, i.e., they fulfil $A1$ and $A2$.

4.3.2 Four-phase Handshaking

Unfortunately, when the communication implemented by U_x and U_y terminates, all handshaking variables are **true**. Hence, we cannot implement the next communication on channel (X, Y) with U_x and U_y. However, the complementary implementation can be used for the next matching pair, namely:

$$D_x \equiv xo\downarrow;\ [\neg xi]$$
$$D_y \equiv [\neg yi];\ yo\downarrow\ .$$

The solution consisting in alternating U_x and D_x as an implementation of X, and U_y and D_y as an implementation of Y, is called *two-phase handshaking*, or *two-cycle signaling*. Since it is in most cases impossible to determine syntactically which X- or Y-actions follow each other in an execution, the general

two-phase handshaking implementations require testing the current value of the variables as follows:

$$xo := \neg xo;\ [xi = xo]$$
$$[yi \neq yo];\ yo := \neg yo\ .$$

In general, we prefer to use a simpler solution, known as *four-phase handshaking*, or *four-cycle signaling*. In a four-phase handshaking protocol, X-actions are implemented as "$U_x; D_x$" and Y-actions as "$U_y; D_y$". Observe that the D-parts in X and Y introduce an extra communication between the two processes whose only purpose is to reset all variables to **false**.

Both protocols have the property that for a matching pair (X, Y) of actions, the implementation is not symmetrical in X and Y. One action is called *active* and the other one *passive*. The four-phase implementation, with X active and Y passive, is:

$$X \equiv xo\uparrow;\ [xi];\ xo\downarrow;\ [\neg xi] \tag{4.6}$$

$$Y \equiv [yi];\ yo\uparrow;\ [\neg yi];\ yo\downarrow\ . \tag{4.7}$$

(We will introduce an alternative form of active implementation, called *lazy active*.) Although four-phase handshaking contains twice as many actions as two-phase handshaking, the actions involved are simpler and are more amenable to the algebraic manipulations we shall introduce later. When operator delays dominate the communication costs, which is the case for communication inside a chip, four-phase handshaking will, in general, lead to more efficient solutions. When transmission delays dominate the communication costs, which is the case for communication between chips, two-phase is preferred.

4.3.3 Probe

A simple implementation of the probe $\overline{X}$ is xi, with X implemented as passive. (Given our definition of suspension, the proof that this implementation of the probe fulfills its definition is straightforward.)

A probed communication action $\overline{X} \rightarrow \ldots X$ is then implemented as

$$xi \rightarrow \ldots xo\uparrow;\ [\neg xi];\ xo\downarrow\ .$$

4.3.4 Choice of Active or Passive Implementation

When no action of a matching pair is probed, the choice of which action should be active and which passive is arbitrary, but a choice has to be made. The choice can be important for the composition of identical circuits. A simple rule is that, for a given channel (X, Y), all actions on one port (called the

active port) are active, and all actions on the other port (called the *passive port*) are passive. If $\overline{X}$ is used, all X-actions are passive—with the obvious restriction that $\overline{Y}$ cannot be used in the same program.

However, we shall see that this criterion for choosing active and passive ports may conflict with another criterion related to the implementation of input and output commands.

4.3.5 Reshuffling

In 4.6 and 4.7, D_x and D_y are used only to reset all variables to **false**. Hence, provided that the cyclic order of the actions of 4.6 and 4.7 is maintained, the sequences D_x and D_y can be inserted at any place in the program of each of the processes without invalidating the semantics of the communication involved. This transformation, called *reshuffling*, may introduce a deadlock.

Reshuffling, which is the source of significant optimizations, will be used extensively. It is therefore important to know when it can be applied without introducing deadlock.

There are two simple cases where the reshuffling of sequence "$U_x; D_x; S$" into sequence "$U_x; S; D_x$" does not introduce deadlock:

- S contains no communication action, or
- X is an internal channel introduced by process decomposition.

4.3.6 Lazy-active protocol

Consider the active implementation of communication command X:

$$xo\uparrow;\ [xi];\ xo\downarrow;\ [\neg xi]\ .$$

We introduce an alternative protocol, called *lazy active*:

$$[\neg xi];\ xo\uparrow;\ [xi];\ xo\downarrow\ .$$

The lazy active protocol is derived from the active one by postponing wait action $[\neg xi]$ until the next communication on X, and by adding a vacuous wait action $[\neg xi]$ at the beginning of the first communication X. Hence, the lazy active protocol is a correct implementation when combined with a passive protocol.

The lazy active protocol is not identical to a passive protocol in which the input variable is replaced with its negation. In a passive protocol, the effective part (the upgoing part) of the protocol is $[xi];\ xo\uparrow$. In a lazy active one, the effective part is $xo\uparrow;\ [xi]$.

4.4 Production-rule Expansion

Production-rule expansion is the transformation from a handshaking expansion to a set of PRs. It is the most crucial and most difficult step of the compilation since it requires enforcing sequencing by semantic means. It consists of three steps: state assignment, guard strengthening, symmetrization.

Consider the handshaking expansion

$$S \equiv *[[w_0];\ t_0;\ [w_1];\ t_1; \ldots ; [w_{n-1}];\ t_{n-1}] \ .$$

The wait-conditions are boolean expressions, possibly identical to **true** , and the t_i are simple assignments. The extension to the case of multiple assignments between the wait-conditions is straightforward.

The production rule expansion of S is the transformation of S into a semantically equivalent set of production rules. Let

$$P \equiv \{b_i \mapsto t_i | 0 \leq i < n\}$$

be such a set.

4.4.1 Notations and Definitions

For an arbitrary PR p, $p.g$ and $p.a$ denote the guard and the assignment of p, respectively. The predicate $R(a)$, the *result* of the simple assignment a, is defined as: $R(x\uparrow) = x$, and $R(x\downarrow) = \neg x$. An execution of a PR that changes the value of the assigned variable is called *effective*, otherwise it is called *vacuous*.

With these definitions, the stability of a PR can be reformulated as follows:

Stability. *A PR p is stable in a computation if and only if $p.g$ can be falsified only in states where $R(p.a)$ holds. As a consequence, $p.g$ holds as a postcondition of any effective firing of p.*

The production-rule expansion algorithm compiles a handshaking expansion S into a set P of PRs, all of which are stable, with the exception of those whose guards contain negated probes. Since, as we shall see, the guards of the PRs are obtained by strengthening the wait-conditions of S, the stability of the wait-conditions is necessary to satisfy the stability of the PRs.

A wait-condition w is stable if once w is **true** , it remains **true** at least until the completion of the following assignment. Unstable wait-conditions can be caused by negated probes or unrestricted shared variables. The case of negated probes will be dealt with separately by introducing synchronizers. We ignore the use of shared variables in these lecture notes.

In particular, the wait-conditions of the handsaking expansions are stable, also after reshuffling.

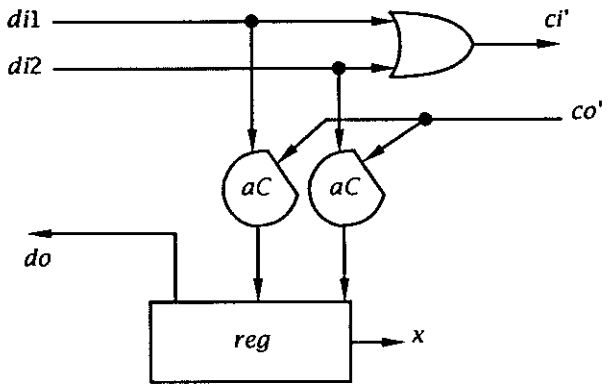

Figure 7.6: Input actions on passive port

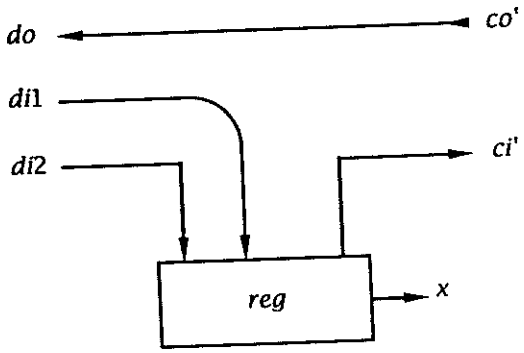

Figure 7.7: Input actions on active port

4.4.2 Sequencing

The set P of PRs implements S when the following conditions are fulfilled:
Guard strengthening The guards of the PRs of P are obtained by strengthening the wait conditions of S: $\forall i :: b_i \Rightarrow w_i$ and, in the initial state, $w_0 \Rightarrow b_0$.
Sequential execution $(\mathbf{N}i :: b_i \wedge \neg R(t_i)) \leq 1$, i.e., at most one effective PR can be executed at a time.
Program-order execution For all i: If w_{i+1} holds eventually as a postcondition of t_i in S, then b_{i+1} holds eventually as a postcondition of t_i in P. (Addition $i+1$ is modulo n.)

The first condition establishes that an execution of PR $b_i \mapsto t_i$ in P is equivalent to an execution of $[w_i]; t_i$ in S. The second and third conditions establish that the order of execution of effective PRs of P is the order specified by S, which we have called the *program-order*, and that no deadlock is introduced in the construction of P.

As we shall see, it is not always possible to construct, for a given handshaking expansion, a PR set that satisfies the above three conditions. In certain cases, the handshaking expansion must be augmented with assignments to new variables, called *state variables*. This transformation, which is always possible, will be explained later.

4.4.3 Acknowledgement

Fulfilling the second and third conditions requires that for any two PRs $p : b \mapsto t$ and $p' : b' \mapsto t'$, such that p immediately precedes p' in the program order,

$$b' \Rightarrow R(t)$$

holds as a postcondition of p. We say that b' is the *acknowledgement* of t. Hence the

Acknowledgement Property. *For a PR set executed in program order, the guard of each PR is an acknowledgement of the immediately preceding assignment.*

We shall see that the acknowledgement property is necessary but not sufficient to ensure program-order execution.

We use two kinds of acknowledgements depending on the type of variable used in the assignment. But other forms of acknowledgments can be envisioned. If t assigns an internal variable, then the acknowledgement is implemented by strengthening b' as $b' \wedge R(t)$. For example, if t is $x\uparrow$, the acknowledgement is $b' \wedge x$.

If t assigns a handshake variable, i.e., a variable implementing a communication command, another kind of acknowledgement can be used as follows.

Acknowledgement of Output Variables. *For xo and xi used in an active protocol, xi is an acknowledgment of $xo\uparrow$, $\neg xi$ is an acknowledgment of $xo\downarrow$. For xo and xi used in a lazy-active protocol, xi is an acknowledgment of $xo\uparrow$. For yo and yi used in the passive protocol of 4.7, $\neg yi$ is an acknowledgment of $yo\uparrow$, yi is an acknowledgment of $yo\downarrow$.*

4.4.4 Implementation of Stability

Consider a PR set P, which implements a given program S. We are going to show that the acknowledgement property, which is necessary to construct a P that implements S, is also sufficient to guarantee stability.

The execution of a PR p of P establishes a path between a constant node (either *VDD* or *GND*), and the node implementing the variable—say, x—assigned by p. Either $p.g$ holds forever after p; or the firing of another PR I, the *invalidating* PR of p, will establish $\neg p.g$, hence cutting the path from the constant node to x.

Let $\tilde{p}$ be the complementary PR of p, i.e., the PR with the complementary assignment. If the PR set contains both p and $\tilde{p}$, then it also contains I because of the non-interference requirement between complementary PRs. And we have the order of execution:

$$p \preceq I \prec \tilde{p} \ .$$

In all the states between I and $\tilde{p}$, the original path to x is cut. In that case, we have to see to it that the assignment to x is completed before the path is cut. Hence the

Completion requirement. *Assignment $p.a$ is completed when a PR q is completed whose guard is an acknowledgement of $p.a$. The execution order of the PR set must satisfy*

$$p \prec q \preceq I \ .$$

vSince this requirement is already implied by the acknowledgement property, the construction of P automatically guarantees stability.

We can add an extra requirement to eliminate the pathological cases of "disguised" self-invalidating PRs, even though such cases rarely arise in practice, and they can be dealt with at the implementation level.

4.4.5 Self-Invalidating PRs

Definition. *A PR p is self-invalidating when $R(p.a) \Rightarrow \neg p.g$.*

For example, $\neg x \mapsto x\uparrow$ is self-invalidating.

Self-invalidating PRs are disallowed since they violate the stability requirement. Fortunately, they are excluded by the completion requirement since it implies $I \neq p$.

For instance, the circuit consisting of an inverter with its output connected to its input is excluded by the completion requirement since it corresponds to the PR set:

$$\begin{aligned} \neg x &\mapsto x\uparrow \\ x &\mapsto x\downarrow \end{aligned}$$

and the two PRs of the set are self-invalidating. However, the PR set

$$\begin{aligned} \neg x &\mapsto y\uparrow \\ y &\mapsto x\uparrow \\ x &\mapsto y\downarrow \\ \neg y &\mapsto x\downarrow \end{aligned}$$

fulfils the completion requirement, although it is the same circuit as previously, since the only change is the addition of the wire $y \underline{w} x$.

We eliminate such "disguised" self-invalidating PRs by adding the

Restoring Acknowledgement Requirement. *There is at least one restoring PR r satisfying $p \prec r \preceq I$, where r is restoring if it is not part of a wire or a fork.*

With this extra requirement, all forms of self-invalidating PRs are eliminated.

It is remarkable that the acknowledgement requirement, which is necessary to enforce the sequential execution of a PR set, is also sufficient to satisfy stability. From now on, we can manipulate PRs as if the transitions were discrete. However, we have made no simplifying assumption on the physical behavior of the system. The only physical requirement so far is that of monotonicity.

Another requirement on the implementation is that the rings of operators that constitute a circuit keep oscillating. It turns out that eliminating self-invalidating PRs enforces the condition that a ring contain at least three restoring operators, which is a necessary (and in practice also sufficient) condition for the ring to oscillate, thanks to the "gain" property of restoring gates. (See [15] for an explanation of gain.).

Chapter 5

Production Rule Expansion

5.1 Introduction

In this chapter, we describe the techniques for production rule expansion in more detail. We first deal with the simple case of a straightline program. The general case of a set of guarded commands is introduced in one example. We also introduce the next step of the compilation, called *operator reduction*, which produces a network of cells from production rules.

5.2 Straightline Programs

As a first example, let us implement the simple process (L/R), where R is an active channel. This process is one of the basic building blocks for implementing sequencing. The handshaking expansion gives:

$$*[[li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ lo\uparrow;\ [\neg li];\ lo\downarrow]\ . \tag{5.1}$$

We now consider the handshaking expansion as the specification of the implementation: Any implementation of the program has to satisfy the ordering defined by 5.1. The next step is to construct a production-rule set that satisfies this ordering. We start with the production-rule set that is syntactically derived from 5.1:

$$\begin{aligned} li &\mapsto ro\uparrow \\ ri &\mapsto ro\downarrow \\ \neg ri &\mapsto lo\uparrow \\ \neg li &\mapsto lo\downarrow \end{aligned}$$

(As a clue to the reader, PRs of a set are listed in program order.)

Since the program is deadlock-free, effective execution of the PRs in program order is always possible. However, some other execution orders may also be possible. If execution orders other than the program order are possible for the production-rule set, the guards of some rules are strengthened so as to eliminate these execution orders.

In our example, program order is not the only execution order for the syntactic production-rule set: Since $\neg ri$ holds initially, the third PR can be executed first. This is also true for the fourth PR; but the execution of the fourth rule in the initial state is vacuous. Because all handshaking variables of R are back to **false** when R is completed, we cannot find a guard for the transition $lo\uparrow$ that holds only as a precondition of $lo\uparrow$ in 5.1. Hence, we cannot distinguish the state following R from the state preceding R, and thus the sequential execution condition introduced in section 8 cannot be satisfied.

In order to fulfil the sequential execution condition, we have to guarantee that each state of the handshaking expansion is unique, i.e., there exists a predicate in terms of variables of the program that holds only in this state. The task of transforming the handshaking expansion so as to make each state unique is called *state assignment.*

5.3 State Assignment With State Variables

The first technique to define uniquely the state in which the transition $lo\uparrow$ is to take place consists in introducing a state variable, say x, initially **false.** Handshaking expansion 5.1 becomes

$$*[[li];\ ro\uparrow;\ [ri];\ x\uparrow;\ [x];\ ro\downarrow;\ [\neg ri];\ lo\uparrow;\ [\neg li];\ x\downarrow;\ [\neg x];\ lo\downarrow]\ . \qquad (5.2)$$

Observe that 5.2 is semantically equivalent to 5.1, since the two sequences of actions that are added to 5.1, namely, $x\uparrow;[x]$ and $x\downarrow;[\neg x]$, are equivalent to a **skip**. (The newly introduced variable x is used nowhere else.) There are several places where the two assignments to the state variable can be introduced. We shall not discuss here the different heuristics that are used in the placement of the variables. But it is important to observe that minimizing the number of state variables is not a relevant criterion in the choice of a state assignment. What counts is minimzing the number of transitions on state variables, and the sizes of the production-rules guards.

5.4 The Basic Algorithm For PR expansion

We consider a straightline handshaking expansion, and assume that state assignment has been performed. Hence, each state of the handshaking expan-

sion is unique, and we can therefore generate a PR set that is semantically equivalent to the handshaking expansion.

For the time being, we assume that each assignment to a variable, such as $x\uparrow$ or $x\downarrow$, occurs at most once in the program. This restriction is easily enforced by renaming; in the case of a program

$$p \equiv \ldots x\uparrow;\ \ldots;\ x\downarrow;\ \ldots;\ x\uparrow;\ldots;\ x\downarrow;\ \ldots$$

we can rename the variable as

$$p' \equiv \ldots\ x1\uparrow;\ \ldots;\ x1\downarrow;\ \ldots;\ x2\uparrow;\ \ldots;x2\downarrow;\ \ldots$$

We first perform the handshaking expansion of p'. We then observe that since $\neg x1 \vee \neg x2$ holds at any time, we can combine $x1$ and $x2$ by the two rules:

$$x1 \vee x2 \mapsto x\uparrow$$
$$\neg x1 \wedge \neg x2 \mapsto x\downarrow.$$

If we treat the cases of selection and repetition separately, we do not have disjunctions in wait-actions. Hence, we can construct all production-rule guards as conjunctions; disjunction will be introduced next in the symmetrization step.

5.4.1 First Method: Weakening Strong Guards

Since each state of the handshaking expansion is uniquely defined, the set of production rules in which each guard is the strongest predicate in this state is ordered.

The set of strongest guards is constructed mechanically by determining in each state the value of all variables that are defined in that state: the strongest predicate in that state is the conjunction of all terms that are true in that state.

We can then simplify the guards of the PRs by using program properties of the form "$P \Rightarrow R$ *holds as a precondition of the PR*" to replace $P \wedge R$ by P. (This method has been proposed and used by Huub Schols.)

5.4.2 Second Method: Strengthening Weak Guards

The second method, which we have been using most of the time, starts with the weakest set of guards and strengthens them until the production rule set is ordered.

For each assignment, the initial guard of the production rule is the wait action that precedes it in the handshaking expansion. When the assignment—say, S— is preceded by another assignment, we introduce the net-effect of the

preceding assignment as wait action:

$$x\uparrow; S \text{ is replaced by } x\uparrow; [x]; S$$

$$x\downarrow; S \text{ is replaced by } x\downarrow; [\neg x]; S$$

For each assignment, we define two sets of states:

- the *firing set*, which is the set of all states in which the guard of the assignment holds; and
- the *conflicting set*, which is the set of all states in which the firing of the assignment must be disallowed. For assignment S, let S' be the complementary assignment. The conflicting set is the set of contiguous states starting at the state preceding S' and ending at the state preceding the assignment that preceeds S.

The "window of$S$" is the intersection of the firing set and the conflicting set of S. The window set must be empty ("the window is closed"). If it is not, we shrink the firing set of S (by strengthening the precondition) until the intersection is empty.

Because each state can be uniquely characterized in terms of the program variables, it is always possible to close the window of each assignment by strengthening the guards. There may be several possible ways to strengthen a guard. We choose the one that is the simplest (least number of variables) and that is best suited for symmetrization of the rules, which is explained later.

As an example of the use of the algorithm, we prove a theorem that identifies standard production rules that need not be strengthened. This result significantly reduces the number of cases to be considered.

Theorem 1. *Production rule $xi \mapsto xo\downarrow$ of the active expansion of communication action X and production rule $\neg xi \mapsto xo\downarrow$ of the passive expansion of communication action X are always ordered.*

Proof. The active handshaking expansion of X is

$$xo\uparrow;\ [xi];\ xo\downarrow;\ [\neg xi]$$

For $xi \mapsto xo\downarrow$, the firing set starts at the precondition of $xo\downarrow$ and ends at the postcondition of $xo\downarrow$. The conflicting set starts at the precondition of $xo\uparrow$ and ends at the postcondition of $xo\uparrow$. Observe that even with reshuffling these two sets are disjoint: the window is closed.

The passive handshaking expansion of X is:

$$[xi];\ xo\uparrow;\ [\neg xi];\ xo\downarrow$$

For $\neg xi \mapsto xo\downarrow$, the firing set starts at the precondition of $xo\downarrow$ and ends at any place before $[xi]$. The conflicting set starts at the precondition of $xo\uparrow$ and

ends at the postcondition of $xo\uparrow$. Again, even with reshuffling, the window is always closed. □

A similar theorem holds for standard production rules involving state variables.

Theorem 2. *For state variable u, introduced as follows in the active handshaking expansion of X:*

$$xo\uparrow;\ [xi];\ u\uparrow;\ [u];\ xo\downarrow;\ [\neg xi] \tag{5.3}$$

the production rules $xi \mapsto u\uparrow$ and $u \mapsto xo\downarrow$ are ordered. For state variable u, introduced as follows in the passive handshaking expansion of X:

$$[xi];\ xo\uparrow;\ [xo];\ u\uparrow;\ [u];\ [\neg xi];\ xo\downarrow \tag{5.4}$$

the production rule $xo \mapsto u\uparrow$ is ordered. The same results hold if any of the variables involved is replaced by its complement.

The proof, which is similar to that of Theorem 1, is omitted. The results of Theorem 2 indicate that passive handshaking is more difficult to deal with than active handshaking.

Let us now complete the production-rule expansion of the Q-element. Since x has been introduced to distinguish the prestate of $ro\uparrow$ from the prestate of $lo\uparrow$, we can immediately strengthen the guard of $ro\uparrow$ with $\neg x$ and the guard of $lo\uparrow$ with x. We get:

$$\neg x \wedge li \mapsto ro\uparrow \tag{5.5}$$
$$ri \mapsto x\uparrow \tag{5.6}$$
$$x \mapsto ro\downarrow \tag{5.7}$$
$$x \wedge \neg ri \mapsto lo\uparrow \tag{5.8}$$
$$\neg li \mapsto x\downarrow \tag{5.9}$$
$$\neg x \mapsto lo\downarrow \tag{5.10}$$

It is easy to check in 5.2 that the strenghtenings of the guards of 5.5 and 5.8 close the two windows. We further observe in 5.2 that the introduction of $x\uparrow$ in the handshaking expansion of R, and the introduction of $x\downarrow$ in the handshaking expansion of L both fulfil property 5.3 of Theorem 2. Hence, according to Theorem 2, 5.6, 5.7, 5.9, and 5.10 are ordered, and the above handshaking expansion is program ordered.

5.5 Operator Reduction

The last step of the compilation, called *operator reduction*, groups together the PRs that assign the same variables. Those PRs are then identified with

(and implemented as) an operator. The program is thus identified with a set of operators.

Since we have enforced the stability of each rule and non-interference between any two complementary rules, we can implement any set of PRs directly. (For reasons of efficiency, we have to see to it that the guards do not contain too many variables in a conjunct, which would lead to too many transistors in series. Hence, the implementation of the set may also involve decomposing a PR into several PRs by introducing new internal variables.)

The direct operator implementation of the PR set is straightforward:

The PRs that set and reset ro correspond to the asymmetric C-element $(\neg x, li)$ $\underline{aC}$ ro.

The PRs that set and reset lo correspond to the asymmetric C-element $(x, \neg ri)$ $\underline{aC}$ lo.

The PRs that set and reset x correspond to the flip-flop (ri, li) $\underline{ff}$ x.

If the above operators are implemented as dynamic, this implementation of process (L/R) is the simplest possible. If static implementations of the operators are required, another implementation might be considered with fewer state-holding elements since, as we have explained in the first part, static state-holding operators are slightly more difficult to realize than combinational operators.

A last transformation, called *symmetrization*, may be performed on the PR set to minimize the number of state-holding operators. However, since symmetrization also introduces inefficiencies of its own, it should not be applied blindly.

5.6 Symmetrization

Symmetrization is performed on the two guards of PRs $b1 \mapsto z\uparrow$ and $b2 \mapsto z\downarrow$, when one of the two guards, say, $b1$, is already in the form $x \wedge \neg b2$. If we replace guard $b2$ with $\neg x \vee b2$, then the two guards are complements of each other; i.e., the operator is combinational. Of course, weakening guard $b2$ is a dangerous transformation since it may introduce a new state where the guard holds. We have to check that this does not occur by checking the following invariant:

Given the new rule $\neg x \vee b2 \mapsto z\downarrow$, $\neg z$ must hold in any state where $\neg x \wedge \neg b2$ holds; i.e., we have to check the invariant truth of

$$x \vee b2 \vee \neg z \ .$$

5.6.1 Operator Reduction of the (L/R)-element

The symmetrization of the PRs of the (L/R)-element gives:

$$\begin{aligned} \neg x \wedge li &\mapsto ro\uparrow \\ ri &\mapsto x\uparrow \\ \neg li \vee x &\mapsto ro\downarrow \\ x \wedge \neg ri &\mapsto lo\uparrow \\ \neg li &\mapsto x\downarrow \\ ri \vee \neg x &\mapsto lo\downarrow \end{aligned}$$

The PRs that set and reset ro correspond to the *and*-operator $(\neg x, li)\ \underline{\wedge}\ ro$.

The PRs that set and reset x correspond to the flip-flop $(ri, li)\ \underline{ff}\ x$.

The PRs that set and reset lo correspond to the *and*-operator $(x, \neg ri)\ \underline{\wedge}\ lo$.

The flip-flop can be replaced with the C-element $(li, ri)\ \underline{C}\ x$.

The resulting circuit is shown in Figure 5.1. (The dot identifies the input that is activated first.) This implementation of (L/R), either with a flip-flop or with a C-element, is called a *Q-element*. The Q-element implementing (L/R) as above is described by the infix notation $(li, lo)\ \underline{Q}\ (ri, ro)$.

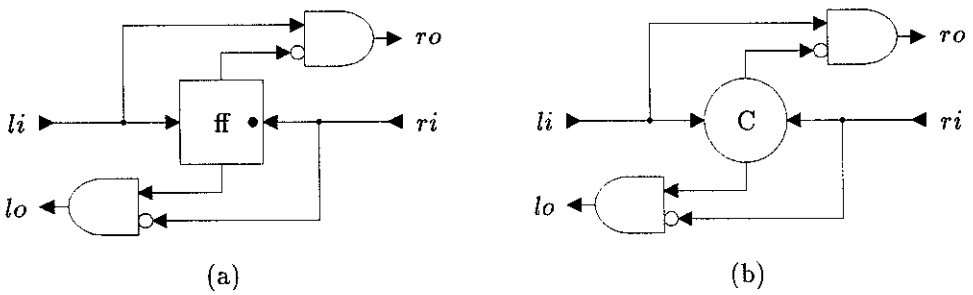

Figure 5.1: Implementation of (L/R) with a Q-element

5.7 Isochronic Forks

In the previous operator reduction, li is an input to the flip-flop $(li, ri)\ \underline{\overline{ff}}\ x$, and to the *and*-operator $(li, \neg x) \underline{\wedge} ro$. Formally, in order to compose the PRs together to form a circuit, we have to introduce the fork $li\underline{f}(l1, l2)$ and replace li by $l1$ as input of the *and*-operator, and by $l2$ as input of the flip-flop. We also have to introduce the forks $ri\ \underline{f}\ (r1, r2)$ and $x\ \underline{f}\ (x1, x2)$ for the same reason.

Let us analyse the effect of the first fork only. The PR set that includes the PRs of the fork is:

$$\begin{aligned} li &\mapsto l1\uparrow, l2\uparrow \\ \neg x \wedge l1 &\mapsto ro\uparrow \\ ri &\mapsto x\uparrow \\ \neg l1 \vee x &\mapsto ro\downarrow \\ x \wedge \neg ri &\mapsto lo\uparrow \\ \neg li &\mapsto l1\downarrow, l2\downarrow \\ \neg l2 &\mapsto x\downarrow \\ ri \vee \neg x &\mapsto lo\downarrow \end{aligned}$$

Now we observe that transition $l1\uparrow$ o is acknowledged by the guard of the textually following PR but $l2\uparrow$ is not; transition $l2\downarrow$ of is acknowledged by the guard of the textually following PR but $l1\downarrow$ is not. Hence, the assignments $l2\uparrow$ and $l1\downarrow$ do not fulfil the completion requirement, and thus are not stable!

We solve this problem by making a simplifying assumption: We assume that the fork is *isochronic, i.e., the difference in delays between the two branches of the fork is shorter than the delays in the operators to which the fork is an input.* Hence, when a transition on one output is acknowledged and thus completed, the transition on the other output is also acknowledged and thus completed.

This is the only timing condition that has to be fulfilled. In general, the constraint is easy to meet because it is one-sided. However, the isochronicity requirement is more difficult to meet when a negated input introduces an inverter on a branch of the fork, since the transition delays of an inverter are of the same order of magnitude as the transition delays of other operators. We have proved that, for the implementation of each language construct, these inverters can always be eliminated from the isochronic forks by simple transformations. (These transformations have not been applied to the circuits presented here as examples, but they are always applied before the circuits are actually implemented.)

In [14], we have proved that the class of entirely delay-insensitive circuits is very limited: Practically all circuits of interest fall outside the class. We believe that the notion of isochronic fork is the weakest compromise to delay-insensitivity sufficient to implement any circuit of interest.

Which forks have to be isochronic is easy to decide by a simple analysis of the PR sets. For instance, the fork $ri\,\underline{f}\,(r1, r2)$ also has to be isochronic, but the fork $x\,\underline{f}\,(x1, x2)$ does not. We shall ignore the issue of isochronic forks in the rest of this presentation.

5.8 Reshuffled Implementations of (L/R)

We illustrate the use of reshuffling by deriving two other implementations of (L/R). If L is an internal channel introduced for process decomposition, we can reshuffle the handshaking expansions of L and R without the risk of introducing deadlock. Let us return to handshaking expansion (14).

5.8.1 First Reshuffling

We postpone the second half of the handshaking expansion of R —i.e., the sequence $ro\downarrow; [\neg ri]$—until after $[\neg li]$. We get:

$$*[[li];\ ro\uparrow;\ [ri];\ lo\uparrow;\ [\neg li];\ ro\downarrow;\ [\neg ri];\ lo\downarrow]\ .$$

The syntactic PR expansion we now derive is already "program ordered":

$$\begin{aligned} li &\mapsto ro\uparrow \\ ri &\mapsto lo\uparrow \\ \neg li &\mapsto ro\downarrow \\ \neg ri &\mapsto lo\downarrow\ . \end{aligned}$$

The first and third rules specify the wire $(li\,\underline{w}\,ro)$, the second and fourth rules specify the wire $(ri\,\underline{w}\,lo)$. Hence, the implementation reduces to two wires!

5.8.2 Second Reshuffling: The D-element

We now postpone the whole handshaking expansion of R until after $[\neg li]$. We get:

$$*[[li];\ lo\uparrow;\ [\neg li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ lo\downarrow]\ .$$

We need to introduce a state variable, say x, as follows:

$$*[[li];\ x\uparrow;\ [x];\ lo\uparrow;\ [\neg li];\ ro\uparrow;\ [ri];\ x\downarrow;\ [\neg x];\ ro\downarrow;\ [\neg ri];\ lo\downarrow]\ .$$

The PR expansion gives:

$$\begin{aligned} li &\mapsto x\uparrow \\ (ri \vee) x &\mapsto lo\uparrow \\ x \wedge \neg li &\mapsto ro\uparrow \\ ri &\mapsto x\downarrow \\ (li \vee) \neg x &\mapsto ro\downarrow \\ \neg x \wedge \neg ri &\mapsto lo\downarrow\ . \end{aligned}$$

The terms between parentheses have been added for symmetrization. The operator reduction gives:

$$(li, \neg ri) \underline{ff}\ x$$
$$(ri, x) \underline{\vee}\ lo$$
$$(x, \neg li) \underline{\wedge}\ ro\ .$$

The flip-flop can be replaced with the C-element $(li, \neg ri)\ \underline{C}\ x$. The circuit is shown in Figure 5.2; it is called a D-element.

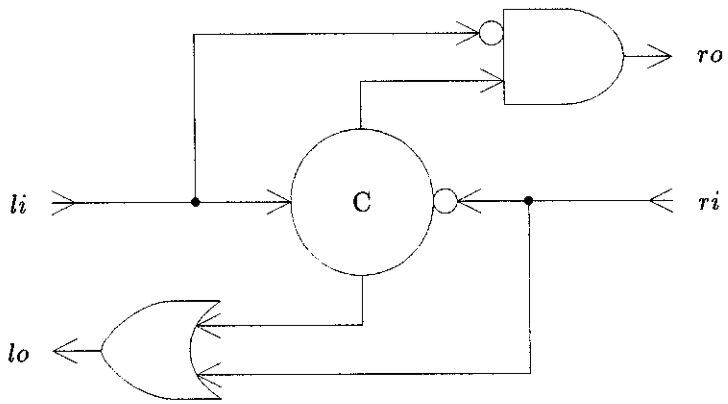

Figure 5.2: A circuit for the D-element

5.9 Example 2: A One-place Buffer

The one-place buffer is the most ubiquitous process. In the processor for example, each stage of the pipeline is a one-place buffer of the type:

$$*[L?x; R!f(x)]\ .$$

Let us ignore the transmission of messages, and implement the "bare" process:

$$*[L; R]\ .$$

One of the most useful implementations of this process is with L lazy-active and R passive. The handshaking expansion gives:

$$*[[\neg li];\ lo\uparrow;\ [li];\ lo\downarrow;\ [ri];\ ro\uparrow;\ [\neg ri];\ ro\downarrow]\ .$$

We choose to include the state variable x in such a way that the transition $x\uparrow$ is concurrent with $lo\uparrow$, and transition $x\downarrow$ is concurrent with $ro\uparrow$. We get:

$$*[[\neg li];\ lo\uparrow;\ x\uparrow;\ [x];\ [li];\ lo\downarrow;\ [ri];\ ro\uparrow;\ x\downarrow;\ [\neg x];\ [\neg ri];\ ro\downarrow]\ .$$

The production rule expansion is:

$$\begin{aligned} \neg x \wedge \neg li \wedge \neg ro &\mapsto lo\uparrow \\ lo &\mapsto x\uparrow \\ x \wedge li &\mapsto lo\downarrow \\ x \wedge \neg lo \wedge ri &\mapsto ro\uparrow \\ ro &\mapsto x\downarrow \\ \neg x \wedge \neg ri &\mapsto ro\downarrow \end{aligned}$$

The direct implementation of this production rule set is shown in Figure 5.3.

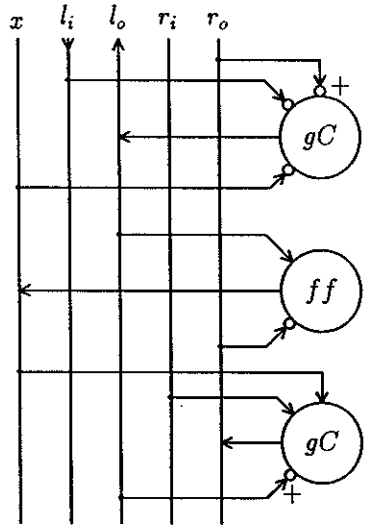

Figure 5.3: A circuit for the one-place buffer

5.10 Boolean Register

Consider the following *register* process that provides read and write access to a simple boolean variable, x:

$$\begin{array}{l} *[[\,\overline{P} \rightarrow P?x \\ \;\;[]\,\overline{Q} \rightarrow Q!x \\ \;]]\,, \end{array} \tag{5.11}$$

where $\neg\overline{P} \vee \neg\overline{Q}$ holds at any time.

The handshaking expansion uses the *double-rail* technique: The boolean value of x is encoded on two wires, one for the value **true** and one for the value **false** . Input channel P has two input wires, $pi1$ for receiving the value **true** , and $pi2$ for receiving the value **false** ; and one output wire, po. Output channel Q has two output wires, $qo1$ for sending the value **true** , and $qo2$ for sending the value **false** ; and one input wire, qi. Each guarded command is expanded to two guarded commands:

$$\begin{array}{l} *[[\,pi1 \rightarrow x\uparrow;\ [x];\ po\uparrow;\ [\neg pi1];\ po\downarrow \\ \;\;[]\,pi2 \rightarrow x\downarrow;\ [\neg x];\ po\uparrow;\ [\neg pi2];\ po\downarrow \\ \;\;[]\,x \wedge qi \rightarrow qo1\uparrow;\ [\neg qi];\ qo1\downarrow \\ \;\;[]\,\neg x \wedge qi \rightarrow qo2\uparrow;\ [\neg qi];\ qo2\downarrow \\ \;]]\ . \end{array} \tag{5.12}$$

5.10.1 Mutual Exclusion Between Guarded Commands

We are now faced with a new problem: enforcing mutual exclusion between the production-rule sets of different guarded commands. (This problem is not concerned with making the *guards* of the different commands mutually exclusive. For the time being, we are considering only examples where the guards of the commands are already mutually exclusive.) Let us illustrate our problem with the compilation of the first two guarded commands. If we just concatenate the production-rule sets of these two commands, we get:

$$\begin{array}{rl} pi1 & \mapsto x\uparrow \\ pi1 \wedge x & \mapsto po\uparrow \\ \neg pi1 & \mapsto po\downarrow \\ pi2 & \mapsto x\downarrow \\ pi2 \wedge \neg x & \mapsto po\uparrow \\ \neg pi2 & \mapsto po\downarrow\ . \end{array}$$

However, the second and the sixth guarded commands are interfering since they set and reset variable po concurrently. For reasons of symmetry, the same holds for the third and the fifth PRs.

The problem of ensuring mutual exclusion between PRs of different guarded commands is the same as enforcing program order between PRs of the same guarded command. We use the same technique, which consists in strengthening the guards of the production rules, if necessary, by introducing state variables to distinguish between the states corresponding to each true guard.

In the case at hand, we can strengthen the guards of the third and the sixth rules by combining the two rules as:

$$\neg pi1 \wedge \neg pi2 \mapsto po\downarrow \ .$$

The non-standard gate implementing the production rules of po is shown in Figure 5.4.

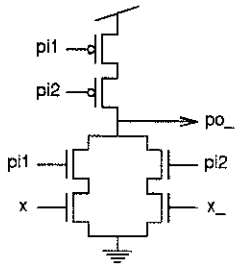

Figure 5.4: Non-standard gate for write acknowledge

We can also strengthen the guards of the third and the sixth rules as:

$$\begin{aligned} x \wedge \neg pi1 &\mapsto po\downarrow \\ \neg x \wedge \neg pi2 &\mapsto po\downarrow \ . \end{aligned}$$

Now, the PRs of po can be transformed into

$$\begin{aligned} (pi1 \wedge x) \vee (pi2 \wedge \neg x) &\mapsto po\uparrow \\ (\neg pi1 \wedge \neg x) \vee (\neg pi2 \wedge x) &\mapsto po\downarrow \ , \end{aligned}$$

which is the definition of the *if*-operator $(pi1, pi2, x)$ $\underline{if}$ po .

The rest of the implementation is straightforward. The first and fourth PRs correspond to the flip-flop $(pi1, \neg pi2)\underline{ff}\,x$. The production-rule expansion of the last two guarded commands gives:

$$\begin{aligned} x \wedge qi &\mapsto qo1\uparrow \\ \neg x \vee \neg qi &\mapsto qo1\downarrow \\ \neg x \wedge qi &\mapsto qo2\uparrow \\ x \vee \neg qi &\mapsto qo2\downarrow \ , \end{aligned}$$

which corresponds to the two operators $(x, qi) \triangle qo1$ and $(\neg x, qi) \triangle qo2$. The circuit is represented in Figure 5.5.

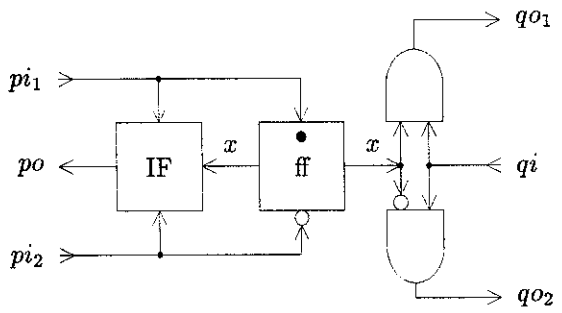

Figure 5.5: Single boolean register

In the next example, we shall refer to the implementation of the first two guarded commands as the *register* operator:

$$(pi1, pi2) \underline{reg} (po, x) .$$

We shall refer to the implementation of the last two guarded commands of (26) as the *read* operator:

$$(qi, x) \underline{read} (qo1, qo2) .$$

5.11 Process Factorization

The next example is used to introduce the technique of process factorization. The idea is to decompose a process, say, p, described as a handshaking expansion into a number of processes $p0$, $p1$ ..., pn such that $(p0 \| p1 \| \ldots \| pn)$ is equivalent to p, i.e., implements the same handshaking sequence as p.

Factorization obeys two rules.

- Rule 1: Each output variable belongs to exactly one factor process. (Hence factorization reduces the number of output variables per process.) Input variables may be shared by several factor processes.

- Rule 2: Two adjacent actions α; β of the original process are put into two different processes during factorization if, and only if, the semicolon between α and β is superfluous. Two cases fulfill this condition:
  1. the two adjacent actions $\{\neg x\}\ x\uparrow;\ [x]$ and the two adjacent actions $\{x\}\ x\downarrow;\ [\neg x]$ for internal variable x, and
  2. the pairs of handshaking actions $xo\uparrow;\ [xi]$ and $xo\downarrow;\ [\neg xi]$ for an active implementation, and the pair of handshaking actions $yo\uparrow;\ [\neg yi]$ for a passive implementation. (This is a direct consequence of Property 1.)

5.11.1 Example: Two-to-Four Phase Converter

The following process converts a passive two-phase handshaking on channel L into an active four-phase handshaking on channel R. First observe that the converter cannot be specified as a buffer $*[L;R]$. Indeed, let (L', R') be the channel on which the converter is to be inserted. This channel maintains the relation $\underline{c}L' = \underline{c}R'$. The converter should leave it unchanged. But if $0 \leq \underline{c}L - \underline{c}R \leq 1$, then $0 \leq \underline{c}L' - \underline{c}R' \leq 1$ holds after insertion of the converter. Hence, we have to implement the converter such that $\underline{c}L = \underline{c}R$, i.e., we have to interleave the handshaking of L and R in such a way that L and R are completed at the same time. We get:

$$conv \equiv *[[li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ lo\uparrow;\ [\neg li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ lo\downarrow]$$

(There are several ways to interleave the handshake sequences of two actions so as to make their completions coincide. Again, we have chosen the one in which the waits and the assignments alternate.) We first try to factorize *conv* into two processes, $p1$ and $p2$. We get

$$p1 \equiv *[[li];\ ro\uparrow; \ldots$$

$$p2 \equiv *[[ri];\ ro\downarrow; \ldots$$

Here the factorization fails since it violates rule 1. Rule 1 is violated because actions $ro\uparrow$ and $ro\downarrow$ follow each other as output actions in *conv*. We can separate the two output actions $ro\uparrow$ and $ro\downarrow$ by inserting a vacuous sequence $u\uparrow;\ [u]$ on a newly introduced internal variable u. (Initially, $u =$ **false**.) We introduce this sequence after the first $[ri]$; for reasons of symmetry, we introduce the sequence $u\downarrow;\ [\neg u]$ after the second $[ri]$. The transformed program is:

$$\begin{aligned} conv' \equiv *[\,&[li]; ro\uparrow; [ri]; u\uparrow; [u]; ro\uparrow; [\neg ri]; lo\uparrow; \\ &[\neg li]; ro\uparrow; [ri]; u\downarrow; [\neg u]; ro\uparrow; [\neg ri]; lo\downarrow \\ ]& \end{aligned}$$

Now, we can apply factorization rule 2 without violating rule 1. We get:

$$p1 \equiv *[[li]; ro\uparrow; [u]; ro\downarrow; [\neg li]; ro\uparrow; [\neg u]; ro\downarrow]$$

$$p2 \equiv *[[ri]; u\uparrow; [\neg ri]; lo\uparrow; [ri]; u\downarrow; [\neg ri]; lo\downarrow]$$

It is easy to verify that $(p1 \| p2) = conv'$. Since the sequences $u\uparrow;\ [u]$ and $u\downarrow;\ [\neg u]$ are both equivalent to a *skip* in $conv'$, $(p1 \| p2) = conv$.

Process $p2$ can immediately be identified as a standard process called a toggle, represented by the infix operator ri **tog** $(u; lo)$. For $p1$, we first strengthen the guards as follows:

$$p1 \equiv *[[\neg u \wedge li]; ro\uparrow; [li \wedge u]; ro\downarrow; [\neg li \wedge u]; ro\uparrow; [\neg li \wedge \neg u]; ro\downarrow]$$

The validity of this transformation relies on invariants from $conv'$; it cannot be justified by properties of $p1$ only.

Now $p1$ can be identified with a difference-operator: (u, li) **dif** ro, also called an *exclusive-or*. The corresponding circuit is shown in Figure 5.6.

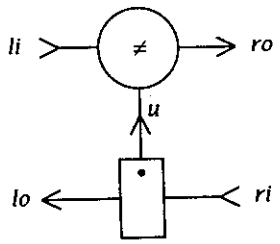

Figure 5.6: Two-to-four phase converter

The kind of process factorization we have described in the previous section is very helpful but can, in principle, be avoided by applying the standard technique for production-rule expansion. One case of process factorization that cannot be avoided is when a process has to be decomposed into two or more processes, one of which is given. For reasons that will become clear in a the following chapters, we call this transformation "process quotient".

5.12 Sequencing

There are many ways to implement the sequencing of n arbitrary actions. We shall introduce the basic operators that are used in the most straightforward implementations.

5.12.1 The Active-Active Buffer

Consider the program $*[S_1; S_2]$, where S_1 and S_2 are two arbitrary program parts. Process decomposition of this program gives

$$*[L; R] \parallel (L'/S_1) \parallel (R'/S_2) .$$

Hence, the basic sequencing operator is the process

$$B(L_a, R_a) \equiv *[L; R] ,$$

where both L and R are active. This process is called an *active-active buffer*. The handshaking expansion gives:

$$*[lo\uparrow;\ [li];\ lo\downarrow; [\neg li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri]] . \tag{5.13}$$

Since ri is **false** initially, we can rewrite 5.13 as:

$$*[[\neg ri];\ lo\uparrow;\ [li];\ lo\downarrow; [\neg li];\ ro\uparrow;\ [ri];\ ro\downarrow] . \tag{5.14}$$

By comparing 5.14 with (14)—the handshaking expansion of the Q-element, we observe that $B(L_a, R_a) \equiv (\neg ri, ro)\ \underline{Q}\ (li, lo)$, which gives the implementation of Figure 5.7.

5.12.2 The (L/A;R)-element

In order to generalize the above construction to the case of an arbitrary number of actions, we need to implement the generalization of the (L/R)-element. Sequence

$$*[S_1; S_2; \ldots; S_n] \tag{5.15}$$

can be decomposed into a number of shorter sequences by repeatedly applying process decomposition. There are as many ways to decompose 5.15 as there are binary trees of n leaves. But observe that, if $n > 2$, all decompositions will require at least one process of the form:

$$(L/A; R) ,$$

where A and R are active communication actions. (The semicolon binds more tightly than the process call.) We shall use two different reshufflings to

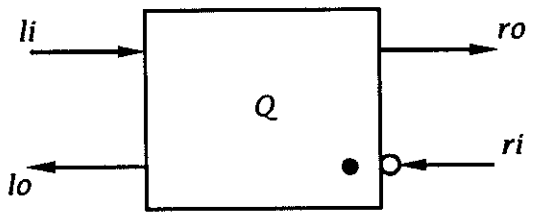

Figure 5.7: Implementation of the active-active buffer with a Q-element

implement this process. Again, these reshufflings maintain the semantics of the original program if the handshaking expansion of L' is not reshuffled.

The first reshuffling is:

$$*[[li];\ ao\uparrow;\ [ai];\ lo\uparrow;\ [\neg li];\ ao\downarrow;\ [\neg ai];\ R;\ lo\downarrow]\ .$$

We decompose it into two sequences by applying a process-factorization decomposition described earlier:

$$\begin{array}{l}(*[[li];\ ao\uparrow;\ [\neg li];\ ao\downarrow] \\ \|*[[ai];\ lo\uparrow;\ [\neg ai];\ R;\ lo\downarrow] \\ )\ .\end{array}$$

The first sequence is the wire $(li\ \underline{w}\ ao)$. The second sequence is the D-element $(ai, lo)\ \underline{D}\ (ri, ro)$.

The second reshuffling is:

$$*[[li];\ A;\ ro\uparrow;\ [ri];\ lo\uparrow;\ [\neg li];\ ro\downarrow;\ [\neg ri];\ lo\downarrow]\ .$$

Again, we decompose it into two sequences by process factorization:

$$
\begin{array}{l}
(\ *[[ri];\ lo\uparrow;\ [\neg ri];\ lo\downarrow] \\
\|*[[li];\ A;\ ro\uparrow;\ [\neg li];\ ro\downarrow] \\
)\ .
\end{array}
$$

The first sequence is the wire $(ri \underline{w}\ lo)$. The second sequence is the Q-element $(li, ro)\ \underline{Q}\ (ai, ao)$. Both implementations are shown in Figure 5.8.

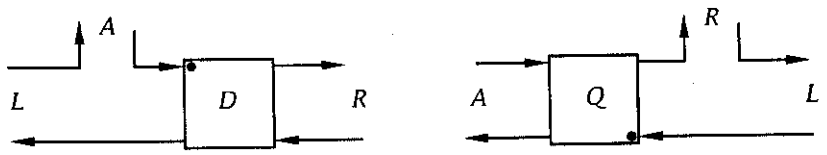

Figure 5.8: Implementations of the $(L/A; R)$-element

Now, the implementation of a sequence of n actions is straightforward. For instance, for $n = 4$, we have two "linear" decompositions of $(L/S_1; S_2; S_3; S_4)$. The first one is

$$((L/S_1; L_1) \parallel (L_1/S_2; L_2) \parallel (L_2/S_3; S_4))\ .$$

The second one is

$$((L/L_2; S_4) \parallel (L_2/L_1; S_3) \parallel (L_1/S_1; S_2))\ .$$

These two decompositions lead to the linear implementations shown in Figure 5.9.

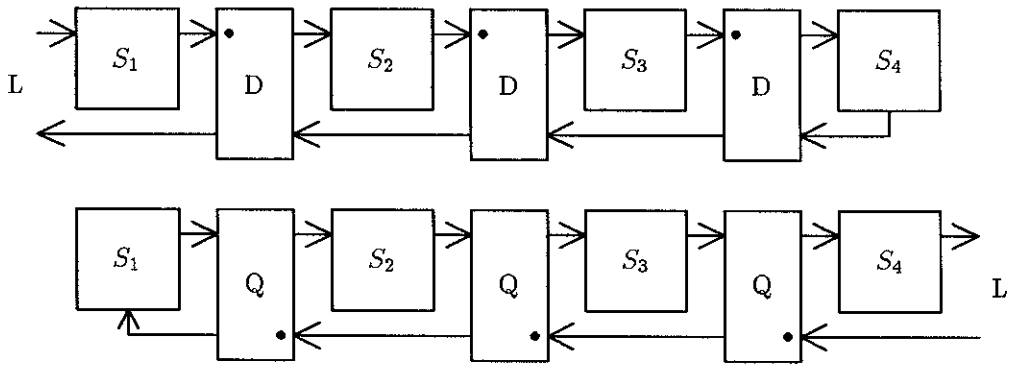

Figure 5.9: Implementations of $(L/S_1; S_2; S_3; S_4)$

5.12.3 The Passive-active Buffer

In order to compose one-place buffers in a linear chain, one channel must be active and the other one passive. We implement the buffer with L passive and R active. This version is denoted by $B(L_p, R_a)$. In order to take advantage of the active-active case, we decompose the buffer into two processes q and t:

$$q \equiv *[D'; R]$$
$$t \equiv (D/L) .$$

Process q is an active-active buffer. The compilation of t is straightforward. The handshaking expansion gives:

$$*[[di];\ [li];\ lo\uparrow;\ [\neg li];\ lo\downarrow;\ do\uparrow;\ [\neg di];\ do\downarrow] .$$

Since D is an internal channel, we can reshuffle the sequence $[\neg li]; lo\downarrow$ with respect to D without introducing deadlock. (Also observe that since $do\downarrow$ remains the last action of the sequence, we have not changed the order of L relative to R.) We get

$$*[[di];\ [li];\ lo\uparrow;\ do\uparrow;\ [\neg di];\ [\neg li];\ lo\downarrow;\ do\downarrow] .$$

The PR expansion leading to the circuit of Figure 6 is

$$di \wedge li \mapsto lo\uparrow, do\uparrow$$
$$\neg di \wedge \neg li \mapsto lo\downarrow, do\downarrow \ .$$

Process t is used to connect the two ports of a channel when they are both active. It is called a "passive-passive adaptor". The complete circuit is shown in Figure 5.10.

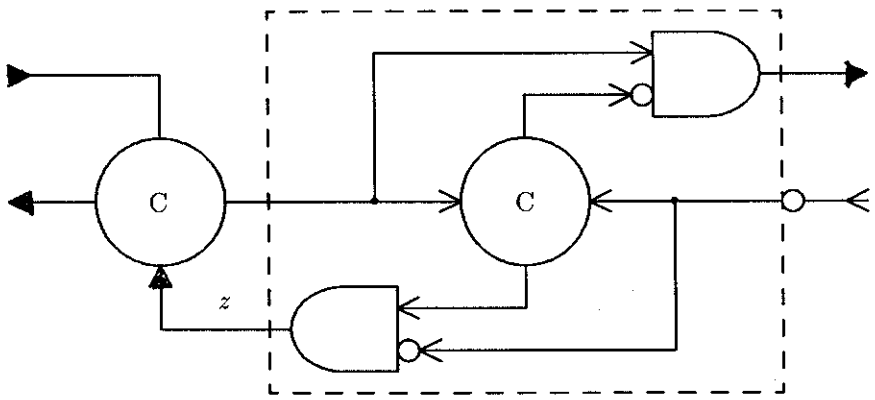

Figure 5.10: An implementation of the passive-active buffer

The passive-active buffer can be compiled directly by introducing a state variable. The circuit obtained is slightly different. See [9].

Chapter 6

Case Study: Two Arbitration Problems

6.1 Introduction

In this chapter, we construct circuits for two difficult control problems involving arbitration among asynchronous events. These examples show how to introduce the two standard building blocks for arbitration circuitry, the arbiter and the synchronizer.

The first example addresses the issues of arbitration between guards and unstable guards. We have already discussed the metastability property of arbiters. But the realization of a delay-insensitive arbiter raises another issue: fairness. An arbiter is *strongly fair* when a pending communication request is granted after a bounded number of other requests are granted. An arbiter is *weakly fair* when a request is granted after a finite number but possibly unbounded number of other requests. Whether it is possible to construct a delay-insensitive fair arbiter has been, so far, an open question. It has been conjectured that delay-insensitive fair arbiters do not exist. In this example, we prove the existence of delay-insensitive fair arbiters by constructing one.

6.1.1 A Fair-Arbiter Program

The process *fsel* described in the first part defines a fair arbitration program between two unrelated inputs. We choose to implement the following simplified version of *fsel*:

$$*[[\overline{A} \rightarrow A \| \neg\overline{A} \rightarrow \mathbf{skip}]; [\overline{B} \rightarrow B \| \neg\overline{B} \rightarrow \mathbf{skip}]] \; . \tag{6.1}$$

According to 6.1, when $\overline{A}$ holds, A will be completed after, at most, one B action, whatever the current state of the computation is. Hence, the arbiter is strongly fair towards requests A and B. Assume that A' is pending at a certain point of the computation. By definition of the probe, $\overline{A}$ is **true** eventually; i.e., a finite but unbounded number of B actions can be completed between the moment $\mathbf{q}A'$ holds and the moment $\overline{A}$ holds. Hence, the arbiter is only *weakly* fair towards requests A' and B'.

Therefore, *with this definition of suspension of an action*, we can say that the arbiter is strongly fair towards requests that have reached the arbiter and weakly fair towards all requests. (We could redefine the suspension of a communication action X such that $\mathbf{q}X$ holds only when the initiation of action X can be observed by the other process. With this definition of suspension, we have $\mathbf{q}A' = \overline{A}$. The arbiter is then strongly fair towards all requests.)

6.1.2 The Compilation

Applying the process decomposition rule, we decompose 6.1 into three processes ($P1 \parallel P2 \parallel P3$). Channels (C, D) between $P1$ and $P2$, and (E, F) between $P1$ and $P3$ are introduced.

$$
\begin{array}{ll}
P1 \equiv & *[E; C] \\
P2 \equiv & *[[\,\overline{D} \wedge \overline{B} \rightarrow B; D \\
& \;\;[]\; \overline{D} \wedge \neg\overline{B} \rightarrow D \\
& \,]] \\
P3 \equiv & *[[\,\overline{F} \wedge \overline{A} \rightarrow A; F \\
& \;\;[]\; \overline{F} \wedge \neg\overline{A} \rightarrow F \\
& \,]]\ .
\end{array}
$$

Ports D and F are implemented as passive; ports C and E are implemented as active. Hence $P1$ is the standard active-active buffer. The handshaking expansion of $P2$ gives:

$$
\begin{array}{ll}
P2 \equiv & *[[\,di \wedge bi \rightarrow bo\uparrow;\ [\neg bi];\ bo\downarrow;\ do\uparrow;\ [\neg di];\ do\downarrow \\
& \;\;[]\; di \wedge \neg bi \rightarrow do\uparrow;\ [\neg di];\ do\downarrow \\
& \,]]\ .
\end{array}
$$

Because bi can change from **false** to **true** asynchronously, the second guard of $P2$ is not stable; i.e., its value can change from **true** to **false** at any time. In order to make both guards of $P2$ stable, we introduce the synchronizer

$$
\begin{array}{ll}
sync \equiv & *[\,[di \wedge bi \rightarrow u\uparrow;\ [\neg di];\ u\downarrow \\
& \;\;[]\; di \wedge \neg bi \rightarrow v\uparrow;\ [\neg di];\ v\downarrow \\
& \,]]\ .
\end{array}
$$

sync is a standard operator that we have described in Part I. We now have to find a process, X, such that Since *sync* is entirely defined, we would like to be able to perform the inverse operation of $\|$, or "process quotient", so as to compute X as $X = (P2 \div sync)$. A way to perform this quotient is to remove all actions of *sync* from $P2$, and then to check whether the result fulfills $(X \parallel sync) = P2$.

To perform the quotient as suggested, $P2$ should be extended to contain all actions of *sync*, so that the orders of actions are compatible in *sync* and in the extended version of $P2$. (This procedure is explained in [10].) The extension of $P2$ gives:

$$
\begin{array}{l}
*[[\, di \wedge bi \rightarrow u\uparrow;\ [u];\ bo\uparrow;\ [\neg bi];\ bo\downarrow;\ do\uparrow;\ [\neg di];\ u\downarrow;\ [\neg u];\ do\downarrow \\
\ \ [\!]\ di \wedge \neg bi \rightarrow v\uparrow;\ [v];\ do\uparrow;\ [\neg di];\ v\downarrow;\ [\neg v];\ do\downarrow \\
\ \ ]] \ .
\end{array}
$$

We obtain for X:

$$
\begin{array}{l}
*[\,[u \rightarrow bo\uparrow;\ [\neg bi];\ bo\downarrow;\ do\uparrow;\ [\neg u];\ do\downarrow \\
\ \ [\!]v \rightarrow do\uparrow;\ [\neg v];\ do\downarrow \\
\ \ ]] \ .
\end{array}
$$

The compilation of the first guarded command is facilitated if transition $bo\downarrow$ is postponed until after $[\neg u]$. This transformation does not introduce deadlock since the completion of D does not depend on the completion of B. After this transformation, the PR expansion gives:

$$
\begin{aligned}
u &\mapsto bo\uparrow \\
u \wedge \neg bi &\mapsto do\uparrow \\
bi \vee \neg u &\mapsto do\downarrow \\
\neg u &\mapsto bo\downarrow \\
v &\mapsto do\uparrow \\
\neg v &\mapsto do\downarrow \ .
\end{aligned}
$$

The operator reduction, which includes introducing auxiliary variables do' and do'', gives

$$
\begin{aligned}
u \quad & \underline{w}\, bo \\
(u, \neg bi) \quad & \underline{\wedge} do' \\
v \quad & \underline{w}\, do'' \\
(do', do'') \quad & \underline{\vee} do \ .
\end{aligned}
$$

The circuit is shown in Figure 6.1. The implementation of P3 is identical.

6.1.3 The Circuit

The final circuit, shown in Figure 6.2, is obtained by composing the two identical circuits implementing $P2$ and $P3$ with the circuit of $P1$. The reshuffled

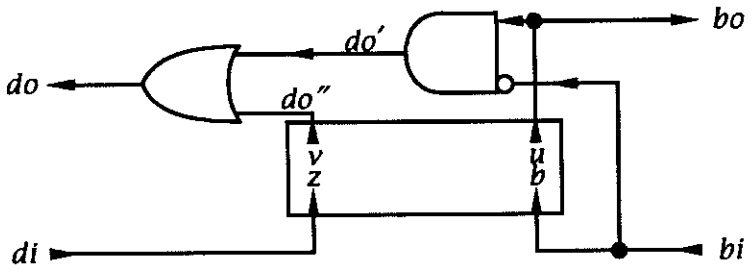

Figure 6.1: Implementation of P2

version of $P1$, consisting of a wire and an inverter, can also be used if it can be proved that the reshuffling does not introduce deadlock. The circuit shown in Figure 6.2 includes a minor optimization that eliminates the negated inputs that are also the output of a fork.

Notice that the solution can be immediately generalized to an arbitrary number of requests.

6.2 Distributed Mutual Exclusion

The first paper describing this method for the synthesis of asynchronous circuits from high-level description was presented at the 1985 Chapel Hill Conference on VLSI [8]. The example used to illustrate the method was the algorithm for distributed mutual exclusion on a ring of processes described in Chapter 2.

Unfortunately, the circuit presented in the Chapel Hill paper is not entirely

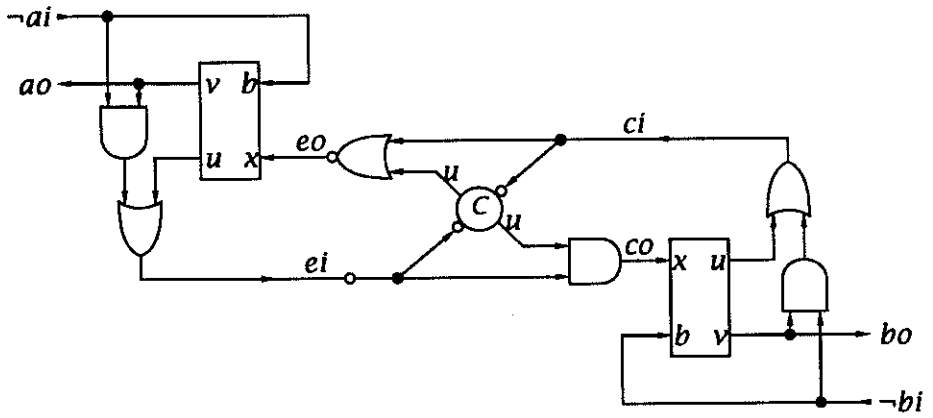

Figure 6.2: Implementation of the fair arbiter

correct: A glitch may appear on the wire named z in the paper. The error is due to my not following the compilation procedure when I defined the variable z. The error was noticed by many people, and the actual CMOS implementation of the circuit realized by Andy Fife the same year is entirely correct.

However, I never took the time to publish the correct solution, and therefore the bug has been rediscovered over and over again, sometimes with great publicity[4]. Since several people have asked me to show them a correct derivation of the circuit, here it is after five years!

As in the original paper, we observe that the two consecutive D commands, and the two consecutive U commands can both be implemented as the two halves of a 4-phase handshaking protocol; and therefore we can replace the two U commands with one single U to be implemented as a 4-phase handshaking protocol.

Next, we decompose process m into two processes A and B as follows:

$$
\begin{array}{l}
A \equiv *[\,[\overline{U} \rightarrow P; Bt; U \\
\quad\;\; [\overline{L} \rightarrow P; Bf; L \\
\quad]\,]
\end{array}
$$

$$
\begin{array}{l}
B \equiv *[\,[\overline{Q} \wedge b \rightarrow Q \\
\quad\;\; [\overline{Q} \wedge \neg b \rightarrow R; Q \\
\quad\;\; [\overline{S} \rightarrow b\uparrow; S \\
\quad\;\; [\overline{T} \rightarrow b\downarrow; T \\
\quad]\,]
\end{array}
$$

The internal channels between A and B are (P, Q), (Bt, S), and (Bf, T).

The technique used to obtain A and B is the standard process decomposition, with one addition. The plain process decomposition would give a process A with U before Bt, and L before Bf. We have inverted the order of these actions, since it is semantically irrelevant whether the assignment to b is the last action of the guarded command provided the assignment follows the selection command. The reason for this transformation is that the program in which U and L are the last actions of the guarded commands is easier to implement. This point will be further explained in the compilation of A.

6.2.1 Compilation of A

Since the guards $\overline{U}$ and $\overline{L}$ are not mutually exclusive, we are introducing an arbiter described by the program:

$$
\begin{array}{l}
Arb \equiv *[\,[ui \rightarrow u'\uparrow;\ [\neg ui];\ u'\downarrow \\
\quad\;\; [li \rightarrow l'\uparrow;\ [\neg li];\ l'\downarrow \\
\quad]\,] .
\end{array}
$$

We know that $A = (Arb \| A')$, where $A' = A^{ui,li}_{u',l'}$.

EXERCISE Prove the correctness of the above result. □

Hence:

$$
\begin{array}{l}
A' \equiv *[\,[u' \rightarrow po\uparrow; [pi]; po\downarrow; [\neg pi]; bto\uparrow; [bti]; bto\downarrow; [\neg bti]; uo\uparrow; [\neg u']; uo\downarrow \\
\quad\;\; [l' \rightarrow po\uparrow; [pi]; po\downarrow; [\neg pi]; bfo\uparrow; [bfi]; bfo\downarrow; [\neg bfi]; lo\uparrow; [\neg l']; lo\downarrow \\
\quad]\,]
\end{array}
$$

6.2.2 Mutual exclusion among guarded commands

The main problem in implementing A' is to enforce the mutual exclusion between the two guarded commands (GCs). By construction of the arbiter circuit Arb, we know that—provided that $\neg u' \wedge \neg l'$ holds initially—$\neg u' \vee \neg l'$ holds at any time. Hence, the mutual exclusion between the *guards* of A' is guaranteed.

However, as soon as $u'\downarrow$ is completed, the first GC of the arbiter can complete, the second GC of the arbiter can start, and consequently, the PR set implementing the second GC of A' can start firing, even though the first GC of A' may not be completed. We shall see that, in order to enforce the mutual exclusion between the implementations of the two GCs of A', it is advantageous to postpone $u'\downarrow$ as long as possible. This explains our decision to modify A such that U is the last action of the first GC, and L the last action of the second GC.

6.3 First Solution

We slightly reshuffle the actions of A' as follows:

$$\begin{aligned} A' \equiv *[\,&[u' \rightarrow po\uparrow;\ [pi];\ po\downarrow;\ [\neg pi];\ bto\uparrow;[bti];uo\uparrow;[\neg u'];bto\downarrow;[\neg bti];uo\downarrow \\ &[\!]\, l' \rightarrow po\uparrow;\ [pi];po\downarrow;\ [\neg pi];\ bfo\uparrow;\ [bfi];lo\uparrow;[\neg l'];bfo\downarrow;[\neg bfi];lo\downarrow \\ &]] \end{aligned}$$

We first ignore the transitions on *bto*, *bti*, *bfo*, and *bfi*, and implement the program:

$$\begin{aligned} A' \equiv *[\,&[u' \rightarrow po\uparrow;\ [pi];\ po\downarrow;\ [\neg pi];\ uo\uparrow;\ [\neg u'];\ uo\downarrow \\ &[\!]\, l' \rightarrow po\uparrow;\ [pi];\ po\downarrow;\ [\neg pi];\ lo\uparrow;\ [\neg l'];\ lo\downarrow \\ &]] \end{aligned}$$

Each guarded command is a Q-element. The transitions on *bto*, *bti*, *bfo*, and *bfi* are added by just "opening" the wires *uo* and *lo*, respectively.

For mutual exclusion between the implementations of the two guarded commands, the guard u' is strengthened as $u' \wedge \neg lo$, and the guard l' is strenghened as $l' \wedge \neg uo$.

6.3.1 Merge

We now have to compose the circuit implementing the first GC with the one implementing the second GC. This composition is a little more than mere juxtaposition because the two circuits use the variables *pi* and *po*. The standard way to deal with this case is to compose the two circuits with a *merge* circuit.

We replace P with $P1$ in the first GC, and with $P2$ in the second GC, and add the *merge* process:

$$\begin{aligned} *[\,&[\overline{P1} \rightarrow P1 \bullet P \\ &[\!]\, \overline{P2} \rightarrow P2 \bullet P \\ &]] \end{aligned}$$

The handshaking expansion gives:

$$*[\,[p1i \rightarrow po\uparrow;\ [pi];\ p1o\uparrow;\ [\neg p1i];\ po\downarrow;\ [\neg p1i];\ p1o\downarrow$$
$$[]\ p2i \rightarrow po\uparrow;\ [pi];\ p2o\uparrow;\ [\neg p2i];\ po\downarrow;\ [\neg p2i];\ p2o\downarrow$$
$$]\,]$$

The production rule expansion gives:

$$\begin{aligned} p1i \vee p2i &\mapsto po\uparrow \\ pi \wedge p1i &\mapsto p1o\uparrow \\ \neg p1i \wedge \neg p2i &\mapsto po\downarrow \\ \neg pi &\mapsto p1o\downarrow \\ pi \wedge p2i &\mapsto p2o\uparrow \\ \neg pi &\mapsto p2o\downarrow\ . \end{aligned}$$

The operators are the or-gate $(p1i, p2i) \underline{\vee}\ po$, and the two asymmetric C-elements $(pi; p1i)\ \underline{aC}\ p1o$ and $(pi; p2i)\ \underline{aC}\ p2o$.

6.3.2 Circuit for A'

Composing the merge circuit and the circuits for the two guarded commands lead to an implementation of A'. But we make two observations. First, the asymmetric C-elements in the merge are not needed in this case. Second, and more importantly, we realize that instead of merging the two circuits after the two Q-elements, we could merge them before the Q-elements so that the two circuits could share the same Q-element. This transformation is formalized by the following program decomposition. We have $A' \equiv (A1 \| Q)$, with:

$$A1 \equiv *[\,[u' \wedge \neg lo \rightarrow po'\uparrow;\ [pi'];\ bto\uparrow; [bti]; uo\uparrow; [\neg u']; po'\downarrow;\ [\neg pi'];\ bto\downarrow; [\neg bti]; uo\downarrow$$
$$[]\ l' \wedge \neg uo \rightarrow po''\uparrow;\ [pi']; bfo\uparrow;\ [bfi]; lo\uparrow; [\neg l']; po''\downarrow;\ [\neg pi'];\ bfo\downarrow; [\neg bfi]; lo\downarrow$$
$$]\,]$$

$$Q \equiv *[[po' \vee po'']];\ po\uparrow;\ [pi];\ po\downarrow;\ [\neg pi];\ pi'\uparrow;\ [\neg po' \wedge \neg po''];\ pi'\downarrow]$$

The first guarded command of $A1$ is compiled as:

$$\begin{aligned} u' \wedge \neg lo &\mapsto po'\uparrow \\ pi' \wedge po' &\mapsto bto\uparrow \\ bti &\mapsto uo\uparrow \\ lo \vee \neg u' &\mapsto po'\downarrow \\ pi' &\mapsto bto\downarrow \\ \neg bti &\mapsto uo\downarrow \end{aligned}$$

The operator reduction gives:

$$(u', \neg lo)\ \underline{\triangle} po'$$
$$(pi'; po')\ \underline{aC}\ bto$$
$$bti\ \underline{w}\ uo$$

The compilation of the second GC of $A1$ is similar.

6.3.3 Compilation of B

The compilation of B is identical to that of the original paper. The handshaking expansion of B with a slight reshuffling of the actions in the second GC gives:

$$\begin{aligned} B \equiv *[\,[&qi \wedge b \rightarrow qo\uparrow; [\neg qi]; qo\downarrow \\ \,[]\,&qi \wedge \neg b \rightarrow ro\uparrow; [ri]; qo\uparrow; [\neg qi]; ro\downarrow; [\neg ri]; qo\downarrow \\ \,[]\,&si \rightarrow b\uparrow; so\uparrow; [\neg si]; so\downarrow \\ \,[]\,&ti \rightarrow b\downarrow; to\uparrow; [\neg ti]; to\downarrow \\ ]\,] \end{aligned}$$

We first observe that the mutual exclusion between the guards and between the guarded commands is guaranteed. The production rule expansion gives:

$$\begin{aligned} qi \wedge b &\mapsto qo\uparrow \\ b \wedge \neg qi &\mapsto qo\downarrow \\ qi \wedge \neg b &\mapsto ro\uparrow \\ ri &\mapsto qo\uparrow \\ \neg qi &\mapsto ro\downarrow \\ \neg b \wedge \neg ri &\mapsto qo\downarrow \end{aligned}$$

The conjunct b is added to the guard of the first PR for mutual exclusion with the second GC. A better strengthening of the two rules that reset qo is $\neg qi \wedge \neg ri \mapsto qo\downarrow$.

Combining all PRs relative to qo gives:

$$\begin{aligned} qi \wedge b \vee ri &\mapsto qo\uparrow \\ \neg qi \wedge \neg ri &\mapsto qo\downarrow \end{aligned}$$

The other operator is $(qi; \neg b)\ \underline{aC}\ ro$. The production rule expansion of the last two GCs is straightforward. It gives:

$$\begin{aligned} si &\mapsto b\uparrow \\ ti &\mapsto b\downarrow \\ si \wedge b &\mapsto so\uparrow \\ \neg si &\mapsto so\downarrow \\ ti \wedge \neg b &\mapsto to\uparrow \\ \neg ti &\mapsto to\downarrow \end{aligned}$$

The set of operators is:

$$(si; \neg ti)\ \underline{ff}\ b$$

$$(si; b)\ \underline{aC}\ so$$

$$(ti; \neg b)\ \underline{aC}\ to$$

The last two operators can be replaced with the and-gates $(si, bi)\ \underline{\wedge}\ so$ and $(ti, \neg b)\ \underline{\wedge}\ to$ by the usual symmetrization. The flip-flop can be replaced with the C-element $(si, \neg ti)\ \underline{C}\ b$, also by symmetrization.

The complete circuit is shown in Figure 6.3.

6.4 Exercise: Implementation without reshuffling

Can we implement the program of A directly without postponing U and L? We have to implement the following version of A:

$$\begin{array}{l} A \equiv *[\,[\overline{U} \rightarrow P; U; Bt \\ \quad [\!]\overline{L} \rightarrow P; L; Bf \\ \quad]\,] \end{array}$$

B is unchanged.

$$A1 \equiv (Arb \| A'),$$

where Arb is unchanged. A' is slightly reshuffled.

$$\begin{array}{l} A' \equiv *[\,[u' \rightarrow po\uparrow; [pi]; uo\uparrow; [\neg u']; po\downarrow; [\neg pi]; uo\downarrow; bto\uparrow; [bti]; bto\downarrow; [\neg bti] \\ \quad [\!]l' \rightarrow po\uparrow; [pi]; lo\uparrow; [\neg l']; po\downarrow; [\neg pi]; lo\downarrow; bfo\uparrow; [bfi]; bfo\downarrow; [\neg bfi] \\ \quad]\,] \end{array}$$

Apart from the opening of the uo wire for the (po, pi) connection, the first guarded command is just the passive-active buffer:

$$*[[u']; uo\uparrow; [\neg u']; uo\downarrow; bto\uparrow; [bti]; bto\downarrow; [\neg bti]]$$

The rest of the compilation is left as an exercise to the reader.

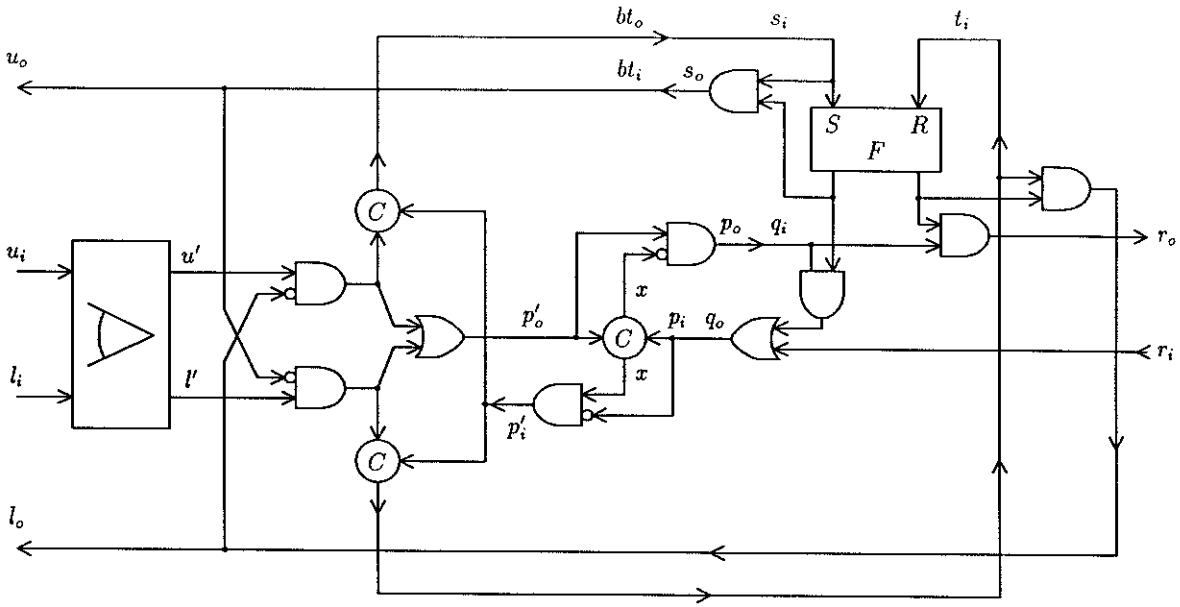

Figure 6.3: Circuit for a server

Chapter 7

Implementation of the Lazy Stack

7.1 Introduction

The design of the stack will be used to explain the general method for implementing communications that involve passing messages. The method relies on the time-honored "divide-and-conquer" principle: We first construct the so-called *control part* of the program, which is the original program in which messages have been removed from each communication action, and all arithmetic operations have been replaced by procedure and function calls . We then combine this control part with a *data path*, which is a collection of processes implementing the assignment parts of the communication actions and the functions and procedures implementing arithmetic operations.

7.2 The Control Part of the Stack

We assume that the stack is empty initially. We introduce the channel (t, t'), so that F can be called from within E by process decomposition. We get

$$
\begin{aligned}
E \equiv\ & *[[\, \overline{in} \rightarrow in?x;\ t \\
& \quad [\!] \ \overline{out} \rightarrow get?x;\ out!x \\
& \,]] \\
F \equiv\ & *[[\, \overline{t'} \wedge \overline{in} \rightarrow put!x;\ in?x \\
& \quad [\!] \ \overline{t'} \wedge \overline{out} \rightarrow out!x;\ t' \\
& \,]] \ .
\end{aligned}
$$

The control part of the stack consists of programs E and F, from which message communication has been removed. We get

$$\begin{array}{ll} E \equiv & *[[\,\overline{in} \rightarrow in;\ t \\ & \ [\!]\ \overline{out} \rightarrow get;\ out \\ &]] \\ F \equiv & *[[\,\overline{t'} \wedge \overline{in} \rightarrow put;\ in \\ & \ [\!]\ \overline{t'} \wedge \overline{out} \rightarrow out;\ t' \\ &]]\ . \end{array}$$

In the handshaking expansion, we let the choice of active and passive communications be dictated by the occurrence of the probes. (However, we will return to this choice later.) We get

$$\begin{array}{ll} E \equiv & *[[\,ini \rightarrow ino\uparrow;\ [\neg ini];\ ino\downarrow;\ to\uparrow;\ [ti];\ to\downarrow;\ [\neg ti] \\ & \ [\!]\ outi \rightarrow geto\uparrow;\ [geti];\ geto\downarrow;\ [\neg geti];\ outo\uparrow;\ [\neg outi];\ outo\downarrow \\ &]] \\ F \equiv & *[[\,ti' \wedge ini \rightarrow puto\uparrow;\ [puti];\ puto\downarrow;\ [\neg puti];\ ino\uparrow;\ [\neg ini];\ ino\downarrow \\ & \ [\!]\ ti' \wedge outi \rightarrow outo\uparrow;\ [\neg outi];\ outo\downarrow;\ to'\uparrow;\ [\neg ti'];\ to'\downarrow \\ &]]\ . \end{array}$$

7.2.1 Compilation of E

The first guarded command, $E1$, is a standard passive-active buffer. The second guarded command, $E2$, is a standard Q-element. The implementation of E must combine the implementations of $E1$ and $E2$ in a way that enforces mutual exclusion between the execution of $E1$ and that of $E2$.

Since the execution of *in* and that of *out* are mutually exclusive, it suffices to guarantee that when *in* is completed in $E1$, $E2$ cannot start until t is completed. On the other hand, we are sure that $E1$ cannot start before $E2$ is completed because $outo\downarrow$ is the last action of $E2$.

In order to prevent $E2$ from starting before $E1$ is completed, we have to introduce an extra variable, or reshuffle the handshaking expansions. We choose to introduce the variable z (initially **true**) in the handshaking expansion of $E1$, and we strengthen the guard of $E2$ with z. We get

$$E1 \equiv z \wedge ini \rightarrow ino\uparrow;\ z\downarrow;\ [\neg z];\ [\neg ini];\ ino\downarrow;\ to\uparrow;\ [ti];\ to\downarrow;\ [\neg ti];\ z\uparrow\ ,$$

$$E2 \equiv \neg ti \wedge outi \wedge z \rightarrow geto\uparrow;\ [geti];\ geto\downarrow;\ [\neg geti];\ outo\uparrow;\ [\neg outi];\ outo\downarrow\ .$$

It turns out that our choice for variable z is quite fortunate as it is already an internal variable of $E1$, as indicated on Figure 7.1.

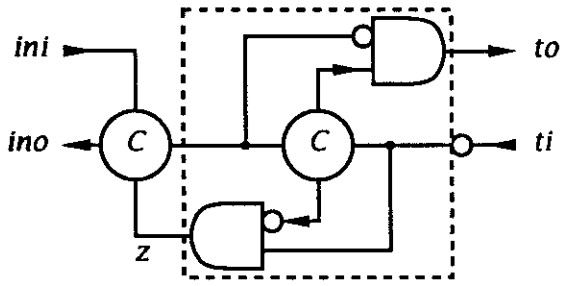

Figure 7.1: Implementation of the first g.c. of E with variable z

Now, $E2$ cannot start until $z\uparrow$ is completed, i.e., until $E1$ is completed. For symmetrization, we also weaken $\neg outi$ as $\neg outi \vee \neg z$. Hence, mutual exclusion is enforced by replacing input $outi$ with the *and*-operator $(outi, z)\ \underline{\wedge}\ outi'$ in the Q-element implementation of $E2$. This gives the circuit of Figure 7.2 as an implementation of E.

7.2.2 Compilation of F

The compilation of the first guarded command $F1$ of F is identical to that of $E2$ with the appropriate change of variables. The compilation of the second guarded command $F2$, however, can be simplified by reshuffling. We reshuffle the handshaking sequence of t' in $F2$ as follows:

$$ti' \wedge outi \rightarrow outo\uparrow;\ to'\uparrow;\ [\neg ti' \wedge \neg outi];\ outo\downarrow;\ to'\downarrow$$

The validity of this reshuffling stems from the fact that we do not reshuffle the initiation or the completion of action t' since $[ti']$ and $to'\downarrow$ are not reshuffled and the reshuffling of the middle two actions of t' does not introduce deadlock. The above sequence compiles immediately into the "forked" C-element $(ti', outi)\ \underline{C}\ (outo, to')$. The reshuffling guarantees that $F1$ cannot be started before $F2$ is completed.

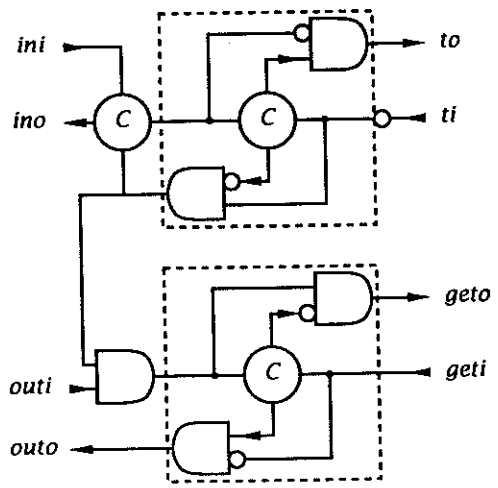

Figure 7.2: Implementation of E

The channels *in* and *out* are used both in E and F, so we need to merge the local copies of *in* and the local copies of *out* in a standard way that we do not describe here. The resulting circuit for the control part of the stack element is shown in Figure 7.3.

7.3 Implementation of the data path

We now have to extend the implementation of the control part $S2$ so as to obtain an implementation of the whole program $S1$. We want to leave $S2$ unchanged by introducing a data path process, P, such that the parallel composition of $S2$ and P implements $S1$.

The channels in, out, get, put of $S2$ are renamed in', out', get', put'. P communicates with $S2$ via in', out', get', put' and with the environment via *in, out, get, put.* (See Figure 7.4.)

Let C be a channel of $S1$, and C' be the renamed channel of $S2$ to which C

corresponds. For $(S2 \parallel P)$ to implement $S1$, each communication on C must coincide with a communication on C'; i.e., P must implement the so-called *channel interface* process

$$I_C \equiv *[C \bullet C'] .$$

Hence, P has to implement the four channel interfaces:

$$*[\,in' \bullet in?x]$$
$$*[\,out' \bullet out!x]$$
$$*[\,get' \bullet get?x]$$
$$*[\,put' \bullet put!x] .$$

7.4 Implementation of Channel Interfaces

There are four types of channel interfaces, depending on whether the port is active or passive, and whether the communication is an input or an output.

7.4.1 Input Actions on a Passive Port

We want to implement the interface I_C for action $C?x$ on the passive port C. I_C communicates with $S2$ by the active port C', and with the environment by the passive port D. Furthermore, in the standard double-rail encoding technique, the two-wire implementation (ci, co) of C has to be interfaced to the three-wire input port D in which the two input wires, $di1$ and $di2$, are used to encode the two values of the incoming message. (See Figure 7.5.)

I_C has to implement an interleaving of the three sequences:

$$S_C \equiv *[ci'\uparrow;\ [co'];\ ci'\downarrow;\ [\neg co']]$$
$$S_D \equiv *[[di1 \vee di2];\ do\uparrow;\ [\neg di1 \wedge \neg di2];\ do\downarrow]$$
$$S_X \equiv *[[di1 \rightarrow x\uparrow;\ [x] \,[]\, di2 \rightarrow x\downarrow;\ [\neg x]]] .$$

We first interleave sequences S_C and S_D so as to implement $C' \bullet D$:

$$*[[di1 \vee di2];\ ci'\uparrow;\ [co'];\ do\uparrow;\ [\neg di1 \wedge \neg di2];\ ci'\downarrow;\ [\neg co'];\ do\downarrow] . \qquad (7.1)$$

Next we interleave (7.1) and S_X. The interleaving has to ensure that the assignment to x is inserted after $[co']$ so that, when the assignment to x is performed in the datapath, communication action C has indeed been started in the control part. This interleaving is the final specification of the interface Ic:

$$*[[di1 \vee di2];\ ci'\uparrow;\ [co' \wedge di1 \rightarrow x\uparrow;\ [x]\,[]\,co' \wedge di2 \rightarrow x\downarrow;\ [\neg x]];\ do\uparrow;\ [\neg di1 \wedge \neg di2];\ ci'\downarrow;\ [\neg co'];\ do\downarrow] . \qquad (7.2)$$

We can implement (7.2) directly as follows:

$$\begin{aligned} di1 \vee di2 &\mapsto ci'\uparrow \\ co' \wedge di1 &\mapsto x\uparrow \\ co' \wedge di2 &\mapsto x\downarrow \\ di1 \wedge x \vee di2 \wedge \neg x &\mapsto do\uparrow \\ \neg di1 \wedge \neg di2 &\mapsto ci'\downarrow \\ \neg co' &\mapsto do\downarrow \end{aligned}$$

We can aslo decompose (7.2) into standard operators. We first decompose (7.2) into the two sequences:

$$*[[di1 \vee di2];\ ci'\uparrow;\ [\neg di1 \wedge \neg di2];\ ci'\downarrow] \tag{7.3}$$

and

$$\begin{aligned} *[[&\ co' \wedge di1 \rightarrow x\uparrow;\ [x];\ do\uparrow;\ [\neg co'];\ do\downarrow \\ [\!]&\ co' \wedge di2 \rightarrow x\downarrow;\ [\neg x];\ do\uparrow;\ [\neg co'];\ do\downarrow \\ ]]&\ . \end{aligned} \tag{7.4}$$

Sequence (7.3) is realized by the operator $(di1, di2)\ \underline{\vee}\ ci'$. We factor (7.4) so as to isolate the register part:

$$\begin{aligned} (co', di1)\ \underline{aC}\ x1 \equiv&\ *[[co' \wedge di1];\ x1\uparrow;\ [\neg co'];\ x1\downarrow] \\ (co', di2)\ \underline{aC}\ x2 \equiv&\ *[[co' \wedge di2];\ x2\uparrow;\ [\neg co'];\ x2\downarrow] \\ (x1, x2)\ \underline{reg}\ (x, do) \equiv&\ *[[\ x1 \rightarrow x\uparrow;\ [x];\ do\uparrow;\ [\neg x1];\ do\downarrow \\ &\ [\!]\ x2 \rightarrow x\downarrow;\ [\neg x];\ do\uparrow;\ [\neg x2];\ do\downarrow \\ &\ ]]\ . \end{aligned}$$

The implementation is shown in Figure 7.6.

7.4.2 Input Actions on an Active Port

For port C active, the communication variables of the interface I_C remain the same. But now the handshaking expansions of C' and D are different, since C' is passive and D is active. We get:

$$\begin{aligned} S_C &\equiv *[[co'];\ ci'\uparrow;\ [\neg co'];\ ci'\downarrow] \\ S_D &\equiv *[do\uparrow;\ [di1 \vee di2];\ do\downarrow;\ [\neg di1 \wedge \neg di2]] \\ S_X &\equiv *[[di1 \rightarrow x\uparrow;\ [x]\ [\!]\ di2 \rightarrow x\downarrow;\ [\neg x]]]\ . \end{aligned}$$

(Observe that S_X is not changed.)

An interleaving of S_C and S_D that implements $C' \bullet D$ is the interleaving corresponding to two wires:

$$*[[co'];\ do\uparrow;\ [di1 \vee di2];\ ci'\uparrow;\ [\neg co'];\ do\downarrow;\ [\neg di1 \wedge \neg di2];\ ci'\downarrow]\ .$$

As to the implementation of the assignment to x, we now observe that, since C and D are active, there is no risk that the assignment to x be started before C is. The interleaving obtained is:

$$\begin{array}{l} *[[co'];\ do\uparrow;\ [di1 \rightarrow x\uparrow \ \|\ di2 \rightarrow x\downarrow]; \\ \qquad\qquad ci'\uparrow;\ [\neg co'];\ do\downarrow;\ [\neg di1 \wedge \neg di2];\ ci'\downarrow]\ , \end{array} \tag{7.5}$$

which can be factored into the wire

$$(co'\ \underline{w}\ do) \equiv *[[co'];\ do\uparrow;\ [\neg co'];\ do\downarrow]$$

and the register

$$\begin{array}{rl} (di1, di2)\ \underline{reg}\ (x, ci') \equiv & *[\,[di1 \rightarrow x\uparrow;\ [x];\ ci'\uparrow;\ [\neg di1];\ ci'\downarrow \\ & \ \ \|di2 \rightarrow x\downarrow;\ [\neg x];\ ci'\uparrow;\ [\neg di2];\ ci'\downarrow \\ &]\,]\ . \end{array}$$

The implementation of the interface is shown in Figure 7.7.

7.5 Output Actions

In the case of an output, like $out!x$ or $put!x$, the implementation turns out to be the same for passive and active ports. Given the same nomenclature as in the input case, port D is now implemented with two output variables, $do1$ and $do2$, and one input variable di. Port C' is not changed. The rest of the derivation is straightforward and is left as an exercise for the reader. It leads to a wire and a *read* operator, which we have introduced in the implementation of the register.

$$di\ \underline{w}\ ci \equiv\ *[[di];\ ci'\uparrow;\ [\neg di];\ ci'\downarrow]$$

$$\begin{array}{rl} (co', x)\ \underline{read}\ (do1, do2) \equiv & *[\,[x \wedge co' \rightarrow do1\uparrow;\ [\neg co'];\ do1\downarrow \\ & \ \ \|\neg x \wedge co' \rightarrow do2\uparrow;\ [\neg co'];\ do2\downarrow \\ &]\,]\ . \end{array}$$

The only difference between the active and the passive cases is that, in the active case, the *read* is activated first. In the passive case, the wire is activated first. The circuit is shown in Figure 7.8.

7.5.1 Active Input and Passive Output

A somewhat surprising result of this implementation of input and output commands is that, contrary to common belief, it is simpler to implement input commands with active ports than with passive ports. The gain is quite important: For n bits of data, the active implementation saves $2 \times n$ asymmetric

C-elements and n *or*-gates. On the other hand, the implementation of output actions is the same for active and passive ports.

Therefore, we shall always implement input actions with active ports. When the input port is probed, like *in* in the stack example, we shall use a slightly more complicated handshaking protocol that makes it possible to probe an active port. A simple version of this protocol consists of replacing the single passive communication, say *in*, with two communications $in1$ and $in2$, with $in1$ passive and probed, and $in2$ active and used for the input action. The two handshaking expansions are usually interleaved as follows:

$$ini \rightarrow \ldots ino\uparrow;\ [\neg ini];\ ino\downarrow$$

is replaced with

$$in1i \rightarrow \ldots in1o\uparrow;\ [in2i];\ in2o\uparrow;\ [\neg in1i];\ in1o\downarrow;\ [\neg in2i];\ in2o\downarrow$$

(In the implementation of the microprocessor, we have used a more efficient version of this protocol.)

7.6 The Complete Circuit for the Stack

The sharing of register x by ports *in* and *get* has to be implemented either by a multiplexer or by a multiport flip-flop. Since only two ports share the register, we choose to use a dual-port flip-flop. The complete data path is shown in Figure 7.9.

The complete circuit obtained by composing the different parts together is shown in Figure 7.10. An important optimization has been added to the design. It concerns the implementation of the second guard of E:

$$\overline{out} \rightarrow get?x;\ out!x.$$

We observe that the value of x involved in the second action ($out!x$) is the same as the value of x involved in the first action ($get?x$). We can therefore replace it with

$$\overline{out} \rightarrow out!(get?).$$

The handshaking expansion is:

$$\begin{array}{l} outi \rightarrow geto\uparrow;\ [geti1 \rightarrow outo1\uparrow \| geti2 \rightarrow outo2\uparrow];\ [\neg outi]; \\ \qquad geto\downarrow;\ [\neg geti1 \rightarrow outo1\downarrow \| \neg geti2 \rightarrow outo2\downarrow] \end{array}$$

The implementation is the three wires $outi$ $\underline{w}$ $geto$, $geti1$ $\underline{w}$ $outo1$, and $geti2$ $\underline{w}$ $outo2$.

The above modification leads to a significant simplification of the circuit since we can eliminate a D-element, and, *for each bit of the data path*, we

can eliminate an IF-element and replace the multiport flip-flop with a simple flip-flop. The chip we have fabricated includes this modification, as well as the optimization that consists in making input port *in* active.

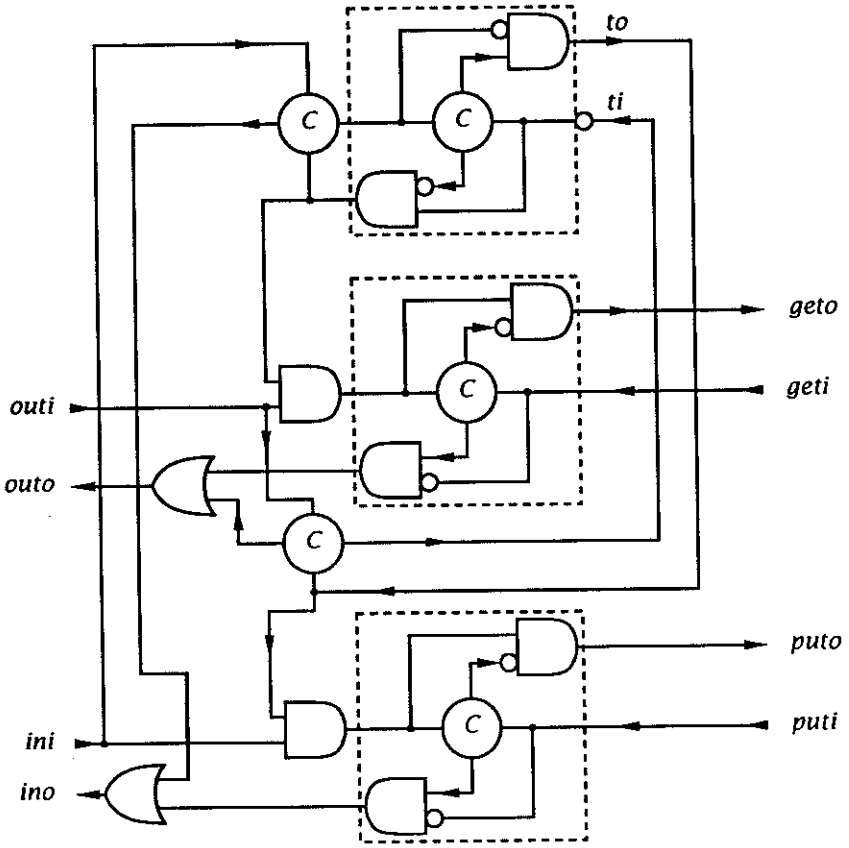

Figure 7.3: The control part of the stack element

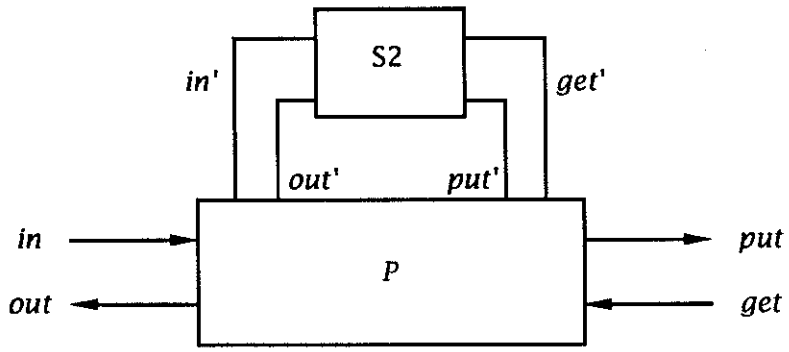

Figure 7.4: Adding the data path

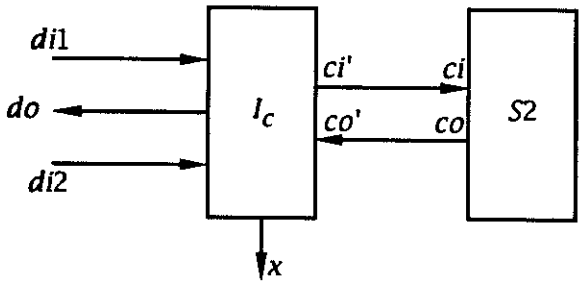

Figure 7.5: Channel interface for input port

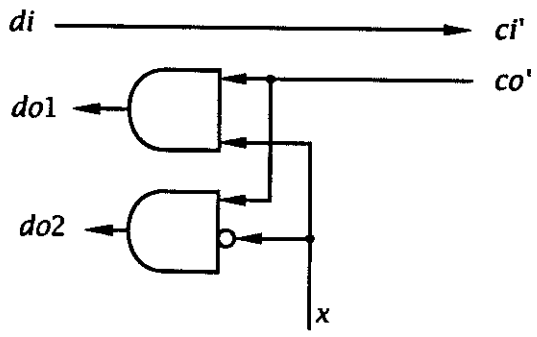

Figure 7.8: Output action interface

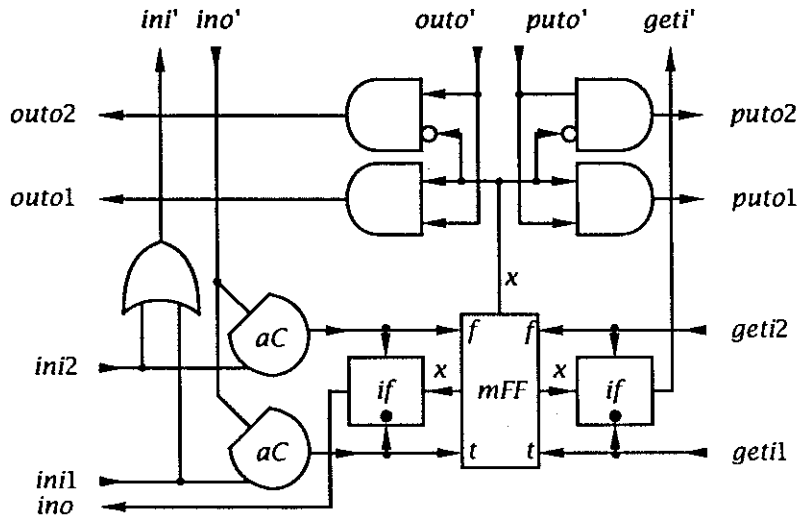

Figure 7.9: The complete data path

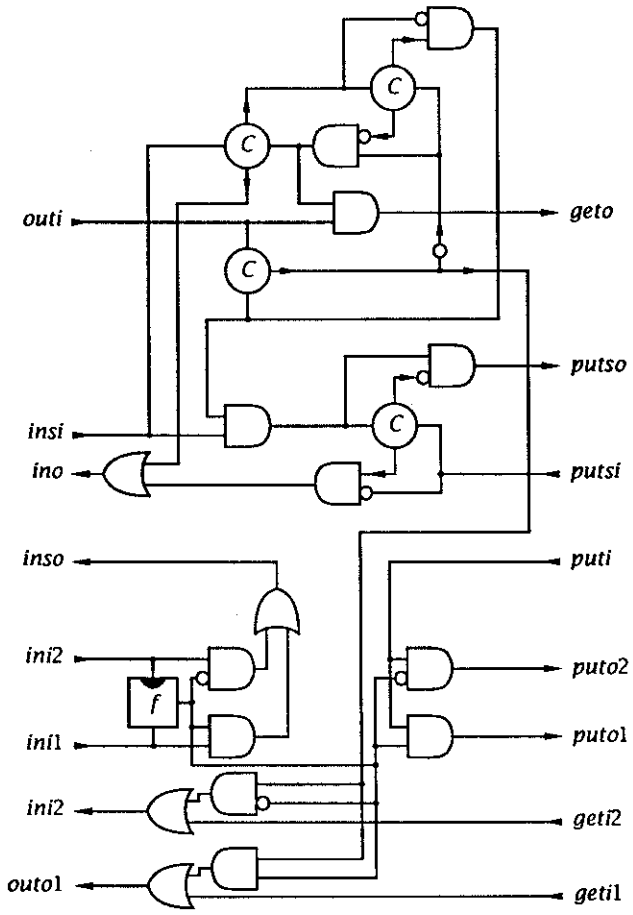

Figure 7.10: The complete circuit for a one-bit stack element

Chapter 8

Asynchronous Adders

8.1 Introduction

The purpose of this chapter is to describe the design of an asynchronous ripple-carry adder as an illustration of the transformations and design decisions that play a role in the construction of asynchronous VLSI circuits for arithmetic functions.

8.2 Function Evaluation

The evaluation of a function, say, $F(X)$, usually appears in a program in the form $Y := F(X)$, *ie*, the function is evaluated and its value is assigned to a variable Y. The first program transformation we perform consists of separating the function evaluation from the assignment. Let P be the program containing the assignment $Y := F(X)$. We apply the transformation:

$$P \triangleright (P_D^{Y:=F(X)} \| (D/C!F(X)) \| *[C?Y])$$

or, alternatively:

$$P \triangleright (P_D^{Y:=F(X)} \| (D/C?Y) \| *[C!F(X)])$$

(C and D are channels introduced for process decomposition. We use the same global name for the two ports of the same channel.)

The two alternative decompositions are equivalent. We leave it to the reader to check that whatever decomposition and handshaking expansion are used, they will contain the process $*[C!F(X)]$ with C passive, or a handshaking expansion equivalent to that process up to the renaming of variable ci.

First, we briefly discuss the general approach to the implementation of this process. The handshaking expansion has to be a generalization of the handshaking expansion of the bare passive communication action C:

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

with ci and co boolean, and the environmemt implementing either the lazy-active protocol:

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

or the usual active protocol. Initially, ci and co are false.

The generalization requires that the single boolean co be replaced with a set C of booleans, and the two assignments $co\uparrow$ and $co\downarrow$ with two multiple assignments to the elements of C, denoted $C\Uparrow$ and $C\Downarrow$, respectively.

The two predicates $\neg co$ and co as used in the wait-actions of the environment have to be replaced by two general predicates $z(C)$—for "zero"—and $v(C)$—for "valid"—, respectively, such that $\neg z(C) \vee \neg v(C)$ is invariantly true. A value of C for which $v(C)$ holds is called a *valid* value. A value of C for which $z(C)$ holds is called a *zero* value.

Furthermore, the two assignments $C\Uparrow$ and $C\Downarrow$ fulfill the requirements:

$$\{z(C)\}C\Uparrow\{v(C)\}$$

and

$$\{v(C)\}C\Downarrow\{z(C)\}\ ,$$

so that the protocol between the process and the environment can now be described by the two handshaking expansions:

$$*[[ci];\ C\Uparrow;\ [\neg ci];\ C\Downarrow]$$

and

$$*[[z(C)];\ ci\uparrow;\ [v(C)];\ ci\downarrow]\ .$$

Initially, ci is false, and $z(C)$ holds.

8.2.1 Delay-Insensitive Codes

The code for C must fulfill the following requirements.

- All values that can be transmitted on the channel, typically all integers from 0 to $2^N - 1$ for some given N, can be coded as valid values, and at least one zero value of C can be coded such that $\neg z(C) \vee \neg v(C)$.
- When C is assigned a valid value by the concurrent assignment $C\Uparrow$, no intermediate value taken by C during the assignment is a valid value. (Otherwise, the wait action $[v(C)]$ of the environment could be completed too early.)

- Symmetrically, when C is assigned a zero value by the concurrent assignment $C\Downarrow$, no intermediate value taken by C during the assignment is a zero value. (Otherwise, the wait action $[z(C)]$ of the environment could be completed too early.)
- Finally, a "side-effect" of the assignment $C\Uparrow$ is to assign to C a (valid) value whose numerical interpretation, say $\nu(C)$, is such that $\nu(C) = F(X)$.

8.2.2 Dual-rail Code

There are many codes for C that fulfil the above requirements. A simple and popular one is the so-called *dual-rail* code. If C represents an N-bit integer, each bit is coded with two boolean variables. Bit c_k is coded with ct_k and cf_k: ct_k is the "true bit" of c_k and is set to true when c_k has to be set to true, cf_k is the "false bit" of c_k and is set to true when c_k has to be set to false.

We have:

$$z(C) \equiv \langle \bigwedge k :: \neg ct_k \wedge \neg cf_k \rangle$$

$$v(C) \equiv \langle \bigwedge k :: ct_k \wedge \neg cf_k \vee cf_k \wedge \neg c_k \rangle$$

$$v(C) \Rightarrow (\forall k :: c_k \equiv ct_k)$$

(In this paper, all quantifications range from 0 to $N-1$, with $N > 0$.) Since $\langle \bigwedge k :: (\neg ct_k \vee \neg cf_k) \rangle$ is maintained as an invariant, $v(C)$ is implied by the simpler condition:

$$\langle \bigwedge k :: ct_k \vee cf_k \rangle \ .$$

Observe that there is only one zero value of C, namely, $\bigwedge k :: \neg ct_k \wedge \neg cf_k$, and there are many values of C that are not valid and not zero.

8.2.3 Stable versus Communicated Inputs

The implementation of $C!F(X)$ described so far relies on the assumption that when ci holds, the input X has the valid value for the evaluation of F and that X is not changed through the evaluation of F. We say that the input is *stable.*

An alternative solution consists in having X being received as a message on channel C, and $F(X)$ being sent as a message on the same channel: C implements the swap of X and $F(X)$. In that case, the input X has to go through the valid/zero cycle and is dual-rail encoded (or encoded with any other delay-insensitive code). We say that the input is *communicated.* The handshaking expansion of $C!F(X)$ is of the form:

$$*[[v(X)];\ C\Uparrow;\ [z(X)];\ C\Downarrow] \ . \tag{8.1}$$

The handshaking expansion of 8.1 requires that all boolean inputs of X be valid before any elementary assignment of $C \Uparrow$ is started, and that all boolean inputs of X be zero before any elementary assignment of $C \Downarrow$ is started. Such an ordering requirement is unnecessarily strong. The following weaker requirement—which we call the *weak handshake rule for communicated input*—is sufficient:

For each boolean x of input X, there is at least one elementary assignment $c\uparrow$ of $C \Uparrow$ such that $\{v(x)\}c\uparrow$, and there is at least one elementary assignment $c'\downarrow$ of $C \downarrow$ such that $\{z(x)\}c'\downarrow$.

Hence, each boolean input variable x is part of the hanshaking sequence:

$$*[[v(x)];\ c\uparrow;\ [z(x)];\ c'\downarrow]\ . \tag{8.2}$$

In the following implementation of the addition, the operands of the addition are assumed to be stable but the carry inputs are communicated.

8.3 Binary Addition

We want to implement the process $*[S!(A+B)]$. Its handshaking expansion is:

$$*[[si];\ S \Uparrow \{\nu(S) = \nu(A) + \nu(B)\};\ [\neg si];\ S \Downarrow]\ . \tag{8.3}$$

A, B, and S are N-bit integers. Inputs A and B are assumed to be stable, but the sum S is dual-rail encoded.

Next, we need to refine the postcondition $\nu(S) = \nu(A) + \nu(B)$ in terms of relations between each bit of S and the corresponding bits of A and B. There are many ways to describe these relations, each corresponding to a particular addition algorithm. Here, we choose the algorithm that is usually called "ripple-carry adder."

8.3.1 Ripple-carry Addition

The value of bit s_k of S can be expressed as a function of the bits a_k and b_k of A and B, and of the carry-in bit c_k. More precisely, the postcondition of the addition can be expressed as:

$$\neg c_0 \wedge (\forall k :: sum_k)$$

where each sum_k is the conjunction of the three predicates:

$$(\neg a_k \wedge \neg b_k) \Rightarrow (s_k, c_{k+1} = c_k, false)$$

$$(a_k \wedge b_k) \Rightarrow (s_k, c_{k+1} = c_k, true)$$

$$(a_k \neq b_k) \Rightarrow (s_k, c_{k+1} = \neg c_k, c_k)$$

The computation of bit s_k of the sum requires the previous computation of carry bit c_k and therefore also produces carry bit c_{k+1}. Hence, the carry bits also have to be dual-rail encoded and used as communicated variables, *ie* we will have to add the waits $[v(c_k)]$ and $[z(c_k)]$ in the handsaking expansion.

We can easily design the program add_k that establishes sum_k. Its inputs are a_k and b_k, and the carry-in bits ct_k and cf_k. Its outputs are st_k and sf_k and the carry-out bits ct_{k+1} and cf_{k+1}. We get:

$$\begin{array}{l} add_k \equiv [\,\neg a_k \wedge \neg b_k \rightarrow ([ct_k \rightarrow st_k\uparrow \| cf_k \rightarrow sf_k\uparrow]) \parallel cf_{k+1}\uparrow \\ \quad [\!]\, a_k \wedge b_k \rightarrow ([ct_k \rightarrow st_k\uparrow \| cf_k \rightarrow sf_k\uparrow]) \parallel ct_{k+1}\uparrow \\ \quad [\!]\, a_k \neq b_k \rightarrow [ct_k \rightarrow sf_k\uparrow, ct_{k+1}\uparrow \| cf_k \rightarrow st_k\uparrow, cf_{k+1}\uparrow] \\ \quad] \end{array}$$

(The comma is used as an alternative to $\parallel$ for the parallel composition of simple assignments.)

8.3.2 Handshaking Expansion

We now replace $S\Uparrow$ with add_k in (8.3) and implement $S\Downarrow$. This refinement gives:

$$*[[si];\ \langle \| k :: add_k \rangle;\ [\neg si];\ \langle \| k :: st_k\downarrow, sf_k\downarrow, ct_{k+1}\downarrow, cf_{k+1}\downarrow \rangle]\ . \qquad (8.4)$$

Furthermore, the first carry-in bits are generated by the program:

$$\begin{array}{l} *[\,[si \rightarrow ct_0\downarrow, cf_0\uparrow \\ \quad [\!]\, \neg si \rightarrow ct_0\downarrow, cf_0\downarrow\ . \\ \quad]] \end{array}$$

Next we have to enforce the weak handshake rule of (8.2) for the inputs c_k. A straightforward solution is:

$$*[[si];\ \langle \| k :: [v(c_k)]; add_k \rangle;\ [\neg si \wedge (\textstyle\bigwedge k :: z(c_k))];\ \langle \| k :: st_k\downarrow, sf_k\downarrow, ct_{k+1}\downarrow, cf_{k+1}\downarrow \rangle]\ . \qquad (8.5)$$

Unfortunately, this solution is entirely sequential since the expressions $v(c_k)$ become true in the order of increasing k. (Observe that including the waits for $v(c_k)$ in the wait for si, as $[si \wedge (\bigwedge k :: v(c_k))]$, would result in a deadlock since only $v(c_0)$ holds initially.) But, we can use the fact that the computation of each bit s_k in add_k requires that $v(c_k)$ hold. We can therefore remove the explicit wait $[v(c_k)]$ from (8.5) and still fulfill the weak handshake rule. Now, the upgoing part of the computation of a carry bit can proceed without waiting for the previous carry bit when $a_k = b_k$. Concurrency in the computation of the carry bits also introduces concurrency in the computation of the sum bits. However, the downgoing part of the computation of the

carry bits is still sequential since the wait for $(\bigwedge k :: z(c_k))$ still precedes all downgoing assignments. We improve this part as follows.

We first apply a transformation rule that takes the parallel quantification $\|k :: \text{out}$ of the process. This transformation results in replacing the single process with N parallel processes:

$$\langle \|k :: *[[si];\ add_k;\ [\neg si \wedge z(c_k)];\ st_k\downarrow, sf_k\downarrow, ct_{k+1}\downarrow, cf_{k+1}\downarrow]\rangle\ . \qquad (8.6)$$

Second, we replace the downgoing sequence of (8.6)

$$[\neg si \wedge z(c_k)];\ st_k\downarrow, sf_k\downarrow, ct_{k+1}\downarrow, cf_{k+1}\downarrow$$

with the sequence

$$[\neg si \rightarrow ct_{k+1}\downarrow, cf_{k+1}\downarrow] \parallel [\neg ct_k \wedge \neg cf_k \rightarrow st_k\downarrow, sf_k\downarrow]\ ,$$

in which we have implemented $z(c_k)$ as $\neg ct_k \wedge \neg cf_k$. The weak handshake rule is still obeyed. We get the final handshaking expansion:

$$\langle \|k :: *[[si];\ add_k;\ ([\neg si \rightarrow ct_{k+1}\downarrow, cf_{k+1}\downarrow] \parallel \neg ct_k \wedge \neg cf_k \rightarrow st_k\downarrow, sf_k\downarrow])]\rangle\ . \qquad (8.7)$$

Now, the downgoing part of the carry-out generation can proceed without waiting for the carry-in; and, as we mentioned before, the upgoing part of the carry-out generation can proceed without waiting for the carry-in when $a_k = b_k$. This optimization of the carry-chain length is the main characteristic of this type of adders.

8.4 Implementation of the Adder Cells

Each program of (8.7) is called an *adder-cell.* For the rest of the implementation of the adder-cells, we can omit the subscripts k and $k+1$. The input variables are a, b, and ct and cf for the carry-in bits. The output variables are st and sf for the sum bits, and dt and df for the carry-out bits.

We first simplify the program of *add* by combining the guards and factoring the parallel composition. We get:

$$\begin{array}{l} add \equiv (\,[\,(\neg a \wedge \neg b) \vee ((a \neq b) \wedge cf) \rightarrow df\uparrow \\ \quad [\!]\ (a \wedge b) \vee ((a \neq b) \wedge ct) \rightarrow dt\uparrow \\ \quad] \\ \quad \|[\,ct \wedge (a = b) \vee cf \wedge (a \neq b) \rightarrow st\uparrow \\ \quad [\!]\ cf \wedge (a = b) \vee ct \wedge (a \neq b) \rightarrow sf\uparrow \\ \quad] \\ \quad) \end{array}$$

The complete program for an adder-cell is:

$$*[[si];\ add;\ ([\neg si \rightarrow dt\downarrow, df\downarrow] \parallel [\neg ct \wedge \neg cf \rightarrow st\downarrow, sf\downarrow])] \tag{8.8}$$

Next, we include the wait $[si]$ into the guards of *add*, *ie* a guard G becomes $G \wedge si$.

The program of an adder-cell becomes:

$$\begin{aligned} adder - cell \equiv *[([\,&(si \wedge \neg a \wedge \neg b) \vee ((a \neq b) \wedge cf) \rightarrow df\uparrow \\ &[\!]\ (si \wedge a \wedge b) \vee ((a \neq b) \wedge ct) \rightarrow dt\uparrow \\ &]\,] \\ \|[\,&ct \wedge (a = b) \vee cf \wedge (a \neq b) \rightarrow st\uparrow \\ &[\!]\ cf \wedge (a = b) \vee ct \wedge (a \neq b) \rightarrow sf\uparrow \\ &]\,] \\ &);\ ([\neg si \rightarrow dt\downarrow, df\downarrow] \parallel [\neg ct \wedge \neg cf \rightarrow st\downarrow, sf\downarrow]) \\ ]& \end{aligned}$$

We have optimized this transformation by adding si only in the terms of the guards of *add* that do not contain ct or cf since we can prove that

$$(ct_k \Rightarrow si) \wedge (cf_k \Rightarrow si) \tag{8.9}$$

holds for all k. The proof of (8.9) is by induction on k: For the base case $k = 0$, (8.9) holds obviously because of the program:

$$\begin{aligned} *[\,&[si \mapsto ct_0\downarrow, cf_0\uparrow \\ &[\!]\neg si \mapsto ct_0\downarrow, cf_0\downarrow\ . \\ &]\,] \end{aligned}$$

For the induction case, we prove that if (8.9) holds for k it holds for $k+1$. The structure of the guarded commands is such that $(dt \Rightarrow (si \vee ct)) \wedge (df \Rightarrow (si \vee cf))$ holds as a postcondition of *adder-cell*. But since, by the induction hypothesis, $(ct \Rightarrow si) \wedge (cf \Rightarrow si)$ holds for k, we have established $(dt \Rightarrow si) \wedge (df \Rightarrow si)$. Since dt for cell k is ct for cell $k+1$, and similarly for df and cf, (8.9) is established for $k+1$.

8.4.1 Production-rule Expansion

We add a minor modification: The guards of $df\uparrow$ and $dt\uparrow$ are equivalent to (and can be replaced with):

$$(si \wedge \neg a \wedge \neg b) \vee (\neg a \vee \neg b) \wedge cf$$

and

$$(si \wedge a \wedge b) \vee (a \vee b) \wedge ct$$

respectively. The production-rule expansion is now straightforward:

$$\begin{aligned}
(si \wedge \neg a \wedge \neg b) \vee cf \wedge (\neg a \vee \neg b) &\mapsto df\uparrow \\
(si \wedge a \wedge b) \vee ct \wedge (a \vee b) &\mapsto dt\uparrow \\
\neg si &\mapsto dt\downarrow, df\downarrow \\
(ct \wedge a = b) \vee (cf \wedge a \neq b) &\mapsto st\uparrow \\
(cf \wedge a = b) \vee (ct \wedge a \neq b) &\mapsto sf\uparrow \\
\neg ct \wedge \neg cf &\mapsto st\downarrow, sf\downarrow
\end{aligned}$$

8.5 Implementation Issues

The CMOS gates for dt and st are shown in Figure 8.1. We use dynamic logic for these state-holding gates since there is no data-dependent delays between the upgoing and downgoing transitions. As usual, the logic is inverting, and thus the gates produce the complementary signals $dt\_$ and $st\_$ of dt and st. Adding an inverter at the output of each gate that produces $dt\_$ is an expensive solution since the carry chain may include up to N inverters in series in addition to the N carry gates. A better solution consists in alternating gates that produce $dt\_$ and $df\_$ (the even-numebred bits) with gates that produce dt and df (the odd-numbered bits).

Finally, we can simplify the design of the adder-cell 0: We can eliminate inputs ct_0 and cf_0 since ct_0 is identically false and $cf_0 = si$. The simplified production rules are:

$$\begin{aligned}
si \wedge (\neg a \vee \neg b) &\mapsto df\uparrow \\
si \wedge (a \vee b) &\mapsto dt\uparrow \\
\neg si &\mapsto dt\downarrow, df\downarrow \\
cf \wedge (a \neq b) &\mapsto st\uparrow \\
cf \wedge (a = b) &\mapsto sf\uparrow \\
\neg cf &\mapsto st\downarrow, sf\downarrow
\end{aligned}$$

Acknowledgments

Acknowledgement is due to my student Tony Lee for his comments and for designing several beautiful asynchronous ALUs that were an inspiration for this paper.

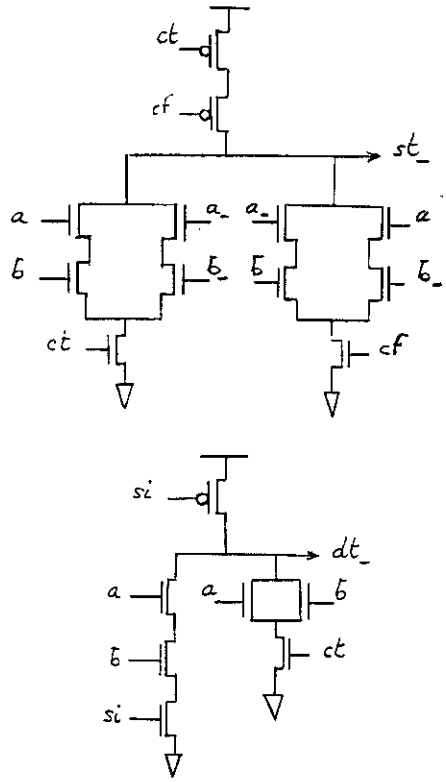

Figure 8.1: CMOS implementation of the true bits of the sum and carry

Chapter 9

The First Asynchronous Microprocessor

9.1 Introduction

In this chapter, we describe a delay-insensitive microprocessor my students and I designed at Caltech in the fall of 1988. It is the first delay-insensitive or even asynchronous microprocessor ever designed. It is a 16-bit, RISC-like architecture. The version implemented in 1.6 micron SCMOS runs at 18 MIPS. The chips were found functional on "first silicon."

As we explained in Section 2.8.6, the processor was first specified as a sequential program, which was then transformed into a concurrent program so as to pipeline instruction execution. The circuits were derived from the concurrent program by semantics-preserving program transformation.

The design was undertaken as a large-scale application of the high-level synthesis method for asynchronous VLSI that we have developed in these notes.

The results of the experiment can be summarized as follows. First, it is possible and advantageous to describe circuits, even of the size and complexity of a microprocessor, in a high-level program notation. With the exception of the ALU function, the complete program takes less less than two pages—let us say that a complete description including all functions would take approximately three pages. The transformations performed on the initial sequential program to introduce pipelining show that the notation is appropriate for a designer to work with efficiently, since all important design decisions can be made at the level of source code.

Second, it is possible to derive the circuit from the program by applying systematic semantics-preserving transformations, and to obtain a circuit that

is correct on first silicon. The compilation procedure is not described here, but can be found in several papers, in particular [6].

Third, the results of the experiment demonstrate that the often accepted "fatalities," that formal design methods and asynchronous techniques lead to inefficient solutions, are simply myths fueled by the natural resistance to change. Not only is the processor surprisingly small and fast for a first design, but it also exhibits a robustness to parameter variations that goes beyond our expectations and almost beyond our understanding: One of the two versions seems still to function with a voltage value of 0.35V for the VDD! Maybe the biggest surprise is the very low power consumption of the chips, which makes this design style ideally suited for use in highly concurrent architectures where a large number of chips are tightly packed.

9.2 The Processor: The Test Results

The processor has a 16-bit, RISC-like instruction set. It has sixteen registers, four buses, an ALU, and two adders. Instruction and data memories are separate. The chip size is about 20,000 transistors. Two versions have been fabricated: one in $2\mu m$ MOSIS SCMOS, and one in $1.6\mu m$ MOSIS SCMOS. (The dimension refers to the minimal width of a wire.) On the $2\mu m$ version, only twelve registers were implemented in order to fit the chip on the 84-pin $6600\mu m \times 4600\mu m$ pad frame.

With the exception of *isochronic forks*, the chips are entirely delay-insensitive. The circuits use neither clocks nor knowledge about delays. The only exception to the design method is the interface with the memories. In the absence of available memories with asynchronous interfaces, we have simulated the completion signal from the memories with an external delay. For testing purposes, the delay on the instruction memory interface is variable.

In spite of the presence of several floating n-wells, the $2\mu m$ version runs at 12 MIPS. The $1.6\mu m$ version runs at 18 MIPS. (Those performance figures are based on measurements from sequences of ALU instructions without carry. They do not take advantage of the overlap between ALU and memory instructions.) Those performances are quite encouraging given that the design is very conservative: It uses static gates, dual-rail encoding of data, completion trees, *etc.*

Only two of the 12 $2\mu m$ chips passed all tests, but 34 of the 50 $1.6\mu m$ chips were found to be functional. (However, within a certain range of values for the instruction memory delay, the $1.6\mu m$ version malfunctions. We will return to this phenomenon, which is related to the implementation of isochronic forks.) It takes less than 700 instructions to test the processors for stuck-at faults. The program counter is the only part that was not tested exhaustively because the memory used for the test did not contain the address required for testing

the most significant bit of the program counter.

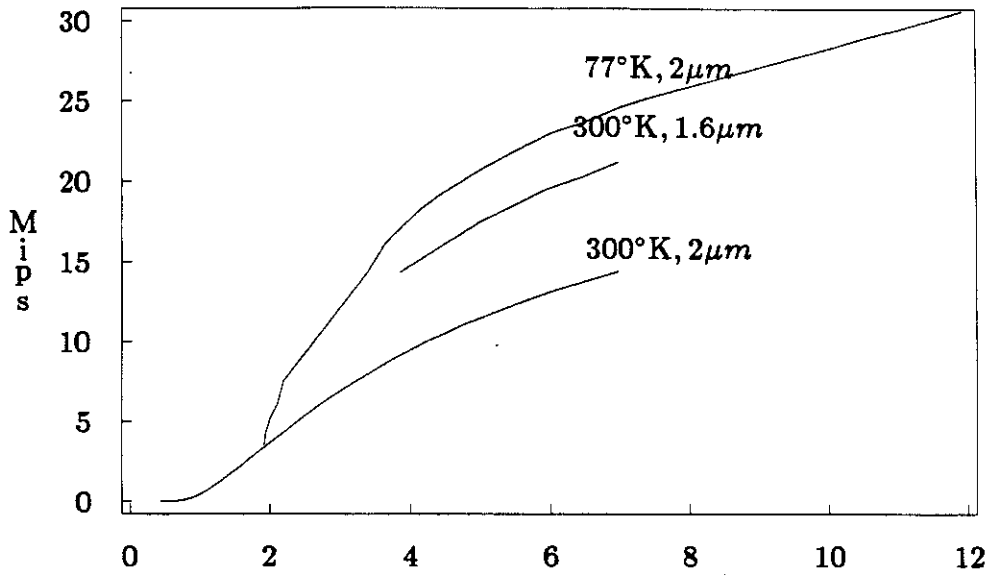

Figure 9.1: MIPS as a function of VDD

We have tested the chips under a wide range of *VDD* voltage values. At room temperature, the $2\mu m$ version is functional in a voltage range from 7V down to 0.35V! And it reaches 15 MIPS at 7V. We have also tested the chips cooled in liquid nitrogen. The $2\mu m$ version reaches 20 MIPS at 5V and 30 MIPS at 12V. The $1.6\mu m$ version reaches 30 MIPS at 5V. Of course, the measurements are made without adjusting any clocks (there are none), but simply by connecting the processor to a memory containing a test program and observing the rate of instruction execution. The results are summarized in Figure 9.1. The power consumption is 145mW at 5V and 6.7mW at 2V.

9.3 Specification of the processor

The instruction set is deliberately not innovative. It is a conventional 16-bit-word instruction set of the *load-store* type. The processor uses two separate memories for instructions and data. There are three types of instructions: ALU, memory, and program-counter (*pc*). All ALU instructions operate on registers; memory instructions involve a register and a data memory word. Certain instructions use the following word as *offset*. The only important omissions, those of an interrupt mechanism and communication ports, are ones we found to be unnecessary distractions in a first design.

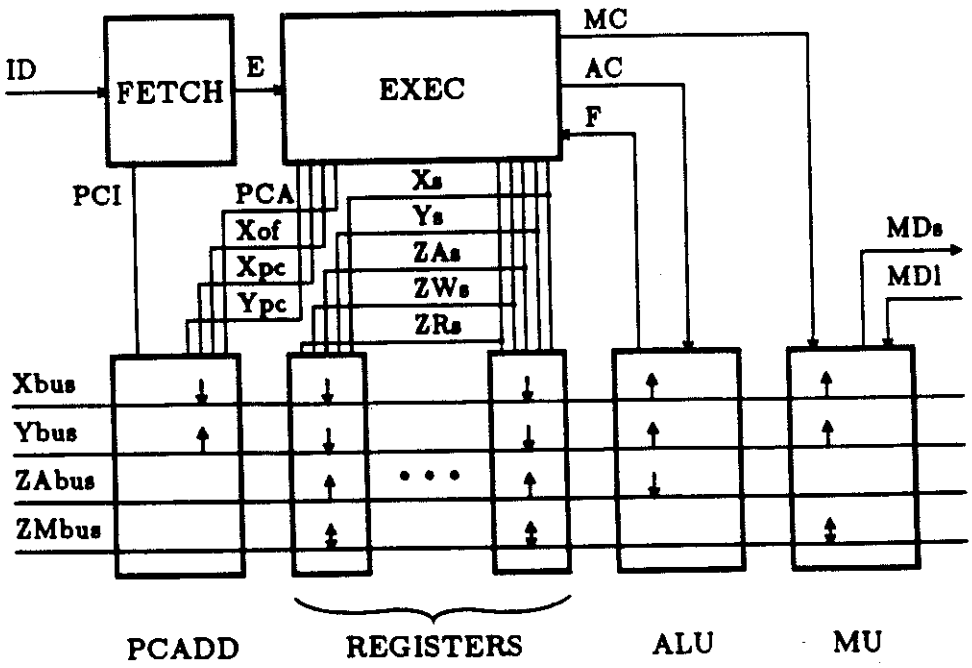

Figure 9.2: Process and channel structure

9.4 Decomposition into Concurrent Processes

The program of Section 2.8.6 is further decomposed into a set of concurrent processes. In this program we have used a restricted form of shared variables. The control channels *Xs*, *Ys*, *ZAs*, *ZWs*, *ZRs*, and the bus *ZA* are one-to-many; the buses *X*, *Y*, *ZM* are many-to-many; the other channels are one-to-one. But all channels are used by only two processes at a time. The structure of processes and channels is shown in Figure 9.2. The final program is shown in Figures 9.3 and 9.4. Process *FETCH* fetches the instructions from the instruction memory, and transmits them to process *EXEC* which decodes them. Process *PCADD* updates the address *pc* of the next instruction concurrently with the instruction fetch, and controls the *offset* register. The execution of an ALU instruction by process *ALU* can overlap with the execution of a memory instruction by process *MU*. The *jump* and *branch* instructions are executed by *EXEC*; *store-pc* is executed by the ALU as the

instruction "add the content of register r to the pc and store it." The array $REG[k]$ of processes implements the register file. Both *MU* and *PCADD* contain their own adder. Processes *IMEM* and *DMEM* describe the instruction memory and data memory, respectively.

9.4.1 Updating the PC

The variable *pc* is updated by process *PCADD*, and is used by *IMEM* as the index of the array *imem* during the *ID* communication—the instruction fetch.

The assignment $pc := pc+1$ is decomposed into $y := pc+1; pc := y$, where y is a local variable of *PCADD* . The overlap of the instruction fetch, *ID*? (either $ID?i$ or $ID?offset$), and the pc increment, $y := pc+1$, can now occur while *pc* is constant. Action *ID*? is enclosed between the two communication actions *PCI1* and *PCI2*, as follows:

$$PCI1; ID?i; PCI2 .$$

In *PCADD*, $y := pc + 1$ is enclosed between the same two communication actions while the updating of *pc* follows *PCI2*:

$$\overline{PCI1} \rightarrow PCI1; y := pc + 1; PCI2; pc := y .$$

Since the completions of *PCI1* and *PCI2* in *FETCH* coincide with the completion of *PCI1* and *PCI2* in *PCADD*, respectively, the execution of $ID?i$ in *FETCH* overlaps the execution of $y := pc + 1$ in *PCADD*. *PCI1* and *PCI2* are implemented as the two halves of the same communication handshaking to minimize the overhead.

In order to concentrate all increments of *pc* inside *PCADD*, we use the same technique to delegate the assignment $pc := pc + offset$ (executed by the *EXEC* part in the sequential program) to *PCADD*.

The guarded command $\overline{Xof} \rightarrow X!offset \bullet Xof$ in *PCADD* has been transformed into a concurrent process since it needs only be mutually exclusive with assignment $y := pc+ offset$, and this mutual exclusion is enforced by the sequencing between $PCA1; PCA2$ and *Xof* within *EXEC*.

9.5 Stalling the Pipeline

When the pc is modified by *EXEC* as part of the execution of a *pc* instruction, (*store-pc*, *jump* or *branch*), fetching the next instruction by *FETCH* is postponed until the correct value of the pc is assigned to *PCADD.pc*.

When the offset is reserved for *MU* by *EXEC*, as part of the execution of some memory instructions, fetching the next instruction, which might be a new offset, is postponed until *MU* has received the value of the current offset.

$$
\begin{array}{ll}
IMEM \equiv & *[ID!imem[pc]] \\
FETCH \equiv & *[\, PCI1; ID?i; PCI2; \\
& \quad [offset(i.op) \rightarrow PCI1; ID?offset; PCI2 \\
& \quad \|\neg offset(i.op) \rightarrow skip \\
& \quad]; E1!i; E2 \\
& \quad] \\
PCADD \equiv & \quad (*[[\, \overline{PCI1} \rightarrow PCI1; y := pc+1; PCI2; pc := y \\
& \qquad \|\overline{PCA1} \rightarrow PCA1; y := pc + offset; PCA2; pc := y \\
& \qquad \|\overline{Xpc} \rightarrow X!pc \bullet Xpc \\
& \qquad \|\overline{Ypc} \rightarrow Y?pc \bullet Ypc \\
& \qquad]] \\
& \quad \|*[[\, \overline{Xof} \rightarrow X!offset \bullet Xof]] \\
& \quad) \\
EXEC \equiv & \quad *[\, E1?j; \\
& \qquad [alu(j.op) \rightarrow E2; Xs \bullet Ys \bullet AC!j.op \bullet ZAs \\
& \qquad \|ld(j.op) \rightarrow E2; Xs \bullet Ys \bullet MC1 \bullet ZRs \\
& \qquad \|st(j.op) \rightarrow E2; Xs \bullet Ys \bullet MC2 \bullet ZWs \\
& \qquad \|ldx(j.op) \rightarrow Xof \bullet Ys \bullet MC1 \bullet ZRs; E2 \\
& \qquad \|stx(j.op) \rightarrow Xof \bullet Ys \bullet MC2 \bullet ZWs; E2 \\
& \qquad \|lda(j.op) \rightarrow Xof \bullet Ys \bullet MC3 \bullet ZRs; E2 \\
& \qquad \|stpc(j.op) \rightarrow Xpc \bullet Ys \bullet AC!add \bullet ZAs; E2 \\
& \qquad \|jmp(j.op) \rightarrow Ypc \bullet Ys; E2 \\
& \qquad \|brch(j.op) \rightarrow F?f; \quad [cond(f, j.cc) \rightarrow PCA1; PCA2 \\
& \qquad\qquad\qquad\qquad\quad \|\neg cond(f, j.cc) \rightarrow skip \\
& \qquad\qquad\qquad\qquad\quad]; E2 \\
& \quad]]
\end{array}
$$

Figure 9.3: The final program, first part

$$
\begin{array}{l}
ALU \equiv\ *[[\overline{AC} \rightarrow\ AC?op \bullet X?x \bullet Y?y; \\
\qquad\qquad \langle z, f\rangle := aluf(x, y, op, f); ZA!z \\
\qquad [\overline{F} \rightarrow F!f \\
\qquad]] \\
MU\ \equiv *[[\overline{MC1} \rightarrow X?x \bullet Y?y \bullet MC1; ma := x + y; MDl?w; ZM!w \\
\qquad [\overline{MC2} \rightarrow X?x \bullet Y?y \bullet MC2 \bullet ZM?w; ma := x + y; MDs!w \\
\qquad [\overline{MC3} \rightarrow X?x \bullet Y?y \bullet MC3; ma := x + y; ZM!ma \\
\qquad]] \\
DMEM \equiv *[[\overline{MDl} \rightarrow MDl!dmem[ma] \\
\qquad\qquad [\overline{MDs} \rightarrow MDs?dmem[ma] \\
\qquad\qquad]] \\
REG[k] \equiv (*[[\neg bk \wedge k = j.x \wedge \overline{Xs} \rightarrow X!r \bullet Xs]] \\
\qquad [*[[\neg bk \wedge k = j.y \wedge \overline{Ys} \rightarrow Y!r \bullet Ys]] \\
\qquad [*[[\neg bk \wedge k = j.z \wedge \overline{ZWs} \rightarrow ZM!r \bullet ZWs]] \\
\qquad [*[[\neg bk \wedge k = j.z \wedge \overline{ZAs} \rightarrow bk\uparrow; ZAs; ZA?r; bk\downarrow]] \\
\qquad [*[[\neg bk \wedge k = j.z \wedge \overline{ZRs} \rightarrow bk\uparrow; ZRs; ZM?r; bk\downarrow]] \\
\qquad)
\end{array}
$$

Figure 9.4: The final program, second part

In the second design, we have refined the protocol to block *FETCH* only when the next instruction is a new offset.

Postponing the start of the next cycle in *FETCH* is achieved by postponing the completion of the previous cycle, i.e., by postponing the completion of the communication action on channel E. As in the case of the PCI communication, E is decomposed into two communications, $E1$ and $E2$. Again, $E1$ and $E2$ are implemented as the two halves of the same handshaking protocol.

In *FETCH*, $E!i$ is replaced with $E1!i; E2$. In *EXEC*, $E2$ is postponed until after either *Xof?offset* or a complete execution of a *pc* instruction has occurred.

9.6 Sharing Registers and Buses

A bus is used by two processes at a time, one of which is a register and the other is *EXEC*, *MU*, *ALU*, or *PCADD*. We therefore decided to introduce enough buses so as not to restrict the concurrent access to different registers. For instance, *ALU* writing a result into a register should not prevent *MU* from using another register at the same time.

The four buses correspond to the four main concurrent activities involving

the registers. The X bus and the Y bus are used to send the parameters of an ALU operation to the ALU, and to send the parameters of address calculation to the memory unit. We also make opportunistic use of them to transmit the pc and the offset to and from *PCADD*.

The ZA bus is used to transmit the result of an ALU operation to the registers. The ZM bus is used by the memory unit to transmit data between the data memory and the registers.

We make a virtue out of necessity by turning the restriction that registers can be accessed only through those four buses into a convenient abstraction mechanism. The ALU uses only the X, Y, and ZA ports without having to reference the particular registers that are used in the communications. It is the task of *EXEC* to reserve the X, Y, and ZA bus for the proper registers before the ALU uses them.

The same holds for the *MU* process, which references only X, Y, and ZM. An additional abstraction is that the X bus is used to send the offset to *MU*, so that the cases for which the first parameter is $i.x$ or *offset* are now identical, since both parameters are sent via the X bus.

9.6.1 Exclusive Use of a Bus

Commands *Xpc*, *Ypc*, and *Xof* are used by *EXEC* to select the X and Y buses for communication of *pc* and *offset*. Commands *Xs*, *Ys*, and *ZAs* are used by *EXEC* to select the X, Y, and ZA buses, respectively, for a register that has to communicate with the ALU as part of the execution of an ALU instruction.

Two commands are needed to select the ZM bus: ZWs if the bus is to be used for writing to the data memory, and ZRs if the bus is to be used for reading from the data memory.

Let us first solve the problem of the mutual exclusion among the different uses of a bus. As long as we have only one ALU and one memory unit, no conflict is possible on the ZA and ZM buses, since only the ALU uses the ZA bus, and only the memory unit uses the ZM bus. But the X and Y buses are used concurrently by the ALU, the memory unit, and the pc unit.

We achieve mutual exclusion on different uses of the X bus as follows. (The same argument holds for Y.) The completion of an X communication is made to coincide with the completion of one of the selection actions Xs, Xof, Xpc; and the occurrences of these selection actions exclude each other in time inside *EXEC* since they appear in different guarded commands.

This coincidence is implemented by the *bullet* command: We recall that, for arbitrary communication commands U and V inside the same process, $U \bullet V$ guarantees that the two actions are completed at the same time. We then say that the two actions coincide. The use of the bullets $X!pc \bullet Xpc$ and $X!offset \bullet Xof$ inside *PCADD* , and $X!r \bullet Xs$ inside the registers enforces the coincidence of X with *Xpc*, *Xof*, and *Xs*, respectively. The bullets in *EXEC*,

ALU, and *MU* have been introduced for reasons of efficiency: Sequencing is avoided.

9.7 Register Selection

Command Xs in *EXEC* selects the X bus for the particular register whose index k is equal to the field $i.x$ of the instruction i being decoded by *EXEC*, and analogously for commands Ys, ZAs, ZRs, and ZWs.

Each register process $REG[k]$, for $0 \leq k < 16$, consists of five elementary processes, one for each selection command. The register that is selected by command Xs is the one that passes the test $k = i.x$. This implementation requires that the variable $i.x$ be shared by all registers and *EXEC*. An alternative solution that does not require shared variables uses demultiplexer processes. (The implementations of the two solutions are almost identical.)

The semicolons in the last two guarded commands of $REG[k]$ are introduced to pipeline the computation of the result of an ALU instruction or memory instruction with the decoding of the next instruction.

9.7.1 Mutual Exclusion on Registers

A register may be used in several arguments (x, y, or z) of the same instruction, and also as an argument in two successive instructions whose executions may overlap. We therefore have to address the issue of the concurrent uses of the same register. Two concurrent actions on the same register are allowed when they are both read actions.

Concurrency within an instruction is not a problem: X and Y communications on the same register may overlap, since they are both read actions, and Z cannot overlap with either X or Y because of the sequencing inside *ALU* and *MU*.

Concurrency in the access to a register during two consecutive overlapping instructions (one instruction is an ALU and the other is a memory instruction) can be a problem: Writing a result into a register (a ZA or a ZR action) in the first instruction can overlap with another action on the same register in the second instruction. But, because the selection of the z register for the first instruction takes place before the selection of the registers for the second instruction, we can use this ordering to impose the same ordering on the different accesses to the same register when a ZA or ZR is involved.

This ordering is implemented as follows: In $REG[k]$, variable bk (initially false) is set to true before the register is selected for ZA or ZR, and it is set back to false only after the register has been actually used. All uses of the register are guarded with the condition $\neg bk$. Hence, all subsequent selections of the register are postponed until the current ZA or ZR is completed.

We must ensure that *bk* is not set to true before the register is selected for an X or a Y action *inside the same instruction*, since this would lead to deadlock. We omit this refinement which does not appear in the program of Figures 9.3 and 9.4.

9.8 Conclusion

Instruction pipelining has been approached as a concurrent programming problem: Starting with a sequential program for the processor, concurrency is introduced through a series of program transformations. However, although the transformations are guided by the intent to overlap the important phases—fetch, decode, execute—of instruction execution, they are neither mechanical nor unique. The designer decides how to decompose a program into several concurrent ones. We do not claim that our solution in this first design is in any way optimal.

Since the choice of an instruction set was not part of the experiment, our design should be judged in two ways: the choice of the concurrent program of Figures 9.3 and 9.4, and its implementation. The implementation, which is described in [7], is satisfactory, but not optimal. The sizing of transistors can be improved and the number of transitions can be decreased, mainly by a better placement of inverters. For instance, the delays due to the control for a buffer are both about twice their theoretical minimum.

The program represents the choice of a pipeline, and of synchronization techniques to implement it. We have deliberately chosen a simple pipeline. In particular, the mechanism for stalling, which places part of the decoding in series with the fetch on the critical path, sacrifices efficiency for simplicity. However, performance evaluations show that the pipeline is well-balanced since the different stages have comparable average delays. Improving the critical path by overlapping fetch and decode requires improving the ALU and memory instruction execution stages by pipelining parts of these stages.

The practicality of overlapping ALU and memory instruction executions remains an open issue. It is not clear whether the gain in performance is worth the complexity of the synchronization involved and the requirement of two separate Z buses.

We find the synchronization techniques used to implement the concurrent activities between the different stages of the pipeline particularly elegant and efficient, since the delays incurred in a synchronization can be of arbitrary length and vary from instruction to instruction.

We foresee excellent performances for asynchronous processors as the feature size keeps decreasing. But the designer must be ready to use new methods based on concurrent programmming, in order to exploit asynchronous techniques to their fullest.

Chapter 10

The Limitations to Delay-Insensitivity

10.1 Introduction

In this chapter, we characterize the class of circuits that are entirely DI, and we show that this class is surprisingly limited: Practically all circuits of interest fall outside the class since closed circuits inside the class may contain only C-elements as multiple-input operators.

We prove that all DI circuits have to fulfill the so-called *Unique-Successor-Set* criterion; and we show that the class of circuits that meet this criterion is very limited. We also give a characterization of the class of computations that admit a DI implementation. Finally, we discuss what we consider to be the weakest compromise to delay-insensitivity, namely, *isochronic forks*.

10.2 Circuits as Networks of Gates

A DI circuit is a network of logical operators, or *gates*. A gate has one or more Boolean inputs and one Boolean output. (Later, we will introduce gates with multiple outputs.) The state of the circuit is entirely characterized by the values of the input and output variables of the gates.

We assume that all circuits are *closed*: Each variable of a circuit is the input of a gate and also the output of a gate. An open circuit is transformed into a closed one by representing the environment of the circuit as gates.

A gate with output variable z is defined by the two *production rules*:

$$\begin{aligned} B_u &\mapsto z\uparrow \\ B_d &\mapsto z\downarrow \end{aligned}$$

We will assume that a guard is in disjunctive-normal form, that is, it is either a *literal*, a *term*, or a disjunction of terms. A literal is a variable or its negation; a term is a conjunction of literals.

The two PRs of a gate must fulfill the non-interference requirement. A gate is a partial function when the non-interference requirement is not a tautology but has to be maintained as a program invariant. The flip-flop is an example of such a gate.

The non-interference requirement eliminates the most obvious case of malfunctioning of a gate. But other forms of malfunctioning, usually called *hazards*, have to be eliminated as well. A hazard is an incomplete transition on the output of a gate caused either by two consecutive transitions on one input variable or by some concurrent changes on several input variables. In our model, all occurrences of hazards are eliminated by the stability requirement.

(The stability of the physical implementation of a PR also requires that the changes in value of the physical quantity—voltage, in MOS technology—representing the Boolean values be *monotonic*. However, monotonicity around the stable values is, in general, neither attainable, because of noise, nor necessary.)

If a circuit fulfills the non-interference and stability criteria, no glitch or hazard can corrupt the value of the variables. At any point in time, the physical quantity representing a variable either has one of the two stable values representing the two Boolean values, or is monotonically changing from one stable value to the other.

Any pair of PRs that set and reset the same output variable defines a valid gate, with the exception of *self-invalidating* PRs. A rule with guard g and result r is self-invalidating if $r \Rightarrow \neg g$ may hold as a postcondition of a transition of that rule. In other words, the execution of the rule may falsify the guard. For example, the rules $x \mapsto x\downarrow$ and $\neg x \mapsto x\uparrow$ are self-invalidating.

It is always possible to modify the guard of a PR so that it does not contain the output variable of the gate. (This is achieved by removing all terms that contain the result as literal. For example, $(x \wedge z) \vee y \mapsto z\uparrow$ can be replaced with $y \mapsto z\uparrow$, since an execution of the PR in the state where $x \wedge z$ holds is vacuous.)

Hence, gates do not contain variables that are both input and output (self-loops). In the sequel, unless specified otherwise, an execution of a PR is an effective execution.

10.2.1 Wires

A priori, a wire with input x and output y is the gate defined by the PRs $x \mapsto y\uparrow$ and $\neg x \mapsto y\downarrow$. But, since the composition of any gate, *including a wire*, with a wire is the gate itself with one of its variables renamed, we can add an arbitrary number of wire gates to a circuit definition without

actually changing the circuit. In order to have a unique network of gates for each circuit, we exclude the wire from the gates; a wire is just a renaming mechanism for variables.

So far all gates except the wire have more inputs than outputs, but most circuits have as many outputs as inputs. We must therefore reset the balance by introducing at least one gate with more outputs than inputs. This gate is the *fork*.

10.2.2 Forks and Multiple-Output Gates

A fork has one input and at least two outputs. The fork, f, with input x and outputs y and z is defined as

$$\begin{array}{ll} x & \mapsto y\uparrow, z\uparrow \\ \neg x & \mapsto y\downarrow, z\downarrow \end{array}$$

where the comma means the execution of the two assignments in any order or concurrently. The generalization to an arbitrary number of outputs is obvious. The gate

$$\begin{array}{ll} B_u & \mapsto x\uparrow \\ B_d & \mapsto x\downarrow \end{array}$$

composed with fork f is equivalent to the gate with outputs y and z

$$\begin{array}{ll} B_u & \mapsto y\uparrow, z\uparrow \\ B_d & \mapsto y\downarrow, z\downarrow . \end{array}$$

Hence, the fork is just a mechanism for replicating the outputs of a gate and for defining gates with an arbitrary number of outputs. The following discussion is somewhat simplified if we eliminate the fork and allow instead the type of multiple-output gates that correspond to the composition of a single-output gate and a fork. But gates defined in this way have an important restriction: *The effective execution of a PR of a gate contains an effective transition on each output of the gate.*

10.2.3 Summary of the Model

The only restriction that these definitions and conventions introduce on the class of circuits being considered is the exclusion of gates with self-loops and of arbitration devices. Unlike models based on the "fundamental mode" of operation, several inputs of a gate may change values simultaneously as long as the stability of the guards of the PRs is preserved.

Also, we do not assume that the transitions are instantaneous: A variable value changes monotonically from the "bottom" value representing one logical value to the "top" value representing the other logical value, and vice-versa.

Because the transitions durations are finite but positive and variable, the ordering of transitions in a circuit has to be defined with care.

10.3 Partial Order of Transitions

The specification of a sequential circuit defines a partial order of actions taken from a repertoire of commands. In order to assert that a circuit fulfills a specification, we must relate this partial order to some other order relation among transitions of the circuit. The partial order of transitions is defined as follows.

Consider an effective execution of a PR causing the transition t, and let C be a term of the guard such that C holds for this execution of the PR.

We attach to C a set, T, of transitions in the following way. Each literal of C uniquely defines a transition: The literal x is the result of a transition of type $x\uparrow$, and the literal $\neg x$ is the result of a transition of type $x\downarrow$. (The initialization of a variable is also considered a transition.) *By definition, we say that transition t is a successor of each transition of T.* In other words, a transition is the successor of the set of transitions that make the guard true, including initializations.

For example, if the PR is $x \wedge y \mapsto z\uparrow$, we say that each transition $z\uparrow$ is the successor of a transition $x\uparrow$ and of a transition $y\uparrow$.

If the guard of the PR is of the form $A \vee B$, the transition is the successor of the set of transitions that make A true, or of the set of transitions that make B true. Hence, the successor relation defined is not unique for a given circuit. A *computation* is a particular successor relation on a set of transitions, such that each computation corresponds to a possible execution of the circuit. The set of transitions of a computation is finite if the corresponding execution of the circuit terminates, and possibly infinite otherwise.

¿From the successor relation, we can now construct a relation $\prec$ that is a pre-order; that is, it is transitive and anti-reflexive. Once we have the pre-order relation $\prec$, we construct the partial order $\preceq$ by defining $t1 \preceq t2$ to mean $t1 \prec t2$ or $t1 = t2$.

Transitivity. *For any two transitions $t1$ and $t2$, we say that $t1 \prec t2$ when $t2$ is a successor of $t1$, or there exists a transition $t3$ such that $t1 \prec t3$ and $t3 \prec t2$.*

Anti-reflexivity. *$t \prec t$ holds for no transition t.*

REMARK: Anti-reflexivity is satisfied if, for each ring of gates in the circuit, there is always at least one PR whose guard is true and whose result is false—the ring "oscillates." Anti-reflexivity excludes rings of gates that are used to maintain constant values of variables, as in cross-coupled device constructions

of storage elements. We therefore assume that the storage elements are parts of "perfect wires," so to speak, that keep the value of a variable until the next transition on the variable. □

Definition. *A chain from a to b is a finite, non-empty set $\{t_i, 0 \leq i < n\}$ of transitions such that $t_0 = a$, $t_n = b$, and for all i, $0 < i < n$, t_i is a successor of t_{i-1}. By construction, $a \preceq b$ means that there is a chain from a to b. If $a \prec b$, we say that b follows a.*

10.4 Implementation of Stability

Consider again an execution of a PR with guard B and transition t. Either B is never falsified once it holds, but then t is the last transition on the variable involved, and we say that the transition is *final*. Or B is falsified after a finite number of transitions following t, in which case, in order to implement the stability of B, we have to see to it that t is completed before B is falsified.

For all transitions i that falsify B, we have to guarantee $t \prec i$. Hence, by definition of the order relation, there must be a transition s such that s is a *successor* of t, and $s \preceq i$. We say that s *acknowledges* t. Hence, the

Acknowledgment Theorem. *In a DI circuit, each non-final transition has a successor transition.*

By construction of multiple-output gates, we have the

Corollary. *In a DI circuit, a non-final transition on an input of a gate has a successor transition on each output of the gate.*

EXAMPLE: Consider the three following gates with two inputs, x and y, and one output, z. The *flip-flop* is defined as $x \mapsto z\uparrow$ and $\neg y \mapsto z\downarrow$, the *asymmetric C-element* as $x \wedge y \mapsto z\uparrow$ and $\neg y \mapsto z\downarrow$, and the *switch* as $x \wedge y \mapsto z\uparrow$ and $x \wedge \neg y \mapsto z\downarrow$.

Since no guard of these gates has a term containing the literal $\neg x$, a transition of type $x\downarrow$ has no successor. Hence, according to the Acknowledgment Theorem, there can be at most two transitions on x in any computation of a DI circuit using any of these three gates. □

10.5 The Unique-Successor-Set Criterion

Later on, we shall give a simple criterion for deciding whether a given circuit—a network of gates—is DI. But such a criterion does not tell us whether there exists a DI circuit for a given specification. We shall therefore formulate a more general theorem that characterizes the partial orders of transitions

that admit a DI implementation. This criterion enables us to decide that a program has no DI implementation without having to construct a circuit.

Successor Set. *In a computation, the successor set of a transition t is the set of variables x such that a transition on x is a successor of t.*

Unique-Successor-Set Property. *A computation has the unique-successor-set (USS) property when all non-final transitions on the same variable have the same successor set. A set of computations has the USS property when all non-final transitions on the same variable have the same successor set in all computations of the set.*

Unique-Successor-Set Theorem. *A set of computations of a DI circuit has the USS property.*

Proof. Consider an arbitrary variable x of a DI circuit. By the corollary of the Acknowledgment Theorem, any non-final transition t on x has a successor transition on each output of the gate, say G, of which x is an input.

By definition of the successor set, the set of output variables of G is the successor set of t. But since the set of output variables of a gate is unique, the successor set is the same for all non-final transitions on x. □

10.6 Characterization of DI computations

Although the Unique-Successor-Set Theorem is a direct consequence of the Acknowledgment Theorem, its formulation in terms of computations instead of gates makes it possible to lift the result from the implementation level to the specification level. Since the partial orders of actions defining a circuit are projections of the partial orders of actions implementing it, we shall investigate whether the USS property is maintained by projection.

Definition. *Given a computation, c, on a set of variables, V, the projection of c on a subset, W, of V is the computation derived from c by removing all transitions on variables of $V \backslash W$ from the chains of c. The projection of a set of computations is the set obtained by projecting each element of the original set.*

Projection Theorem. *If a set of computations has the USS property, then its projection on a subset of variables has the USS property.*

Proof. By definition, the projection of a set of computations on W can be obtained by removing the elements of $V \backslash W$ one for one from all chains of each computation of the set. We prove the theorem by showing that removing all transitions on one variable, say, w, maintains the USS property of the set.

Let x be another variable, and let X be the USS of (all transitions on) x in all computations of the set. Either w does not belong to X and X is left unchanged by the transformation, or w is removed from X. But then, for each transition tx on x, the successor set of the transition on w that follows tx must be added to the successor set of tx. Since all transitions on w have the same successor set in all computations of the set, the new X is the same for all transitions and all computations of the set. □

10.6.1 Example: One-Place Buffer

The cyclic program $*[X; Y]$, where X and Y are communication commands, is called a *one-place buffer*[1]. It is a basic building block of asynchronous circuit design since it is used to implement the sequencing of any two actions. With a four-phase handshaking protocol for implementing the communications, an expansion of the program in terms of elementary variables is:

$$*[[xi]; xo\uparrow; [\neg xi]; xo\downarrow; yo\uparrow; [yi]; yo\downarrow; [\neg yi]],$$

where xi and yi are the input variables, and xo and yo are the output variables[2]. (See Figure 1.) The environment of the circuit can be simply modeled as the two programs:

$$*[xi\uparrow; [xo]; xi\downarrow; [\neg xo]]$$
$$*[[yo]; yi\uparrow; [\neg yo]; yi\downarrow].$$

These three programs are concurrent. Now observe that the projection of a computation on the output variables of the first program gives the computation described by the program

$$*[xo\uparrow; xo\downarrow; yo\uparrow; yo\downarrow].$$

Obviously, this computation does not have the USS property; therefore, by the Projection Theorem, the closed circuit implementing the three programs is not DI. But the two environment programs can be implemented with an inverter gate and an identity gate, which are DI circuits. Hence, there is no DI circuit implementing this version of the one-place buffer with four-phase handshaking.

We can state a more general result. We observe that, for whatever four-phase handshaking is chosen for X and Y, the projection on the output variables is always $*[xo\uparrow; xo\downarrow; yo\uparrow; yo\downarrow]$, unless the handshaking actions of X are reordered ("shuffled") with respect to the handshaking actions of Y. Hence, the

[1] The notation $*[S]$ stands for the non-terminating repetition of the program S.

[2] For an arbitrary Boolean expression B, the command $[B]$ is a shorthand notation for $[B \rightarrow skip]$, and can be informally defined as "wait until B holds."

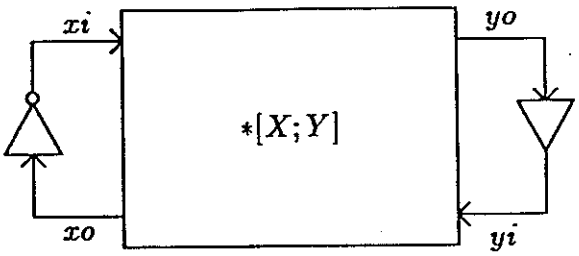

Figure 10.1: A one-place buffer and its interface

Theorem. *There is no DI circuit implementing a one-place buffer with unshuffled four-phase handshaking.*

We can shuffle the handshaking actions of X with respect to the handshaking actions of Y, so that the projection on the output variables is the sequence

$$*[xo\uparrow; yo\uparrow; xo\downarrow; yo\downarrow].$$

Now, the sequence has the USS property, and we can implement the one-place buffer as a DI circuit. An example is shown in Figure 2.

10.7 Specifications and the USS Property

The Projection Theorem is very useful because we can also define when a specification has the USS property. If a specification does not have the property, we can immediately conclude that there exists no DI implementation of the specification. The projection from implementation to specification occurs as follows.

We assume that, whatever specification notation is used, whether programs, traces, or regular expressions, it is possible to derive from the speci-

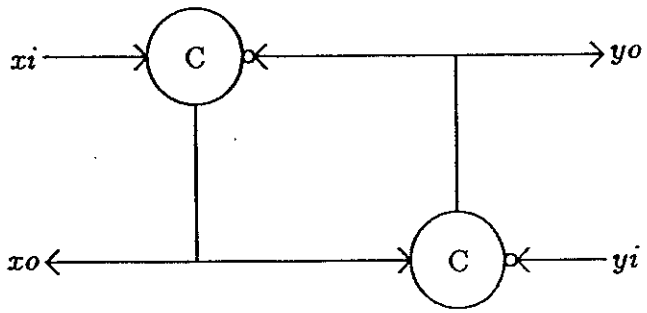

Figure 10.2: A DI circuit for the one-place buffer

fication certain properties of the partial order of actions involved. Hence, in the sequel, a *specification* is a set of partial orders of actions, where an action is an execution of a command taken from some given repertoire.

We also assume that an elementary variable can be uniquely identified with (the implementation of) each command: The transitions on the variable occur only in the executions of the command, and each execution of the command contains a transition on the variable. This (in theory, slightly restrictive) assumption is needed only for the following

Specification Theorem. *If the specification of a circuit does not have the USS property, the circuit is not DI.*

Proof. Consider a specification, S, of a circuit. For each command, X, of S, we substitute a transition on the elementary variable x that is uniquely associated with X. We obtain a set, s, of partial orders of transitions on elementary variables. Since the existence of the USS property is independent of whether the transitions are upgoing or downgoing (that is, the "direction" of the transitions), we can decide whether s has the USS property even though the direction of the transitions in s is undefined.

By definition, we say that specification S has the USS property if and only if the set, s, thus defined has the USS property. By construction, s is a

projection of the set of computations of the circuit specified by S. Hence, by the Projection Theorem and the USS Theorem, if s does not have the USS property, the circuit is not DI. □

EXAMPLES: The following examples, which we give without proofs, show how limited is the class of programs that admit a DI implementation. (In the examples, all commands are different from *skip*.) We assume that the semantics of the program notation are clear enough that we can identify the programs with the partial order of actions they represent.

• Let $P \equiv *[S_1; S_2; \ldots S_n]$, and assume that there is no equivalent program

$$*[S_1; S_2; \ldots S_k]$$

with $k < n$. (We say that P is a minimal representation. For instance, $*[X; X]$ is not minimal since $*[X]$ is an equivalent program.)

Then P has the USS property if and only if $S_i \neq S_j$ for $i \neq j$. Hence, the "modulo-2 counter" $*[X; X; Y]$ and all other "modulo-k counters" have no DI implementation. A similar result has been proved by C. J. Seger[22].

• The program $*[S_1; [B_1 \rightarrow S_2 [] B_2 \rightarrow S_3]; S_4]$, with $S_2 \neq S_3$, does not have the USS property. Hence, there is no DI circuit implementing such a selection command. □

10.8 Gate Characterization of DI Circuits

We have already seen that, apart from the trivial case where one input of the gates changes at most twice, there is no DI circuit that contains either a flip-flop, or an asymmetric C-element, or a switch. In the same way, we can use the USS and the Projection Theorems to show that there is no DI circuit containing either an *or-gate*, or an *and-gate*, or an *exclusive-or*, in which each input of the gates changes more than a minimum number of times specific to each case. Consider an or-gate with inputs x and y and output z. The only sequence[3] in which each transition on an input is acknowledged is:

$$((x\uparrow; z\uparrow; x\downarrow; z\downarrow)^*; (y\uparrow; z\uparrow; y\downarrow; z\downarrow)^*)^*$$

We easily see that any computation that contains a transition on both inputs does not have the USS property.

The cases of the and-gate and of the exclusive-or are treated similarly and are left as an exercise for the reader. After having eliminated all gates with at most two inputs except the inverter and the Muller-C element, we are led to conjecture that a DI circuit contains only C-elements. C-elements are defined as follows.

[3]The notation $(S)^*$ is the Kleene-star notation standing for an arbitrary number of actions S in sequence.

Definition. *An n-input gate in which B_u is the conjunction of the n input variables and B_d is the conjunction of the negations of the n input variables is called an n-input C-element. A gate derived from a C-element by negating one or more literals in B_u or B_d is also a C-element.*

The Muller-C element is a two-input C-element according to our definition. A one-input C-element reduces to either a wire or an inverter.

C-Element Theorem. *If a DI circuit has only one computation, and if the computation contains at least three transitions on each variable, then the circuit can be constructed with C-elements only.*

Proof. Let x be an arbitrary variable of the circuit; x is the input of gate g with output z. We shall prove that g can be implemented as a C-element. Since there are no self-loops, x and z are different variables.

First, observe that because of the non-interference, all transitions on the same variable are totally ordered. And because all transitions are effective, upgoing and downgoing transitions on the same variable alternate.

Since the circuit contains at least three (effective) transitions on each variable, at least one transition of type $x\uparrow$ is followed by a transition of type $x\downarrow$, and at least one transition of type $x\downarrow$ is followed by a transition of type $x\uparrow$.

Let $t1$ be a transition of type $x\uparrow$ and $t2$ be the transition of type $x\downarrow$ following it. For the guard of the PR of $t1$ to be stable, there must be a transition tz on z such that $t1 \prec tz \prec t2$. We also know that tz is a successor of $t1$.

By the USS Theorem and the Projection Theorem, there is exactly one transition tz on z such that $t1 \prec tz \prec t2$. By the same argument, there is exactly one transition on z between a transition of type $x\downarrow$ and the transition of type $x\uparrow$ following it.

Without loss of generality, assume that the first transition on x is of type $x\uparrow$ and the first transition on z is of type $z\uparrow$. Then, because of the alternation of upgoing and downgoing transitions on each variable, each transition of type $z\uparrow$ is the successor of a transition of type $x\uparrow$, and each transition of type $z\downarrow$ is the successor of a transition of type $x\downarrow$.

By definition of the successor relation, x holds as a precondition of each transition $z\uparrow$; thus, guard B_u of g can be formulated so that all terms contain x, since a term that is never true can be removed. Hence, B_u can be chosen of the form $x \wedge C_u$, where C_u does not contain x. Symmetrically, guard B_d of g can be chosen of the form $\neg x \wedge C_d$, where C_d does not contain x. Since this property of B_u and B_d holds for each input of g, g is a C-element or can be replaced with a C-element. □

10.9 Isochronic Forks

Since the class of DI circuits is so limited, we must have compromised the delay-insensitivity in the circuits that we designed using the synthesis method described, for instance, in [12] and [11]. Let us analyze a standard sequencing circuit used in this design style. (It is similar to the one-place buffer, but is simpler to use as an example.) This circuit (Figure 3) is an implementation of the sequence of elementary actions:

$$*[[xi]; yo\uparrow; [yi]; u\uparrow; [u]; yo\downarrow; [\neg yi]; xo\uparrow; [\neg xi]; u\downarrow; [\neg u]; xo\downarrow].$$

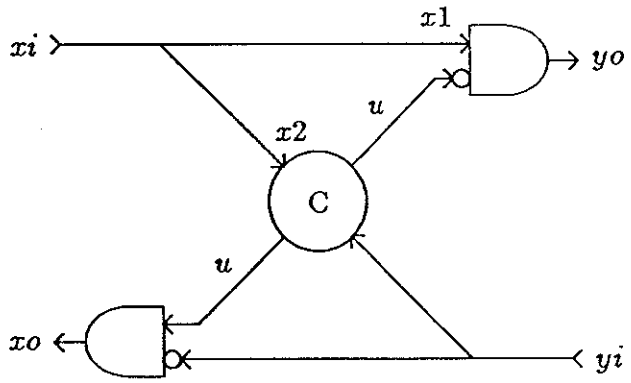

Figure 10.3: A sequencing element containing isochronic forks

The environment of the circuit is the same as that of the one-place buffer. The x- and y-variables are each parts of a four-phase handshaking sequence, and u is a state variable—without u, it would not be possible to encode each state of the circuit uniquely. Since the projection of this sequence on the variables xo, yo, and u lacks the USS property, and since the environment of the circuit can be implemented as an inverter and an identity, the circuit is not DI.

In order to find out where we have cheated, we must look at the forks. We observe that xi is an input both of the and-gate with output yo and of the C-element. Hence, the circuit actually contains a fork with input xi and two outputs, say, $x1$ and $x2$. Similarly, the circuit contains a fork with input

yi, and a fork with input u. Let us analyze the behavior of the first fork by introducing it explicitly into the set of PRs of the circuit. For the sake of simplicity, we ignore the other two forks. We get:

$$\begin{array}{ll} xi & \mapsto x1\uparrow, x2\uparrow \\ x1 \wedge \neg u & \mapsto yo\uparrow \\ x2 \wedge yi & \mapsto u\uparrow \\ \neg x1 \vee u & \mapsto yo\downarrow \\ \neg yi \wedge u & \mapsto xo\uparrow \\ \neg xi & \mapsto x1\downarrow, x2\downarrow \\ \neg x2 \wedge \neg yi & \mapsto u\downarrow \\ yi \vee \neg u & \mapsto xo\downarrow \end{array}$$

Transitions $x1\uparrow$ and $x2\uparrow$ are both acknowledged by the two PRs that follow. But only transition $x2\downarrow$ is acknowledged. Transition $x1\downarrow$ is *not* acknowledged. Hence, the circuit is not DI, because the Acknowledgment Theorem is not satisfied. Therefore, the completion of transition $x1\downarrow$ is not guaranteed unless we implement the fork as an *isochronic fork*, which is defined as follows.

In an isochronic fork, when a transition on one output is acknowledged, and thus completed, the transitions on all outputs are acknowledged, and thus completed.

(We leave it as an exercise to the reader to check that the fork with input yi must also be isochronic, but not the fork with input u.)

The implementation of an isochronic fork relies on two types of assumptions about delays. First, we have to assume that the difference between the delays in the branches of the fork is negligible compared to the delays in the gates. This requirement is easy to meet in current MOS technology except when there is an inverter on one branch of the fork and not on the other branch(es). The fork with input yi has such an inverter, and therefore, the inverter must be removed by proper circuit transformations.

Second, and more important in current technology, we have to assume that the switching thresholds in the different gates to which the fork is an input are close enough to each other. This requirement is more difficult to meet than the first one because, on the one hand, the thresholds of individual transistors are difficult to control—in particular in CMOS; on the other hand, the switching thresholds of a gate vary greatly with the logical design of the gate. For these reasons, this requirement may impose a design style in which all gates are implemented as combinational gates, so that the fight between pull-up and pull-down during the switching of the gate keeps the switching threshold around *VDD/2*. Observe that, unlike what is advocated in other compromises to delay-insensitivity, enforcing the locality of the wires offers little help in implementing isochronicity because locality is irrelevant to the issue of threshold voltages!

10.10 For Whom the Bell Tolls?

Are these results tolling the bell for DI design? Actually, not. At worst, they may slightly embarrass those researchers who claim to have a design method for entirely DI circuits. At best, they vindicate the compromises to delay-insensitivity adopted by several asynchronous design methods. Most likely, they are sobering reminders of the difficulty of VLSI design and the novelty of asynchronous design.

We have proved elsewhere that extending a standard repertoire of DI gates with isochronic forks is sufficient to construct any circuit of interest. The proof consists in giving a circuit implementation for each construct of the progam notation we use (see [2]). I believe the isochronic fork to be the weakest possible compromise to delay-insensitivity in the sense that all other compromises also include isochronic forks: For instance, in speed-independent design[19], all forks are supposed to be isochronic; in self-timed design[23], all forks inside a certain region—called an *equipotential region*—are assumed to be isochronic.

Chapter 11

Conclusion

We have described a method for implementing a concurrent program (a set of communicating processes) as a network of digital operators that can be directly mapped into a delay-insensitive VLSI circuit. The circuit is derived from the program by applying a series of systematic, semantics-preserving transformations that we have compared to compiling. Hence, the circuits are correct by construction, and their logical correctness is independent of the delays in operators and wires, with the exception of isochronic forks.

The most encouraging aspect of the method is that it is really a synthesis technique: it allows designers to construct solutions that they would never have found had they not applied the method. Different applications of the transformations lead to different circuits for the same program. Although all circuits are semantically equivalent, they may exhibit different behaviors in terms of speed or size (number of operators used). The method therefore includes the trade-offs between simplicity and efficiency that should be available to the VLSI designer.

Using concurrency to implement a sequential computation may seem wasteful at first sight. But VLSI is essentially a concurrent medium: concurrency is implemented at no cost by mere juxtaposition of the concurrent parts. On the other hand, implementing sequencing requires synchronization and is, in general, more expensive. We shall therefore implement sequencing as restricted concurrency. Once a process has been transformed into a semantically equivalent set, the problem of implementing sequencing has disappeared!

This technique entails one of the main novelties of the method. Other techniques implement sequencing by transforming the computation into a finite-state machine, and realizing each state with a state-holding element. In our technique, some state-holding elements may be needed, but the number of those elements is drastically less than in techniques using finite-state machines.

Since the issue of isochronic forks seems to have confused certain readers of previous papers, let us make clear a number of points. First, most forks need not be isochronic, as, for instance, the fork that distributes a control signal to all bits of a register. Second, the isochronicity requirement is easy to meet when there is no inverter on the branches of the fork, and in practice, it is usually easy to move the inverters so as to remove them from the branches of isochronic forks. Third, isochronic forks are necessary to implement the sequencing of two four-phase handshaking protocols; therefore, methods that claim to dispense with isochronic forks just hide them inside building blocks.

The proofs that the transformations preserve the semantics of the algorithms rely on properties of the four-phase handshaking protocol with which the communication primitives are implemented. Although rigorous proofs of these properties have been omitted, the reader should have no difficulty in being convinced of their correctness, and thus of the correctness of the transformations performed.

The examples cover most constructs of the language but not all of them: We have not shown how to implement an arbitrary set of guards. Therefore, we have not quite shown that *any* program in the language can be compiled. Such a proof has been given in [1] and [2], where the compilation of each construct is described as part of the basic algorithm for an automatic compiler. It is shown that any program in a subset of the language can be implemented as a delay-insensitive circuit using only a small set of basic elements: the 2-input C-element, the 2-input or-gate or 2-input and-gate, the synchronizer, the inverter, and the isochronic fork.

However, there is no reason for confining the designer to a minimal set of operators. On the contrary, since an advantage of VLSI is the possibility to create operators at no cost, introducing the special purpose operator that exactly implements an arbitrary set of production rules often simplifies a circuit drastically.

In order to convince the VLSI community of the practicality of our method, it was essential that we fabricated the circuits we had designed. Hence, all significant examples that we have used in our research—distributed mutual exclusion, queues, stack, routing automata for communication network, $3X + 1$ engine, microprocessor—have been fabricated in SCMOS using the MOSIS foundry service. They have all be found to be correct on "first silicon". They are also very robust, and surprisingly fast, given the low-level of circuit optimization applied. The $3x + 1$ engine, constructed by Tony Lee, is a special-purpose processor consisting of a state-machine and an 80-bit-wide datapath. It contains approximately 40,000 transistors and operates at over 8 MIPS (million instructions per second) in $2MU$m MOSIS SCMOS technology.

We have designed the first asynchronous general-purpose microprocessor in CMOS. The results of this experiment, described in Chapter 9, are very en-

couraging and contradict the long-held belief that asynchronous techniques are too slow and too wasteful in area for something as demanding as a pipelined general-purpose microprocessor. We have just finished a GaAs version of the same microprocessor. It is now being fabricated by Vitesse through the MOSIS service. Although it is too early to report any performance results, this experiment already demonstrates how easy it is with such a synthesis method to transport a complete design across very different technologies.

11.1 Acknowledgments

I am indebted to my students Dražen Borković, Steve Burns, Marcel van der Goot, Pieter Hazewindus, Tony Lee, and José Tierno for their contributions to the research and for their comments on the manuscript.

Bibliography

[1] J.A. Brzozowski and J.C. Ebergen. Recent Developments in the Design of Asynchronous Circuits. Research Report CS-89-18, Computer Science Department, University of Waterloo, 1989.

[2] Steven M. Burns and Alain J. Martin. Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. *Proc. Fifth MIT Conference on Advanced Research in VLSI*, ed. J. Allen and F. Leighton, MIT Press, 35-40, 1988.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading MA, 1988.

[4] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs NJ, 1976.

[5] David L. Dill. *Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1989.

[6] C.A.R. Hoare. Communicating Sequential Processes. *Comm. ACM* 21,8, pp 666-677, 1978.

[7] Alain J. Martin. The Probe: An Addition to Communication Primitives. *Information Processing letters* 20, pp 125-130, 1985.

[8] Alain J. Martin. Distributed Mutual Exclusion on a Ring of Processes. *Science of Computer Programming*, 5, 265-276, 1985.

[9] Alain J. Martin. The Design of a Self-timed Circuit for Distributed Mutual Exclusion. *1985 Chapel Hill Conference on VLSI*, ed. Henry Fuchs, Computer Science Press, 247-260, 1985.

[10] Alain J. Martin. A Delay-insensitive Fair Arbiter. Technical Report 5193:TR:85, Computer Science Department, California Institute of Technology, 1985.

[11] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, P.J. Hazewindus. The Design of an Asynchronous Microprocessor. *Decennial Caltech Conference on VLSI*, ed. C.L. Seitz, MIT Press, 351-273, 1989.

[12] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, 1,(4), 1986.

[13] Alain J. Martin. Formal Program Transformations for VLSI Circuit Synthesis. *UT Year of Programming Institute on Formal Developments of Programs and Proofs*, ed. E.W. Dijkstra, Addison-Wesley, Reading MA, 1989.

[14] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. *Formal Methods for VLSI Design*, ed. J. Staunstrup, North-Holland, 1990.

[15] Alain J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conference on Advanced Research in VLSI*, ed. W.J. Dally, MIT Press, 1990.

[16] Raymond E. Miller. *Switching Theory*, Vol. 2, Wiley, 1965.

[17] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*, Addison-Wesley, Reading MA, 1980.

[18] Teresa H. Meng, Robert W. Brodersen, David G. Messerchmitt. Automatic Synthesis of Asynchronous Circuits from High-Level Specifications. *IEEE Trans. on CAD*, 8:11, 1185-1205, 1989.

[19] Raymond E. Miller. *Switching Theory*, Vol. 2, Wiley, 1965.

[20] David E. Muller and W.S. Bartky. A Theory of Asynchronous Circuits. *Annals of the Computation Laboratory of Harvard University*, Vol. 29, Harvard University Press, Cambridge, Mass., 204-243, 1959.

[21] J. Staunstrup and M.R. Greenstreet. Designing Delay-Insensitive Circuits using "Synchronized Transitions." *IMEC IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[22] Carl-Johan Seger. On the Existence of Speed-Independent Circuits. Research Report CS-87-63, Computer Science Department, University of Waterloo, 1987.

[23] Charles L. Seitz. System Timing. Chapter 7 in Mead & Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA, 1980.

[24] Snepscheut, J.v.d., *Trace Theory and VLSI Design* LNCS 200, Springer-Verlag Berlin Heidelberg (1985)

[25] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*, Addison-Wesley Reading MA, 1985.