Asynchronous VLSI Systems

Rajit Manohar

Contents

Acknowledgments

The material in these notes is the result of over 20 years of research on asynchronous VLSI design performed at the California Institute of Technology and Cornell University. Some of the students that made significant contributions to research on asynchronous VLSI presented in this book include Dražen Borković, Steve Burns, Marcel van der Goot, Pieter Hazewindus, Tak-Kwan Lee, Andrew Lines, Rajit Manohar, Mika Nyström, and José Tierno. (XXX: this is out of date)

The team that designed the first asynchronous microprocessor included Steve Burns, Tak-Kwan Lee, Dražen Borković, and Pieter Hazewindus. This project provided the first convincing demonstration of the synthesis methodology for control and datapath circuits, including the concept of process decomposition and control/data separation. The asynchronous MIPS processor team included Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri Cummings, and Tak-Kwan Lee. The contributions of this project included circuits and design methodologies for highly pipelined asynchronous circuits, including the concepts of projection, slack matching, and slack elasticity.

The exercises and notes were compiled from lecture notes and slides from CS 185 (Asynchronous VLSI Design) at Caltech taught by Alain J. Martin, and from EE 571 (Asynchronous VLSI Design) at Cornell taught by Rajit Manohar.

This research was primarily supported by the Defence and Advanced Research Projects Agency under several contracts including the Submicron Systems Architecture Project and the Asynchronous Systems Architecture Project. Additional support was provided by the National Science Foundation, the Office of Naval Research, and by several graduate fellowships from IBM, Intel Corporation, the Okinawa Foundation, and National Semiconductor Corporation.

Chapter 1.

Introduction

"It is a little off the beaten track, isn't it?"
—Sir Arthur Conan Doyle, *The Red Headed League*

A VLSI circuit is said to be *asynchronous* when it does not use a clock signal for sequencing. Asynchronous switching circuits have been in use since the 1940's. The Illiac, designed by the University of Illinois Digital Computer Laboratory in the late 1950's,[34] is an example of a computer that contained both synchronous and asynchronous switching circuits. The computer had "end signals" (now called "acknowledge" signals) that indicated the completion of an action.

Early concepts in the design of asynchronous circuits were contributed by D.A. Huffman in the 1950's.[34] As complex asynchronous circuits became difficult to design because of the problem of hazards (glitches) in switching signals, they were replaced by synchronous circuits. By the time computers became widespread in the 1970's, synchronous switching circuits had emerged as the prevalent design style.

Modern asynchronous circuit design probably began when concerns arose regarding problems with the physical realization of large-scale synchronous system. While concerns about clock distribution and wire delays motivated initial investigation, the main challenge facing designers of VLSI systems today is the management of complexity. Some recent (in) famous examples include the error introduced in the Pentium division algorithm and the `F00F` bug that could deadlock a Pentium processor.

One of the most difficult parts of the design of a clocked system is reasoning about the synchronization required for its correct operation. A clocked system comprising of a large number of concurrent pipeline stages is difficult to reason about when examined at the circuit level. To combat this problem we will use a high level of abstraction when describing complex systems. In order to achieve this level of abstraction we must ignore the physical implementation details of the VLSI system until the final stages of the design. We treat the physical details of the implementation as parameters that we will tune for performance, not as parameters that affect the correctness of the design.

An important physical parameter we would like to ignore is time, because this parameter is only known once the physical design of a computation is complete. We choose asynchronous circuits as the preferred implementation strategy for a computation, because they permit us to completely abstract away from the notion of time.

We say a circuit is *delay-insensitive* when its correct operation is independent of the delays in gates and in the wires connecting the gates, assuming that the delays are finite and positive. For the reasons mentioned above, we would like to use delay-insensitive circuits to abstract away from all timing information. Unfortunately, the class of truly delay-insensitive circuits is very limited.

A theory of *speed-independent* asynchronous switching circuits was developed by D.E. Muller in the early 1960's as an attempt to abstract from the difficulties of designing circuits that depended heavily on their precise physical implementation. His model assumed that transistor networks may have arbitrary delay, and that the propagation delay through wires is negligible compared to the delay through the network. However, his design methodology was limited to simple sequential circuits.

In 1979, Seitz proposed a design methodology for *self-timed* circuits wherein the circuit was to be decomposed into equipotential regions—regions where delays in wires could be considered negligible, with explicit modeling of signal propagation delay between such regions.[37]

The first method for the synthesis of asynchronous circuits whose correct functioning did not depend on the delays of gates and which permitted multiple concurrent switching signals was introduced by Alain J. Martin.[28] His approach is inspired by the observation that a VLSI chip is a fine-grained concurrent computation. VLSI computations are modeled using CHP (Communicating Hardware Processes) programs that describe their behavior algorithmically. Asynchronous *quasi delay-insensitive* (QDI) circuits are synthesized from these programs using semantics-preserving transformations. The circuit design methodology assumes that gates have arbitrary delay, and only makes relative timing assumptions on the propagation delay of some signals that fanout to multiple gates. We use this approach to the design of asynchronous circuits in these notes.

Asynchronous QDI circuits are robust to variations in temperature, voltage, and fabrication process parameters. For example, the Caltech microprocessor fabricated in $1.6\mu m$ CMOS technology (design voltage $5\,V$) is functional at all voltages from $0.6\,V$ (sub-threshold) to $12\,V$ (punch-through).[31] More recently, an asynchronous pipelined lattice-structure filter[5] was fabricated and is functional in $0.8\mu m$ CMOS technology (design voltage $3.3\,V$) from $1.1\,V$ to $4.9\,V$. Both chips are also functional at temperatures ranging from $77K$ to $350K$. In 1998, a high-performance asynchronous MIPS core was fabricated in $0.6\mu m$ CMOS technology, and found to be functional at voltages ranging from $1.1\,V$ to $6.9\,V$.

Asynchronous implementation strategies for complex VLSI systems are interesting for other reasons as well. Asynchronous circuits exhibit average-case behavior. As a result, we can choose implementations that improve the overall performance of the circuit even if they make the worst-case performance worse. For example, an N-bit ripple-carry asynchronous adder has an average case latency of $O(\log N)$, the same order as a more complex synchronous carry-lookahead adder.[2]

Asynchronous implementations of a system consume less power than synchronous implementations. If we assume that the physical implementation of a circuit dissipates power when a particular signal in the circuit changes, we can show that any computation must dissipate an amount of power that depends on the entropy of the specification of the circuit.[41] What may not be so obvious is that we can achieve this bound within a constant factor by using an asynchronous implementation.[41]

In synchronous design, the performance of the system is determined by the clock frequency. If any component of the system is slow, the entire system must be slowed down to ensure the system operates correctly. This affects system throughput if some part of it does not operate at the desired clock frequency. A well-designed asynchronous circuit with the same properties will operate at the speed of the slow component only when the slow component is used.

Since the clock is used to discretize the time domain, differences in performance among components are measured in clock intervals. Asynchronous implementation methods take advantage of subtle performance differences—differences that arise when the number of transistors in series vary depending on the data, for instance. Normally, this difference is too small to be utilized by a synchronous design, whereas an asynchronous circuit will adapt its performance based on the value of its input even in cases where such small variations occur. Therefore, asynchronous design is the ideal vehicle to implement one of the most pervasive principles of computer architecture: make the common case fast.

Chapter 2.

Communicating Hardware Processes

One of the main issues with designing complex VLSI systems is managing complexity. When a number of devices are placed side-by-side on a chip, they all execute concurrently. While understanding the resulting behavior might be relatively simple for a small number of devices, it is exceedingly difficult to determine what a large collection of simple devices that cooperate in some manner is doing. Our approach to managing complexity is to abstract away from the details of the implementation of a computation as far as possible.

Complex systems can be concisely described using high-level languages. Therefore, we use a programming notation to describe asynchronous systems. We call this notation CHP, for communicating hardware processes. The notation we use has "free syntax"—we do not restrict variable types or operators that can be used in the language. We only provide a basic set of constructs that suffice for a large class of programs. For instance, it may be convenient to extend the language by introducing more complex operators that simplify the description of a certain design. (Typically this is done by introducing a new basic data type and operators and constructors for the data type.) Permitting the base language to be extended in this manner is convenient when describing complex systems. However, one should keep in mind that all data types must eventually be reduced to collections of Boolean-valued variables, since we will eventually construct a digital VLSI implementation of the computation.

The core of the language is a sequential programming notation based on Dijkstra's language of *guarded commands*. The language supports assignment statements, and two control structures: the loop and selection statement. A feature that distinguishes this language from most programming languages is that it includes a notion of *non-determinism*. Intuitively, a program is said to be non-deterministic when its execution can have more than one outcome for the same set of inputs.

2.1. Simple Statements

The statement **skip** does nothing. It does not modify any variables in the system. The statement $x := E$ where x is a variable and E is an expression denotes an assignment statement. Its behavior is described as follows: first, the expression E is evaluated; next, the value of the expression is assigned to variable x. For example, the statement $x := x + 1$ increments the value stored in variable x. Boolean-valued variables are frequently encountered when writing programs that correspond to VLSI systems. Therefore, we abbreviate $x :=$ **true** to $x\uparrow$ and $x :=$ **false** to $x\downarrow$.

The statement $x_1, x_2, \ldots, x_n := E_1, E_2, \ldots, E_n$ where $x_1, \ldots, x_n$ are variables and $E_1, \ldots, E_n$ are expressions denotes a *multiple-assignment* statement. Its behavior is described as follows: first, all the expressions are evaluated; next, the result of evaluating expression E_i is assigned to variable x_i, for all i. For example, the statement $x, y := y, x$ swaps the values stored in variables x and y. The statement is well-formed only when all the x_i variables are distinct.

Example. Consider the following multiple-assignment statement:

$$x, y := y, x$$

Since the expression on the right-hand side is evaluated before any variables are changed, the statement has the effect of swapping the values of x and y. ▮

Integers of length n can take on values in the range $[0, 2^n - 1]$. Signed integers of length n can take on values in the range $[-2^{n-1}, 2^{n-1} - 1]$. Both these integers are represented using n Boolean-valued variables. If x is an unsigned integer of length n, its most significant bit is denoted x_{n-1} and its least significant bit is x_0. The assignment $x := y$ where both x and y are integers of length n corresponds to the multiple-assignment statement $x_0, \ldots, x_{n-1} := y_0, \ldots, y_{n-1}$.

2.2. Assertions

An assertion statement is written $\{B\}$, where B is a Boolean-valued expression. An assertion can be placed before or after a statement. It states that the condition B must hold at that point in the program.

Example. The following assertions are guaranteed to be true:

$$x := 0 \ \{x = 0\}$$
$$x := y + 1\{x > y\}$$

▮

Assertions are useful for program development, because they can be used to write down conditions that one relies on for the correctness of the program. In some cases, these conditions must be satisfied by the program input.

Example. Consider a register file core that accepts commands to read two registers and write one register. The register file core performs all operations in parallel. The control to this register file core must guarantee that a register that is being read cannot be written at the same time. We can express this as $\{rs \neq rd \wedge rt \neq rd\}$, where *rs* and *rt* are the register numbers being read, and *rd* is the register number being written. ▮

2.3. Arrays, Bit-Fields and Records

Given an array a, the variable $a[i]$ denotes element i of array a. Array variables should only be used when the index i is not known beforehand. The array indexing mechanism introduces circuit overhead, and should be avoided if possible. The array mechanism can be used to describe access to the register file of a microprocessor and access to memories such as instruction caches, data caches, register rename tables, and translation lookaside buffers.

A common operation when describing hardware is to extract certain bits from an integer. If x is an integer, $x_{j..i}$ specifies an integer of length $(j - i + 1)$ comprising of bits $x_i, x_{i+1}, \ldots, x_j$ of x.

Example. The following statement describes the execution of an `add` instruction in a microprocessor.

$$\mathit{reg}[i_{15..11}] := \mathit{reg}[i_{25..21}] + \mathit{reg}[i_{20..16}]$$

The array *reg* is the register file, and variable i has the instruction being executed. Bits 15 to 11 specify the index of the register where the result of the addition is stored, and bits 25 to 21 and 20 to 16 specify the registers that are to be added. ▮

Often different bits of a variable have specific meanings. In the example above, bits 25 through 21 of variable i specify the destination register index. It is convenient to name these bit-fields to make the program more readable. We use *records* as a mechanism for naming fields of a variable. For instance, we can define

```
regfmt ≡ struct {
             int op: 6;
             int rs: 5;
             int rt: 5;
             int rd: 5;
             int sa: 5;
             int func: 6;
         }
```

which would correspond to the encoding of register-format MIPS instructions. The *op* field corresponds to bits 31 to 26, the *rs* field corresponds to bits 25 to 21, etc. If variable i is treated as a *regfmt* integer, we can rewrite the example above as shown below:

$$\mathit{reg}[i.\mathit{rd}] := \mathit{reg}[i.\mathit{rs}] + \mathit{reg}[i.\mathit{rt}]$$

2.4. Sequential Composition

Given two program statements S and T, the sequential composition of S and T is denoted by $S; T$. Its behavior is described as follows: first execute S, and once S has completed execution, execute T. The semicolon is an associative operation, i.e., $(S; T); U$ is the same as $S; (T; U)$. Therefore we can write a number of sequential operations without any parentheses. The operation is clearly not commutative, i.e. $S; T$ is not guaranteed to be the same as $T; S$.

2.5. Control Structures

The guarded command language has two basic control structures: selection and repetition. The former provides the functionality of **if**-statements in programming languages and the latter provides the functionality of **while**-loops.

2.5.1. Selection

The *deterministic selection* statement is shown below.

$$[\ G_1 \longrightarrow S_1 \ \talloblong\ G_2 \longrightarrow S_2 \ \talloblong \cdots \talloblong\ G_n \longrightarrow S_n\ ]$$

G_1, G_2, …, G_n are Boolean-valued expressions called *guards*, and S_1, S_2, …, S_n are program parts. The program fragment $G_i \rightarrow S_i$ is called a *guarded command*. The execution of this selection statement can be described as follows: wait for at least one of the guards to become true; execute an arbitrary S_i for which G_i holds. The deterministic selection statement has a further restriction that at most one guard can be true at any point in a computation.

Example. The following program sets y to be the absolute value of x.

$$[\ x \geq 0 \longrightarrow y := x \ \talloblong\ x < 0 \longrightarrow y := -x\ ]$$

▮

If multiple guards could be true, we use the *non-deterministic selection* statement shown below.

$$[\ G_1 \longrightarrow S_1 \ \mid\ G_2 \longrightarrow S_2 \ \mid \cdots \mid\ G_n \longrightarrow S_n\ ]$$

The execution of this statement is identical to the deterministic selection statement. When multiple guards are true, the statement picks any one true guard G_i and executes the corresponding S_i. We do not assume anything about the selection mechanism, and the choice of which guard is selected for execution is said to be *demonic*. (The idea here is that there is a demon waiting to pick the alternative that makes the program fail! Therefore, the only way to ensure that the program always behaves correctly is to not assume anything about which true guard is chosen when multiple guards are true.) Note that the only difference between the execution of the deterministic and non-deterministic selection is that in the former case at most one guard can be true.

Example. The following program sets x to either **true** or **false**.

$$[\mathbf{true} \longrightarrow x\uparrow \;\mid\; \mathbf{true} \longrightarrow x\downarrow\;]$$

When multiple guards are true, we do not assume anything about which guard is chosen for execution. Therefore, we cannot say anything about the final value of x. ∎

Example. The presence of non-determinism can make programs that are normally written in an asymmetric fashion symmetric.

$$[x \leq y \longrightarrow min, max := x, y \;\mid\; y \leq x \longrightarrow min, max := y, x\;]$$

The program fragment shown above sets *min* to the minimum of x and y, and *max* to the maximum of x and y. When $x = y$, the execution path is non-deterministic (though the result is same). Although such symmetry is pleasing, we will strive to eliminate it when describing asynchronous systems. The reason for this is that non-deterministic choice is costly to implement when compared to deterministic choice. ∎

When describing asynchronous circuits, we are often faced with a situation in which we are simply waiting for some condition to be true before proceeding. The statement $[G]$, an abbreviation for $[G \rightarrow \mathbf{skip}]$, corresponds to this behavior and can be read "wait for G to become true."

2.5.2. Repetition

The *deterministic repetition* statement is shown below.

$$*[\; G_1 \longrightarrow S_1 \;[]\; G_2 \longrightarrow S_2 \;[]\cdots[]\; G_n \longrightarrow S_n \;]$$

G_1, G_2, …, G_n are Boolean-valued expressions called *guards*, and S_1, S_2, …, S_n are program parts. The execution of this repetition statement can be described as follows: repeatedly execute some S_i for which G_i is true. If all guards are false, terminate. The deterministic repetition statement has a further restriction that at most one guard can be true at any point in a computation.

Example. The following program executes the program fragment S ten times.

```
   $i := 0;$
*[ $i < 10 \longrightarrow S; i := i + 1$ ]
```

∎

Example. The following program describes Euclid's algorithm for computing the greatest common divisor of two numbers. We use assertions to specify conditions on the input to the algorithm, as well as to specify the result of the computation.

```
 $\{X > 0 \wedge Y > 0\}$
 $x, y := X, Y;$
*[$x > y \longrightarrow x := x - y$
 ▯$x < y \longrightarrow y := y - x$
 ]
 $\{x = gcd(X, Y)\}$
```

▮

If multiple guards can be true, we use the *non-deterministic repetition* statement shown below.

*[$G_1 \longrightarrow S_1$ | $G_2 \longrightarrow S_2$ |$\cdots$| $G_n \longrightarrow S_n$]

The execution of this statement is identical to the deterministic repetition statement. The statement picks some true guard G_i, and executes the corresponding S_i. If all guards are false, the statement terminates. Once again, we do not assume anything about which true guard is chosen for execution when multiple guards are true.

Most CHP programs that describe hardware correspond to non-terminating computations. The statement *[S], an abbreviation for *[**true** $\rightarrow S$], corresponds to this behavior and can be read as "repeat S forever."

2.6. Replication Construct

The *syntactic replication construct* is shown below:

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

i is the running index, n and m are constant integer expressions, and $S(i)$ is a statement parameterized by the running index. When $n \leq m$, the construct is equivalent to:

$S(n)\, \mathbf{op}\, S(n+1)\, \mathbf{op} \cdots \mathbf{op}\, S(m)$

This operator is useful when describing classes of programs that are parameterized by an integer variable.

Example. A *decoder* takes an integer whose value ranges from 1 to N and sets one of N different Boolean-valued variables to **true**. It can be written as a deterministic selection statement that examines the value of an integer variable y and sets the appropriate Boolean-valued variable **true**. Assuming all the x_i variables are false initially, we write a decoder as the following selection:

[$\langle$▯$i : 1..N : y = i \longrightarrow x_i\uparrow \rangle$]

▮

2.7. Concurrency

In order to describe a computation, we use the notion of *sequencing*, i.e., the description of how different actions in the system are ordered in time. The sequential composition operator ";" is one way to order two actions. However, we do not make assumptions about the *time* at which actions are executed. Relying on a notion of time requires a complete understanding of the physical devices used to implement the computation in order to determine whether the computation is correct. Ignoring timing considerations for reasoning about the correctness of the computation permits us to freely adjust physical parameters for performance without having to worry about correctness. It also makes the design relatively technology-independent—another desirable property. An important consequence of not making assumptions about time for the purposes of describing the behavior of a computation is that the relative speeds at which two sequential parts of a concurrent computations are carried out are arbitrary.

We denote the concurrent composition of two sequential programs p and q by $p \parallel q$. Intuitively, this means that both p and q could execute at the same time. In order to define precisely what we mean by concurrency, let us examine the following three examples of concurrent programs where both x and y are integer variables.

1. $\{x = 0\}\ \ (x := 1 \ \ \parallel \ \ y := 2)$
2. $\{x = 0\}\ \ (x := 1 \ \ \parallel \ \ x := 2)$
3. $\{x = 0\}\ \ (x := x + 1 \ \ \parallel \ \ x := 3)$

We have used assertions to indicate that $x = 0$ initially.

In example 1, it seems reasonable to assume that the result of executing $x := 1$ and $y := 2$ in parallel causes both x to be set to 1 and y to be set to 2. This brings us to our first assumption about concurrent composition:

> **Non-interference.** *The concurrent activities of two programs that do not share variables do not interfere with each other.*

It is important to realize that we are indeed making a real assumption; this assumption isn't "obvious." The assumption we have made is that the physical implementation of actions that do not share variables do not interfere with one another. Some effects we are ignoring by this assumption include charge-sharing, cross-talk, ground bounce, capacitive coupling, inductance phenomenon, and other sources of noise in the circuit. It is possible to violate this assumption by careless physical design. Therefore, we need to ensure that these effects are in fact negligible during the final stages of circuit design and physical layout.

In the second example, we attempt to set x to both 1 and 2. A typical assumption made by models of concurrency used to describe software is that the two actions will be *interleaved* in a non-deterministic manner. In other words, the effect of the parallel composition of $x := 1$ and

$x := 2$ can be deduced by examining the different possible *interleavings* of actions. In this case, we have two interleavings: $x := 1; x := 2$ and $x := 2; x := 1$. This assumption is known as the *atomicity postulate*—we have assumed that the two write operations $x := 1$ and $x := 2$ cannot be split into smaller actions, and therefore must be interleaved in some way. Therefore, we might be tempted to conclude that $x = 1 \vee x = 2$ is the final state in the second example.

However, if we examine what might happen when the physical implementation of a variable x attempts to set x to two different values simultaneously we find that the result of the operation cannot be determined. Indeed, the result of such an operation could cause the voltages levels for logic 1 and logic 0 to be connected together, resulting in a short-circuit and possibly permanent damage to the physical implementation! Therefore, we are forced to conclude that the second example is erroneous—we must ensure that we do not attempt to assign two different values to one variable concurrently.

In the third example, we attempt to increment x and set it to 3 in parallel. However, the operation $x := x + 1$ consists of *two* operations on variable x: a read followed by a write. We can analyze the behavior of $x := x + 1$ by using our non-interference postulate. Recall that an assignment statement evaluates the expression on the right hand side of the assignment, and then assigns the result to the variable on the left hand side. We introduce a fresh variable r that is not used anywhere else in the program, and break the statement $x := x + 1$ into $r := x; r := r + 1; x := r$. Now each assignment performs at most one operation on the shared variable x. Using the interleaving model of concurrency, there are four possible executions where at most one action occurs at a time.

$$\{x = 0\}\ \ x := 3; r := x; r := r + 1; x := r\ \ \{x = 4\}$$
$$\{x = 0\}\ \ r := x; x := 3; r := r + 1; x := r\ \ \{x = 1\}$$
$$\{x = 0\}\ \ r := x; r := r + 1; x := 3; x := r\ \ \{x = 1\}$$
$$\{x = 0\}\ \ r := x; r := r + 1; x := r; x := 3\ \ \{x = 3\}$$

However, because there is the possibility of two possible values being assigned to x simultaneously (i.e., $x := 3$ and $x := r$ can execute in parallel), we must conclude that the third example is also erroneous for the reasons outlined above. Once again, we cannot assume that two concurrent write operation to one variable are ordered in any way. Therefore, our model of concurrency does not postulate that writes are atomic operations. This is a significant departure from traditional models of concurrency.

To continue exploring the definition of parallel composition, consider the following four examples, where x and y are Boolean-valued variables.

4. $\{\neg x \wedge \neg y\}\ \ (x\!\uparrow\ \ \|\ \ y := x)$
5. $\{\neg x \wedge \neg y\}\ \ (x\!\uparrow\ \ \|\ \ [x]; y := x)$

In example 4, we have a concurrent read and write operation on variable x. We can conclude that x will be set to **true** in the final state, because there is only one write operation on x. However, we cannot say anything about the final value of y. Indeed, if we think of the physical implementation of x being a voltage that represents the logic value of x, y is attempting to sample the value of x when it is changing from logic 0 to logic 1. The implementation of this operation is complex, and requires a special circuit called a *synchronizer* that guarantees that y will be assigned the value **true** or **false**. We almost never assign the value of a changing variable to another due to the complexity of its implementation.

In example 5, the wait operation $[x]$ guarantees that $y := x$ will only be executed after $x\uparrow$ has completed. Therefore, the final state in example 5 is one in which both x and y are **true**. Although x is changing when it is being read by the wait operation, we do not attempt to assign it to y until it has changed to **true**. This only requires that the change of x from **false** to **true** be *monotonic*. Therefore, this is the only assumption we make about the underlying physical implementation of the computation.

Programs that describe VLSI circuits are typically infinite computations. The last two examples we consider comprise non-terminating programs.

6. $\{\neg x \wedge \neg y\}$ $(*[\ x\uparrow; x\downarrow\ ] \parallel *[\ y\uparrow; y\downarrow\ ])$
7. $\{\neg x \wedge \neg y\}$ $(*[\ x\uparrow; x\downarrow\ ] \parallel *[\ [x]; y\uparrow; y\downarrow\ ])$

In example 6, we have two concurrent programs that toggle the value of variables x and y respectively. The result of their concurrent composition is an infinite computation. Since the concurrent programs do not interact via shared variables, we assume that actions from one program cannot artificially prevent actions from another program from executing. We postulate that actions from *both* processes get a chance to execute eventually; for instance, the following two interleavings are assumed to be impossible:

$x\uparrow; x\downarrow; x\uparrow; x\downarrow; \ldots$

$y\uparrow; y\downarrow; y\uparrow; y\downarrow; \ldots$

The first interleaving only has actions from process $*[x\uparrow; x\downarrow]$, and the second only has actions from $*[y\uparrow; y\downarrow]$. We also disallow any interleaving that only contains a finite number of operations on x or a finite number of actions on y for similar reasons. If interleavings having a finite number of operations on x are permissible, then this would mean that after a finite number of actions of the computation have completed actions from $*[x\uparrow; x\downarrow]$ no longer get a chance to execute. Therefore, the only permissible interleavings are those that contain an infinite number of operations on both x and y—i.e., those that contain an infinite number of actions from both processes.

However, it is important to realize that we do not rule out *any other* interleaving because we make no assumptions about the relative speeds of the two concurrent programs. The only requirement of a valid execution is that both programs make forward progress.

In example 7, each iteration of $y\uparrow; y\downarrow$ is preceded by an operation that waits for x to be **true**. However, the wait operation $[x]$ is not guaranteed to complete because the value of x keeps changing. The implementation might be constructed in a manner where x is found to be **false** whenever we evaluate the condition x in the wait statement. Therefore, in example 7, we permit executions where only a finite (possibly zero) number of y actions occur. These assumptions are captured by the following *fairness* postulate.

> **Weak Fairness.** *Any action that is enabled to execute and stays enabled will get a chance to execute eventually.*

We say that the parallel composition operation "$\|$" is *weakly fair*. We sometimes denote $S_1 \| S_2$ by S_1, S_2 using "," instead of "$\|$." The two operators are identical, except they have different precedence rules.

2.8. Communication and Synchronization

As we have seen above, reading and writing shared variables concurrently can be problematic. To prevent such problems, we avoid using shared variables as far as possible. Therefore, we need another mechanism that permits concurrently executing processes to exchange information. The mechanism we use is *message-passing*.

Two processes can exchange messages if they are connected by a communication *channel*. The two endpoints of the channel are known as *ports*. A process can send a message on an outgoing communication port, and receive a message on an incoming communication port. To avoid introducing sharing, there can be only one process that can send a message on a channel, and one process that can receive a message from a channel. If X is an outgoing communication port, we can send a message on port X by the command

$$X!e$$

where e is an expression. The action evaluates the expression, and sends its value on port X. If Y is an incoming communication port, we can receive a message from Y into local variable v by the command

$$Y?v$$

The action removes the next message from the communication channel connected to Y and stores it in variable v.

Channels are first-in first-out and unidirectional. In addition, since channels are going to be implemented on a VLSI chip in a controlled environment, we assume that messages are neither generated, lost, nor corrupted by the channel. Since messages cannot be received before they are sent, sends and receives also implement a form of *synchronization*.

To any command X we can attach a counter $\mathbf{c}X$ which tells us how often the command has executed. More formally, the counter $\mathbf{c}X$ is the number of *completed* X-actions in the computation. This counter is a form of *ghost variable*—it is an auxiliary variable introduced to reason about the computation. The value of this variable is zero initially, and increments every time an X-action completes.

We reason about the synchronization behavior of two actions A and B by examining the difference $\mathbf{c}A - \mathbf{c}B$. Two commands A and B are said to be *synchronized* if the difference $\mathbf{c}A - \mathbf{c}B$ is bounded.

Example. Consider the following three CHP processes.

1. $*[\ A\ ]\ \|\ *[\ B\ ]$
2. $*[\ A; B\ ]$
3. $*[\ (A \| B)\ ]$

In process 1, the difference $\mathbf{c}A - \mathbf{c}B$ cannot be bounded in any manner. The difference can take on any positive or negative integer value. (This is a consequence of our weak fairness assumption.) Inserting assertions into process 2 as shown below, it is clear that $0 \leq \mathbf{c}A - \mathbf{c}B \leq 1$.

$$*[\ \{\mathbf{c}A - \mathbf{c}B = 0\} A; \{\mathbf{c}A - \mathbf{c}B = 1\} B\ ]$$

Similarly, in process 3 we have $-1 \leq \mathbf{c}A - \mathbf{c}B \leq 1$. In the second and third processes, the two actions A and B are synchronized by the semicolons in the text of the program. ∎

Two commands A and B form a pair of *synchronization primitives* if the difference $\mathbf{c}A - \mathbf{c}B$ is bounded in all contexts. In the examples shown above, we would not conclude that A and B were synchronization primitives because there is a context (namely the first CHP program in the example above) where the difference $\mathbf{c}A - \mathbf{c}B$ is not bounded.

Let S be a send action, and R be a receive action. Since we have assumed that a message cannot be received before it is sent, we conclude that[†]

$$0 \leq \mathbf{c}S - \mathbf{c}R \quad .$$

The maximum difference $\mathbf{c}S - \mathbf{c}R$ is called the *slack* of the communication channel. There are three categories of the value of this slack:

- infinite slack
- positive, finite slack
- zero slack

[†] $\mathbf{c}R$ can be greater than $\mathbf{c}S$ if the communication channel has data in it in the initial state. We assume such initialization is explicit in the CHP program.

It is easy to establish that these three categories have the same expressive power as far as their synchronization behavior is concerned (to implement infinite slack primitives with finite/zero slack ones, we introduce arbitrary precision integer variables and an infinite array for storing data).

The synchronization slack tells us the number of sends that can complete before a receive must complete. Every time a send completes, the message it sends must be stored in the channel. Therefore, the synchronization slack corresponds to the amount of storage present in the communication channel.

Since we will eventually translate CHP programs into VLSI circuits, we have to implement any storage present in the channel. This rules out infinite slack as our model because it is not implementable. Instead we assume that all channels have zero slack unless otherwise explicitly mentioned. Assuming slack zero channels removes any "hidden" storage we may need to implement once we translate our programs into circuits. For slack zero channels, we have the following simple constraint:

$$\mathbf{c}S = \mathbf{c}R$$

In other words, the completion of the Nth send action coincides with the completion of the Nth receive action. Matching sends and receives implement a form of distributed assignment statement. If S and R are ports connected together by a communication channel,

$$(S!e \| R?v) \quad \equiv \quad v := e \qquad .$$

Namespace. When we write a CHP process, all variable names are local to that process unless explicitly stated otherwise. This prevents shared variables by default. We follow the convention that ports that are connected by a channel are given the same local name. This makes programs a bit easier to read. In addition, we typically parameterize a process by its channels. For instance, the following describes a process parameterized by two ports A and B:

$$P(A,B) \quad \equiv \quad *[\ A?x; B!x\ ]$$

When communication actions are only used for synchronization and don't carry data, we can drop the "!" and "?" symbols. In other words, we can write process $*[A?; B!]$ as $*[A; B]$. This notation also captures the fact that for slack zero communication channels which do not carry data, the send and receive actions are symmetric.

2.9. Probes

If a send (receive) action is reached, and the matching receive (send) action has not yet been reached, the send (receive) cannot complete because otherwise the constraint $\mathbf{c}S = \mathbf{c}R$ would be violated. In this case, the action is forced to block. Note that since we have only one sender and one receiver for a channel, there can be at most one process blocked on a given send/receive action. We introduce a ghost variable $\mathbf{q}X$ to reason about a blocked action. The variable $\mathbf{q}X$ is

Boolean-valued and is **true** just when there is a pending X-action. If S and R are communication ports connected by a slack zero channel, $\neg\mathbf{q}S \vee \neg\mathbf{q}R$ is invariant. This invariant captures the property that we cannot artificially suspend two matching communication actions simultaneously.

Sometimes it is useful to know if there is a pending communication action on a port. We use the *probe* to determine if there is a pending communication action on a channel. Let S and R be two ports connected by a slack zero communication channel. The probe of S, denoted $\overline{S}$, has the following two properties:

$$\overline{S} \Rightarrow \mathbf{q}R$$
$$\mathbf{q}R \Rightarrow \Diamond\overline{S}$$

The first property states that if the probe of S is **true**, then $\mathbf{q}R$ must be **true** as well—i.e., there is a pending R-action. The second property states that if $\mathbf{q}R$ is **true**, then *eventually* (denoted by $\Diamond$) the probe of S will become **true**. The probe is symmetric, so we can probe either S or R. The probe of a port gives us information about the **q**-value of the other end of the channel. We restrict the use of probes by only permitting them to be used in guards.

For channels with non-zero slack and with multiple senders and receivers, we can define $\mathbf{q}S$ as the number of S-actions that are blocked. If the slack of the channel is k, the safety property for the channel is:

$$0 \le \mathbf{c}S - \mathbf{c}R \le k$$

The quantity k is denoted $\mathbf{k}S$. The progress requirement is:

$$(\mathbf{q}S = 0 \vee \mathbf{q}R = 0) \wedge (\mathbf{q}S > 0 \Rightarrow \mathbf{c}S - \mathbf{c}R = k) \wedge (\mathbf{q}R > 0 \Rightarrow \mathbf{c}S = \mathbf{c}R)$$

When $k = 0$, the second and third conjunct in the statement above reduce to **true**, thereby reducing the progress requirement to the one we have stated above. The probe can be defined by the following properties:

$$\overline{S} \Rightarrow (\mathbf{c}S - \mathbf{c}R < k \vee \mathbf{q}R > 0) \wedge (\mathbf{c}S - \mathbf{c}R < k \vee \mathbf{q}R > 0) \Rightarrow \Diamond\overline{S}$$
$$\overline{R} \Rightarrow (\mathbf{c}S > \mathbf{c}R \vee \mathbf{q}S > 0) \wedge (\mathbf{c}S > \mathbf{c}R \vee \mathbf{q}S > 0) \Rightarrow \Diamond\overline{R}$$

Substituting $k = 0$ and using the safety property, we have the same conditions as stated above.

If the probe of port S is **true**, this implies that $\mathbf{q}R$ is **true**, i.e. there is a pending R-action. The only way this R action can complete is for S to be executed (since $\mathbf{c}R = \mathbf{c}S$). Since ports are not shared and a channel has only one sender port and one receiver port, the only process that can execute an S-action is the process that is probing S. Therefore, the only way the probe $\overline{S}$ can become **false** is for S to be executed by the process probing S. There is no action the environment can perform that can make $\overline{S}$ become **false**. However, when $\overline{S}$ is **false**, it can become **true** at any time by the execution of an R action. This property of probes is known as *stability*.

Stability. *If $\overline{X}$ is* **true**, *it remains* **true** *until the next X-action. The* **true** *value of a probe is stable. If $\overline{X}$ is* **false**, *it can change from* **false** *to* **true** *at any point. The* **false** *value of a probe is unstable.*

If the probe of port S is **true**, $\mathbf{q}R$ must be **true**. This implies that there is an R-action pending. Therefore, executing an S action is guaranteed not to suspend. For example, consider the program fragment shown below:

$$\overline{S} \longrightarrow ...\ S!0$$

Since $\overline{S}$ is **true** when the guarded command executes and the **true** value of the probe is stable, $\overline{S}$ remains **true** until we execute $S!0$. As a result, $S!0$ cannot suspend because of the pending R action. We state this property of probes as follows:

Suspension. *A probed communication action cannot suspend.*

Probes are useful only when we do not know what communication action is to be performed next. As an example of how not to use probes, consider the following program:

$$[\overline{S} \longrightarrow S!0]$$

The program waits for $\overline{S}$ to be **true** and then completes the communication on port S. Such a construct is redundant (and as we will see later, restricts the implementation choice for S) and should never be used. In this case we could have simply used $S!0$ without any selection statement and probe, because if the receiver is not ready the action automatically blocks.

2.10. Simultaneous Composition

During program development, it is useful to keep actions tightly synchronized because it makes the computation easy to reason about. We introduce a specialized synchronization construct known as the *bullet* that is used to synchronize two communication actions. If S_1 and S_2 are two communication actions, the statement

$$S_1 \bullet S_2$$

enforces the condition $\mathbf{c}S_1 = \mathbf{c}S_2$. Simultaneous composition is associative and commutative.

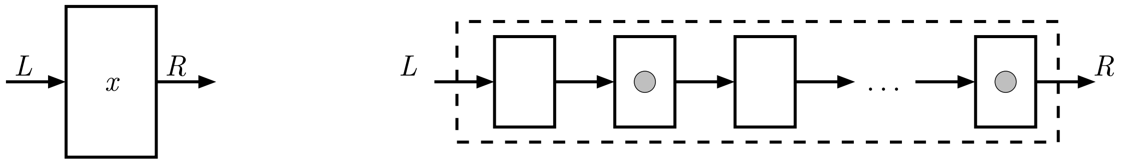
When we implement a single synchronization action using multiple actions on shared variables, we define the bullet operator by interleaving the actions that implement the two communication actions. This is consistent with our semantics because the environment cannot detect that we are not "simultaneously" executing the two communication actions. This is because probes—the only mechanism by which the environment can inspect the state of the communication channel—are not instantaneous.

Table 2.1 shows the precedence of the different composition operators. As an example, consider

$$A; B, C \| D \bullet E; F, G$$

Inserting parentheses to indicate grouping by using the precedence table results in

$$(A; (B, C)) \| ((D \bullet E); (F, G))$$



A simple one-place buffer

Linear chain of buffers

Figure 2.1. Linear buffers, showing a single buffer element as well as a linear array of buffers. Shaded circles correspond to data items.

2.11. CHP Examples

The programming notation we have introduced can succinctly describe complex asynchronous computations. In this section we look at several examples that occur frequently when designing asynchronous systems.

2.11.1. Linear Buffers

A *one-place buffer* is a process that has two ports: an input port L, and an output port R. The process reads data from L and sends it out on port R. The CHP that corresponds to this description is given by:

$$*[\ L?x;\ \ R!x\ ]$$

The reason the process acts as a buffer is because when the process is at the semicolon between the actions $L?x$ and $R!x$, the process holds a data item in local variable x. If we use a ghost variable *empty* that is **true** just when the process holds valid data, we can annotate the CHP with assertions as follows:

$$*[\ \{empty\}L?x\{\neg empty\};\ \ \{\neg empty\}R!x\{empty\}\ ]$$

We can construct an N-place buffer by connecting N one-place buffers in a linear array, where each process runs concurrently. We can write this (for $N \geq 2$) as follows:

$$BUF(L,R) \ \equiv\ *[\ L?x;\ \ R!x\ ]$$
$$NBUF(L,R,N) \ \equiv\ (\|i : 1..N-2 :\ \ BUF(C_i, C_{i+1}))\ \|\ BUF(L, C_1)\|BUF(C_{N-1}, R)$$

Notice the use of the same name to connect two ports by a channel. An N-place buffer is an implementation of slack N for a channel. Whenever we use channels with non-zero slack, the

Operator	**Meaning**
•	bullet, simultaneous composition
,	comma, parallel composition
;	semicolon, sequential composition
‖	parallel, parallel composition

Table 2.1: Operator precedence, from highest to lowest.

default implementation of the slack is a linear buffer. Figure 2.1 shows the linear buffer.

2.11.2. Non-deterministic Merge

Suppose two processes want to send messages on the same channel. Since we have restricted ourselves to channels with one sender and one receiver, we cannot allow two processes to directly access a single sender port. The non-deterministic merge can be used to resolve this issue. The process has two input ports, and one output port. It receives an input along any of the two input ports, and sends the value it received on the output port. The CHP for this process is:

```
*[[$\overline{X} \longrightarrow X?a; Z!a$
 |$\overline{Y} \longrightarrow Y?a; Z!a$
 ]]
```

Observe that we are using probes only because we do not know which communication action can occur. This process functions correctly even if there is no communication on port X. Figure 2.2 pictorially depicts the behavior of this process.

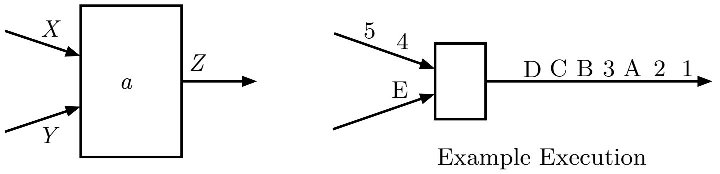
The merge also introduces some buffering because local variable a can hold a data item. To eliminate this buffering, we use the special syntax shown below:

```
*[[$\overline{X} \longrightarrow Z!(X?)$
 |$\overline{Y} \longrightarrow Z!(Y?)$
 ]]
```

The statement $Z!(X?)$ can be read as: "send the value received on port X on port Z." When written in this form, the merge process has the property that $\mathbf{c}X + \mathbf{c}Y = \mathbf{c}Z$.

One problem with the merge process is that it is possible for it to never receive a value on port X, even if $\overline{X}$ is **true** because there might always be data pending on channel Y. The reason for this is that we have assumed that the selection statement is unfair. One way to make the merge process fair is to change it as follows:

```
*[[$\overline{X} \longrightarrow Z!(X?)$;  [$\overline{Y} \longrightarrow Z!(Y?)$  |$\neg\overline{Y} \longrightarrow$ skip]
 |$\overline{Y} \longrightarrow Z!(Y?)$;  [$\overline{X} \longrightarrow Z!(X?)$  |$\neg\overline{X} \longrightarrow$ skip]
 ]]
```



Example Execution

Merge Process

Figure 2.2. Non-deterministic merge, showing the merge process as well as a possible partial execution.

After receiving a value on either X or Y, this process checks the other port; if there is a pending communication on that port, it completes it and sends the value out on Z. The reason this process is a fair merge is that once $\overline{X}$ becomes **true**, it remains **true**, and the selection statement $[\overline{X} \rightarrow Z!(X?) \mid \neg\overline{X} \rightarrow \textbf{skip}]$ will eventually have only one true guard. An examination of the process shows that once $\overline{X}$ is stable **true**, all execution paths include a $Z!(X?)$ action, thereby establishing fairness. The explicit introduction of fairness typically leads to the use of negated probes.

Note that the selection statement

$$[\overline{Y} \longrightarrow Z!(Y?) \;\mid\; \neg\overline{Y} \longrightarrow \textbf{skip}]$$

was written using a non-deterministic selection. The reason for this is that we have not assumed that evaluating guards in a selection statement is an atomic action. We could evaluate the guard $\neg\overline{Y}$, and find it to be **true** (because $\overline{Y}$ was **false**). Since the **false** value of the probe is unstable, when we evaluate the guard $\overline{Y}$ we may find that it is also **true**. Therefore, the two guards are not mutually exclusive and we must use the non-deterministic selection statement.

Another way to make the merge fair is to use a "round robin" strategy. We repeatedly probe ports X and Y, completing any pending communication if possible. The CHP for this strategy is:

```
$M1 \;\equiv\; *[[\overline{X} \longrightarrow Z!(X?) \;\mid\; \neg\overline{X} \longrightarrow \textbf{skip}];$
                 $[\overline{Y} \longrightarrow Z!(Y?) \;\mid\; \neg\overline{Y} \longrightarrow \textbf{skip}]$
                $]$
```

A problem with the fair merge $M1$ is that it "busy waits"—if both $\overline{X}$ and $\overline{Y}$ are **false**, the loop still executes. At the circuit level, this translates to wasted power. We can avoid this problem by adding a statement that waits for either $\overline{X}$ or $\overline{Y}$ to be **true**. The resulting merge is shown below:

```
$M2 \;\equiv\; *[[\overline{X} \vee \overline{Y}];$
                 $[\overline{X} \longrightarrow Z!(X?) \;\mid\; \neg\overline{X} \longrightarrow \textbf{skip}];$
                 $[\overline{Y} \longrightarrow Z!(Y?) \;\mid\; \neg\overline{Y} \longrightarrow \textbf{skip}]$
                $]$
```

Depending on the environment, $M1$ may be more or less efficient than $M2$. If the environment is very active and we find that the condition $\overline{X} \vee \overline{Y}$ remains **true**, then the extra wait translates to extra circuitry that has to switch on each iteration of the loop, resulting in additional power dissipation. However, if the environment is mostly idle, the additional power dissipated per iteration might be worthwhile compared to the power wasted by iterating through the loop without doing any useful work.

An N-way non-deterministic merge can be described using the syntactic replication construct.

$$*[\;\;[\;\;(\mid i : 0..9 : \overline{X_i} \longrightarrow X_i?a;\;\; Z!a)\;\;]\;\;]$$

A generalized fair merge is given by:

```
*[[$(\vee i : 0..9 : \overline{X_i})$];
  $(; i : 0..9 :$ [$\overline{X_i} \longrightarrow Z!(X_i?)$ | $\neg\overline{X_i} \longrightarrow$ **skip**]$)$
 ]
```

2.11.3. Copy and Alternating Split/Merge

A commonly occurring CHP process is one that makes a copy of a particular input value. This is useful because we cannot read the same value from a channel multiple times. The process has one input and two outputs, and is shown below:

```
*[ $X?a$;  $Y!a, Z!a$ ]
```

Observe that we have used a comma to indicate parallel composition, thereby avoiding the use of parentheses.

Another process that occurs in high-performance asynchronous circuits is one which distributes an input stream evenly among two outputs. This process is called an *alternating split*, and is shown below:

```
*[ $X?a; Y!a$;  $X?a; Z!a$ ]
```

An alternating split is typically used in conjunction with an *alternating merge*, shown below:

```
*[ $X?a; Z!a$;  $Y?a; Z!a$ ]
```

This reads values from input ports X and Y, and sends them on channel Z. It differs from the non-deterministic merge because it merges the values received on ports X and Y in a fixed manner.

2.11.4. Controlled Split/Merge

Controlled splits and merges are used to construct data routing networks. A controlled split takes a control value and a data item. The control value indicates where the data item is to be sent. The item is then sent along the appropriate port. The following CHP process describes a controlled split with two outputs:

```
*[ $C?c, X?a$;
   [$c \longrightarrow Y!a$ ▯$\neg c \longrightarrow Z!a$ ]
 ]
```

A controlled merge performs the dual function. It receives a control value indicating where a data item is to be received from. It reads the appropriate input port and sends the data item on a fixed output port. The following CHP process describes a controlled merge with two inputs:

```
*[ $C?c$;  [$c \longrightarrow X?a$ ▯$\neg c \longrightarrow Y?a$];
   $Z!a$
 ]
```

Both the controlled split and merge use auxiliary variables to store the control value c. To eliminate this storage, we use a special notation. The notation permits us to probe the next *value* to be received on a channel. This restricted probe can only be used for input ports that carry data. The syntax for this special probe is $\overline{C_1, C_2, \ldots, C_n : E}$, where C_1, …, C_n are the input ports, and E is an expression. Channels from the list C_1,…,C_n used in E evaluate to the value of the pending input. The probe expression is **false** if $\overline{C_i}$ is **false**, for any i. Otherwise it evaluates to E. Given this extended syntax, we can defined $\overline{X}$ to be an abbreviation for $\overline{X : \textbf{true}}$. Using this new syntax, we can write the controlled split without using any auxiliary storage as ∗[[$\overline{C : C} \rightarrow Y!(X?), C?$▯$\overline{C : \neg C} \rightarrow Z!(X?), C?$]]. Note the use of $C?$ to complete the communication action without the use of any variable. The controlled merge is ∗[[$\overline{C : C} \rightarrow Z!(X?), C?$▯$\overline{C : \neg C} \rightarrow Z!(Y?), C?$]].

2.11.5. Reactive Process Structure

A commonly occurring program structure is the *reactive process structure* shown below:

```
*[[$G_1 \longrightarrow S_1$
   ▯$G_2 \longrightarrow S_2$
   ▯...
   ▯$G_n \longrightarrow S_n$
  ]]
```

Operationally, this process structure can be described as follows: "repeatedly execute the following: wait for some G_i to be **true**; execute the corresponding S_i." A process having this structure waits for its environment to execute some action, and reacts to it. The guards typically contain probes, and the environment selects different actions from the process by executing different communication actions.

2.12. Recursive Constructions

A powerful technique used in the construction of complex asynchronous programs is the technique of induction. When faced with a design problem of size N, we break it up into design problems of size $< N$ and use additional processes to construct the solution to the problem of size N. Once we solve the base case (typically $N = 1$) directly, we can construct solutions for any value of N.

We have already seen an example of such a construction in the form of the N-place buffer. The base case $N = 1$ is the process ∗[$L?x; R!x$]. We can extend a buffer of size $N - 1$ to a buffer of size N by adding process ∗[$L?x; R!x$], and connecting R to the input of the buffer of size $N - 1$.

Processes designed by this method are said to be *recursively constructed*. In this section we look at some more examples constructed in this manner.

2.12.1. Lazy Stack

A stack is a last-in first-out structure, with two ports connected to the environment:

- *push*: An input port by which the environment adds elements to the stack;

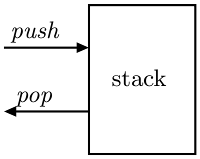


Figure 2.3. Stack Interface.

- *pop*: An output port by which the environment removes elements from the stack.

We assume that the environment guarantees mutual exclusion between *push* and *pop* operations. Figure 2.3 shows the interface the stack presents to the environment.

Using an array to store the values in the stack, we can solve the problem as follows:

```
$n := 0;$
$*[[\overline{push} \wedge n < N \longrightarrow push?x[n]; n := n + 1$
  $[] \overline{pop} \wedge n > 0 \longrightarrow pop!x[n-1]; n := n - 1$
$]]$
```

The process maintains the property that the elements stored in the stack are $x[0]$, $x[1]$, …, $x[n-1]$, and that $x[n-1]$ was the last element inserted into the stack. (To show these properties hold requires a stronger invariant, namely that the elements $x[0]$, …, $x[n-1]$ are ordered by insertion time.)

To add concurrency, we would like to construct the stack as the parallel composition of a number of processes. We do so by using a recursive construction. We construct an N-place stack by assuming the existence of a $(N-1)$-place stack.

The base case for our construction is a 1-place stack. When a *push* operation is requested on a 1-place stack, the stack simply accepts the *push* operation and stores the data value in variable x. When a *pop* is requested, it returns the stored data value. The CHP process that does this is shown below:

```
$*[[\overline{push} \longrightarrow push?x$
  $[] \overline{pop} \longrightarrow pop!x$
$]]$
```

When the 1-place stack overflows (i.e. we attempt 2 *push* operations in sequence), this process overwrites the old data value. When the stack underflows (i.e. we attempt two *pop* operations in sequence), the stack provides duplicate data items. We can change this behavior using the following 1-place stack:

```
$*[\ \{stack\ \ is\ \ empty\}\ push?x;$
    $\{stack\ \ is\ \ full\}\quad pop!x$
$]$
```

The assertions in the program indicate the state of the stack. When the stack is empty, the only

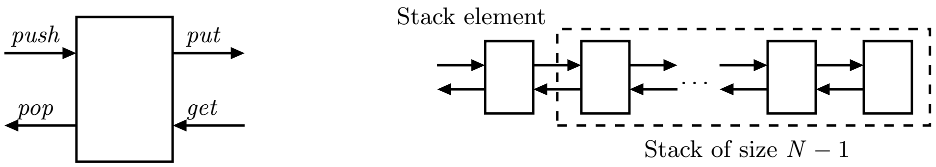


Figure 2.4. Stack element, showing the recursive construction.

operation permitted is a *push*. When the stack is full, the only operation permitted is a *pop*. A *push* operation when the stack is full deadlocks the system!

Now that we have completed the base case, we move on to the induction step. Assuming the existence of an $(N-1)$-place stack, we extend it to a N-place stack by adding a *stack element*. The stack element has the following four communication ports:

- *push*, *pop*: interface to the N-place stack
- *put*, *get*: interface to the $(N-1)$-place stack

Figure 2.4 shows the communication ports of the stack element and the recursive construction of the stack. To construct a stack element, we consider two cases: the stack element stores no data item (empty), and the stack element stores one data item (full).

Assuming the stack element is empty, the stack element can accept a *push* request and enter the full state. If the stack element receives a *pop* request, it must request data from the stack of size $N-1$ because it does not store any data itself. The value received from the smaller stack is then sent on the *pop* channel, resulting in the stack element remaining in the empty state. Therefore, the *push* and *pop* operations in the empty state can be written as follows:

```
[$\overline{push} \longrightarrow push?x \;\; \{full\}$
[]$\overline{pop} \longrightarrow get?x; pop!x \;\; \{empty\}$
]
```

Assuming the stack element is full, the stack element can accept a *pop* operation and enter the empty state. If the stack element receives a *push* request, it must first send the data item it has stored locally to the stack of size $N-1$. It can then accept the *push* operation and remain in the full state. Note that the data value sent to the smaller stack is the one previously held by the stack element, thereby maintaining the last-in first-out property. The *push* and *pop* operations in the full state can be written as follows:

```
[$\overline{push} \longrightarrow put!x; push?x \;\; \{full\}$
[]$\overline{pop} \longrightarrow pop!x \;\; \{empty\}$
]
```

We encode the state of the stack element using a Boolean-valued variable b. When b is **true**, the stack element is empty; when b is **false**, the stack element is full. The CHP for the complete stack

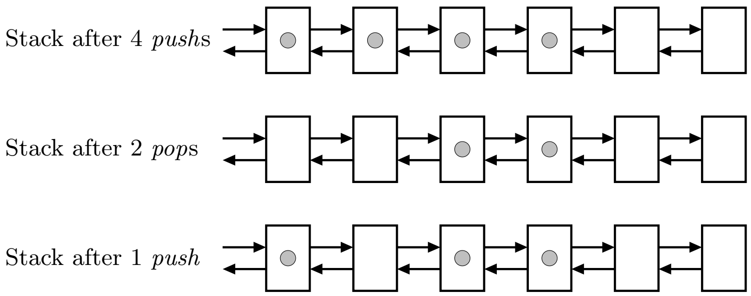


Figure 2.5. Lazy stack showing different possible configurations.

element can be written as follows:

```
$b\uparrow$;
$*[[b \wedge \overline{push} \longrightarrow push?x; b\downarrow$
  $[]\, b \wedge \overline{pop} \longrightarrow get?x; pop!x$
  $[]\, \neg b \wedge \overline{push} \longrightarrow put!x; push?x$
  $[]\, \neg b \wedge \overline{pop} \longrightarrow pop!x; b\uparrow$
$]]$
```

A neater coding of this same program is shown below:

```
$*[\{empty\}$  $[\overline{push} \longrightarrow push?x$  $[]\,\overline{pop} \longrightarrow get?x$ $]$;
   $\{full\}$   $[\overline{push} \longrightarrow put!x$   $[]\,\overline{pop} \longrightarrow pop!x$ $]$
 $]$
```

This stack is called *lazy* because it does not move data items unnecessarily. The stack can contain "holes," as depicted in Figure 2.5. Therefore, the time taken by a *push* or *pop* operation on the stack can vary depending on the configuration of the holes and data items in the stack.

2.12.2. Eager Stack

The time taken by a lazy stack to complete a *push* (*pop*) operation depends on the distance of the first "hole" (data item) from the output of the stack. An eager stack is one which attempts to keep the data items in a compact configuration to make sure that *pop* operations take constant time. In other words, we attempt to keep the stack in the first configuration in Figure 2.5, avoiding the second and third configurations. The base case for an eager stack is the same as that for a lazy stack. We now provide a construction for the eager stack element. (It has the same interface as a lazy stack element.)

Assuming the stack element is empty, the stack element can accept a *push* request and enter the full state. However, if the stack element receives a *pop* request, it can no longer request data from the stack of size $N-1$! The reason is that since the stack is in a compact configuration, we maintain the property that if a stack element is empty, the rest of the stack is empty as well. Therefore, because the stack element is empty, the stack of size $N-1$ is empty too.

Assuming the stack element is full, the stack element can accept a *push* request by sending the value stored locally to the stack of size $N-1$ and reading the value sent on the *push* channel into a local variable. If the stack element receives a *pop* request, it sends the data value it stores locally to the environment. However, we are now faced with the following dilemma: should the next operation be a *get* from the stack of size $N-1$? The reason we should be attempting a *get* is because the eager stack is supposed to store data items in a compact configuration. The problem is that a *get* should only be performed if the stack of size $N-1$ stores some data.

Instead, we adopt the following approach. The *pop* channel not only sends a data item, but also a Boolean flag that indicates whether the data item is valid or not. Now, when the stack element is empty, a *pop* request can be satisfied by sending a **false** value. If the stack element is full, it sends the locally stored value to the environment and executes a *get* operation. If the flag returned by *get* is **true**, the stack element has received a valid data item and it stays in the full state; if *get* returns a **false** value, the stack element enters the empty state. Using local variable *full* that is **true** whenever the stack element is is full, we can write the eager stack element as follows:

```
 $full\downarrow;$
*[[$\overline{push} \wedge \neg full \longrightarrow push?x; full\uparrow$
  []$\overline{pop} \wedge \neg full \longrightarrow pop!(x, \textbf{false})$
  []$\overline{push} \wedge full \longrightarrow put!x; push?x$
  []$\overline{pop} \wedge full \longrightarrow pop!(x, \textbf{true}); get?(x, full)$
 ]]
```

An examination of the CHP shows that *pop* operations now always complete in constant time. *push* operations are not constant time if the stack element is in the full state. We introduce a temporary variable y as additional storage and make the *push* operations constant time as follows:

```
 $full\downarrow;$
*[[$\overline{push} \wedge \neg full \longrightarrow push?x; full\uparrow$
  []$\overline{pop} \wedge \neg full \longrightarrow pop!(x, \textbf{false})$
  []$\overline{push} \wedge full \longrightarrow push?y; put!x; x := y$
  []$\overline{pop} \wedge full \longrightarrow pop!(x, \textbf{true}); get?(x, full)$
 ]]
```

While this stack can complete communications on both *push* and *pop* in constant time, it does waste some storage because the variable y only holds data temporarily.

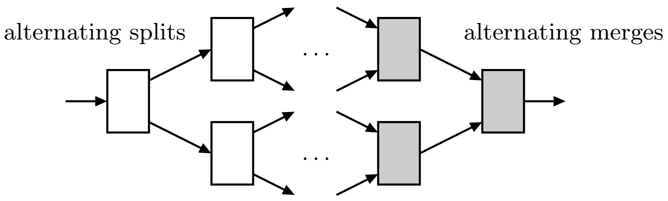


Figure 2.6. Binary Tree FIFO.

2.12.3. Tree Buffers

An N-place buffer has poor performance when N is very large. The problem arises if the buffer is mostly empty. The time taken for a data item to travel from the input to the output is proportional to the length of the buffer. Similarly, when the buffer is mostly empty, its performance is limited by the rate at which "holes" can be transported from the output back to the input. In this section, we construct a buffer of size $O(N)$ with $O(\log N)$ latency using a recursive construction.

We attempt to construct a buffer of size $2^n - 2$. When $n = 1$, this is a buffer of size zero, which corresponds to connecting the input port to the output without any intervening process. Assuming we have two buffers of size $2^n - 2$, we can construct a buffer of size $2^{n+1} - 2$ by connecting the two outputs of an alternating split to the inputs of the two buffers of size $2^n - 2$, and connecting the two outputs of the buffers of size $2^n - 2$ to the inputs of an alternating merge. The amount of buffering we have is $2 \cdot (2^n - 2) + 1 + 1 = 2^{n+1} - 2$, completing the construction. The resulting buffer has a latency of $2n - 2$ steps. This tree buffer is shown in Figure 2.6.

The tree buffer consists of a tree of alternating splits followed by a tree of alternating merges. Suppose we eliminate the tree of alternating merges and examine the outputs produced at the leaves of the alternating split tree. Figure 2.7 shows the resulting process structure and the outputs produced at the leaves of the alternating split tree for a tree with four leaves. If we permute the outputs of the tree, this process structure can be used to convert a serial input stream into a parallel output stream.

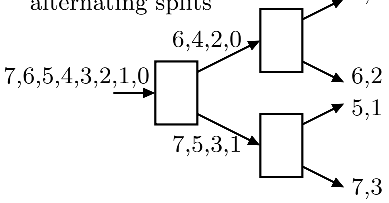


Figure 2.7. Serial-to-parallel Converter.

2.13. Performance Estimation

When designing large asynchronous systems, the largest performance gains occur due to design decisions made at the CHP level. In this section, we explain how to estimate the performance of individual CHP processes by induction on the process structure. The reasoning behind these estimates is based on a canonical implementation of different CHP constructs. We assume that processes are never blocked at a synchronization action or selection statement to make the analysis simple, and we ignore wire delays.

Because we are *estimating* the performance at the CHP level without actually designing the final circuit implementation, we do not estimate the performance in units of physical time. Rather, we are counting the number of sequential steps that must be performed by the computation before it completes.

Consider the following two CHP programs:

1. $x\uparrow$
2. $x\uparrow; y\downarrow$

Here x and y are Boolean-valued variables. We assume that the operation $x\uparrow$ takes one time unit, or $O(1)$ time. We say that $\mathcal{T}(x\uparrow) = O(1)$. Similarly, $x\uparrow; y\downarrow$ take two time units, which is still $O(1)$ time. If x is an N-bit integer, we can implement the assignment

$$x := E$$

as the concurrent assignment

$$x_0, x_1, \ldots, x_N := E_0, E_1, \ldots, E_N$$

Therefore, the time taken by the assignment $x := E$ where x is an N-bit integer is also $O(1)$, i.e.,

$$\mathcal{T}(x := E) = O(1) + O(\mathcal{T}(E))$$

where $\mathcal{T}(E)$ is the time taken to evaluate the expression E. The time $\mathcal{T}(E)$ is given by the minimum number of *elementary* operations necessary to evaluate E. An elementary operation is a binary or unary operation on a Boolean-valued variable.

We have assumed that the assignment can be implemented as $x_0 := E_0, x_1 := E_1, \ldots, x_N := E_N$. If this is not possible because bits of x are used in E, then this assignment could have a cost that is more than $O(1)$ because we have to implement it as $r_0 := E_0, \ldots, r_N := E_N; x_0 := r_0, \ldots, x_N := r_N$ where r is a fresh variable. This has cost $O(\log N)$.

We can optimize the implementation $r_0 := E_0, \ldots, r_N := E_N; x_0 := r_0, \ldots, x_N := r_N$ by determining the dependencies of the bits of E on the bits of x. If E_i depends on x_j, we cannot execute $x_i := E_i$ and $x_j := E_j$ concurrently. The two operations must be ordered either using semicolons, or by introducing temporary variables. The total time taken by the assignment depends on the total number of sequential steps required to perform the assignment.

The time taken to evaluate an expression also depends on the number of copies of a particular bit that it needs. For instance, if x is an N-bit integer and y is a Boolean-valued variable, the assignment $x_0, x_1, \ldots, x_N := y, y, \ldots, y$ takes time that is more than $O(1)$ because N copies of y are required. In general, it takes $O(\log m)$

steps to make m copies of a Boolean-valued variable. The copies of different variables can be performed in parallel.

The concurrent composition of N statements completes when the longest task completes. Under the assumption that the statements never block at any synchronization action, this is just the time taken by the longest task. We are relying on the fact that, in VLSI, concurrency does not introduce any additional time penalty because the circuit implementation of concurrent actions is given by the union of the individual circuits. Therefore,

$$\mathcal{T}(S_1 \| \cdots \| S_N) \;=\; \max_i \mathcal{T}(S_i)$$

Consider two sets of statements $S_1, \ldots, S_N$ and $T_1, \ldots, T_M$. We consider the time taken by the concurrent composition of S-statements followed by the concurrent composition of T-statements. The time taken by the concurrent composition of the S-statements is given by $\max_i \mathcal{T}(S_i)$ as before, and the time taken by the concurrent composition of the T-statements is given by $\max_i \mathcal{T}(T_i)$. So the combined statement takes at least time $\max_i \mathcal{T}(S_i) + \max_i \mathcal{T}(T_i)$. There is an additional cost to the combined statement, and that cost arises from the time it takes to sequence the two sets of statements.

None of the T-statements can begin execution until *every* S-statement completes. If we think of each S-statement providing us a bit of information that lets us know when it completes execution, we have to examine N bits to know that the statement $S_1 \parallel \cdots \parallel S_N$ has completed. This inspection takes $O(\log N)$ time. Since this information must also be communicated to each T-statement, M copies of this piece of information are required. The copying operation takes $O(\log M)$ time. Therefore, we conclude that:

$$\mathcal{T}((S_1 \| \cdots \| S_N);(T_1 \| \cdots \| T_M)) \;=\; \max_i \mathcal{T}(S_i) + \max_i \mathcal{T}(T_i) + O(\log M + \log N)$$

The time taken for sending a one-bit expression over a channel is given by the time taken to evaluate the expression plus a constant amount, since we assumed that communication actions never block. N bits of information can be communicated by sending each bit in parallel. The receive operation is similar, and we conclude that

$$\begin{aligned}\mathcal{T}(C!e) &= \mathcal{T}(e) + O(1)\\ \mathcal{T}(C?v) &= O(1)\end{aligned}$$

Consider the deterministic selection statement $[G_1 \rightarrow S_1 [\!] \cdots [\!] G_N \rightarrow S_N]$ where we find $G_i \equiv \textbf{true}$. The execution of this statement corresponds to evaluating all the guards in parallel, determining which guard is true, and selecting the corresponding statement for execution. Once we have determined the index of the true guard (in this case i), we have to *stop* evaluation of the other guards because executing S_i might result in another guard G_j $(j \neq i)$ becoming **true**. The control signal that stops guard evaluation must be propagated to N different places in the circuit, a process that takes $O(\log N)$ time. Therefore, assuming that G_i evaluates to **true**, we have:

$$\mathcal{T}(\,[\,G_1 \longrightarrow S_1 \;\square\; \cdots \;\square\, G_N \longrightarrow S_N\,]\,) \;=\; \max_i \mathcal{T}(G_i) + \mathcal{T}(S_i) + O(\log N)$$

Access to array variables is more complex than access to simple variables. Reading or writing $x[i]$, where x is an array of N elements is not a constant time operation. Arrays have to be implemented using circuitry that uses the value of i to select a particular array element and then assigns to it. We can implement the assignment $x[i] := v$ with the selection statement:

```
[$i = 0 \longrightarrow x[0] := v$
$\square\, i = 1 \longrightarrow x[1] := v$
. . .
$\square\, i = N - 1 \longrightarrow x[N-1] := v$
]
```

We have changed an array indexing operation into constant assignments. An assignment of the form $x[0] := v$ takes $O(1)$ time because we can treat $x[0]$ as an ordinary variable rather than an array access. By using our analysis of guarded commands, this statement takes $O(\log N)$ time plus the time taken to evaluate the guards—$O(\log\log N)$ since i can be represented with $\lceil \log_2 N \rceil$ bits. Therefore, the entire array access takes $O(\log N)$ time.

2.13.1. Response Time

Many CHP processes have a reactive process structure, where the environment selects a particular operation by communicating on channels that are probed by the process. The environment can continue executing when the communication action completes. An important performance metric is the throughput of various operations the environment can perform. We can estimate this by measuring the time between successive operations performed by the environment.

The *response time* of a CHP program is the time between two successive communication actions performed by its environment. In the case of the lazy stack, the response time of a *pop* operation is $O(m)$, where m is the distance of the first data item from the output of the stack. As opposed to this, the response time of a *pop* operation for the eager stack is $O(1)$.

A CHP program is said to exhibit *constant response time* (CRT) when the time between two consecutive commands (the response time) is independent of the data and of the number of processes. The eager stack is an example of a process where both *push* and *pop* operations exhibit constant response time.

Example. The response time of an $L!$ operation for a linear buffer is $O(1)$ if the buffer is empty. The time between inserting and removing a particular data item is $O(N)$, and is the latency of the buffer. The response time of an $R?$ operation is $O(1)$ if the buffer is full. ∎

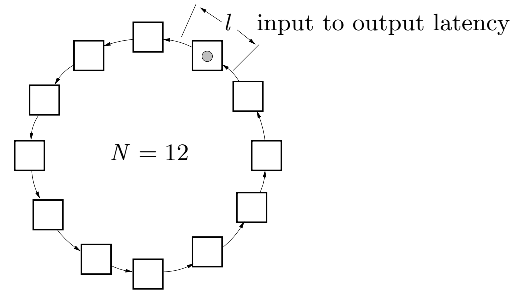


Figure 2.8. A ring of buffers with a single data item.

Example. Consider a simple ring of buffers, with $N-1$ processes of the form $*[L?x;R!x]$ and one process executing $*[R!x;L?x]$. Let τ be the time taken by one iteration of the loop $*[L?x;R!x]$, assuming that both L and R communication actions do not block. Let l be the delay between $L?x$ and $R!x$, assuming that the two actions do not block. The ring has one data item in it that moves through the buffers. We are interested in the throughput of the ring, i.e., the rate at which any one process executes the loop $*[L?x;R!x]$. Figure 2.8 shows this process structure.

The time taken by the data item to travel around the ring is Nl, and that is one source of throughput limitation. The second source is the time taken by a single loop iteration, namely τ. Therefore, the system operates at the highest possible throughput when $\tau = Nl$, i.e., when $N = \tau/l$. ∎

Example. Consider two linear pipelines PA and PB with input channels Ain and Bin, and with output channels $Aout$ and $Bout$ respectively. We can construct a larger linear pipeline out of them in various ways.

Suppose we introduce the following two processes:

$$*[\ L?x;\ \ Ain!x, Bin!x\ \ ]\ \|\ *[\ Aout?x, Bout?y;\ \ R!x\ \ ]$$

The first process is a copy, whereas the second reads the outputs of both pipelines and produces one of them on channel R. For simplicity, we have used port names that match the channel names they connect to. This new pipeline structure has a throughput that is limited by the throughput of PA and PB, and a latency that is the maximum of the latencies of the individual pipelines.

If instead we use processes

$$*[\ L?x;\ \ Ain!x;\ \ L?x;\ \ Bin!x\ \ ]\ \|\ *[\ Aout?x;\ \ R!x;\ \ Bout?x;\ \ R!x\ \ ]$$

instead of the two processes shown above, the resulting pipeline has very different characteristics. The latency of a data item depends on which pipeline it travels through. However, the throughput of the complete system is no longer limited by the throughput of the individual pipelines because we alternate between PA and PB. ∎

2.14. Energy Estimation

In CMOS technology, energy is dissipated only when a node of the circuit is switched—when a capacitor is charged or discharged. Therefore, an energy model for VLSI computation based on CMOS may assume that energy is dissipated only when the state of the computation changes—the process of waiting does not dissipate any power. This assumption is satisfied by a CMOS asynchronous circuit, but is not in general satisfied by a clocked circuit.

We derive a simple energy model from the CHP program. We can do so because the final asynchronous circuit corresponds very closely to its CHP program: for each assignment, communication and guard evaluation executed by the CHP program there is a corresponding assignment, communication, and guard evaluation computed by the CMOS implementation. The CMOS implementation dissipates energy only during the execution of various parts of the CHP program; therefore, this energy can be attributed to the energy required to execute the corresponding CHP statement. The purpose of the model presented here is to study architectural trade-offs (e.g., comparison of bit-serial and parallel implementations of a function) or to determine architectural parameters (e.g., the optimal width of a cache memory) with respect to energy consumption. A detailed model with a large number of parameters can be intractable without significantly increasing the accuracy of the model, especially if the parameters are layout-dependent (and, therefore, not well known before the layout is complete). A simpler model is desirable at the design stage; we base this model on the cost of communication, assignment, and selections.

CMOS circuits have three main sources of energy dissipation: leakage currents, short-circuit currents, and dynamic currents. The total energy dissipated during the execution of one operation, E_T, can be calculated as:

$$E_T = E_s + E_d + E_{sc}$$

where E_s is the energy dissipated by the sub-threshold leakage currents, E_d is the energy used for charging and discharging capacitors, and E_{sc} is the energy dissipated by the short-circuit currents.

Leakage currents come from the sub-threshold behavior of MOSFET's, and currently constitute a small part of the total power dissipation in CMOS processes. Short-circuit currents originate in the short transients that occur when both pull-up and pull-down transistors conduct while the input signal switches between V_{tn} and $V_{DD} - V_{tp}$. We assimilate this switching energy to the dynamic energy dissipation that represents the bulk of the total energy dissipation in a standard CMOS circuit. Dynamic energy dissipation, E_d, comes from the energy used to charge the capacitors in the circuit. The capacitors are then discharged to ground, and the energy is not recuperated. E_d

can be computed as:

$$E_d = \sum_{C_i} n_i C_i V_{DD}^2$$

where the C_i's are all the capacitors in the circuit, and n_i is the number of times capacitor C_i is switched in the execution of one operation. We rewrite this as

$$E_d = K_L V_{DD}^2$$

Based on these results, we use K_L as an *energy index* for an asynchronous CMOS circuit. This index is independent of the power-supply voltage and the speed of operation; furthermore, K_L is additive: we can calculate the index corresponding to an operation by adding the indices of all of its sub-operations.

A basic postulate of the model is that parallel composition is free: no extra circuits are required in the implementation. If there is no synchronization between the statements S_1, ..., S_N, then

$\mathcal{E}(S_1 \| \cdots \| S_N) = \sum_i \mathcal{E}(S_i)$

where $\mathcal{E}(S)$ is the energy index of statement S.

The energy index of $(S_1 \| \cdots \| S_N); (T_1 \| \cdots \| T_M)$ is given by the sum of the energy indices of the component statements, plus the energy index of the implementation of the semicolon. As previously discussed, the implementation of the semicolon involves gathering N signals that indicate the completion of execution of each S_i statement, and broadcasting this information to M places. The energy index of this operation is proportional to $M + N$. Therefore,

$\mathcal{E}((S_1 \| \cdots \| S_N); (T_1 \| \cdots \| T_M)) = \sum_i \mathcal{E}(S_i) + \sum_i \mathcal{E}(T_i) + O(M + N)$

The energy index of communication is proportional to the number of bits being communicated. If x is an N-bit quantity, then

$\mathcal{E}(C!x) = O(N)$

$\mathcal{E}(C?x) = O(N)$

The energy index of a selection statement is proportional to the sum of the indices of its guards, the statement that is selected for execution, and $O(N)$ where N is the number of guards. The last term comes from the circuitry that stops guards from evaluating, once one of the guards evaluates to true. Therefore, we write

$\mathcal{E}([G_1 \longrightarrow S_1 \ [] \ \cdots \ [] G_N \longrightarrow S_N]) = \sum_i \mathcal{E}(G_i) + \mathcal{E}(S_i) + O(N)$

where i is the index of the guard that evaluates to **true**.

The simple selection process that implements assignment $x[i] := v$ is shown below.

```
[$i = 0 \longrightarrow x[0] := v$
▯$i = 1 \longrightarrow x[1] := v$
$\ldots$
▯$i = N-1 \longrightarrow x[N-1] := v$
]
```

This process has energy index that is $O(N)$. However, we can improve this by changing the implementation of the assignment to:

```
[$\neg i_0 \longrightarrow$ [$\neg i_1 \longrightarrow \ldots$ [$\neg i_{N-1} \longrightarrow x[0] := v$ ▯ $i_{N-1} \longrightarrow x[1] := v$]
            ▯$i_1 \longrightarrow \ldots$
            ]
▯$i_0 \longrightarrow \ldots$
]
```

Effectively, instead of selecting on the result of N comparisons, we examine each bit of i separately and select the appropriate assignment statement hierarchically. While this does not change the time complexity of the operation, the energy index is reduced to $O(\log N)$. Therefore, a read or write access to an array has an energy index that is $O(\log N)$, where N is the size of the array.

Example. Consider a linear fifo of size N. Each data item travels through N processes. The fifo dissipates $O(N)$ energy per data item transported through it. ▮

2.15. Slack Elasticity*

The CHP specification of a process completely characterizes both the computation it performs as well as its synchronization behavior. For instance, we can specify a process that performs addition with the following CHP:

```
*[ $A?x, B?y;\ \ C!(x+y)$ ]
```

Unfortunately, for performance reasons, this specification can be very restrictive in practice. The specification includes the property that

$$0 \leq \mathbf{c}A - \mathbf{c}C \leq 1$$

In other words, the specification includes the fact that an implementation cannot accept its next set of inputs on channel A without producing an output on channel C. This restriction causes the throughput of an asynchronous delay-insensitive circuit that implements the computation to degrade as $1/\log N$, where N is the number of bits used in the representation of x. However, it is possible that this property of the specification is not critical—namely, modifying it to the weaker

$$0 \leq \mathbf{c}A - \mathbf{c}C \leq \log N$$

does not affect the correctness of the computation. In that case, we can prevent the throughput degradation by pipelining the computation—a significant improvement.

It is often necessary to adjust the amount of pipelining in an asynchronous computation to optimize its performance based on the timing behavior of components of the system.[42] Ideally this should be a transformation applied after the high-level design is completed, since we may not have the necessary timing information until the physical design of the system has been simulated. Such transformations, in general, involve examining the entire asynchronous system instead of just a single process.

We address the issues raised above by examining the following question: when can we change the slack of communication channels in the system without modifying behaviors of the system? This single transformation can be used to show the correctness (or lack thereof) of a number of different program transformation techniques. Changing the slack of a synchronization channel is a non-trivial operation. Consider the following example in which channels A, X, and Y are slack-zero channels.

$$X; A \;\|\; A; Y \;\|\; [\; \overline{X} \;\longrightarrow\; X; Y;\; \text{"}\mathit{good}\text{"} \;[\!]\; \overline{Y} \;\longrightarrow\; Y; X;\; \text{"}\mathit{bad}\text{"} \;]$$

The only possible computation is the sequence $X; A; Y;$"*good*". However, if we introduce slack on channel A, we now have the possibility $A; Y; X;$"*bad*".

When we are permitted to add slack to a channel in the system, we say that the particular channel is *slack elastic*. If every channel in the system is slack elastic, the system is said to be slack elastic.

2.15.1. Model

CHP communication channels that carry data can be described using this framework. A CHP channel C has two *ports* associated with it: a sender $C!$, and a receiver $C?$. $(C!, C?)$ form a pair of synchronization primitives. We define $\mathbf{s}C!$ to be the sequence of data values that have been sent on the sender port, and $\mathbf{s}C?$ the sequence of received values. Let $|s|$ be the length of sequence s. Then, $|\mathbf{s}C!| = \mathbf{c}C!$ and $|\mathbf{s}C?| = \mathbf{c}C?$.

We restrict our attention to systems that satisfy the four properties listed below; their need will become evident in the sections that follow.

- the system is closed, i.e., we have specified the CHP processes of interest and their environment;
- the system is deadlock-free;
- negated probes of the sender port of channels are not used in the computation;
- in an infinite execution, if a sender port is probed, the probe will be true infinitely often.

An execution *trace* is a particular interleaving of atomic actions that can occur during execution of the system. The system is completely characterized by the set of possible traces that can occur [9,39]. We only consider the *complete* traces of the system.[10]

Given a concurrent system, we are not interested in the possible interleavings of actions that occur in a trace. Rather, we are interested in the sequence of data values that are sent on certain channels of the system, given the sequence of values being sent on other channels. For instance, for the process that computes the sum of x and y we showed earlier, we might only be interested in the fact that the data values sent on channel C correspond to the sum of the values received on channels A and B. To this end, we define a *behavior* of a system in terms of the possible traces that can occur.

A behavior in our model is primarily characterized by the sequence of values that are sent and received on the channels of the system. Since processes in the system can only interact using communication channels, behaviors capture the data values that are exchanged by interacting processes. Therefore behaviors can be used to describe the input/output characteristics of processes in the system. In addition, we would like to specify a computation without specifying its synchronization behavior as far as possible. In our model, the only ordering between values that have been sent on various channels that can be inferred from the behavior itself is the ordering preserved by the FIFO nature of the individual channels.

Since the sequences of values sent and received on channels can be infinite, behaviors capture the notion of weakly fair execution. The notion of weak fairness in traces corresponds to enabled actions in a process being executed eventually; the notion of weak fairness in behaviors corresponds to the next value (if any) that can be sent/received on a channel being sent/received eventually.

The other component of a behavior is the sequence of non-deterministic choices made by processes in the system, since these choices can affect the data values being sent on channels. The only construct in CHP that introduces such choices is the selection statement.

The assignment of initial values to variables and the initialization of channels is assumed to be part of the CHP program for each process. Therefore, the actual initial values of variables do not affect the behavior, because every variable is assigned a value initially (or the value the variable has initially is not used).

Given the sequence of choices made by a process and the sequence of values that have been received by the process, we can completely determine the local state of a process. Therefore, our model does not include the local state of the process as part of a behavior.

Definition 2.1. (*decision point*)
Given a trace, a decision point for a process p is a point between two actions in the trace where p

has selected a guard of a selection statement for execution and several guards of the selection are true.

A decision point is characterized by a tuple $(n, sel, gset, alt)$*, where* n *is the occurrence index of the selection statement in the execution of* p*,* sel *denotes the selection statement,* $gset$ *is the set of guards of the selection statement that are true, and* alt *is the alternative chosen by* p*.*

Decision points of the system correspond to places where a non-deterministic choice is made. We assume we have no control over the mechanism used to implement this choice; therefore, the choice made by the computation is assumed to be unfair.

Definition 2.2. (*behavior*)
Given a trace, the corresponding behavior $\mathcal{B}$ *of a system is a function that maps channels* c *in the system to pairs of sequences of values* $(\mathbf{s}c?, \mathbf{s}c!)$ *that occurred in the trace, and processes to their set of decision points in the trace.*

Given a channel c and process p, we denote $(\mathbf{s}c?, \mathbf{s}c!)$ by $\mathcal{B}.c$, and the set of decision points corresponding to p by $\mathcal{B}.p$. The behavior corresponding to a trace is unique. However, multiple traces can map onto the same behavior, since different interleavings of actions that do not interact will be reduced to the same behavior if they do not affect the sequence of values sent on the channels in the system.

Definition 2.3. (*system*)
A system is a closed, deadlock-free collection of CHP processes and is defined by the set of behaviors that can occur during execution.

Note that a system will be the empty set just when it does not contain any processes.

Example. The system

```
   *[ $X!0$ ] || *[ $Y!1$ ] || *[ $Z?w$ ]
|| *[[$\overline{X} \longrightarrow X?x;\ Z!x;$ [$\overline{Y} \longrightarrow Y?x; Z!x$ | $\neg\overline{Y} \longrightarrow$ skip]
    |$\overline{Y} \longrightarrow Y?y;\ Z!y;$ [$\overline{X} \longrightarrow X?y; Z!y$ | $\neg\overline{X} \longrightarrow$ skip]
   ]]
```

has an execution that corresponds to the sequence $X!0, X?x;\ Z!0, Z?w;\ Y!1, Y?x;\ Z!1, Z?w \ldots$ where the first guard $\overline{X} \rightarrow \ldots$ is chosen for execution with $\overline{Y}$ being true in the outer selection statement, and $\overline{Y} \rightarrow \ldots$ is chosen in the inner selection statement. The behavior corresponding to this trace maps Y to the pair of infinite sequences $([1, 1, \ldots], [1, 1, \ldots])$, X to $([0, 0, \ldots], [0, 0, \ldots])$, Z to $([0, 1, 0, 1, \ldots], [0, 1, 0, 1, \ldots])$, and the process with the selection statements to the set $\{(0, selout, \{\overline{X}, \overline{Y}\}, \overline{X}),\ (1, selout, \{\overline{X}, \overline{Y}\}, \overline{X}), \ldots\}$, where $selout$ is the outer selection statement that selects between $\overline{X}$ and $\overline{Y}$, and the labels $\overline{X}$ and $\overline{Y}$ refer to the alternatives in the selection statement. ∎

2.15.2. Specifications and Observability

The specification of a closed CHP program is a set of behaviors. Usually a specification does not completely specify the sequence of values sent and received on all channels of the system. Accordingly, we classify the channels of the system into *internal* and *external* channels, depending on whether or not the data values sent on those channels are part of the specification. All properties of interest must be specified only using the quantities $\mathbf{s}E!$ and $\mathbf{s}E?$, where E is an external channel.

Example. It is possible that we may not be able to observe certain properties of a computation, since behaviors do not contain as much information as the sequence of actions in the computation. For example, consider the following two processes:

```
   *[ $NCS_1$;  $CS_1$ ]
|| *[ $NCS_2$;  $CS_2$ ]
```

We cannot directly observe the property that two processes access their critical sections CS_i in an exclusive manner since we can only observe the sequence of values on channels. However, we can make the mutual exclusion property visible by the introduction of a third process and an external channel C as follows:

```
   *[ $NCS_1$;  $A!1; CS_1; A!1$ ]
|| *[ $NCS_2$;  $B!2; CS_2; B!2$ ]
|| *[[$\overline{A} \longrightarrow A?x$ ▯ $\overline{B} \longrightarrow B?x$];  $C!x$ ]
```

By observing the sequence of values on channel C, we can determine if mutual exclusion is maintained. For instance, if sequence $1, 2, 1, 2, \ldots$ is possible, we have violated the mutual exclusion requirement. ∎

Definition 2.4. (*smaller set of decision points*)
Given two sets of decision points D_1 and D_2 for a process p, we say that $D_1 \sqsubseteq D_2$ iff for every decision point $(n, sel, gset_1, alt) \in D_1$, there exists $(n, sel, gset_2, alt) \in D_2$ such that $gset_1 \subseteq gset_2$.

The relation "$\sqsubseteq$" on sets of decision points orders them in terms of the number of nondeterministic choices that were possible.

Definition 2.5. (*implementation*)
A system implements a specification if, for each behavior $\mathcal{B}_{sys}$ of the system, there exists a behavior $\mathcal{B}_{spec}$ in the specification such that the sequence of values on all external channels in $\mathcal{B}_{spec}$ is the same as in $\mathcal{B}_{sys}$, and $(\forall p :: \mathcal{B}_{sys}.p \sqsubseteq \mathcal{B}_{spec}.p)$.

This implementation relation is different from the traditional implementation relations used in trace theory and other models of concurrent programming because it does not directly include the synchronization behavior of the computation.

Example. Consider the following two systems:

$S_0 \equiv$ `*[` $X!0$ `]` $\|$ `*[` $Y!0$ `]` $\|$ `*[` $X?x$ `]` $\|$ `*[` $Y?y$ `]`

$S_1 \equiv$ `*[` $X!0;\ \ Y!0$ `]` $\|$ **skip** $\|$ `*[` $X?x$ `]` $\|$ `*[` $Y?y$ `]`

The computations specified by S_0 and S_1 are indistinguishable under our model because the sequences of values sent and received on channels X and Y are identical in both systems, and both systems have no decision points. Standard concurrency models will differentiate them because the communication on X and Y cannot be executed in parallel, and because of the additional bound $0 \le \mathbf{c}X - \mathbf{c}Y \le 1$ in system S_1. ▮

We now present the theorems that enable a large number of transformations, including the introduction and elimination of pipelining, data-flow style process decomposition, and pipelined completion detection.

2.15.3. Main Theorems

Throughout this section, we use $\mathcal{S}$ to denote the set of possible behaviors of the system of interest, p to denote a process in the system, and c to denote a channel in the system.

Lemma 2.6. (*monotonicity*)
Let $\mathcal{S}^+$ be the system obtained from $\mathcal{S}$ by increasing the slack on a particular channel. Then $\mathcal{S} \subseteq \mathcal{S}^+$.

Proof: Consider any behavior of $\mathcal{S}$. This behavior corresponds to some execution of system $\mathcal{S}$. It suffices to show that this execution is possible in $\mathcal{S}^+$. Let c be the channel whose slack was increased from $\mathbf{k}c!$ to $\mathbf{k}c! + n$. By definition, computations from $\mathcal{S}$ satisfy $\mathbf{c}c! - \mathbf{c}c? \le \mathbf{k}c!$. These computations still exist in $\mathcal{S}^+$ because we can postpone any attempted send action on c so that this condition is satisfied, since $\mathcal{S}$ is deadlock-free. We now show that the communication actions that were attempted in $\mathcal{S}$ can also occur in $\mathcal{S}^+$.

The only construct in CHP which can affect control flow behavior is the selection statement. Increasing slack does not change the probe of the receiver end of the channel (by definition). The probe of a sender is monotonic with slack (by definition). Since we disallow negated probes of sender ports, this implies that all guards of selection statements are monotonic with slack. Also, a true probe on a sender port can be postponed (since probes only become true eventually) in $\mathcal{S}^+$ until the point when it becomes true in $\mathcal{S}$. Therefore, the guards true in $\mathcal{S}$ will eventually become true in $\mathcal{S}^+$, and so any behavior from $\mathcal{S}$ could occur in $\mathcal{S}^+$. □

Lemma 2.6 shows that the set of behaviors is monotonic with the slack on the channels. We now show that the only way in which increasing the slack on a channel can affect the computation is by increasing non-determinism. Note that all restrictions on computations that were mentioned in the previous section are needed for this proof.

Theorem 2.7. (*decreasing slack*)
Decreasing the slack of a channel does not affect the correctness of computations if and only if it does not introduce deadlock.

Proof: Let $\mathcal{S}^-$ be the system obtained from $\mathcal{S}$ by decreasing the slack of a channel. If $\mathcal{S}^-$ is deadlock-free, $\mathcal{S}^- \subseteq \mathcal{S}$ by lemma 2.6. By definition 2.5, $\mathcal{S}^-$ implements $\mathcal{S}$. □

Definition 2.8. (*extension*)
A behavior $\mathcal{B}'$ is said to be the extension of behavior $\mathcal{B}$ iff:

$$(\forall c :: \mathcal{B}.c = \mathcal{B}'.c) \wedge$$
$$(\exists p_0 :: (\forall p : p \neq p_0 : \mathcal{B}.p = \mathcal{B}'.p) \wedge \mathcal{B}.p_0 \neq \mathcal{B}'.p_0 \wedge \mathcal{B}.p_0 \sqsubseteq \mathcal{B}'.p_0)$$

Intuitively, the extension of a behavior corresponds to the same data behavior but with at least one additional choice which did not exist in the original behavior.

Theorem 2.9. (*increasing slack*)
Let $\mathcal{S}^+$ be the system obtained from $\mathcal{S}$ by increasing the slack of a channel. Then either $\mathcal{S} = \mathcal{S}^+$, or there exists a behavior $\mathcal{B}^+ \in (\mathcal{S}^+ - \mathcal{S})$ that is the extension of a behavior in $\mathcal{S}$.

Proof: By lemma 2.6, $\mathcal{S} \subseteq \mathcal{S}^+$. Therefore either $\mathcal{S} = \mathcal{S}^+$, or there exists $\mathcal{B}_0 \in \mathcal{S}^+ - \mathcal{S}$. Assume such a $\mathcal{B}_0$ exists. Now $\mathcal{B}_0$ differs from every behavior in $\mathcal{S}$ in either the sequence of values sent on some channel or in the set of decision points for some process in $\mathcal{S}$. This implies that the local state of some process from $\mathcal{S}^+$ differs from the local state that could occur in $\mathcal{S}$. Consider the first point in execution when this occurs. The only non-deterministic construct in CHP is the selection statement, and therefore the only way a new local state could occur is because of a new true guard in a selection statement. By the same argument as in lemma 2.6, the guards true in $\mathcal{S}$ will eventually become true in $\mathcal{S}^+$. Therefore, we can pick an alternative of the selection statement that is possible in $\mathcal{S}$, and continue execution as in the original system $\mathcal{S}$. This new behavior is the required extension. □

The strength of Theorem 2.9 lies in the fact that if we can show that we cannot possibly introduce new decision points, this implies that adding slack does not change the behavior of a computation.

2.15.4. Local Slack Elasticity

So far, we have only considered the slack elasticity of closed systems. Such an approach is feasible when systems can be easily identified as being slack elastic. In this section we extend the notion of slack elasticity to open systems, permitting the design of slack elastic systems in a modular fashion.

Definition 2.10. (*local slack elasticity*)
Let $\mathcal{S}$ be an open system, and let C be a set of channels that are in $\mathcal{S}$. Consider any closed, deadlock-free system $(\mathcal{S} \parallel \mathcal{S}')$. By theorem 2.9, adding slack to any channel not in C either preserves the set of behaviors, or extends some behavior of the original system. If, for all such extensions, the process with additional decision points is always from $\mathcal{S}'$, then $\mathcal{S}_C$ is said to be locally slack elastic.

When $\mathcal{S}$ is closed, i.e., when there are no “dangling” channels (channels which are not connected to two processes), then local slack elasticity of $\mathcal{S}_\emptyset$ is equivalent to slack elasticity of $\mathcal{S}$.

Theorem 2.11. (*composition*)
Let $\mathcal{S}^1_{C_1}, \ldots, \mathcal{S}^n_{C_n}$ be locally slack elastic. Let $\mathcal{S} = \mathcal{S}^1 \parallel \cdots \parallel \mathcal{S}^n$, and $C = C_1 \cup \cdots \cup C_n$. Then, $\mathcal{S}_C$ is locally slack elastic.

Proof: Follows from the definition of local slack elasticity. □

Theorem 2.11 shows that the parallel composition of locally slack elastic systems is locally slack elastic. Given a collection of locally slack elastic processes, we can compose them to form a closed system that remains locally slack elastic. Therefore, we can construct large systems that are slack elastic (except for the specified set of channels) in a modular fashion.

When a collection of processes is entirely deterministic, we can introduce slack on any channel without affecting correctness.

Let $\mathcal{S}$ be an open system in which the guards in selection statements are syntactically mutually exclusive and there are no probed channels. $\mathcal{S}_\emptyset$ is locally slack elastic. Consider a deadlock-free system $\mathcal{S} \parallel \mathcal{S}'$. Increasing the slack of a channel cannot introduce more true guards for a process in $\mathcal{S}$, because $\mathcal{S}$ has no probed channels and all guards of selection statements are syntactically mutually exclusive. Therefore, $\mathcal{S}_\emptyset$ is locally slack elastic.

References

The guarded command notation for describing sequential programs was introduced by Edsgar W. Dijkstra.[6] C.A.R. Hoare extended the notation to include concurrent programs by the introduction of communication actions.[8] The notation was further extended by Alain J. Martin with the introduction of the probe as a synchronization construct,[27] and with the bullet operator for describing simultaneous composition.[28] The presentation of synchronization primitives is taken from a paper by Alain J. Martin.[24] The section on energy estimation is taken from a paper on energy estimation by Tierno et al.[41] The notion of slack elasticity was introduced by Rajit Manohar[23] and Alain J. Martin to simplify correctness arguments for highly pipelined asynchronous computations.

Chapter 3.

Circuit Synthesis

The digital abstraction used to describe a CMOS circuit treats the transistor as a switch. The low voltage level (ground, often abbreviated *GND*) is treated as Boolean **false** (or logic 0), and the high voltage level (supply voltage, often abbreviated *Vdd*) is treated as Boolean **true** (or logic 1). However, voltages are continuous quantities and they can take on intermediate values.

Figure 3.1 shows how simple clocked circuits enforce the digital abstraction by using the clock signal to discretize the time domain. When the circuit is exhibiting non-ideal behavior, i.e., when signals are changing, the clock is used to ignore the transient state of the system. The light grey regions in Figure 3.1 show the times at which the circuit can exhibit transient behavior. Once that period is over, the voltages of all signals must be in the dark grey bands representing logic levels 0 and 1. The width of each band corresponds to the *noise margin* of the circuit.

An asynchronous circuit cannot ignore the intermediate states that occur when signals are changing. An important observation is that we do not need transistors to behave as discrete devices; voltages can be viewed as continuous quantities, as long as changes from a stable value in

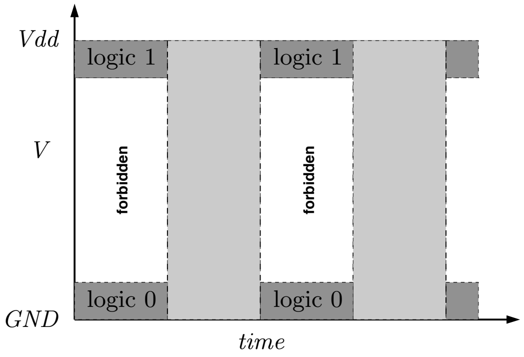


Figure 3.1. Clocked system showing discretization in time.

one logic region to another are *monotonic*. This monotonicity assumption is implemented during the circuit design process.

A change in voltage of the input to a gate can be detected when the voltage passes the switching threshold of the gate. Therefore, switching thresholds of operators determine the noise margin of their inputs.

Strict monotonicity is not necessary—which is fortunate because it is impossible to implement in the presence of noise. All that is required is that fluctuations in the input near the switching threshold of an operator are transient. The property of *stability* guarantees that the network implementing the input will eventually change the value of the input node to Vdd or GND.

3.1. Production Rules

Our digital abstraction for CMOS circuits is the *production rule set*, which consists of a set of *production rules*. A production rule is a construct of the form $G \mapsto S$, where S is either a simple assignment statement or a list of simple assignment statements, and G is a Boolean expression called the *guard* of the production rule. A simple assignment is one that assigns a constant value (**true** or **false**) to a Boolean-valued variable.

Example. The following are examples of production rules:

$$\begin{aligned} x \wedge y \ &\mapsto \ z\uparrow \\ \neg x \ &\mapsto \ u\downarrow, v\uparrow \end{aligned}$$

Note that the only assignment statements on the right hand side of a production rule arrow are of the form v := **true** or v := **false** (where v is a variable), and are abbreviated $v\uparrow$ and $v\downarrow$ respectively. ▮

A production rule $G \mapsto s_1, s_2, \ldots, s_n$ where all the s_i's are simple assignment statements is an abbreviation for the production rule set $\{G \mapsto s_1, G \mapsto s_2, \ldots, G \mapsto s_n\}$. Therefore, we only consider production rules that have a simple assignment statement on the right hand side of the arrow.

Example. The production rule $\neg x \mapsto u\downarrow, v\uparrow$ shown above is equivalent to the production rule set

$$\neg x \mapsto u\downarrow \qquad\qquad \neg x \mapsto v\uparrow$$

▮

The *execution* of a production rule $G \mapsto t$ where t is a simple assignment statement (sometimes called a *transition*) is defined as an unbounded sequence of *firings*. A firing of $G \mapsto t$ with $G \equiv$ **true** amounts to the execution of t; A firing of $G \mapsto t$ with $G \equiv$ **false** amounts to **skip**. The execution of a production rule set is the weakly fair concurrent composition of the execution of the individual production rules in the set.

We use the predicate R on transitions to denote the result of a transition: $R(v\uparrow)\equiv v$ and $R(v\downarrow)\equiv\neg v$. A firing of a production rule $G \mapsto t$ in a state where $G \wedge \neg R(t)$ holds is called an *effective firing*. An effective firing changes the state of the computation. A firing of $G \mapsto t$ in a state where $G \wedge R(t)$ holds is called a *vacuous firing*.

This definition of production rules and their execution was chosen so as to make them easily implementable in CMOS. Consider a production rule of the form $G \mapsto v\uparrow$. If we implement a switching network that is conducting just when G is **true**, and connect v to one end and Vdd (which represents **true**) to the other end, we will have implemented the production rule $G \mapsto v\uparrow$ correctly. Whenever the switching network is not conducting, i.e., $G \equiv$ **false**, v will not be changed by the action of $G \mapsto v\uparrow$. If the switching network is conducting, i.e., $G \equiv$ **true**, then v will be connected to Vdd (logic value **true**), and will therefore eventually be set to **true**.

3.1.1. Stability and Non-interference

Two production rules are said to be *complementary* when they set the same variable to two different values. Complementary production rules are of the form $G^+ \mapsto v\uparrow$ and $G^- \mapsto v\downarrow$. Two complementary production rules $G^+ \mapsto v\uparrow$ and $G^- \mapsto v\downarrow$ are said to be *non-interfering* when $\neg G^+ \vee \neg G^-$ is invariant. A production rule set is said to be non-interfering when all pairs of complementary production rules in the set are non-interfering.

Example. The production rules

$$\begin{array}{rcl} x & \mapsto & u\uparrow \\ y & \mapsto & u\downarrow \end{array}$$

are interfering if $x \wedge y$ holds at any point in the computation. ∎

A production rule $G \mapsto t$ is *stable* in a computation just when G can change from **true** to **false** only in states where $R(t)$ holds. If the computation can change state from $G \wedge \neg R(t)$ to $\neg G \wedge \neg R(t)$, the production rule $G \mapsto t$ is *unstable*. A production rule set is said to be stable when all production rules in it are stable.

Example. Consider the production rule set

$$\begin{array}{rclcrcl} a & \mapsto & b\downarrow & \quad & c & \mapsto & a\downarrow \\ \neg a & \mapsto & b\uparrow & & \neg c & \mapsto & a\uparrow \\ \\ b & \mapsto & c\downarrow & & a & \mapsto & x\uparrow \\ \neg b & \mapsto & c\uparrow & & \neg a & \mapsto & x\downarrow \end{array}$$

Assume the initial state is $\neg a \wedge \neg x \wedge b \wedge \neg c$. The production rule $a \mapsto x\uparrow$ is unstable, because the system can change from state $a \wedge \neg b \wedge c \wedge \neg x$ to $\neg a \wedge \neg b \wedge c \wedge \neg x$, making the guard of $a \mapsto x\uparrow$

change from **true** to **false** in a state where x is **false**. ∎

The only valid production rule sets are those that are stable and non-interfering. The examples demonstrate that stability and non-interference do not hold for arbitrary production rule sets. The synthesis method described in this chapter will guarantee these two properties.

It can be shown that, under the stability and non-interference of a production rule set, the execution of the production rules of a set is equivalent to the following sequential execution:

```
*[ select a production rule with a true guard;
   fire the production rule
 ]
```

where the selection operation is weakly fair. This equivalence simplifies the analysis of production rule sets. Circuits generated from stable, non-interfering production rule sets are known as *quasi delay-insensitive* (QDI) circuits. Chapter 9 describes the difference between these circuits are purely delay-insensitive circuits in detail.

3.2. CMOS Implementation of Production Rules

A CMOS circuit is a network of electrical *nodes*, interconnected by transistors and wires. The circuit contains the special node *Vdd* that provides the constant high-voltage value used to represent **true**/logic 1, and the special node *GND* that provides the constant low-voltage value used to represent **false**/logic 0. A node in the circuit can take on voltage values between the high voltage and low voltage.

Our design methodology will guarantee that the production rules generated are stable and non-interfering. These two properties simplify the CMOS implementation of production rules. Non-interference will guarantee that the resulting CMOS circuit has no stable states where *Vdd* and *GND* are connected (shorted), and stability will ensure that the circuit has no switching *hazards*. The result is that asynchronous QDI circuits are robust to variations in the underlying fabrication technology, and can be mapped to different fabrication processes with little or no modification.

3.2.1. Digital Abstraction

Boolean variables in our computation are implemented as nodes in the CMOS circuit. The value of the variable is represented by the voltage of the corresponding node. Voltages that are greater than a certain value v_1 are interpreted as logic 1 (**true**), and voltages below a certain value v_0 are interpreted as logic 0 (**false**). The value of v_0 and v_1 vary from node to node.

The following is a highly simplified presentation of CMOS transistors. For a comprehensive treatment, the reader is referred to other standard texts.[33,40]

A CMOS transistor is either of n-type or of p-type. An n-type transistor can be considered to be a three-terminal device (with the fourth terminal, the "bulk," being held at GND), having terminals g, the "gate", and two other terminals x and y. When the voltage of the gate $v(g)$ exceeds $\min(v(x), v(y)) + v_{tn}$, the region between x and y becomes conducting and a current flows between x and y until $v(x) = v(y)$, or $v(g) \leq \min(v(x), v(y)) + v_{tn}$. The value v_{tn} is called the *threshold voltage* of the n-type transistor. For a p-type transistor, when the voltage $v(g)$ is greater than $\max(v(x), v(y)) - v_{tp}$, the region between x and y becomes conducting. The value v_{tp} is the threshold voltage of the p-type transistor. The values of v_{tn} and v_{tp} are technology dependent.

Hence, a transistor approximates a switch that either opens or closes a connection between two electrical nodes. The state of the switch is controlled by the voltage of the gate of the transistor. If a node is connected to Vdd or GND through a network of transistors, we say that the node is *driven*; otherwise the node is said to be *floating*. A node in a CMOS circuit is said to be *dynamic* if the circuit can enter a state where the node is floating.

Given an n-type transistor with gate g and two other terminals x and y, the effect of $v(g)$ being set to Vdd can be easily determined if x is driven and y is floating (or vice-versa). If $v(x) = GND$, then setting $v(g)$ to Vdd sets $v(y)$ to GND. If $v(x) = Vdd$, the result is to set $v(y)$ to $Vdd - v_{tn}$. For p-type transistors, setting $v(g)$ to GND sets $v(y)$ to Vdd if $v(x) = Vdd$, and $v(y)$ to v_{tp} if $v(x) = GND$. Observe that even with our simple model, an n-transistor can only raise the voltage of a node to $Vdd - v_{tn}$, and a p-transistor can only lower the voltage of a node to v_{tp}. To prevent such voltage drops, we will use n-transistors exclusively to lower the voltage of a node—in *pull-down* networks, and p-transistors exclusively to raise the voltage of a node—in *pull-up* networks.

3.2.2. Switching Networks

Consider a stable production rule

$$B \mapsto z\downarrow$$

The variables of B are used as control voltages for the gates of a switching network S of transistors that implements the condition B. Since we have to set z to **false** when B is **true**, the two ends of the switching network S are connected to GND and z respectively. To avoid any voltage drops, we implement the switching network S exclusively with n-type transistors. We can implement B with a standard series/parallel combination of transistors. Similarly, a production rule of the form

$$B \mapsto z\uparrow$$

is implemented with a switching network that connects z to Vdd, where the switching network comprises only p-type transistors.

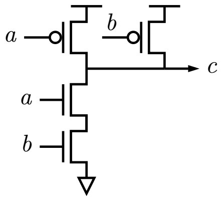


Figure 3.2. CMOS implementation of a NAND gate.

Example. Figure 3.2 shows the CMOS implementation of a NAND gate. The NAND gate is a valid implementation of the following production rule set:

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

The switching network for the pull-up implements the Boolean expression $\neg a \vee \neg b$ using p-transistors, and the switching network for the pull-down implements expression $a \wedge b$ using n-transistors. ▮

Consider the production rule

$$a \mapsto c\uparrow$$

We cannot directly implement a switching network with p-type transistors that only uses variable a as a gate voltage. The problem is that we need the *negated version of* a to build the switching network. If $\_a$ were the negated version of a, we could implement the production rule $\neg\_a \mapsto c\uparrow$ without difficulty.

To check whether the guard of a production rule is CMOS-implementable, we write all guards in *negation-normal form.* A guard is said to be in negation-normal form just when negation symbols in the guard only appear on variables. We can convert a guard into negation-normal form using DeMorgan's laws.

Example. The Boolean expression $\neg(a \vee b)$ is not in negation-normal form since it contains a $\neg$ symbol attached to an expression. By applying DeMorgan's laws, we can rewrite it as $\neg a \wedge \neg b$, which is in negation-normal form since $\neg$ symbols only appear on variables. ▮

A production rule $B \mapsto z\uparrow$ with B in negation-normal form is CMOS-implementable just when every variable in B is negated. A production rule $B \mapsto z\downarrow$ with B in negation-normal form is CMOS-implementable just when every variable in B is non-negated.

Example. The production rule $x \wedge y \mapsto z\downarrow$ is CMOS-implementable because variables x and y are non-negated. However, $\neg x \wedge y \vee x \wedge u \mapsto z\downarrow$ is not CMOS-implementable because there is an instance of x that is negated. ▮

3.2.3. Operators

Complementary production rules are implemented as one *operator* (also known as a gate). Consider the set of complementary production rules

$$b_1 \mapsto z\uparrow$$
$$b_2 \mapsto z\downarrow$$

where both b_1 and b_2 are CMOS-implementable by switching networks s_1 and s_2 respectively. The two switching networks are connected by node z. Since the production rules are non-interfering, $\neg b_1 \vee \neg b_2$ is invariant, implying that there is no (digital) state in the computation where Vdd and GND are connected.

An operator is said to be *combinational* whenever $b_1 \vee b_2 \equiv$ **true**. An operator that is not combinational is said to be *state-holding*.

If an operator is combinational, either b_1 or b_2 holds in any state of the computation. Therefore, either switching network s_1 or switching network s_2 is always conducting. As a result, the node z is always driven. The CMOS implementation of a combinational operator is shown in Figure 3.3. For an operator to be combinational, $b_1 \equiv \neg b_2$, and so the set variables used in the pull-up and pull-down are identical. As a result, whenever $b_1 \mapsto z\uparrow$ ($b_2 \mapsto z\downarrow$) has an effective firing, it is because a variable occurring in b_1 (b_2) changed and made $b_1 \equiv$ **true** ($b_2 \equiv$ **true**) and $b_2 \equiv$ **false** ($b_1 \equiv$ **false**). Since the voltage of the changing variable changes continuously from one logic value to another, there is an intermediate, transient state where both switching networks are partially conducting. This results in short-circuit currents that dissipate power. If the output z is changing to **true** (**false**), there is some opposition to this change by the pull-down (pull-up) network for z because the network is partially conducting. By picking appropriate transistor sizes, we can determine the voltage a changing input must cross before it can affect the output z (see any standard textbook on CMOS design for details). When the p-transistors and n-transistors are sized to have comparable drive strengths, this voltage tends to be close to $Vdd/2$.

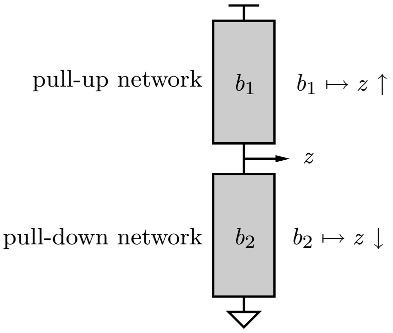


Figure 3.3. CMOS implementation of a combinational operator.

The values v_0 and v_1 for the input to an operator are determined by the transistor sizes used to implement the switching network for the operator.

Consider the case when the operator

$$\begin{aligned} b_1 &\mapsto z\uparrow \\ b_2 &\mapsto z\downarrow \end{aligned}$$

is state-holding. If we use the circuit shown in Figure 3.3 as the CMOS implementation of the operator, node z is not driven in the state where $\neg b_1 \wedge \neg b_2$ holds. This is problematic if the state $\neg b_1 \wedge \neg b_2$ is persistent. If this state persists, the charge stored on z can change due to leakage currents thereby changing the value of z without b_1 or b_2 being **true** causing an erroneous execution. The node is also susceptible to noise when it is floating.

There are several ways we can solve this problem. One method is known as the technique of *symmetrization* where we attempt to convert a state-holding operator into a combinational one. This technique is discussed later in this chapter when we talk about compilation. This technique suffers from two drawbacks: it is not always applicable, and it can result in circuits that are much more complicated than the original state-holding operator.

The standard way to make sure that node z is always driven is to use the circuit implementation shown in Figure 3.4. The additional circuitry consists of two cross-coupled inverters attached to node z. The transistors drawn in Figure 3.4 implement a feedback inverter that ensures that node z is always driven. Given the circuitry in Figure 3.4, assume the circuit is in a state in which $z \wedge \neg b_1 \wedge \neg b_2$ holds. The node z is at Vdd, and both switching networks s_1 and s_2 are not conducting. The node $\_z$ (shown in Figure 3.4) is at GND, and the feedback inverter is driving z to Vdd. If the system enters a state where $z \wedge \neg b_1 \wedge b_2$ holds, the pull-down network for z becomes conducting. At this point, the feedback p-transistor is still driving z high. Therefore, we have a state where z is being driven to Vdd and to GND. The final value of z is determined by the electrical properties of the circuit. Since the correct final value of z is **false**, we can ensure that

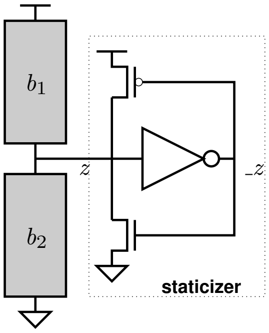


Figure 3.4. Standard implementation of a state-holding operator.

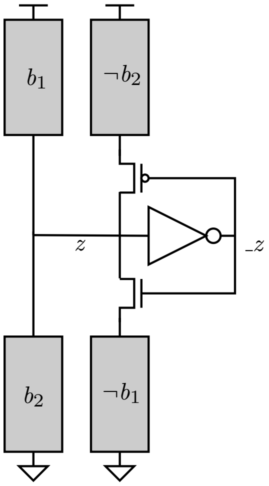


Figure 3.5. Implementation of state-holding operators using combinational feedback.

the network s_2 overcomes the opposing feedback inverter by making the feedback transistors weak compared to the switching networks s_1 and s_2. Given this circuit structure, there is always some opposition to any change in z by the feedback transistors. This opposition helps combat noise, because any perturbation in z will be opposed by the feedback inverter.

The actual change in input voltage necessary for z to change its value depends on the relative strengths of the switching network the input is connected to and the opposing feedback transistor on z. However, since we have to make the feedback transistor weak to ensure the circuit functions correctly, the change in input that is required to change z tends to be small compared to the change required for combinational operators. Therefore, the inputs to state-holding operators implemented as shown in Figure 3.4 are more susceptible to noise. We can change the strength of the feedback transistors to improve the noise margin of the input, but we cannot make the feedback transistors have arbitrary strength.

To avoid this problem, we can adopt the circuit implementation shown in Figure 3.5. In this circuit structure, the opposing feedback transistors are connected in series with a switching network that prevents z from being driven to *Vdd* and *GND* simultaneously. When both b_2 and b_1 are **false**, the switching networks connected in series with the feedback transistors are both conducting, thereby causing z to be driven. Whenever b_1 or b_2 become **true**, the opposing feedback path is cut-off. With this configuration, we can *pick* the voltage an input must cross before it can change z, thereby permitting us to set the noise margin for the inputs to arbitrary values. This solution is known as *combinational feedback*.

The drawback of combinational feedback is that the resulting circuit is much more complex

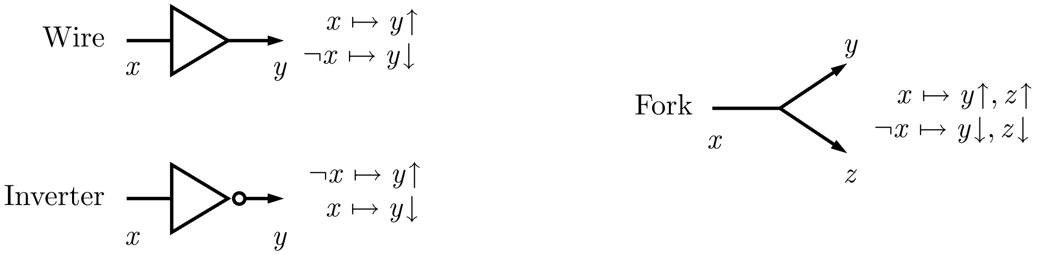

Wire
x
y
$x \mapsto y\!\uparrow$
$\neg x \mapsto y\!\downarrow$
Fork
x
y
z
$x \mapsto y\!\uparrow, z\!\uparrow$
$\neg x \mapsto y\!\downarrow, z\!\downarrow$
Inverter
x
y
$\neg x \mapsto y\!\uparrow$
$x \mapsto y\!\downarrow$

Figure 3.6. One-input operators.

than the simple staticizer shown in Figure 3.4. Therefore, we rarely use this solution. Our preferred implementation strategy for state-holding operators will be the one shown in Figure 3.4. We use combinational feedback only when we need to set the noise margin of the input to a state-holding operator. This level of paranoia is typically necessary when the inputs to the operator are being generated off-chip.

3.2.4. Standard Operators

We list all the standard state-holding and combinational operators in this section, together with their circuit symbols. These operators are frequently encountered in circuit synthesis.

Figure 3.6 shows all the one-input operators together with their circuit symbols. The explicit *wire* between x and y is written *wire*(x, y). The introduction of an explicit production rule for the wire implies that we can add arbitrary delay between a change in x and the corresponding change in y without affecting the correctness of the computation. The other one-input operators are the inverter and the fork, shown in Figure 3.6.

Figure 3.7 shows all two-input state-holding operators along with their circuit symbols. The C-element (consensus element) is one of the most common state-holding circuit elements. The operator waits for both its inputs to be equal, and then sets its output to be equal to its input.

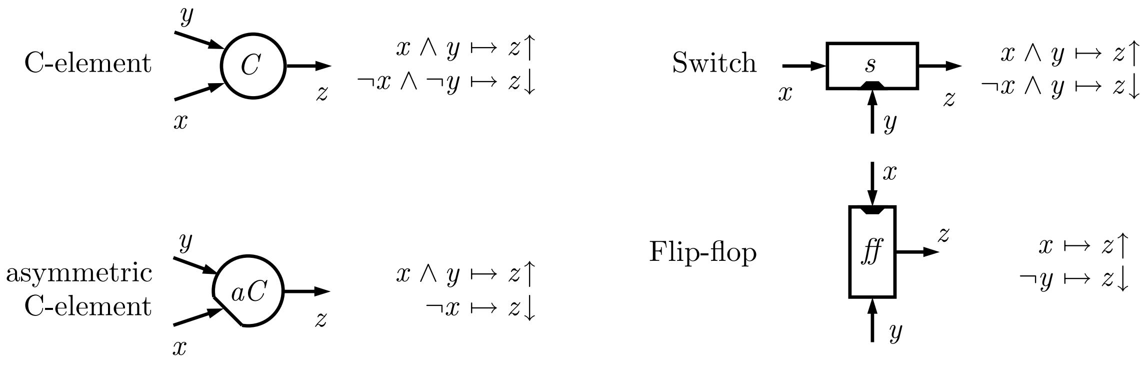

C-element
y
C
x
z
$x \wedge y \mapsto z\!\uparrow$
$\neg x \wedge \neg y \mapsto z\!\downarrow$
Switch
s
x
z
y
$x \wedge y \mapsto z\!\uparrow$
$\neg x \wedge y \mapsto z\!\downarrow$
asymmetric C-element
y
aC
x
z
$x \wedge y \mapsto z\!\uparrow$
$\neg x \mapsto z\!\downarrow$
Flip-flop
x
ff
z
y
$x \mapsto z\!\uparrow$
$\neg y \mapsto z\!\downarrow$

Figure 3.7. Two-input state-holding operators.

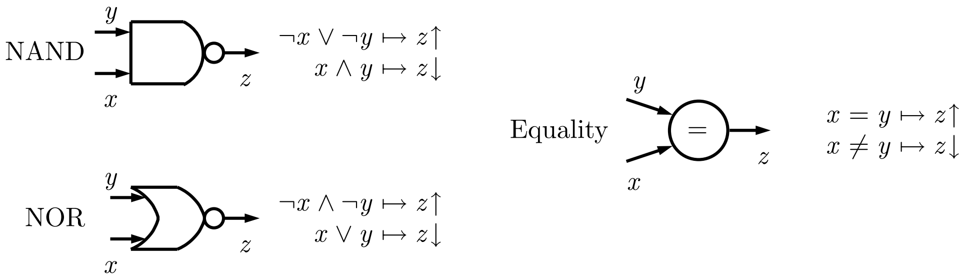

NAND
y
x
z
$\neg x \vee \neg y \mapsto z\uparrow$
$x \wedge y \mapsto z\downarrow$
NOR
y
x
z
$\neg x \wedge \neg y \mapsto z\uparrow$
$x \vee y \mapsto z\downarrow$
Equality
y
x
=
z
$x = y \mapsto z\uparrow$
$x \neq y \mapsto z\downarrow$

Figure 3.8. Two-input combinational operators.

The asymmetric C-element is similar to a C-element, except one of the inputs only occurs in the production rule for $z\uparrow$. Note that these two operators are not directly CMOS-implementable. However, if we replace $z\uparrow$ with $z\downarrow$ and $z\downarrow$ with $z\uparrow$ in the two production rules, they are CMOS-implementable. This modification results in the *inverting* C-element and *inverting* asymmetric C-element respectively; their circuit symbols have a "bubble" on the output. The other two state-holding operators that are possible are the switch and the flip-flop, shown in Figure 3.7. The C-element, the asymmetric C-element, and the switch are guaranteed to be non-interfering. If a circuit uses the flip-flop, non-interference implies that $\neg x \vee y$ is invariant.

There are three combinational two-input operators: NAND, NOR, and equality. The three operators are shown in Figure 3.8. The inverting equality operator is exclusive-or. Note that while the NAND and NOR operators are CMOS-implementable, the equality operator is not directly implementable in CMOS.

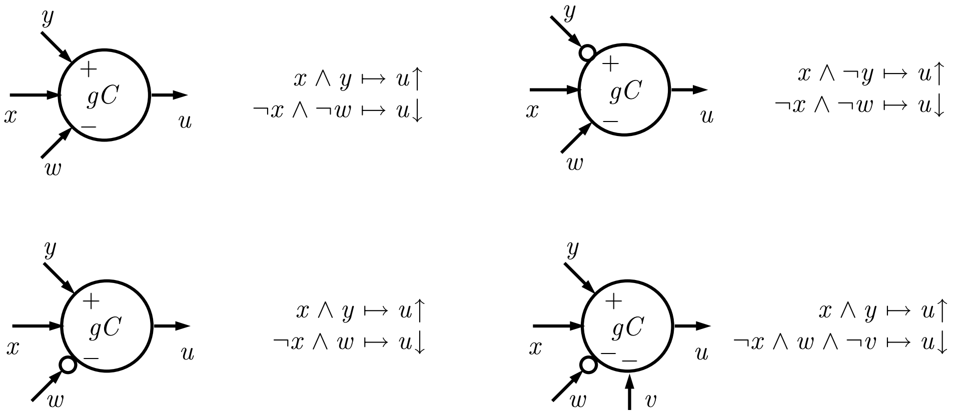
The NAND gate, NOR gate, and C-element can be generalized to N-input operators in the obvious way. The circuit symbol is the same, except there are more than two inputs drawn. The production rules for the N-input NAND, NOR, and C-element are given below:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
(\vee i :: \neg x_i) & \mapsto & z\uparrow & (\wedge i :: x_i) & \mapsto & z\uparrow \\
(\wedge i :: x_i) & \mapsto & z\downarrow & (\wedge i :: \neg x_i) & \mapsto & z\downarrow
\end{array}$$

$$\begin{array}{rcl}
(\wedge i :: \neg x_i) & \mapsto & z\uparrow \\
(\vee i :: x_i) & \mapsto & z\downarrow
\end{array}$$

The *generalized C-element* is a special multi-input operator. The pull-up for the generalized C-element is the conjunction of N terms, where each term is either a variable or a negated variable. The pull-down is the conjunction of M terms, each term being a variable or a negated variable. The C-element is a special case of a generalized C-element with the pull-up consisting of the conjunction of non-negated variables, and the pull-down consisting of the conjunction of the negated version of the variables occurring in the pull-up.

Example. The operator



$$x \wedge y \mapsto u{\uparrow} \qquad \neg x \wedge \neg w \mapsto u{\downarrow}$$

$$x \wedge \neg y \mapsto u{\uparrow} \qquad \neg x \wedge \neg w \mapsto u{\downarrow}$$

$$x \wedge y \mapsto u{\uparrow} \qquad \neg x \wedge w \mapsto u{\downarrow}$$

$$x \wedge y \mapsto u{\uparrow} \qquad \neg x \wedge w \wedge \neg v \mapsto u{\downarrow}$$

Figure 3.9. Several generalized C-elements and their circuit symbols.

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

is a generalized C-element. ∎

The circuit symbol for some generalized C-elements are shown in Figure 3.9. When an input is only used in a pull-up, it is drawn with a "+" symbol; if it is only used in the pull-down, it is drawn with a "−" symbol. If the pull-up uses a negated input variable, or the pull-down uses a non-negated input variable, then the input is drawn with a "bubble." If a variable is used in both the pull-up and pull-down and is used with a negation in one and without a negation in the other, then it is shown without either a "+" or a "−."

3.3. Handshaking Expansions

The high-level design of asynchronous VLSI systems is done in CHP; the final target of the synthesis is production rules. *Handshaking expansions* (HSE) are an intermediate form used to aid the compilation of CHP into production rules.

Handshaking expansions are CHP programs with the following restrictions:

- All variables are Boolean-valued;
- There are no communication actions;
- All assignment statements only have constant expressions (**true** or **false**).

The purpose of these restrictions is to close the gap between production rules and CHP. Note that handshaking expansions still permit the use of selection statements, loops, and sequential composition.

The restriction on assignment statements is more a matter of syntax than a restriction. Consider the assignment statement

$$x := y$$

where x and y are both Boolean-valued variables. We can replace this assignment statement with

$$[y \longrightarrow x\uparrow \ [\!]\neg y \longrightarrow x\downarrow \ ]$$

and satisfy the restriction on assignment statements.

The variables used in HSE are classified into three types, based on their usage: *internal* variables are those that are only read and written by the handshaking expansion under consideration; *input* variables are those that are not internal, but are only read by the handshaking expansion; *output* variables are those that are not internal, but are written by the handshaking expansion.

3.3.1. Handshake Protocols

An important restriction on handshaking expansions is the elimination of all communication actions. We focus on this particular part of the transformation from CHP to HSE in this section.

We first consider the case when we have bare communication actions—communication actions that do not exchange data but only implement slack zero synchronization. Let X and Y two communication actions connected by a slack zero channel. We implement the communication channel with two wires $wire(xo, yi)$ and $wire(yo, xi)$ as shown in Figure 3.10. If X is executed in process P_1, then the variables xo and xi belong to process P_1; if Y is executed in process P_2, then variables yo and yi belong to process P_2. xo and yo are output variables, and xi and yi are input variables.

Assume the program begins in a state where all variables are **false**. We can replace X and Y with the actions U_x and U_y defined as:

$$U_x \equiv \ xo\uparrow; [xi]$$
$$U_y \equiv \ [yi]; yo\uparrow$$

In addition, we have the two wires that correspond to the production rules

$$xo \mapsto yi\uparrow \qquad yo \mapsto xi\uparrow$$
$$\neg xo \mapsto yi\downarrow \qquad \neg yo \mapsto xi\downarrow$$

by the definition of the *wire* operator. If we execute the actions X and Y concurrently together with wires, the only possible execution sequence is:

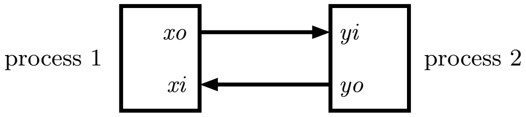


Figure 3.10. Two-wire implementation of a communication channel.

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

Since X cannot terminate until Y is initiated, and Y cannot terminate unless X is initiated, these two operations implement the necessary synchronization. Since the synchronization action is implemented using more than one action, we need the notion of *completion* of a non-atomic action.

Consider an action that consists of a sequence of several atomic actions. This non-atomic action is *initiated* when the first atomic action is executed. The non-atomic action is *terminated* when the final action in the sequence terminates. For non-atomic actions, the notion of completion does not coincide with termination.

If we examine a trace of the computation, then we can pick a point in this trace as the *completion* point for a non-atomic action if it lies between the initiation and termination point of the action, and if the action is guaranteed to terminate after the completion point in all possible valid environments containing the non-atomic action. Given this notion, the completion point of the sequence

$xo{\uparrow}; yi{\uparrow}; yo{\uparrow}; xi{\uparrow}$

can be defined as the point when the action $yo{\uparrow}$ is initiated. In terms of the implementation of the pair of synchronization primitives, the communication action is defined to be *completed* when Y reaches the semicolon between $[yi]$ and $yo{\uparrow}$.

When the communication implemented by X and Y terminates, the computation is in a state where variables xo, yo, xi, and yi are **true**. We cannot implement the next X action with U_x and the next Y action with U_y; instead we use D_x and D_y shown below:

$$D_x \equiv \quad xo{\downarrow}; [\neg xi]$$

$$D_y \equiv \quad [\neg yi]; yo{\downarrow}$$

The solution of alternating U_x and D_x as the implementation of X and alternating U_y and D_y as the implementation of Y is known as *two-phase handshaking*, and actions X and Y are said to be implemented using a *two-phase handshake protocol*. Since U_x initiates the communication action by changing a variable, X is said to be implemented using an *active* protocol; Y is said to be implemented using a *passive* protocol.

Often it is not possible to syntactically determine which X actions can be replaced with U_x, and which X actions can be replaced with D_x. CHP program

$*[\ C?c;\ \ [c \longrightarrow X\ \ [\!]\neg c \longrightarrow \textbf{skip}]\ \ ]$

is an instance of where we cannot syntactically replace X with U_x or U_y. In general, two-phase handshaking implementations require testing the current value of the variables as follows:

$$X \equiv \quad xo := \neg xo; [xi = xo]$$

$$Y \equiv \quad [yi \neq yo]; yo := \neg yo$$

This tends to lead to complex circuit implementations. In general, we prefer the *four-phase handshake protocol*. In a four-phase handshake protocol, X is implemented as $U_x; D_x$ and Y is implemented as $U_y; D_y$. X is said to be implemented using an *active* protocol, and Y is said to be implemented using a *passive* protocol. Observe that the D-parts in the implementation introduce an additional synchronization between the two processes whose sole purpose is to reset all the variables to **false**.

Although four-phase handshaking consists of more actions in sequence, the actions are simpler and the resulting circuits tend to be faster. There are also other transformations we will introduce later that can hide the second half of the four-phase handshake. Two-phase handshaking is preferred in cases where wire delays are dominant such as off-chip communication.

Example. Consider the one-place buffer $*[L; R]$. The handshaking expansion for this process with L implemented using a four-phase passive protocol, and R implemented with a four-phase active protocol is:

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

where (li, lo) and (ri, ro) are the variables that are used to implement L and R respectively. A two-phase implementation of this same process would be:

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

∎

3.3.2. Probes

Let Y be implemented using a four-phase passive communication protocol on (yi, yo). The probe $\overline{Y}$ can be implemented as yi. A probed communication action

$$\overline{Y} \longrightarrow ...; Y$$

is implemented

$$yi \longrightarrow ...; [yi];\, yo\uparrow;\, [\neg yi];\, yo\downarrow$$

Since probes are stable, the wait $[yi]$ is redundant. We therefore implement the probed communication as:

$$yi \longrightarrow ...; yo\uparrow;\, [\neg yi];\, yo\downarrow$$

Given a matching pair of communication actions X and Y where both X and Y are not probed, the choice of active or passive protocol is arbitrary but a choice must be made. If one of the two actions, say X, is probed, then X must be passive, which forces Y to be active. Additional constraints are imposed by the way processes are connected. For instance, if we are to implement a linear chain of one-place buffers of the type $*[L; R]$, then since the R port of one buffer is connected to the L port of the adjacent buffer, L and R must use different protocols.

Example. Consider the CHP program

```
*[[$\overline{C} \longrightarrow x\uparrow; C$
  ▯$\overline{D} \longrightarrow x\downarrow; D$
 ]]
```

Since both channels C and D are probed, we implement them with a four-phase passive protocol as shown below:

```
*[[$ci \longrightarrow x\uparrow; co\uparrow; [\neg ci]; co\downarrow$
  ▯$di \longrightarrow x\downarrow; do\uparrow; [\neg di]; do\downarrow$
 ]]
```

∎

3.3.3. Lazy-Active Protocol

Let two communications X and Y be implemented by a four-phase handshake protocol. Since the first half of the protocol synchronizes actions X and Y, we could *postpone* the last part of the four-phase protocol while still implementing the synchronization. This transformation is called *reshuffling*.

The four-phase *lazy-active* handshake protocol is a commonly used reshuffling of a four-phase active protocol. Let X be implemented by a four-phase active protocol on variables (xi, xo) as follows:

$$X \equiv \quad xo\uparrow; [xi]; xo\downarrow; [\neg xi]$$

If X is executed repeatedly, the execution of the (unreshuffled) handshaking expansion containing X is of the form

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

We postpone the wait $[\neg xi]$ until the beginning of the next occurrence of $xo\uparrow$. The result is:

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

Since xi is initially **false**, we can view this reshuffling as replacing the action X with

$$X \equiv \quad [\neg xi]; xo\uparrow; [xi]; xo\downarrow \qquad .$$

X is said to be implemented using a lazy-active protocol. The protocol is "lazy" because the wait $[\neg xi]$ is postponed until the next time X is executed.

Using a lazy-active protocol rather than an active protocol typically improves performance, because the second wait in the handshake is postponed till the last possible point (the start of the next handshake on X).

3.3.4. Single-Wire Handshaking*

Let X and Y be two synchronization actions. Consider the following implementation for X:

$$X \equiv \;\; w{\uparrow};\, \texttt{[}\neg w\texttt{]}$$
$$Y \equiv \;\; \texttt{[}w\texttt{]};\, w{\downarrow}$$

This implementation uses a single wire that is written by the processes that contain X and Y. However, using this handshaking expansion precludes the possibility of a stable, non-interfering production rule set.

We can make use of this handshaking expansion if we are willing to make timing assumptions. The operator used by X to detect that $w{\uparrow}$ has executed must have a lower threshold compared to the operator used by Y. Similarly, the operator used by Y to detect that $w{\downarrow}$ has executed must have a lower threshold compared to the operator used by X.

3.4. Production Rule Expansion

Production rule expansion transforms a handshaking expansion into a stable, non-interfering production rule set. It is the most difficult part of the synthesis method since the sequencing specified by the handshaking expansion must be implemented by selecting appropriate guards for the production rules. Production rule expansion consists of three main steps:

- state assignment;
- guard strengthening; and
- symmetrization.

We first consider the case of a *straight-line* HSE, defined to be of the form:

$$*\texttt{[}\;\; \texttt{[}w_0\texttt{]};\, t_0;\, \texttt{[}w_1\texttt{]};\, t_1;\, \ldots;\, \texttt{[}w_{n-1}\texttt{]};\, t_{n-1} \;\; \texttt{]}$$

where t_i's are transitions and w_i's are Boolean expressions (possible **true**), for $0 \leq i < n$. The production rule expansion must translate this HSE to a semantically equivalent production rule set

$$P \equiv \{G_i \mapsto t_i \mid 0 \leq i < n\} \quad .$$

Since the execution of t_i is preceded by wait $\texttt{[}w_i\texttt{]}$, the guard G_i must contain this condition—i.e., $G_i \Rightarrow w_i$ for all i. Another way of writing this property is that $G_i \equiv G_i \wedge w_i$, implying that we can replace each guard G_i with $G_i \wedge w_i$ without changing the meaning of the production rule set P. Therefore, for P to be a stable production rule set, the wait-conditions in the HSE must be *stable*.

A wait-condition w is said to be stable if once w becomes **true** and the HSE is at the semicolon preceding the wait $\texttt{[}w\texttt{]}$, w remains **true** until the completion of the following assignment in the HSE. Observe that all the waits introduced by the handshaking expansion are stable.

Unstable waits are caused by the incorrect use of shared variables, and by negated probes. The case of negated probes will be handled separately in Chapter 6. If the CHP contains no shared variables, then the only shared variables introduced in the HSE are from communication channels.

Therefore, a HSE generated from a CHP program without negated probes and shared variables is guaranteed to have stable waits.

For P to implement the HSE, the production rules must obey the execution order specified by the HSE. Since the HSE is sequential, at most one production rule can have an effective firing in any state, i.e., the number of production rules for which $G_i \wedge \neg R(t_i)$ holds is at most one. In addition, if rule $G_i \mapsto t_i$ has an effective firing, then the next effective firing must be due to production rule $G_{i+1} \mapsto t_{i+1}$. Also, if w_{i+1} holds eventually after t_i has completed execution, then G_{i+1} must also hold eventually (to avoid deadlock).

We describe the compilation procedure with the help of an example. Consider the process

$$*[[\overline{L} \longrightarrow R; L]]$$

To translate this into HSE, we use a passive four-phase protocol on L (since L is probed), and we pick an active four-phase protocol on R. The resulting HSE is:

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

For this example, the choice of an active protocol on R is arbitrary; however, if this process were designed to be connected to a passive channel, then we would be forced to pick an active protocol on R.

Syntactic Guards. The first step in the compilation is to write down *syntactic guards*. The syntactic guard for a transition is defined to be its preceding wait (corresponding to the w_i's in the earlier example). The resulting production rule set is:

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

Since the HSE is deadlock-free, it is always possible to execute the syntactic production rules in program order. However, other execution orders may be possible. For instance, the production rule $\neg ri \mapsto lo\uparrow$ can fire in the initial state modifying lo, because ri is initially **false**.

Guard Strengthening. In order to prevent incorrect effective firings of production rules, we must reduce the number of states in which the production rules can fire. We do so by adding conjuncts to the syntactic guards for the incorrect production rules. This procedure is known as *guard strengthening*. To determine what we need to add to the guard for $lo\uparrow$ to prevent its incorrect firing, we annotate the HSE with the values of all its variables. The HSE annotated with the values of (li, lo, ri, ro) at each semicolon is shown below:

$$\begin{array}{l} *[\{\mathtt{X000}\}[li]; \{\mathtt{1000}\}ro\uparrow; \{\mathtt{10X1}\}[ri]; \{\mathtt{1011}\}ro\downarrow; \\ \quad \{\mathtt{10X0}\}[\neg ri]; \{\mathtt{1000}\}lo\uparrow; \{\mathtt{X100}\}[\neg li]; \{\mathtt{0100}\}lo\downarrow \\ ] \end{array}$$

The value `0` is used to represent **false**, and `1` is used to represent **true**. A variable is `X` when it could be either **true** or **false**. The state of variables (li, lo, ri, ro) at each semi-colon depends on the HSE for the environment of the HSE. We assume the *weakest possible environment*, i.e., we assume that the environment simply performs a four-phase handshake on each channel. In this example, the environment is assumed to be of the form

$$\begin{aligned} & *[\ li\!\uparrow;\, [lo];\, li\!\downarrow;\, [\neg lo]\ \ ] \\ \| \quad & *[\ [ro];\, ri\!\uparrow;\, [\neg ro];\, ri\!\downarrow\ ] \end{aligned}$$

When all shared variables obey the four-phase handshake protocol, determining the values of all variables at each state is a relatively simple task: when an output variable changes, the corresponding input variable is set to X until the HSE waits for the input variable to change; all other changes to variables are specified locally in the HSE, and can be easily determined. Once we have determined the state of all variables at each semi-colon, we can determine the states in which a particular syntactic production rule can fire.

Consider the rule $\neg ri \mapsto lo\!\uparrow$. The following HSE uses "$\bullet$" to mark those states in which this production rule can fire.

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

We have marked the states where the guard $\neg ri$ is **true**. Of those states, the states in which the rule $\neg ri \mapsto lo\!\uparrow$ has effective firings is given by:

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

These are the only states we need to consider, because in all other cases firing $\neg ri \mapsto lo\!\uparrow$ would not change the state of the system.

Of these states, the state just before $lo\!\uparrow$ is a valid firing because it is permitted by the HSE. Indeed, without this firing the production rule set would be erroneous. Also, because the waits are stable, we need not consider the state before $[\neg ri]$. This is because the production rule $\neg ri \mapsto lo\!\uparrow$ already includes this wait, and so it can only fire in those states where $\neg ri$ holds. So if we are in the state just before $[\neg ri]$ and the production rule fires, this means that $\neg ri$ holds implying that we are in a state where the HSE can pass the wait and execute $lo\!\uparrow$. Therefore, the set of states in which $\neg ri \mapsto lo\!\uparrow$ misfires can be reduced to:

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

To ensure non-interference, the state just before $lo\!\downarrow$ must also be treated as an undesirable state, because although firing $lo\!\uparrow$ there would be vacuous, non-interference requires that both $lo\!\uparrow$ and $lo\!\downarrow$ should not fire in the same state. Therefore, the final set of states in which $\neg ri \mapsto lo\!\uparrow$ can misfire is:

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

This set of states is known as the *conflicting set* for production rule $\neg ri \mapsto lo\!\uparrow$. The goal of production rule expansion is to ensure that the conflicting set for each production rule is empty.

In general, consider the production rule $B \mapsto x\!\uparrow$ in a handshaking expansion

$$\ldots\{\textcircled{1}\}; x{\uparrow}\{\textcircled{2}\};\ \ \ldots\ \ \{\textcircled{3}\}; x{\downarrow}\{\textcircled{4}\}; \ldots\{\textcircled{5}\}; x{\uparrow}$$

The *firing set* for the production rule $B \mapsto x\uparrow$ is the set of states in which the guard B is **true**. The *disallowed set* is the set of states in which B should not fire. In the HSE shown above, the disallowed set for $B \mapsto x\uparrow$ is the set of states starting from the state marked "③" (inclusive) up to but not including the state "⑤." The production rule must fire in state "①," and the firing is vacuous from state "②" up to but not including state "③." The conflicting set is defined to be the intersection of the firing set and disallowed set, and is empty for a correct production rule.

The firing set for a production rule $B \mapsto x\uparrow$ is determined by the guard for the production rule. It is possible that there is no guard for $x\uparrow$ that makes the conflicting set for the production rule empty. This occurs when two semicolons in the handshaking expansion cannot be distinguished by the values of variables used in the handshaking expansion. Two semicolons in the handshaking expansion where all variables could have the same value are said to be *indistinguishable*. In the handshaking expansion shown earlier, the underline states shown below are indistinguishable.

$$\begin{array}{l} *[\underline{\{\texttt{X000}\}}[\mathit{li}]; \underline{\{\texttt{1000}\}}\mathit{ro}{\uparrow}; \{\texttt{10X1}\}[\mathit{ri}]; \{\texttt{1011}\}\mathit{ro}{\downarrow}; \\ \quad \underline{\{\texttt{10X0}\}}[\neg \mathit{ri}]; \underline{\{\texttt{1000}\}}\mathit{lo}{\uparrow}; \{\texttt{X100}\}[\neg \mathit{li}]; \{\texttt{0100}\}\mathit{lo}{\downarrow} \\ \quad] \end{array}$$

However, we do not consider the states before and after the waits as being indistinguishable, because they only differ in the value of the variables occurring in the wait condition. Given a stable wait $[w_i]$ in the handshaking expansion with initial state A and final state B, $B \equiv A \wedge w_i$. Therefore, the only truly indistinguishable states are:

$$\begin{array}{l} *[\underline{\{\texttt{X000}\}}[\mathit{li}]; \{\texttt{1000}\}\mathit{ro}{\uparrow}; \{\texttt{10X1}\}[\mathit{ri}]; \{\texttt{1011}\}\mathit{ro}{\downarrow}; \\ \quad \underline{\{\texttt{10X0}\}}[\neg \mathit{ri}]; \{\texttt{1000}\}\mathit{lo}{\uparrow}; \{\texttt{X100}\}[\neg \mathit{li}]; \{\texttt{0100}\}\mathit{lo}{\downarrow} \\ \quad] \end{array}$$

There are certain cases when we are guaranteed that there are indistinguishable states. If a handshaking expansion contains a full four-phase handshake with no intervening action, then the initial and final state of the handshake protocol are indistinguishable.

$$\ldots; \bullet \mathit{xo}{\uparrow}; [\mathit{xi}]; \mathit{xo}{\downarrow}; [\neg \mathit{xi}]; \bullet\ldots$$

In the protocol shown above, the states marked with a "•" are indistinguishable since the only changing variables are xi and xo, and they can take on the same values in the two states that are marked.

State Variable Insertion. When a handshaking expansion has indistinguishable states, we introduce *state variables*—new variables that eliminate indistinguishable states. In the HSE above, we introduce a new state variable x that is initially **false**, and use it to eliminate indistinguishable states. The HSE is:

$$*[[\mathit{li}]; \mathit{ro}{\uparrow}; [\mathit{ri}]; x{\uparrow}; \mathit{ro}{\downarrow}; [\neg \mathit{ri}]; \mathit{lo}{\uparrow}; [\neg \mathit{li}]; x{\downarrow}; \mathit{lo}{\downarrow}\ \ ]$$

The HSE annotated with the values of (x, li, lo, ri, ro) at each semicolon is shown below:

$$\begin{array}{l} *[\{\texttt{0X000}\}\,[li];\{\texttt{01000}\}ro\uparrow;\{\texttt{010X1}\}\,[ri];\{\texttt{01011}\}x\uparrow;\{\texttt{11011}\}ro\downarrow; \\ \quad \{\texttt{110X0}\}\,[\neg ri];\{\texttt{11000}\}lo\uparrow;\{\texttt{1X100}\}\,[\neg li];\{\texttt{10100}\}x\downarrow;\{\texttt{00100}\}lo\downarrow \\ \quad] \end{array}$$

This HSE has no indistinguishable states. The conflicting states for $\neg ri \mapsto lo\uparrow$ are:

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

We can make the conflicting set for this production rule empty by using x. The resulting production rule is $x \wedge \neg ri \mapsto lo\uparrow$, and it makes the conflicting set empty. In addition, this production rule can fire in the state where $lo\uparrow$ occurs in the HSE. Therefore, the production rule is correct. Other production rules for this HSE can be determined in a similar fashion. The result is:

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

Symmetrization. Symmetrization is technique where we attempt to convert a state-holding operator into a combinational one so that we do not have to introduce feedback circuitry when implementing the operator in CMOS. In this example, we can replace the production rule for $lo\downarrow$ with $ri \vee \neg x \mapsto lo\downarrow$, and the production rule for $ro\downarrow$ with $\neg li \vee x \mapsto ro\downarrow$, thereby symmetrizing the two operators. The symmetrized rules are:

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

The reason $\neg x \mapsto lo\downarrow$ can be replaced with $ri \vee \neg x \mapsto lo\downarrow$ is that ri is only true when the production rule would have a vacuous firing. The conflicting set for $ri \vee \neg x \mapsto lo\downarrow$ is empty.

In general, consider the operator

$$\begin{array}{rcl} x \wedge B & \mapsto & z\uparrow \\ \neg B & \mapsto & z\downarrow \end{array}$$

If x holds whenever B holds, then we can replace the production rule for $z\downarrow$ with $\neg x \vee \neg B \mapsto z\downarrow$, thereby symmetrizing the operator for z. We can replace $\neg B \mapsto z\downarrow$ with $\neg x \vee \neg B \mapsto z\downarrow$ whenever no new effective firings have been introduced, i.e., whenever $(x \vee \neg B \vee \neg z)$ is invariant. The condition arises from three reasons: the states where x is **true** don't introduce any new firings; the states where $\neg B$ is **true** don't introduce any new firings either; the states where $\neg z$ is **true** only result in effective firings.

In general, if $\mathcal{I}$ describes the invariant of the system (it specifies all the states the system can be in), then $\mathcal{I} \Rightarrow (x \vee \neg B \vee \neg z)$ is the precise condition that must hold for the above mentioned symmetrization to be valid.

Let $G^+ \mapsto x\uparrow$ and $G^- \mapsto x\downarrow$ be two stable, non-interfering production rules that specify the operator for x. We examine the conditions under which we can replace these with the combinational operator $C^+ \mapsto x\uparrow$ and $C^- \mapsto x\downarrow$. The effective firings of $G^+ \mapsto x\uparrow$ occur when $G^+ \wedge \neg x$ holds. These firings *must occur* for a valid implementation of the operator. Other than that, any vacuous firing can occur. Similarly, the production rule for $x\downarrow$ must have effective firings when $G^- \wedge x$ holds.

Since the states where $G^+ \wedge \neg x$ hold are the only states that can be part of the rule for $x\uparrow$ when x is **false**, all the other states where x is **false** must be in the rule for $x\downarrow$. Therefore, $D^- \equiv (G^- \wedge x) \vee (\neg x \wedge \neg G^+)$ is the set of states where C^- must hold. Similarly, $D^+ \equiv (G^+ \wedge \neg x) \vee (x \wedge \neg G^-)$ is the set of states where C^+ must hold. By construction, the expressions D^+ and D^- are non-interfering, and they correspond to a combinational implementation for the operator x. Assuming non-interference of G^+ and G^-, we can simplify these expressions to $D^+ \equiv G^+ \vee (x \wedge \neg G^-)$, and $D^- \equiv G^- \vee (\neg x \wedge \neg G^+)$.

The problem is that the operator specified by $D^+ \mapsto x\uparrow$ and $D^- \mapsto x\downarrow$ uses x in the guard. To avoid this, we must eliminate variable x from the guards by using the invariant $\mathcal{I}$. We can obtain a combinational operator for x if we can find C^+ and C^- that do not use variable x, but where $\mathcal{I} \Rightarrow (D^+ \equiv C^+) \wedge (D^- \equiv C^-)$.

In the example above, we attempted to symmetrize the guard for the operator $x \wedge B \mapsto z\uparrow$ and $\neg B \mapsto z\downarrow$. The conditions D^+ and D^- are $(x \wedge B) \vee (z \wedge B)$ and $\neg B \vee (\neg z \wedge \neg x)$. Assuming that $(x \vee \ B \vee \ z)$ is invariant, we can prove that $D^+ \equiv x \wedge B$, and that $D^- \equiv \neg x \vee \neg B$, showing that our earlier intuition about symmetrizing that particular operator was correct.

Operator Reduction. The process of *operator reduction* is one where standard operators shown earlier are identified in the production rule set. This process is useful during the physical design of the circuit, since standard operators are typically taken from a library.

3.5. Reshuffling

The compilation of process

$$*[[\overline{L} \longrightarrow R; L]]$$

results in the HSE

$$H \quad\equiv\quad *[[li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; lo\downarrow] \qquad .$$

HSE H has indistinguishable states, because a full four-phase handshake on (ri, ro) without any intervening actions occurs as part of H. We introduced a state variable to translate H into production rules.

Alternatively, since the four-phase handshake consists of two synchronization actions, we could attempt to *reshuffle* H by postponing part of the synchronization on (ri, ro). Reshuffling is the transformation where actions in a handshaking expansion are moved relative to each other while preserving ordering among actions in individual handshake protocol. Moving actions in a HSE results in different production rules, and can eliminate indistinguishable states.

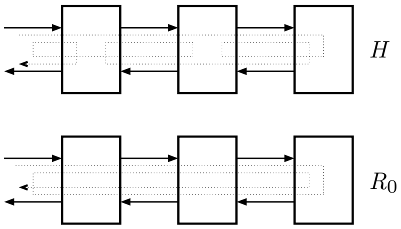


Figure 3.11. Reshufflings H and R_0. The dotted lines trace the sequence of transitions that occur in each reshuffling, showing the different synchronization patterns.

For the HSE shown above, a possible reshuffling is:

$$R_0 \;\equiv\; *\texttt{[}\texttt{[}li\texttt{]};ro{\uparrow};\texttt{[}ri\texttt{]};lo{\uparrow};\texttt{[}\neg li\texttt{]};ro{\downarrow};\texttt{[}\neg ri\texttt{]};lo{\downarrow}\texttt{]}$$

Observe that the actions on (li, lo) and (ri, ro) still obey the four-phase handshake protocol even though we have moved the actions relative to one another. Another reshuffling of the same HSE is:

$$R_1 \;\equiv\; *\texttt{[}\texttt{[}li\texttt{]};lo{\uparrow};\texttt{[}\neg li\texttt{]};lo{\downarrow};ro{\uparrow};\texttt{[}ri\texttt{]};ro{\downarrow};\texttt{[}\neg ri\texttt{]}\texttt{]}$$

Reshuffling R_1 also corresponds to the direct translation of process $*\texttt{[}L;R\texttt{]}$ into handshaking expansions. Since this CHP is different from the CHP for the original process, we conclude that reshuffling can change the semantics of the computation. Therefore, a more careful analysis is required to determine whether a particular reshuffling is valid.

Reshuffling R_0 can be translated into the following valid production rule set:

$$\begin{array}{lll} li \mapsto ro{\uparrow} & \qquad & \neg li \mapsto ro{\downarrow} \\ ri \mapsto lo{\uparrow} & \qquad & \neg ri \mapsto lo{\downarrow} \end{array}$$

This implementation corresponds to two wires. Clearly this implementation has different properties when compared to the unreshuffled solution H. Figure 3.11 shows the sequence of transitions that occur in H and R_0, demonstrating that the two reshufflings have different synchronization patterns.

3.5.1. Analysis of Reshuffling

We observe that a four-phase handshaking expansion comprises *two* synchronization actions. Therefore, our first step in analyzing the relation between CHP programs and reshuffled handshaking expansions is to represent a four-phase communication action as two two-phase communication actions. We rewrite a single synchronization action L as $L^+;L^-$, where L^+ and L^- refers to the two synchronization actions in the four-phase protocol for L. For instance, if L were implemented with an active protocol, then the HSE for L^+ and L^- is given by:

$$\begin{array}{lll} L^+ & \equiv & lo{\uparrow};\texttt{[}li\texttt{]} \\ L^- & \equiv & lo{\downarrow};\texttt{[}\neg li\texttt{]} \end{array}$$

Example. The handshaking expansion

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

corresponds to the following two-phase CHP:

$$*[L^+;R^+;R^-;L^-\ ]$$

In this example, there is a clear relationship between the CHP and the HSE. □

Consider the HSE

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

It is harder to determine which CHP this HSE corresponds to since the protocols for L and R have been mixed. Using the two-phase CHP notation, we can partially write the CHP for this HSE as:

$$*[\ \ [li]\,;ro\uparrow;lo\uparrow;\underbrace{[\neg li]\,;lo\downarrow}_{L^-};\,[ri]\,;\underbrace{ro\downarrow;\,[\neg ri]}_{R^-}\ \ ]$$

$$\triangleright\quad *[\ \ [li]\,;ro\uparrow;lo\uparrow;L^-;\,[ri]\,;R^-\ \ ]$$

Consider the sequence $ro\uparrow;lo\uparrow$. Since $lo\uparrow$ cannot be blocked by the environment in any way, it would be semantically equivalent to replace this fragment with $lo\uparrow;ro\uparrow$ because the two fragments have the same synchronization behavior. With this observation, we can rewrite the HSE as:

$$*[\ \ \underbrace{[li]\,;lo\uparrow}_{L^+};ro\uparrow;L^-;\,[ri]\,;R^-\ \ ]$$

$$\triangleright\quad *[\ \ L^+;ro\uparrow;L^-;\,[ri]\,;R^-\ \ ]$$

To analyze the rest of the handshaking expansion, we make the following observation: we can determine the synchronization effect of two reshuffled two-phase handshaking expansions by composing them with different *environments* having known synchronization behavior. This can be summarized as follows. Assume the two synchronization actions are L^+ and R^+, and we are examining reshuffled versions of their handshaking expansions. Then the reshuffling corresponds to:

- $L^+;R^+$ if it deadlocks with environment $R^+;L^+$ but not with $L^+;R^+$
- $R^+;L^+$ if it deadlocks with environment $L^+;R^+$ but not with $R^+;L^+$
- $L^+\|R^+$ if it does not deadlock with environments $R^+;L^+$ or $L^+;R^+$

Using this observation, we can determine the CHP for any two adjacent synchronization actions. Figure 3.12 shows the result of examining all possible combinations of two adjacent two-phase actions. Therefore, the final CHP for the HSE is:

L Passive, R Passive		**L Active, R Active**		**L Active, R Passive**	
$[li]\,;lo\uparrow;\,[ri]\,;ro\uparrow$	$\triangleright\ L^+;R^+$	$lo\uparrow;\,[li]\,;ro\uparrow;\,[ri]$	$\triangleright\ L^+;R^+$	$lo\uparrow;\,[li\wedge ri]\,;ro\uparrow$	$\triangleright\ L^+;R^+$
$[li\wedge ri]\,;lo\uparrow,ro\uparrow$	$\triangleright\ [\overline{L^+}\wedge\overline{R^+}]\,;L^+\|R^+$	$lo\uparrow,ro\uparrow;\,[li\wedge ri]$	$\triangleright\ L^+\|R^+$	$lo\uparrow;\,[ri]\,;ro\uparrow;\,[li]$	$\triangleright\ L^+\|R^+$
$[ri]\,;ro\uparrow;\,[li]\,;lo\uparrow$	$\triangleright\ R^+;L^+$	$ro\uparrow;\,[ri]\,;lo\uparrow;\,[li]$	$\triangleright\ R^+;L^+$	$[ri]\,;lo\uparrow;\,[li]\,;ro\uparrow$	$\triangleright\ [\overline{R^+}]\,;L^+;R^+$
				$[ri]\,;ro\uparrow,lo\uparrow;\,[li]$	$\triangleright\ R^+;L^+$

Figure 3.12. Reshuffled two-phase communications and their corresponding CHP.

$$*[\ L^+; \underbrace{ro{\uparrow}; L^-; [ri]}_{L^- \| R^+}; R^-\ ]$$

$$\triangleright\quad *[\ L^+; (L^- \parallel R^+); R^-\ ]$$

This technique can be used to analyze the synchronization patterns of reshuffled handshaking expansions.

3.5.2. Bullets

Since a four-phase communication action can be thought of as two synchronization actions, the bullet operator "•" between two communication actions can be implemented by reshuffling the four-phase implementations of the two channels. The following two-phase CHP programs show two possible implementation of the bullet:

$$L \bullet R \qquad \triangleright \quad L^+; R^+; L^-; R^-$$
$$L \bullet R \qquad \triangleright \quad L^+; R^+; R^-; L^-$$

These implement the bullet because neither L nor R can complete unless the other action has started.

3.6. Communication With Data

The handshake protocols we have seen implement slack zero synchronization between two actions. The second function of communication actions is transmitting data. In this section we examine the standard way to send data over a communication channel using a *dual-rail protocol*.

Let X and Y be two one-bit communication ports connected by a channel, with X being the sender port and Y being the receiver port. If data being sent by X is encoded by the values of a set of variables, the receiver must be able to detect when there is valid data being transmitted. This detection operation must be possible no matter what the delay on the wires connecting X and Y is.

The dual-rail code achieves this by encoding a single bit of information with two Boolean-valued variables (sometimes called *rails*). Let $(d0, d1)$ be the dual-rail encoding of a single bit. $d0$ is called the zero (or **false**) rail, and $d1$ is known as the one (or **true**) rail. Initially, both variables are **false**. This is called the neutral state for the encoding. The value **false** is encoded by setting $d0$ to **true** and $d1$ to **false**. The value **true** is encoded by setting $d1$ to **true** and $d0$ to **false**. The state where both $d0$ and $d1$ are **true** is forbidden. Figure 3.13 pictorially depicts the valid state transitions for this protocol.

An active send protocol on port X along with the matching receive protocol on port Y is shown below:

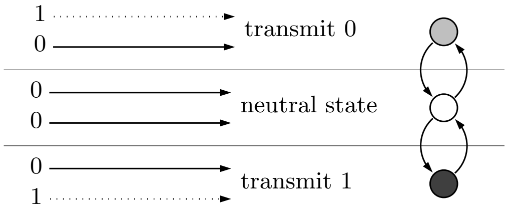


Figure 3.13. Dual-rail data encoding.

$$\begin{aligned}
X!b &\equiv \texttt{[}b \longrightarrow xto{\uparrow} \,\square\, \neg b \longrightarrow xfo{\uparrow}\texttt{]};\, \texttt{[}xi\texttt{]};\, xto{\downarrow}, xfo{\downarrow};\, \texttt{[}\neg xi\texttt{]}\\
Y?v &\equiv \texttt{[}yti \longrightarrow v{\uparrow} \,\square\, yfi \longrightarrow v{\downarrow}\texttt{]};\, yo{\uparrow};\, \texttt{[}\neg yti \wedge \neg yfi\texttt{]};\, yo{\downarrow}
\end{aligned}$$

The two ports are connected by wires $\mathit{wire}(xto, yti)$, $\mathit{wire}(xfo, yfi)$, and (yo, xi). The protocol used for port X is active because it initiates the communication action between X and Y. The variable b is dual-rail encoded so that the process executing $Y?v$ can unambiguously determine the value being transmitted on port X. Figure 3.14 shows the connections between the two communicating processes.

The send and receive actions are asymmetric since data is transferred from the sender to the receiver. Therefore, we also need to consider the case of a passive sender protocol and active receiver protocol. These two protocols are shown below:

$$\begin{aligned}
X!b &\equiv \texttt{[}xi\texttt{]};\, \texttt{[}b \longrightarrow xto{\uparrow} \,\square\, \neg b \longrightarrow xfo{\uparrow}\texttt{]};\, \texttt{[}\neg xi\texttt{]};\, xto{\downarrow}, xfo{\downarrow}\\
Y?v &\equiv yo{\uparrow};\, \texttt{[}yti \longrightarrow v{\uparrow} \,\square\, yfi \longrightarrow v{\downarrow}\texttt{]};\, yo{\downarrow};\, \texttt{[}\neg yti \wedge \neg yfi\texttt{]}
\end{aligned}$$

In the passive implementation for X, the probe of X is implemented by variable xi. The probe of Y is implemented by the condition $(yti \vee yfi)$, since either yti or yfi could be **true** when there is a communication action pending on port X.

Example. The process

$$\begin{array}{l}
*\texttt{[[}\overline{P} \longrightarrow P?x\\
\quad \square\, \overline{Q} \longrightarrow Q!x\\
\quad \texttt{]]}
\end{array}$$

is called a *single-variable register*, since it stores x and provides access ports by which the variable can be read and written. The handshaking expansion for this process is:

$$\begin{array}{l}
*\texttt{[[}pti \vee pfi \longrightarrow \quad \texttt{[}pti \longrightarrow x{\uparrow} \;\square\; \neg pti \longrightarrow x{\downarrow}\texttt{]};\, po{\uparrow};\, \texttt{[}\neg pti \wedge \neg pfi\texttt{]};\, po{\downarrow}\\
\quad \square\, qi \longrightarrow \texttt{[}x \longrightarrow qto{\uparrow} \,\square\, \neg x \longrightarrow qfo{\uparrow}\texttt{]};\, \texttt{[}\neg qi\texttt{]};\, qto{\downarrow}, qfo{\downarrow}\\
\quad \texttt{]]}
\end{array}$$

We use a passive protocol on both ports P and Q since they are probed. $\blacksquare$

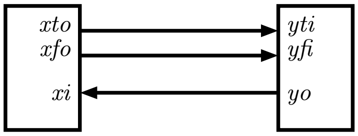


Figure 3.14. Two processes connected by a dual-rail channel.

3.7. Production Rule Expansion: Selections

The handshaking expansion for a CHP program may not be straight-line. In this section, we examine another HSE structure that can also be simply translated into production rules. The form is shown below:

```
$*[[B_1 \longrightarrow S_1$
  $[\!]\, B_2 \longrightarrow S_2$
  $...$
  $[\!]\, B_n \longrightarrow S_n$
 $]]$
```

Each S_i is a straight-line HSE, and the entire HSE has a reactive process structure. This HSE structure occurs quite frequently from CHP processes that have reactive process structures.

The standard strategy to compile this HSE is to implement each guarded command as:

$$*[[B_i]\,; S_i]$$

(a straight-line program) and to enforce mutual exclusion among the guards using variables that occur in the HSE. The original HSE is implemented by the parallel composition of the straight-line programs. We illustrate this strategy by compiling the single-variable register.

The HSE for the single-variable register is:

```
$*[[pti \lor pfi \longrightarrow\ \ [pti \longrightarrow x\uparrow\ \ [\!]\ \ \neg pti \longrightarrow x\downarrow]\,; po\uparrow; [\neg pti \land \neg pfi]\,; po\downarrow$
  $[\!]\, qi \longrightarrow [x \longrightarrow qto\uparrow [\!] \neg x \longrightarrow qfo\uparrow]\,; [\neg qi]\,; qto\downarrow, qfo\downarrow$
 $]]$
```

We rewrite this HSE as:

```
$*[[pti \longrightarrow x\uparrow; po\uparrow; [\neg pti]\,; po\downarrow$
  $[\!]\, pfi \longrightarrow x\downarrow; po\uparrow; [\neg pfi]\,; po\downarrow$
  $[\!]\, qi \land x \longrightarrow qto\uparrow; [\neg qi]\,; qto\downarrow$
  $[\!]\, qi \land \neg x \longrightarrow qfo\uparrow; [\neg qi]\,; qfo\downarrow$
 $]]$
```

The reader can verify that these two handshaking expansions are identical. Each individual guarded command can be translated into production rules, and the result of simply writing down all the production rules for the first two guarded commands is:

$$\begin{array}{lll}
(1) & \mathit{pti} & \mapsto\ x\uparrow \\
(2) & \mathit{pti} \wedge x & \mapsto\ \mathit{po}\uparrow \\
(3) & \neg \mathit{pti} & \mapsto\ \mathit{po}\downarrow \\
(4) & \mathit{pfi} & \mapsto\ x\downarrow \\
(5) & \mathit{pfi} \wedge \neg x & \mapsto\ \mathit{po}\uparrow \\
(6) & \neg \mathit{pfi} & \mapsto\ \mathit{po}\downarrow
\end{array}$$

In this HSE, rules 2 and 6 are interfering, and so are rules 3 and 5. We need to implement mutual exclusion among the actions from the two guarded commands when we combine the production rules.

The problem of implementing mutual exclusion is the same as enforcing program order in a straight-line HSE, and is resolved in the same way: by strengthening guards until the conflicting states are removed, and if necessary, by the introduction of state variables. In this example, we can strengthen the guard for $\mathit{po}\downarrow$ and replace both rules 3 and 6 with:

$$\neg \mathit{pti} \wedge \neg \mathit{pfi}\ \mapsto\ \mathit{po}\downarrow$$

The final combined production rules for the first two guards are

$$\begin{array}{lr}
\mathit{pti}\ \mapsto\ x\uparrow & \mathit{pti} \wedge x \vee \mathit{pfi} \wedge \neg x\ \mapsto\ \mathit{po}\uparrow \\
\mathit{pfi}\ \mapsto\ x\downarrow & \neg \mathit{pti} \wedge \neg \mathit{pfi}\ \mapsto\ \mathit{po}\downarrow
\end{array}$$

The production rules for the third and fourth guarded command are:

$$\begin{array}{rr}
x \wedge \mathit{qi}\ \mapsto\ \mathit{qto}\uparrow & \neg \mathit{qi}\ \mapsto\ \mathit{qto}\downarrow \\
\neg x \wedge \mathit{qi}\ \mapsto\ \mathit{qfo}\uparrow & \neg \mathit{qi}\ \mapsto\ \mathit{qfo}\downarrow
\end{array}$$

which completes the production-rule generation for this HSE.

Alternatively, we can use the original HSE and take the first guarded command from it:

$$*[[\mathit{pti} \longrightarrow x\uparrow\ [\!]\, \mathit{pfi} \longrightarrow x\downarrow]; \mathit{po}\uparrow; [\neg \mathit{pti} \wedge \neg \mathit{pfi}]; \mathit{po}\downarrow]$$

The production rules for this HSE can be directly written as:

$$\begin{array}{lr}
\mathit{pti}\ \mapsto\ x\uparrow & \mathit{pti} \wedge x \vee \mathit{pfi} \wedge \neg x\ \mapsto\ \mathit{po}\uparrow \\
\mathit{pfi}\ \mapsto\ x\downarrow & \neg \mathit{pti} \wedge \neg \mathit{pfi}\ \mapsto\ \mathit{po}\downarrow
\end{array}$$

The second guarded command (the part that reads x) corresponds to the following HSE:

$$*[[x \wedge \mathit{qi} \longrightarrow \mathit{qto}\uparrow [\!] \neg x \wedge \mathit{qi} \longrightarrow \mathit{qfo}\uparrow];\ \ [\neg \mathit{qi}]; \mathit{qto}\downarrow, \mathit{qfo}\downarrow]$$

The production rules for this HSE are:

$$\begin{array}{rr}
x \wedge \mathit{qi}\ \mapsto\ \mathit{qto}\uparrow & \neg \mathit{qi}\ \mapsto\ \mathit{qto}\downarrow \\
\neg x \wedge \mathit{qi}\ \mapsto\ \mathit{qfo}\uparrow & \neg \mathit{qi}\ \mapsto\ \mathit{qfo}\downarrow
\end{array}$$

Environment Considerations. In this single-variable register, we are given that $\overline{P}$ and $\overline{Q}$ are mutually exclusive. Consider the case when $\overline{P}$ becomes **true**. Once $\mathit{po}\uparrow$ executes, the synchronization between the register and the environment is complete. Therefore, there are environments that might begin executing the handshake on Q *before* the complete handshake on P has finished. The production rules given above are correct only if we assume that the *complete* four-phase handshake protocol on P and Q is non-overlapping.

If the environment can start the next communication before the previous handshake is finished, we must strengthen the guards to prevent misfirings. An examination of the handshaking expansion shows that to prevent $\mathit{qto}{\uparrow}$ or $\mathit{qfo}{\uparrow}$ from misfiring, we can strengthen their guards with $\neg \mathit{po}$. Similary, the guards for $x{\uparrow}$, $x{\downarrow}$, and $\mathit{po}{\uparrow}$ can be strengthened with $\neg \mathit{qto} \wedge \neg \mathit{qfo}$ to avoid misfirings. The final production rule set is:

$$
\begin{array}{rclcrcl}
\neg \mathit{qto} \wedge \neg \mathit{qfo} \wedge \mathit{pti} & \mapsto & x{\uparrow} & \qquad & \neg \mathit{po} \wedge x \wedge \mathit{qi} & \mapsto & \mathit{qto}{\uparrow} \\
\neg \mathit{qto} \wedge \neg \mathit{qfo} \wedge \mathit{pfi} & \mapsto & x{\downarrow} & & \neg \mathit{po} \wedge \neg x \wedge \mathit{qi} & \mapsto & \mathit{qfo}{\uparrow} \\
 & & & & & & \\
\mathit{pti} \wedge x \vee \mathit{pfi} \wedge \neg x & \mapsto & \mathit{po}{\uparrow} & & \neg \mathit{qi} & \mapsto & \mathit{qto}{\downarrow} \\
\neg \mathit{pti} \wedge \neg \mathit{pfi} & \mapsto & \mathit{po}{\downarrow} & & \neg \mathit{qi} & \mapsto & \mathit{qfo}{\downarrow}
\end{array}
$$

We can apply the state-assignment and guard-strengthening technique to arbitrary handshaking expansions. However, in the general case, states may not be easy to calculate. For arbitrary handshaking expansions, states cannot be represented as simple bit vectors as we have done for straight-line HSE. The relationships between different variables becomes important and the full generality of Boolean expressions is necessary to represent each state.

3.8. Bubble Reshuffling

We have seen how to translate certain types of CHP programs into handshaking expansions, handshaking expansions into production rules, and the CMOS implementation of production rules. For the production rule set to be CMOS-implementable, all variables used in the pull-up must be inverted, and all variables used in the pull-down must be uninverted. The process of converting a production rule set into one that is directly implementable in CMOS is called *bubble reshuffling*. Note that there are production rule sets which cannot be bubble reshuffled. In this case, we have to change the handshaking expansion by either reshuffling actions, or adding/changing state variables.

Consider the production rule $x \wedge \mathit{qi} \mapsto \mathit{qto}{\uparrow}$. Since the production rule uses the uninverted forms of x and qi, it is not CMOS-implementable. In this case we could change the sense of x and qi, by globally replacing x with $\neg \_x$ and qi with $\neg \_\mathit{qi}$. This transformation would make this particular production rule CMOS implementable.

Suppose in addition to $x \wedge \mathit{qi} \mapsto \mathit{qto}{\uparrow}$ we had rule $\neg x \wedge \mathit{qi} \mapsto \mathit{qfo}{\uparrow}$. Now no matter what sense we pick for x, either the first or the second rule will not be CMOS-implementable. As a naive solution, we might attempt to add an inverter, generate $\_x$ from x, and use $\_x$ in the guard for one of the rules and x in the guard for the other. However due to the semantics of production rule execution, x and $\_x$ are *not* always inverses of each other. Therefore, there is no guarantee that the resulting production rule set is stable and non-interfering.

3.8.1. Acknowledgment

A production rule $G \mapsto t$ is stable when the guard G cannot become **false** in those states where $G \wedge \neg R(t)$ holds. To examine what makes a production rule stable, we introduce the notion of a *successor* relation among different transitions in a computation.

Consider a computation described by the HSE

$$*[\ x\uparrow; y\uparrow; x\downarrow; y\downarrow\ ] \quad .$$

This HSE imposes a total ordering on the transitions in the computation. Let $\langle t, k\rangle$ denote the kth occurrence of transition t. Then we write $\langle t, k\rangle \prec \langle t', k'\rangle$ just when the kth occurrence of transition t is guaranteed to be executed before the k'th occurrence of transition t'. For the HSE above, we can say that:

$$\langle x\uparrow, 0\rangle \prec \langle y\uparrow, 0\rangle \prec \langle x\downarrow, 0\rangle \prec \langle y\downarrow, 0\rangle \prec \langle x\uparrow, 1\rangle \prec \cdots$$

In general, there will be transitions that occur in parallel. The successor relation only holds between two transitions that are truly ordered. Given this successor relationship between transitions, we can examine what makes a production rule stable.

Consider the production rule $G^+ \mapsto x\uparrow$, and let $G^+ \wedge \neg x$ hold. Suppose that when this production rule fires, it would be the kth occurrence of $x\uparrow$. If the opposing production rule $G^- \mapsto x\downarrow$ is to fire in a future state, then G^+ must become **false** before G^- becomes **true** to prevent interference. The transition that makes G^+ **false** *must* execute in a state where $G^+ \wedge x$ holds due to stability. This implies that the transition that makes G^+ **false**, say $\langle a, k_a\rangle$, *must be ordered* with the corresponding $\langle x\uparrow, k\rangle$ action. In other words, we are guaranteed that

$$\langle x\uparrow, k\rangle \prec \langle a, k_a\rangle \prec \langle x\downarrow, k\rangle$$

(assuming that x is **false** initially). We say that this intervening transition *acknowledges* the kth occurrence of $x\uparrow$. Notice that a particular transition in the system is always acknowledged by another transition in the computation. We can summarize this property as follows:

> **Acknowledgment Theorem.** *For a production rule set to be stable and non-interfering, every transition in the system that is followed by another transition that changes the same variable must be acknowledged.*

3.8.2. Isochronic Forks and Branches

Figure 3.15 shows operators for variables x, y, and z. The variable x is an input to the operators for y and z. Since the production rule for x is stable, it satisfies the acknowledgment theorem and the output of a gate *must be acknowledged* by the output of some gate where x is an input. In Figure 3.15, any transition on x must be acknowledged by a transition on y or z.

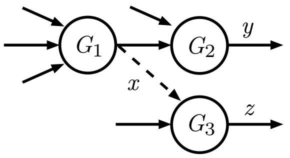


Figure 3.15. Three operators, with an isochronic branch from x to z shown with dotted lines.

Suppose transition $x\uparrow$ are acknowledged by transitions $y\downarrow$ and $z\downarrow$, but transition $x\downarrow$ is only acknowledged by transition $y\uparrow$. Since both transitions on x are acknowledged by the branch of the circuit connecting x with gate G_2, we can add arbitrary delay on that branch of the circuit without violating stability or non-interference. If we explicitly model the delay with a wire operator, then the wire operator will be stable because every transition on the input of the wire is acknowledged by the output of the wire, and every transition on the output of the wire is acknowledged by y. If, instead, we add arbitrary delay on the branch connecting x with G_3, then the resulting production rules will not be stable because the output of the explicit wire connecting x and G_3 will not be acknowledged when $x\downarrow$ occurs.

The branch of the circuit connecting x to G_3 is called an *isochronic branch*, and the branch connecting x to G_2 is called *non-isochronic*. For the circuit to function correctly, we make the *isochronic fork assumption*; we assume that the relative delay between the two wires connecting x to gates G_3 and G_2 is negligible compared to the time taken by y to change.

The assumption is actually more liberal. Since transitions on y acknowledge transitions on x, a change in y must eventually result in a change the the input to the gate for x that causes x to change again. At this point, the original change in x must have propagated to the input of gate G_3.

There is another possiblity—namely, that the change in y actually propagates through a sequence of gates and then affects an input to gate G_3 causing z to change. In this case, we must assume that the change in x will have propagated to the input of G_3 before this occurs.

In either case, the isochronic fork assumption states that the relative delay between two wires is less than the delay through a sequence of gates.

When we examine analog effects, the definition of "x has changed" depends on the input threshold of the gate that x is connected to. Therefore, the gates G_2 and G_3 might have different definitions of the voltage x must cross for x to have changed. For instance, if x is connected to a p-transistor in G_2, but to only an n-transistor in G_3, then the transition $x\downarrow$ might be "detected" by G_2 when x drops below $Vdd - v_{tp}$; however, the change in x will only be detected by G_3 when x falls below v_{tn}. In this case, the isochronic fork assumption requires that the voltage of x to be below v_{tn} before the change in y propagates to the input of G_3.

3.8.3. Transformations

Given the definition of isochronic forks and branches, we conclude that we can introduce inverters only on non-isochronic branches of a fork in the circuit. If an inverter is placed on an

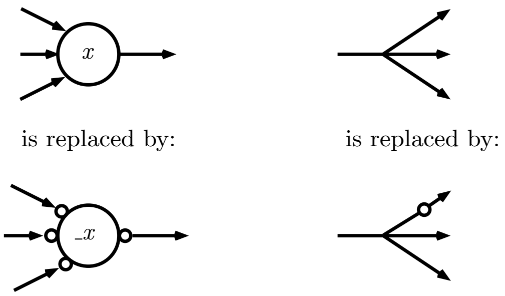


Figure 3.16. Bubble-reshuffling transformations.

isochronic branch, then transitions on the output of the inverter will not always be acknowledged, making the inverter unstable.

This restriction always permits the placement of an inverter on the *output* of a gate (before the fork), because all transitions on the output of a gate are acknowledged when the production rule set is stable and non-interfering.

When transforming a production rule set into one that is CMOS-implementable by bubble-reshuffling, we attempt to pick senses of different signals so that any inverters we need only occur on either the output of a gate or on non-isochronic branches. Given these two criteria, we permit two bubble-reshuffling transformations: (1) Invert the sense of a variable; and (2) Add an inverter to a non-isochronic branch.

When we invert the sense of a variable, the pull-up and pull-down networks are exchanged. If $G^+ \mapsto x\uparrow$ and $G^- \mapsto x\downarrow$ is an operator and we change the sense of x to $\_x$, then we have the operator $G^+ \mapsto \_x\downarrow$ and $G^- \mapsto \_x\uparrow$. For $G^+ \mapsto x\uparrow$ to be CMOS-implementable, every variable in G^+ had to appear in negated form; however, for $G^+ \mapsto \_x\downarrow$ to be CMOS-implementable, every variable must appear in non-negated form. Therefore, changing the sense of $\_x$ also changes the senses of all the inputs required to make the operator CMOS-implementable. The two transformations used for bubble-reshuffling are shown pictorally in Figure 3.16.

The production rules for the HSE

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

are given by:

$$\begin{array}{rcl@{\qquad\qquad}rcl} x & \mapsto & lo\uparrow & x \wedge \neg li & \mapsto & ro\uparrow \\ \neg x \wedge \neg ri & \mapsto & lo\downarrow & \neg x & \mapsto & ro\downarrow \end{array}$$

$$\begin{array}{rcl} li & \mapsto & x\uparrow \\ ri & \mapsto & x\downarrow \end{array}$$

Since applying a bubble-reshuffling transformation to a single gate can affect the senses of other variables in the circuit, we examine all gates in this production rule set using a graph notation.

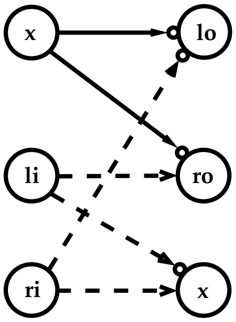


Figure 3.17. Bubble-reshuffling the production rules for HSE *LR*.

(We can keep this transformation local because the wires implementing the handshake protocols are non-isochronic.)

The graph notation describing the circuit for HSE *LR* is shown in Figure 3.17. The left column (x, li, and ri) corresponds to variables used as inputs to operators, and the right column corresponds to the output variables. Each input has an arrow connecting it to an output whose gate uses the variable. Note that even though we draw edges on the arrows, the graph is treated as an undirected graph. If the gate requires the inverted version of the variable to make it CMOS-implementable, then the arrow has a bubble on it. Isochronic branches are shown with dashes rather than solid lines. For instance, consider the operator $ri \mapsto x\downarrow$ and $li \mapsto x\uparrow$. We need the non-negated version of ri, and the negated version of li to make this operator CMOS-implementable, as shown in Figure 3.17.

To determine whether a branch is non-isochronic, we must examine the handshaking expansions as well as the production rules. A branch from an output a to the input of another operator b is non-isochronic whenever every transition on a is acknowledged by a transition on b. Since we do not use $\neg li$ in the operator for x, the transition $li\downarrow$ is not acknowledged by a change in x, making the branch from li to x isochronic.

Once we have the graph, we attempt to move all bubbles from isochronic branches onto non-isochronic branches by applying transformation (1), possibly eliminating bubbles. However, observe that transformation (1) never changes the parity of the number of bubbles on any cycle in the graph. Since bubbles correspond to inverters in the circuit, the final form of the graph that corresponds to the CMOS circuit cannot have any bubbles on isochronic branches.

If we are faced with a situation where the graph contains a cycle of isochronic branches that has an odd number of bubbles on it—known as a *negative cycle*—there will always be one isochronic branch that has a bubble on it. If a graph has a negative cycle, it can never be bubble-reshuffled to a configuration that has a valid CMOS implementation.

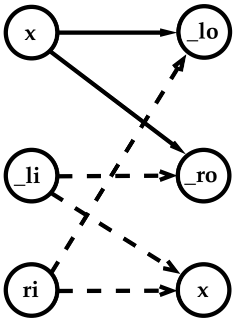


Figure 3.18. Bubble-reshuffled version of the production rules for *LR*.

Figure 3.18 shows the graph after bubble-reshuffling. Flipping the senses of *lo*, *ro*, and *li* eliminated all bubbles from isochronic branches. This choice results in the production rule set:

$$\begin{array}{rcl@{\qquad}rcl}
x & \mapsto & \_lo\downarrow & x \wedge \_li & \mapsto & \_ro\downarrow \\
\neg x \wedge \neg ri & \mapsto & \_lo\uparrow & \neg x & \mapsto & \_ro\uparrow
\end{array}$$

$$\begin{array}{rcl}
\neg\_li & \mapsto & x\uparrow \\
ri & \mapsto & x\downarrow
\end{array}$$

We also have another choice—obtained by complementing every variable in the circuit. This choice results in production rules

$$\begin{array}{rcl@{\qquad}rcl}
\neg\_x & \mapsto & lo\uparrow & \neg\_x \wedge \neg li & \mapsto & ro\uparrow \\
\_x \wedge \_ri & \mapsto & lo\downarrow & \_x & \mapsto & ro\downarrow
\end{array}$$

$$\begin{array}{rcl}
li & \mapsto & \_x\downarrow \\
\neg\_ri & \mapsto & \_x\uparrow
\end{array}$$

3.8.4. Direct Bubble Reshuffling

Given a variable x, a set of production rules that use x and that are CMOS-implementable may need x exclusively, its complement $\_x$ exclusively, or both senses of x.

If only x is used in a set of production rules that is stable, non-interfering, and CMOS-implementable, then all transitions on x are acknowledged and the circuit does not need modification.

If the inverted sense of x, namely $\_x$, is used exclusively in a set of production rules, then we can safely invert x to obtain $\_x$, and then use $\_x$ throughout. This is clearly safe since the original production rule set was stable and, therefore, all transitions on $\_x$ are acknowledged. Furthermore, all transitions on x are also acknowledged by transitions on $\_x$, since x is only connected to the inverter that generates $\_x$.

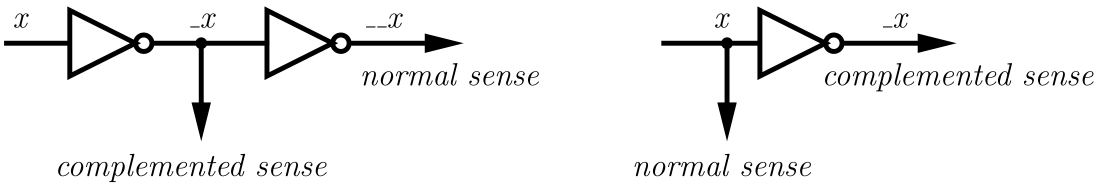


Figure 3.19. Direct Generation of Senses of x.

If both x and $\_x$ are used in a CMOS-implementable production rule set, then there are four transitions to consider: (a) A $0 \to 1$ transition on x; (b) A $1 \to 0$ transition on x; (c) A $0 \to 1$ transition on $\_x$; (d) A $1 \to 0$ transition on $\_x$. If the production rule set is such that transitions (a) and (b) are both acknowledged, then we can generate $\_x$ from x using an inverter, generate $\_\_x$ from $\_x$ using another inverter, and use $\_\_x$ in place of x in the circuit. The net result is that all transitions on $\_\_x$ are acknowledged (by assumption), which implies that all transitions on $\_x$ and x are also acknowledged through the two inverters. Therefore, we can use $\_x$ and $\_\_x$ for the two senses of x without having to worry about stability. Similarly, if (c) and (d) are both acknowledged, then we can generate $\_x$ from x using an inverter and avoid stability concerns. These two cases are shown in Figure 3.19.

Therefore, the only case that is problematic in terms of bubble reshuffling is one where transitions (a) and (c) are acknowledged but (b) and (d) are not, or vice versa. (Note that the case where only transitions (a) and (d) are acknowledged is not possible, since that implies that one of the transitions on x in the original production rule set (before bubble reshuffling) is not acknowledged.) The graph-based technique given above can be used to handle this case.

3.8.5. Dual-rail Variables

When attempting to bubble-reshuffle the production rules for a single-variable register, we are faced with the following operator:

$$\begin{aligned} \mathit{pti} \wedge x \vee \mathit{pfi} \wedge \neg x \;&\mapsto\; \mathit{po}\!\uparrow \\ \neg \mathit{pti} \wedge \neg \mathit{pfi} \;&\mapsto\; \mathit{po}\!\downarrow \end{aligned}$$

To make this production rule CMOS implementable, we need both x and its inverse. This inverter cannot be introduced by the bubble-reshuffling transformation, because we can only pick one sense of x as an input to a particular operator. To circumvent this problem, we return to the handshaking expansion and introduce a variable that can be used in place of $\neg x$.

We replace x with variables u and v. The operations $x\!\uparrow$ and $x\!\downarrow$ are implemented by changing both variables u and v. We adopt the same convention as that used by the dual-rail encoding for communication, namely that the state where both u and v are **true** is forbidden. Given this restriction, changes in x are implemented as:

$$\begin{array}{lcl} x{\uparrow} & \triangleright & u{\downarrow}; v{\uparrow} \\ x{\downarrow} & \triangleright & v{\downarrow}; u{\uparrow} \end{array}$$

After applying this transformation, the handshaking expansion for the single-variable register is

$$\begin{array}{l} *\mathtt{[[}\mathit{pti} \vee \mathit{pfi} \longrightarrow \mathtt{[}\mathit{pti} \longrightarrow u{\downarrow}; v{\uparrow} \;\square\, \mathit{pfi} \longrightarrow v{\downarrow}; u{\uparrow}\mathtt{]}\,; \mathit{po}{\uparrow}; \mathtt{[}\neg \mathit{pti} \wedge \neg \mathit{pfi}\mathtt{]}\,; \mathit{po}{\downarrow} \\ \quad \square\;\; \mathit{qi} \longrightarrow \mathtt{[}v \longrightarrow \mathit{qto}{\uparrow} \square\, u \longrightarrow \mathit{qfo}{\uparrow}\mathtt{]}\,; \;\; \mathtt{[}\neg \mathit{qi}\mathtt{]}\,; \mathit{qto}{\downarrow}, \mathit{qfo}{\downarrow} \\ \;\; \mathtt{]]} \end{array}$$

and results in production rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathit{pti} & \mapsto & u{\downarrow} & v \wedge \mathit{qi} & \mapsto & \mathit{qto}{\uparrow} \\ \mathit{pfi} & \mapsto & v{\downarrow} & u \wedge \mathit{qi} & \mapsto & \mathit{qfo}{\uparrow} \\ \neg \mathit{pfi} \wedge \neg u & \mapsto & v{\uparrow} & & & \\ \neg \mathit{pti} \wedge \neg v & \mapsto & u{\uparrow} & \neg \mathit{qi} & \mapsto & \mathit{qto}{\downarrow} \\ & & & \neg \mathit{qi} & \mapsto & \mathit{qfo}{\downarrow} \\ \mathit{pti} \wedge v \vee \mathit{pfi} \wedge u & \mapsto & \mathit{po}{\uparrow} & & & \\ \neg \mathit{pti} \wedge \neg \mathit{pfi} & \mapsto & \mathit{po}{\downarrow} & & & \end{array}$$

3.9. Initialization: Reset Circuits

So far we have seen how to design QDI asynchronous circuits that can be bubble-reshuffled to make them CMOS implementable. However, there is an important aspect that we have not addressed: what happens when you begin execution in an asynchronous circuit? If the circuit does not begin in a valid initial state, the analysis we performed during circuit synthesis no longer applies. To ensure correct circuit operation, we introduce a global *Reset* signal that is used to bring the circuit into a known state. We think of the execution of the circuit in two possible situations: (i) During the "reset" phase, when we initialize the circuit; (ii) During normal execution. The goal of introducing reset circuitry is to implement the reset phase correctly, and also to ensure that we can exit the reset phase and begin normal execution according to the production rules that have been designed. A valid initial state of the circuit specifies the values of each variable in the entire circuit. (Note that some variables may be "don't cares." Examples include register values that are guaranteed to be written before they are read.)

Given a variable x implemented by production rules $G^+ \mapsto x{\uparrow}$ and $G^- \mapsto x{\downarrow}$, we can set x to **false** during the reset phase by modifying the rules to $G^+ \wedge \neg \mathit{Reset} \mapsto x{\uparrow}$ and $G^- \vee \mathit{Reset} \mapsto x{\downarrow}$. Note that the *Reset* variable is used in the correct sense for the rule to remain CMOS-implementable. If we have to initialize x in the **true** state, we can use the complemented sense of *Reset* (we use "*_Reset*" for this variable) to perform the same operation. The production rules for this case are $G^+ \vee \neg \_\mathit{Reset} \mapsto x{\uparrow}$ and $G^- \wedge \_\mathit{Reset} \mapsto x{\downarrow}$.

A general observation is that to set a variable to a particular value, the reset phase must ensure that:

- The guard for the production rule that sets the variable to its correct initial value will become true eventually during the reset phase. If a reset transistor is used for this purpose, it is referred to as a "parallel" reset transistor.
- The opposing production rule will have its guard become false eventually during the reset phase. If a reset transistor is used for this purpose, it is referred to as a "series cut-off" reset transistor.

Note that both conditions are necessary, otherwise we cannot guarantee that the variable will reset to a deterministic value. There are some exceptions where we can reason about the relative strenghts of transistor networks, but we will ignore those for the purposes of this section.

3.9.1. Simplifying Reset Circuits

There are many optimizations possible during the reset phase that allow us to design circuits that do not require a large number of reset transistors. To illustrate this, consider the following production rule set where x must be reset to **true** and y must be reset to **false**.

$$\begin{array}{rcl@{\qquad\qquad}rcl} G^{+} & \mapsto & x\uparrow & x & \mapsto & y\downarrow \\ G^{-} & \mapsto & x\downarrow & \neg x & \mapsto & y\uparrow \end{array}$$

Once x has been reset to the correct value, the circuit will automatically reset y without having to explicitly add transistors to the inverter that implements y. This observation can be generalized as follows.

Definition 3.1. (*indirect reset*)

The set of variables $\{v_1, \ldots, v_k\}$ indirectly resets variable x iff the production rule for x sets x to the correct initial value when variables v_1, …, v_k are in the initial state.

In the example above, y is indirectly reset by $\{x\}$ because x is **true** in the initial state, and this is sufficient to set y to its correct initial value. Even if a variable is not indirectly reset, we might be able to simplify the reset circuit for it if we know that part of the reset condition is satisfied.

Definition 3.2. (*indirect cut-off*)

The set of variables $\{v_1, \ldots, v_k\}$ indirectly cuts-off variable x iff the production rule guard that sets x to its non-reset value is guaranteed to be **false** *when variables v_1, …, v_k are in the initial state.*

Example. The following example that extends the one above illustrates these definitions. Suppose the initial state requires that $x \wedge \neg y \wedge \neg z \wedge \neg w$ holds.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
G^+ & \mapsto & x\uparrow & x \wedge w & \mapsto & z\downarrow \\
G^- & \mapsto & x\downarrow & \neg x \wedge \neg w & \mapsto & z\uparrow
\end{array}$$

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

As before, $\{x\}$ indirectly resets y. The set $\{x, w\}$ does not indirectly reset z because the initial state corresponds to the C-element for z being state-holding. However, the set $\{x\}$ indirectly cuts-off z, because x being **true** ensures that the pull-up for z is **false**. $\{w\}$ does not indirectly cut-off z, because the initial value of w (**false**) does not guarantee that the guard for $z\uparrow$ will be **false**.

∎

Notice that we did not introduce a symmetric definition for a variable being "indirectly set." Such a definition might look like the following: "The set of variables $\{v_1, \ldots, v_k\}$ indirectly sets variable x iff the production rule guard that sets x to its reset value is guaranteed to be **true** when variables v_1, …, v_k are in the initial state." If this definition were to hold and the production rule set is non-interfering, then the initial state *must guarantee* that the opposing guard is **false**. This would make the variable satisfy definition 3.1, eliminating the need for a separate definition.

Once we have characterized the dependencies of variables on sets ofother variables, we can construct the *reset graph*—a bi-partite graph that captures the reset dependencies. We include edges from sets of variables to variables when we have an "indirectly resets" relationship between them. Specifically, we use set

$$RE1 = \{(S, v) \,|\, S \text{ indirectly resets } v\}$$

The set of vertices in the graph correspond to $\{x \,|\, (x, \_) \in RE1 \vee (\_, x) \in RE1\}$, and the set of edges is given by $RE1 \cup \{(x, S) \,|\, x \in S\}$. Once we have constructed this graph, reducing the number of reset transistors is relatively straightforward. We use the following procedure, constructing the set of variables R that we must reset in order for the entire circuit to be correctly initialized.

(i) Every variable that does not appear in the reset graph is added to set R;

(ii) All variables that have in-degree zero are added to R and deleted from the graph.

(iii) A deleted variable is removed from the reset graph, and it is also eliminated from every set that contains it.

(iv) Finally, all edges of the form $(\emptyset, v)$ are also eliminated. The variable v is treated as a deleted variable as in (iii). This scenario captures the fact that the variable v will be indirectly reset, since all its reset dependencies have been satisfied. (It is not added to R.) Note that in the case where multiple subsets might indirectly reset a variable, it sufficies to have one path reset it. Continue this as far as possible by going back to step (ii).

(v) Every remaining variable must be part of a cycle. Pick a variable in a cycle, add it to R, and delete it from the graph as in step (iii).

Once this process is complete, we have a set of variables R that (when reset) will completely initialize the entire circuit. In particular, note that we will end up initializing one variable per cycle of reset dependencies.

Finally, we need to design the reset circuit for each variable in R. We assign a total order to the variables in R, corresponding to the order we added the variables to set R. The first variable requires the full reset circuit described above, with both a cut-off reset transistor as well as a parallel reset transistor to set its value. For all subsequent variables, we can determine the minimal reset circuitry needed—either the full reset circuit or, in the case when the variable is indirectly cut-off by the variables that have already been reset, a simple parallel reset transistor to set the value of the variable.

Example. The bubble-reshuffled production rules for a specific reshuffling of an *L-R* buffer

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

are given by:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\neg\_x & \mapsto & lo\uparrow & li & \mapsto & \_x\downarrow \\
\_x \wedge \_ri & \mapsto & lo\downarrow & \neg\_ri & \mapsto & \_x\uparrow \\
 & & & & & \\
\neg\_x \wedge \neg li & \mapsto & ro\uparrow & & & \\
\_x & \mapsto & ro\downarrow & & &
\end{array}$$

From the handshaking expansion, the initial state satisfies $\neg x \wedge \neg lo \wedge \neg ro \wedge \neg ri$, which is the same (given the senses of signals in the bubble-reshuffled PRS) as $\_x \wedge \neg lo \wedge \neg ro \wedge \_ri$. The variable li could be either **true** or **false** in the initial state. Figure 3.20 shows the reset graph given the initial state and production rules.

Since $\_ri$ will be reset by the environment, resetting $\_x$ is sufficient to reset the entire circuit. This *assumes* that $\_ri$ will be reset *without* requiring lo. Otherwise there would be a cycle in the

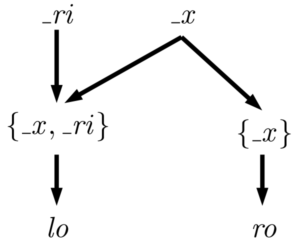


Figure 3.20. Reset graph for the *LR* buffer.

reset graph once we included the reset graph for the environment. The final production rule set with reset circuitry added is:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\neg\_x & \mapsto & lo\uparrow & \_Reset \wedge li & \mapsto & \_x\downarrow \\
\_x \wedge \_ri & \mapsto & lo\downarrow & \neg\_Reset \vee \neg\_ri & \mapsto & \_x\uparrow \\
\\
\neg\_x \wedge \neg li & \mapsto & ro\uparrow & & & \\
\_x & \mapsto & ro\downarrow & & &
\end{array}$$

∎

The example illustrates another major issue when designing the reset circuitry for a large system. Since production rule synthesis will be done per process, it makes sense to use a global reset *convention* so that we can design reset circuits on a per-process basis. While this might result in some inefficiency, this is more than compensated for by the fact that the synthesis procedure remains local.

Channel variables are the ones where reset conventions are required. These variables will inevitably appear on cycles due to the fundamental dependency from one transition to the next in a handshake protocol. Since we must guarantee that there is at least one variable on each cycle that is reset, a safe convention to adopt is to ensure that each acknowledge signal in a set of channel variables is unconditionally reset—without any assumptions on whether the data/request signals in the channel are reset. In other words, given a channel implemented with a set of data rails d_0, …, d_{n-1} and an acknowledge rail d_a, we assume that the environment can *assume* that d_a is reset regardless of whether d_0, …, d_{n-1} is reset. Also, we will assume that if we reset *all the acknowledge signals* that are connected to a process correctly, then all the output data rails will reset. In the example above, we reset *ro* unconditionally, and since the L and R channels are connected to different processes, we can use the fact that *ri* will reset to reset *lo*.

Note that if the buffers described above were connected in a ring, then we would no longer have the property that some process would have all its acknowledges reset unconditionally. For this example, connecting the processes in a ring would result in deadlock. To prevent this, some process would contain a data token and and it could be used to break the cycle in the reset graph.

3.9.2. Stability of Reset*

Given the initial state, some variables might require *Reset* for initialization while others may require *_Reset*. So far we have assumed that *Reset* and *_Reset* are idealized signals, i.e. they are perfect complements. However this is not a safe assumption because both signals are globally distributed across a chip. The number of transistors connected to *Reset*/*_Reset* will be large, which means that the slew rate on these signals may not be high unless appropriate reset drivers are designed.

The transition $\mathit{Reset}{\uparrow}$ is safe, because once reset is high and $\_\mathit{Reset}$ is low, the circuit will eventually reach its initial state. However, the transition $\mathit{Reset}{\downarrow}$ could be unsafe if all parts of the circuit do not "see" this transition at the same time, and especially if the relative delay between $\mathit{Reset}{\downarrow}$ and $\_\mathit{Reset}{\uparrow}$ is high. We now consider this transition in more detail.

Let $G^+ \mapsto x{\uparrow}$ and $G^- \mapsto x{\downarrow}$ be a gate that contains at least one reset transistor. The initial state of the system is given by $I \wedge \mathit{Reset} \wedge \neg\_\mathit{Reset}$, where I is a predicate that captures the values of all variables other than Reset and $\_\mathit{Reset}$. A gate is said to be *reset-passive* iff $I \wedge \neg\mathit{Reset} \wedge \_\mathit{Reset} \Rightarrow (\neg G^+ \vee x) \wedge (\neg G^- \vee \neg x)$. In other words, assuming all variables are in their initial state and $\mathit{Reset}{\downarrow}$ has occured, there are no effective firings of x possible (each guard is either **false** or the transition would have been vacuous). If $I \wedge \neg\mathit{Reset} \wedge \_\mathit{Reset} \not\Rightarrow (\neg G^+ \vee x) \wedge (\neg G^- \vee \neg x)$, then the gate is said to be *reset-active*. Reset-active rules initiate circuit operation with effective firings as soon as $\mathit{Reset}{\downarrow}$ occurs. Note that these rules *must* have both a series cut-off and parallel reset transistor.

Under a digital model, we either have $\mathit{Reset}{\downarrow}$ occuring before $\_\mathit{Reset}{\uparrow}$ or vice versa. (The case when they are simultaneous does not create any problems for the $\mathit{Reset}{\downarrow}$ transition, and can be ignored for the purposes of this discussion.) Also, if we do not wish to make an isochronic fork assumption about $\mathit{Reset}/\_\mathit{Reset}$ (which would be wise, since they are both global signals), we must assume that the arrival time of reset at each gate may be different. We examine the sequence of transitions from a reset-active gate to another gate that contains a reset transistor.

Suppose a reset-active gate initiates a sequence of firings that leads to another reset-active gate g. Without loss of generality we assume that no intervening gate that contains a reset signal, and that gate g is still in the reset state. We claim that this cannot cause an execution error. For this to cause an execution error, the sequence of firings must lead to a state where gate g no longer has an effective firing when it observes that reset is now low. However, if gate g had not actually fired even if it had observed reset low in this entire time period (because the circuit is QDI, we can arbitrary delay any transition without changing the behavior of the computation), this effective transition would be disabled—the definition of an instability. Since we know the original production rule set is stable and non-interfering, this cannot occur. Therefore, the only problematic case occurs when a reset-active gate initiates a sequence of firings that leads to a reset-passive gate.

Suppose a reset-active gate initiates a sequence of firings that leads to a reset-passive gate g. Let the gate g be, without loss of generality, $\_\mathit{Reset} \wedge G^- \mapsto x{\downarrow}$ and $\neg\_\mathit{Reset} \vee G^+ \mapsto x{\uparrow}$. (We will discuss the case where $\_\mathit{Reset}$ is omitted in each guard as well.) Note that problematic cases can only arise if a guard becomes **true**, since any other change cannot create any new effective firings.

This means that the series cut-off can never create any problems, no matter how slow the gate is to observe the *_Reset*↑ transition, since its only function is to disable the $x\downarrow$ transition. However, a parallel reset transistor *without* a series cut-off can create difficulties because the sequence of firings might lead to a state where G^- might become **true**, creating temporary interference that could lead to an invalid state transition. We now outline a number of different solutions to this problem.

Always use cut-off transistors. One solution to this problem is to always use a series cut-off transistor when a parallel reset is needed. This is not ideal, because adding a transistor in series can significantly degrade circuit performance.

Reset convention and timing. The problematic case only arises when a parallel reset transistor is used without a series cut-off. This implies that if the parallel reset transistor was turned off *before* any sequence of firings caused the gate to malfunction, errors would be avoided. One possible convention that can be used is the following: use only one reset signal (either *_Reset* or *Reset*) for parallel reset transistors that have no cut-off. If the signal used is *_Reset*, then the external reset signal is always *_Reset*, and *Reset* is generated from *_Reset* after a delay that is sufficient to ensure that *_Reset* has changed state. As long as the *_Reset* slew rate is high (in order to prevent a reset-active gate that uses *_Reset* from creating a malfunction in a reset-passive gate that has only a series *_Reset* transistor), the circuit will safely exit the reset state. Slew rates can be improved by using on-chip drivers.

Additional Reset signals. A solution that does not require an on-chip timing assumption is to use two flavors of reset: *sReset* and *pReset*. *sReset* is always connected to series cut-offs, while *pReset* is connected to parallel reset transistors for gates that do not have any series cut-offs. A gate that requires both a series cut-off as well as a parallel reset transistor uses *sReset* for both purposes. (Note that bubble reshuffling might require *_sReset* and *_pReset* in addition to the normal *sReset* and *pReset* signals.) The circuit is put into the reset state by setting both *sReset* and *pReset* to **true**. To exit the reset state, first *pReset* is set to **false**. Then after a delay, *sReset* is set to **false**. This solution does not require any timing assumption between *sReset*/*pReset* and *_sReset*/*_pReset*. The only assumption is that when *sReset* is set to **false**, *pReset* is already **false**.

3.10. Analog Considerations*

In the entire synthesis procedure, we have assumed a digital abstraction for circuit operation. Except for the stability requirement where we appealed to the notion of a monotonic signal transition, we assumed digital operation. We now examine some of the analog circuit issues that arise given a stable, non-interfering production rule set. They fall into issues that can affect the correctness of the circuit, and those that affect performance and energy per operation. The goal of this examination is to look at some of the issues that arise when we attempt to provide a digital abstraction of the circuit.

3.10.1. Reset and Staticizers

There is an interesting interaction between parallel reset transistors and state-holding operators. Since parallel reset transistors add diffusion capacitance to the output of a gate, one may be tempted to make these transistors as small as possible. However, if these transistors are made too small, they may not be able to overcome the opposing staticizer. If the relative strength ratios of a reset transistor and its the opposing staticizer is not adjusted, another option is to cut-off the opposing staticizer by incorporating a reset signal as part of the weak driver.

3.10.2. Passive Pull-up/Pull-down Networks

The performance of the circuit implementation of an operator

$$\begin{aligned} G^+ &\mapsto x\uparrow \\ G^- &\mapsto x\downarrow \end{aligned}$$

depends on the number of transistors in series required to change the output. Since n-transistors normally have a lower effective on-resistance when compared to p-transistors, it is preferred to use n-transistors for the network that requires the longer series transistor chain. In this section we consider the option of using a passive resistor implementation for a pull-up network.

The effect of using a passive resistive pull-up is to replace $G^+ \mapsto x\uparrow$ with $\textbf{true} \mapsto x\uparrow$. Since this implementation must rely on the relative on-resistance of the pull-up and pull-down, we assume that if G^- is true, then $x\downarrow$ will fire even though there is interference with $\textbf{true} \mapsto x\uparrow$. The first consequence of using a passive pull-up is that when G^- holds, the circuit will dissipate static power due to interference. Therefore, if we do not want static power dissipation, the circuit should never stay in a state where G^- holds for long periods of time.

When G^- is **false**, the resistive pull-up will cause the rule $\textbf{true} \mapsto x\uparrow$ to fire; therefore, the circuit behaves as if we had production rules:

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

which is a combinational operator. If this production rule is a valid implementation of the original gate for x, and the interference issues are not a concern, then we can use a passive pull-up to replace the network for G^+.

3.10.3. Self-Invalidating Rules

Production rules of the form $x \wedge B \mapsto x\!\downarrow$ or of the form $\neg x \wedge B \mapsto x\!\uparrow$ are said to be *self-invalidating*. This form merits special attention because of the way we implement production rules in CMOS. For the rest of this section, we consider self-invalidating rules of the form $x \wedge B \mapsto x\!\downarrow$; the other case is similar.

Consider the CMOS implementation of the rule $x \wedge B \mapsto x\!\downarrow$, and consider the case when x is **true**. Suppose the circuit enters a state where B is **true**. Now the operator that implements x starts changing the value of x in a continuous manner. However, as the operator sets x to GND, it *disables* the switching network that is setting x to GND. In this case, x might never reach GND.

☡ The precise final voltage of x depends on the slew rate of of x—determined by the strength of the switching network and the capacitive load on x, and the wire delay from the output of the switching network that implements x to the input gate controlled by x.

To prevent this, an additional constraint we impose on production rule sets is that they not contain any self-invalidating production rules. Formally, a stable production rule $G \mapsto t$ is said to be self-invalidating just when $R(t) \Rightarrow \neg G$.

It is normally simple to avoid self-invalidating rules. Consider the operator

$$\begin{aligned} x \wedge B &\mapsto x\!\downarrow \\ B' &\mapsto x\!\uparrow \end{aligned}$$

with self-invalidating rule $x \wedge B \mapsto x\!\downarrow$. Suppose we change the production rules to:

$$\begin{aligned} B &\mapsto x\!\downarrow \\ B' &\mapsto x\!\uparrow \end{aligned}$$

Since $\neg x \wedge B \vee x \wedge B \equiv B$, we can consider this transformation to be equivalent to adding $\neg x \wedge B$ as a disjunct to the guard for $x\!\downarrow$. Note that the only additional states we have introduced where the guard for $x\!\downarrow$ is **true** are states in which $\neg x$ holds. Therefore, all new firings we have introduced are *vacuous*. The only problem we may have introduced is interference. If $\neg x \wedge B \wedge B'$ is **false**, there is no interference and the new production rules are not self-invalidating.

☡☡ It is possible that the value of a node might not switch all the way to Vdd or GND even if the circuit has no self-invalidating production rules. Consider production rule $B \mapsto x\!\uparrow$, and a sequence of effective firings starting with $x\!\uparrow$ that lead to a state in which B is **false** (called a *cut-off path*). If the time taken by this sequence of firings is short compared to the slew rate of the transition $x\!\uparrow$, then x might never reach Vdd. This situation can be avoided by eliminating short (3 transitions or less in modern CMOS processes) cut-off paths in the circuit, and by sizing transistors so that all operators have reasonably fast output slew-rates (fast compared to the cut-off path for the operator). Note that eliminating self-invalidating rules is a special case that removes all 1-transition cut-off paths.

Another technique to avoid self-invalidating rules is to introduce new variables x' and x'', replacing

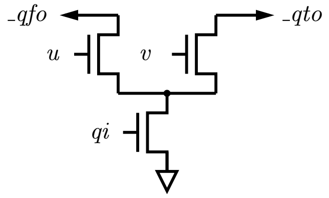


Figure 3.21. Shared-gate network implementation.

$$\begin{aligned} x \wedge B &\mapsto x\uparrow \\ B' &\mapsto x\downarrow \end{aligned}$$

with:

$$\begin{aligned} x' &\mapsto x''\downarrow \qquad & x \wedge B &\mapsto x'\uparrow \\ \neg x' &\mapsto x''\uparrow \qquad & B' &\mapsto x'\downarrow \end{aligned}$$

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

3.10.4. Shared-Gate Networks

Dual-rail and other data encoding schemes used to implement QDI circuits lead to situations that lend themselves to shared-gate network implementations. Knowledge of signal encoding and valid states should be kept in mind when designing transistor implementations of QDI circuits. To illustrate this, consider the following production rule fragment from the single variable register

$$\begin{aligned} v \wedge qi &\mapsto qto\uparrow \\ u \wedge qi &\mapsto qfo\uparrow \end{aligned}$$

To make these CMOS-implementable, we switch the sense of *qto* and *qfo* to obtain

$$\begin{aligned} v \wedge qi &\mapsto \_qto\downarrow \\ u \wedge qi &\mapsto \_qfo\downarrow \end{aligned}$$

Since the pair (u, v) form a dual-rail state variable and never enter the $(\mathbf{true}, \mathbf{true})$ state, we can implement the production-rules for *_qto* and *_qfo* by *sharing* the *qi* transistor in the pull-down network. The circuit that results in shown in Figure 3.21. Since u and v are exclusive high, there is no sneak path from *_qto* to *_qfo*. This transformation reduces the gate capacitance on *qi*. In later chapters, shared gate networks will be used to improve circuit performance for a variety of arithmetic circuits.

3.10.5. Charge Sharing

Charge sharing is a problem that occurs in complex transistor networks, especially those that arise due to shared gates as outlined in the previous section. The basic problem can be described as follows. Consider the production rule:

$a \wedge b \wedge c \;\mapsto\; d\downarrow$

The CMOS implementation of this rule is shown in Figure 3.22. The problem that concerns us can arise in the following situation. Consider the case when the voltages on internal nodes i and j as well as d are at GND, i.e., $V(i) = V(j) = V(d) = Vdd$. Assume that the pull-up network executes transition $d\uparrow$, so that d is now at Vdd. Now consider the case when a and b both become **true**, but c is still **false**. At this point, d, a, and b are all at Vdd, and j and i are at GND. This is not a stable situation, and charge will re-distribute until either $V(i) = V(j) = V(d)$, which, by charge conservation, would be $VddC_{out}/(C_1 + C_2 + C_{out})$, or until transistor with gate a turns off. If this voltage is low enough, then another gate in the circuit may think that d is now **false**, and misfire. This phenomenon is known as *charge sharing*, and affects clocked circuits as well as asynchronous ones. In particular, this phenomenon may be repeated; for instance, a could be **false**, and c could become **true**, thereby making $V(i) = V(j) = GND$. Now we can repeat the problem by making c **false**, and then making a **true**. The repeated version of this phenomenon is referred to as *charge pumping*. These problems depend on a number of things: (a) the relative capacitances of the internal nodes in a transistor network versus the load on the output; (b) the precise transistor network implementation for a production rule; (c) the possible states for the circuit.

We discuss the charge sharing problem for n-transistor networks only; the p-transistor charge sharing problem is analogous.

Ordinary charge sharing. Consider the CMOS-implementable production rule:

$b_1 \;\mapsto\; x\uparrow$

$b_2 \;\mapsto\; x\downarrow$

If $b_1 \vee b_2 \equiv$ **true**, i.e., if the production rule is combinational, then charge sharing does not arise. This is because the output is always connected to either Vdd or GND. Therefore, our first condition for charge sharing to occur is that $\neg b_1 \wedge \neg b_2$ must hold, i.e., x is in a state-holding state. Also this means that we must have a staticizer on x, which actually helps combat charge sharing since it is

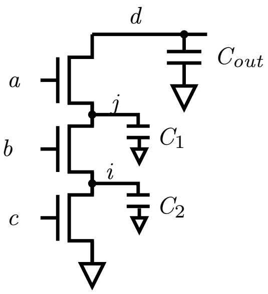


Figure 3.22. Pull-down network illustrating charge-sharing.

continuously driving x to the correct value even though it is doing so using a weak inverter.

Given an internal node i in the network that implements b_2, let $guard(i)$ be the Boolean expression that must be **true** for i to be connected to x, let $supply(i)$ be the Boolean experssion that must be **true** for i to be connected to GND, and let $C(i)$ be the node capacitance of i. For a particular state S of the circuit, the total internal capacitance that is exposed to the output x is given by:

$$C(S) = \sum_{\substack{i \in b_2 \\ S \Rightarrow guard(i)}} C(i)$$

Therefore, the ratio of output to internal capacitance for state S is given by

$$R(S) = \frac{C(x)}{C(S)}$$

If x is at Vdd and all the internal nodes are at GND, then the output voltage may drop to

$$V'(x) = \frac{Vdd \times C(x)}{C(x) + C(S)} = \frac{Vdd \times R(S)}{1 + R(S)}$$

making $V'(x) - Vdd = \Delta V = -Vdd/(1 + R(S))$. Therefore, to evaluate charge sharing problems, we must examine:

- The possible states that the circuit can be in. This is determined by examining the handshaking expansion.
- The states that the circuit can be in where $x \wedge \neg b_1 \wedge \neg b_2$ holds. These are potential charge-sharing states.
- For each charge-sharing state, we must determine the ratio $R(S)$. If the resulting $|\Delta V|$ is larger than a threshold determined by the fanout of signal x, then there is a charge sharing problem.

Charge sharing to the other supply. The internal nodes in the n-transistor network could also be near Vdd. For instance, of $d\uparrow$ fires in Figure 3.22 and a is **true**, then $V(j)$ could become as high as $Vdd - v_{tn}$. To consider this type of charge-sharing, we must examine states when $\neg x \wedge \neg b_1 \wedge \neg b_2$ holds. In those states, we can compute the same ratio as before. However, the internal nodes will be atmost $Vdd - v_{tn}$, while the output is at GND. The final output voltage would be

$$V'(x) = \frac{(Vdd - v_{tn})C(S)}{C(x) + C(S)} = \frac{(Vdd - v_{tn})}{1 + R(S)}$$

making $V'(x) - GND = \Delta V = (Vdd - v_{tn})/(1 + R(S))$. To check this type of charge sharing, we must examine:

- The possible states that the circuit can be in. This is determined by examining the handshaking expansion.
- The states that the circuit can be in where $\neg x \wedge \neg b_1 \wedge \neg b_2$ holds. These are potential charge-sharing states.
- For each charge-sharing state, we must determine the ratio $R(S)$. If the resulting $|\Delta V|$ is larger than a threshold determined by the fanout of signal x, then there is a charge sharing problem.

Precharging internal nodes. Once we have identified problematic states in the handshaking expansion, we can attempt to reduce the charge sharing problem by setting the charge on the internal nodes of the transistor implementation. We only consider one type of charge sharing (ordinary); the other case is symmetric.

If the internal node voltage is at or near Vdd, then this internal node combats the charge sharing problem. In the formula for external node voltage, the capacitance of node i is added to the numerator. If we use $PC(S)$ to denote the sum of the exposed precharged capacitance to the output, then the final output voltage changes to:

$$V'(x) = \frac{Vdd \times (C(x) + PC(S))}{C(x) + C(S)}$$

This may suffice to make $|\Delta V|$ low enough for the circuit to operate correctly. If i is precharged to Vdd through n-transistors, then the formula must be adjusted to account for the fact that the voltage on the internal nodes will not be Vdd, but $Vdd - v_{tn}$.

Given an internal node i, what circuit can we use to precharge i to Vdd? Let $pchg(i)$ be the condition when we precharge i, i.e., we have a "production rule" $pchg(i) \mapsto i\uparrow$. We must ensure that introducing this circuit modification does not affect the correct operation of the original circuit. In particular:

1. **No shorts.** There should be no stable shorts; namely, $supply(i) \wedge pchg(i) \equiv \textbf{false}$, otherwise we could connect the two power supplies through the precharge circuit.
2. **No spurious output.** The precharge should not change the output x. Namely, $\neg b_1 \wedge \neg b_2 \wedge pchg(i) \wedge guard(i) \Rightarrow x$.

Note that both those conditions are trivially satisfied by $pchg(i) = \textbf{false}$. We therefore need a progress condition, which can be written as follows:

3. **Precharge opportunity.** There must be a state in the handshaking expansion where $pchg(i) \wedge \neg x \wedge \neg b_1 \wedge \neg b_2$ holds *before* the state where charge sharing becomes a problem.

If the circuit can enter a state where $pchg(i)$ becomes **false** and can stay there for an arbitrary amount of time before the problematic charge sharing scenario, i may need to be staticized to keep it precharged.

Buffering by the opposite network. The analysis above is conservative in the way it estimates charge sharing. In particular, when considering charge sharing in the n-transistor network, we may have capacitance that is exposed to the output in the p-transistor network that we may aid (or harm) us. Therefore, we should generalize the internal nodes being examined to consider *all* internal nodes—not just those in the n-transistor network but also those in the p-transistor network.

Tracking internal nodes. Finally, we have been very conservative in our charge sharing calculation by assuming that all the internal nodes in the transistor stack have the worst possible value. Instead, we could determine more information about the values of the internal nodes in the transistor network as the circuit executes the handshaking expansion. Once we know this, we can take approximate node voltages into account when performing a charge sharing calculation.

XXX: Lots of examples from the MiniMIPS. Full buffering. PCHB.

3.10.6. Leakage

XXX: Staticizer sizing.

3.11. Standard Compilations

XXX: FIXME

Passive to Lazy active transformation.

3.11.1. Sequence Of Unreshuffled Active Control Actions

$*[\ A; B; C\ ]$

$*[\ ao\uparrow; [ai]; ao\downarrow; [\neg ai]; bo\uparrow; [bi]; bo\downarrow; [\neg bi]; co\uparrow; [ci]; co\downarrow; [\neg ci]\ ]$

$*[\ ao\uparrow; x2\downarrow; x0\uparrow; [ai]; ao\downarrow; [\neg ai]; bo\uparrow; x0\downarrow; x1\uparrow; [bi]; bo\downarrow; [\neg bi];$
$\quad co\uparrow; x1\downarrow; x2\uparrow; [ci]; co\downarrow; [\neg ci]\ ]$

$$\begin{array}{rclcrcl}
x2 \wedge \neg co \wedge \neg ci & \mapsto & ao\uparrow & \qquad & \neg x0 \wedge bo & \mapsto & x1\uparrow \\
x0 \wedge ai & \mapsto & ao\downarrow & & co & \mapsto & x1\downarrow \\
\\
ao \wedge \neg x2 & \mapsto & x0\uparrow & & \neg bo \wedge x1 \wedge \neg bi & \mapsto & co\uparrow \\
bo & \mapsto & x0\downarrow & & x2 \wedge ci & \mapsto & co\downarrow \\
\\
\neg ao \wedge x0 \wedge \neg ai & \mapsto & bo\uparrow & & \neg x1 \wedge co & \mapsto & x2\uparrow \\
x1 \wedge bi & \mapsto & bo\downarrow & & ao & \mapsto & x2\downarrow
\end{array}$$

Blah.

Another version

$$*[\ ao{\uparrow}; x2{\downarrow}; [ai]; x0{\uparrow}; ao{\downarrow}; [\neg ai]; bo{\uparrow}; x0{\downarrow}; [bi]; x1{\uparrow}; bo{\downarrow}; [\neg bi]; co{\uparrow}; x1{\downarrow};$$
$$[ci]; x2{\uparrow}; co{\downarrow}; [\neg ci]\ ]$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
x2 \wedge \neg ci & \mapsto & ao{\uparrow} & \neg x0 \wedge bi & \mapsto & x1{\uparrow} \\
x0 & \mapsto & ao{\downarrow} & co & \mapsto & x1{\downarrow} \\
\\
\neg x2 \wedge ai & \mapsto & x0{\uparrow} & \neg bi \wedge x1 & \mapsto & co{\uparrow} \\
bo & \mapsto & x0{\downarrow} & x2 & \mapsto & co{\downarrow} \\
\\
\neg ai \wedge x0 & \mapsto & bo{\uparrow} & \neg x1 \wedge ci & \mapsto & x2{\uparrow} \\
x1 & \mapsto & bo{\downarrow} & ao & \mapsto & x2{\downarrow}
\end{array}$$

3.11.2. Sequence Of Two-Phase Active Control Actions

$$*[\ ao{\uparrow}; [ai]; bo{\uparrow}; [bi]; co{\uparrow}; [ci]; ao{\downarrow}; [\neg ai]; bo{\downarrow}; [\neg bi]; co{\downarrow}; [\neg ci]\ ]$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\neg ci & \mapsto & ao{\uparrow} & bi & \mapsto & co{\uparrow} \\
ci & \mapsto & ao{\downarrow} & \neg bi & \mapsto & co{\downarrow} \\
\\
ai & \mapsto & bo{\uparrow} & & & \\
\neg ai & \mapsto & bo{\downarrow} & & &
\end{array}$$

3.11.3. Sequence Of Unreshuffled Lazy-Active Control Actions

$$*[\ A; B; C; D\ ]$$

$$*[\ [\neg ai]; ao{\uparrow}; [ai]; ao{\downarrow}; [\neg bi]; bo{\uparrow}; [bi]; bo{\downarrow}; [\neg ci]; co{\uparrow}; [ci]; co{\downarrow}\ ]$$

$$*[\ [\neg ai]; ao{\uparrow}; x2{\downarrow}; x0{\uparrow}; [ai]; ao{\downarrow}; [\neg bi]; bo{\uparrow}; x0{\downarrow}; x1{\uparrow}; [bi]; bo{\downarrow};$$
$$[\neg ci]; co{\uparrow}; x1{\downarrow}; x2{\uparrow}; [ci]; co{\downarrow}\ ]$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\neg ai \wedge x2 \wedge \neg co & \mapsto & ao{\uparrow} & \neg ao \wedge x0 \wedge \neg bi & \mapsto & bo{\uparrow} \\
ai \wedge x0 & \mapsto & ao{\downarrow} & bi \wedge x1 & \mapsto & bo{\downarrow} \\
\\
\neg x1 \wedge co & \mapsto & x2{\uparrow} & \neg x0 \wedge bo & \mapsto & x1{\uparrow} \\
ao & \mapsto & x2{\downarrow} & co & \mapsto & x1{\downarrow} \\
\\
ao \wedge \neg x2 & \mapsto & x0{\uparrow} & \neg bo \wedge x1 \wedge \neg ci & \mapsto & co{\uparrow} \\
bo & \mapsto & x0{\downarrow} & x2 \wedge ci & \mapsto & co{\downarrow}
\end{array}$$

Blah.

References

The proposal of using a single-wire protocol for synchronization was made by van Berkel.[1] The section on the analysis of handshaking expansions is taken from Manohar.[19]

Chapter 4.

Decomposition Techniques

The technique introduced for synthesizing handshaking expansions into production rules can be applied to arbitrary handshaking expansions. The two cases of straight-line handshaking expansions and selections were discussed in detail. In this chapter we will show how one can systematically transform any CHP program into one of these two forms. The techniques we will discuss also permit the introduction of concurrency, improving the overall performance of the computation.

4.1. Process Decomposition

Process decomposition is a standard technique for breaking up a CHP program into multiple parts. Given a process P containing a program part S, we can replace P with the concurrent composition of two processes: P with all occurrences of S replaced with a communication action on a new synchronization channel C, and the process $\texttt{*[[}\overline{C} \to S; C\texttt{]]}$.

Example. We can apply process decomposition to replace $\texttt{*[}x := 0; y := x + 1; x := y + 1\texttt{]}$ with:

$$\texttt{*[}\ x := 0;\ \ C;\ \ x := y + 1\ \texttt{]}\ \ \|\ \ \texttt{*[[}\overline{C} \longrightarrow y := x + 1; C\texttt{]]}$$

∎

The example above illustrates that the use of process decomposition can introduce shared variables. Note, however, that the variables are accessed in a mutually exclusive manner since both processes do not execute any part of the original process concurrently. However, this can complicate production rule generation and, in general, we will avoid sharing variables if possible.

The structure $\texttt{*[[}\overline{C} \to S; C\texttt{]]}$ is used frequently, and we denote it using the *call operator* (C/S), and we say that channel C *calls* or activates S. There are different handshaking expansions possible for this CHP program, and we will examine some of these later.

The statement S can be of different forms. In the simplest case, S would be an assignment statement, a communication action, or a sequence of elementary actions. When S is an assignment statement of the form $x := E$, where x is a Boolean-valued variable, the call $(C/x := E)$ is:

```
    *[[$\overline{C} \longrightarrow x := E; C$]]
▷
    *[[$\overline{C} \wedge E \longrightarrow x\uparrow; C$
      ▯$\overline{C} \wedge \neg E \longrightarrow x\downarrow; C$
     ]]
```

When S is a selection statement of the form [$B_1 \rightarrow S_1$▯$B_2 \rightarrow S_2$], the call (C/S) can be written:

```
    *[[$\overline{C} \longrightarrow$ [$B_1 \longrightarrow S_1$ ▯ $B_2 \longrightarrow S_2$]$; C$]]
▷
    *[[$\overline{C} \wedge B_1 \longrightarrow S_1; C$
      ▯$\overline{C} \wedge B_2 \longrightarrow S_2; C$
     ]]
```

The case of repetition is complicated because we would like to replace the entire repetition statement by a single call. When S is the statement *[$B_1 \rightarrow S_1$▯$B_2 \rightarrow S_2$], the call (C/S) is given by:

```
    *[[$\overline{C} \longrightarrow$ *[$B_1 \longrightarrow S_1$ ▯ $B_2 \longrightarrow S_2$]$; C$]]
▷
    *[[$\overline{C} \wedge B_1 \longrightarrow S_1$
      ▯$\overline{C} \wedge B_2 \longrightarrow S_2$
      ▯$\overline{C} \wedge \neg B_1 \wedge \neg B_2 \longrightarrow C$
     ]]
```

Since the probe $\overline{C}$ is stable, once it becomes **true**, it stays **true** until the execution of communication action C. The action C only executes when both guards B_1 and B_2 are **false**, thereby providing an implementation of the repetition construct.

4.1.1. Avoiding Shared Variables

Consider the CHP program

$P \;\equiv\;$ *[$S_1; x := x + 1; S_2; x := 0; S_3; y := x$]

We can apply process decomposition to break this up into four processes, as shown below.

$$P \;\equiv\; *[\; S_1; I; S_2; J; S_3; K \;] \;\|\; (I/x := x+1) \;\|\; (J/x := 0) \;\|\; (K/y := x)$$

However, this transformation introduces shared variables. Instead, we can combine the three processes obtained by process decomposition into a single process, to obtain the following:

```
$P \;\equiv\; *[\; S_1; I; S_2; J; S_3; K \;]$
    $\| \;\; *[[\overline{I} \longrightarrow x := x+1;\;\; I$
          $[] \overline{J} \longrightarrow x := 0;\;\; J$
          $[] \overline{K} \longrightarrow y := x; K$
          $]]$
```

Assuming that statements S_1, S_2, and S_3 do not use variable x, the resulting system no longer shares x. The decomposition still uses y as part of the assignment $y := x$. We can eliminate this dependence on y by using the following decomposition:

```
$P \;\equiv\; *[\; S_1; I; S_2; J; S_3; K?y \;]$
    $\| \;\; *[[\overline{I} \longrightarrow x := x+1;\;\; I$
          $[] \overline{J} \longrightarrow x := 0;\;\; J$
          $[] \overline{K} \longrightarrow K!x$
          $]]$
```

The channel K now carries data, and communicates the value of x the original process. Action $K?y$ when executed concurrently with $K!x$ implements the distributed assignment $y := x$.

Example. The following process describes a solution to the distributed mutual exclusion problem.

```
$S \;\equiv\; *[[\overline{U} \longrightarrow [b \longrightarrow \textbf{skip} [] \neg b \longrightarrow R]; U; U; b\uparrow$
          $| \overline{L} \longrightarrow [b \longrightarrow \textbf{skip} [] \neg b \longrightarrow R]; L; b\downarrow$
          $]]$
```

Applying process decomposition, we can transform this into the concurrent composition of two processes, as shown below.

```
$S \;\equiv\; *[[\overline{U} \longrightarrow P; U; U; Bu$
          $| \overline{L} \longrightarrow P; L; Bd$
          $]]$
    $\| \;\; *[[\overline{P} \wedge b \longrightarrow P$
          $[] \overline{P} \wedge \neg b \longrightarrow R; P$
          $[] \overline{Bu} \longrightarrow b\uparrow; Bu$
          $[] \overline{Bd} \longrightarrow b\downarrow; Bd$
          $]]$
```

The transformation ensures that b is not shared. ∎

In the example above, the program fragment "[$b \rightarrow$ **skip**▯$\neg b \rightarrow R$]" that occurs twice in the original program is controlled by a single channel P. This transformation permits the circuit implementation of the statement [$b \rightarrow$ **skip**▯$\neg b \rightarrow R$] to be shared by the two call sites.

A standard way to avoid introducing shared variables is to use a register process of the form:

```
*[[$\overline{R_1} \longrightarrow R_1!x$ ▯... ▯$\overline{R_n} \longrightarrow R_n!x$
  ▯$\overline{W_1} \longrightarrow W_1?x$ ▯... ▯$\overline{W_m} \longrightarrow W_m?x$
 ]]
```

This process has n read channels and m write channels. Every read of the variable is implemented by a communication on a R_i channel, and every write is implemented as a communication on a W_j channel. x is no longer shared, and each process that needs to read or write x uses one of the read or write ports.

4.1.2. Overlapping Computation

Consider the program shown below.

```
$P \equiv P_1 \parallel P_2$
$P_1 \equiv$ *[ $S_1; I; S_2; J; S_3; K?y$ ]
$P_2 \equiv$ *[[$\overline{I} \longrightarrow x := x + 1;\ I$
             ▯$\overline{J} \longrightarrow x := 0;\ J$
             ▯$\overline{K} \longrightarrow K!x$
           ]]
```

Furthermore, assume that this program does not have any shared variables. In particular, assume all operations on x are contained in process P_2. Since x is local to P_2, we can reshuffle it with the other actions in P_2 without affecting correctness. Therefore, we can replace P_2 with:

```
$P_2 \equiv$ *[[$\overline{I} \longrightarrow I;\ x := x + 1$
             ▯$\overline{J} \longrightarrow J;\ x := 0$
             ▯$\overline{K} \longrightarrow K!x$
           ]]
```

This new process completes the communication on channels I and J before changing x, thereby permitting P_1 to continue execution while P_2 is updating x. The operation $x := x + 1$ can now overlap with the execution of S_2 in process P_1. This simple transformation provides a way to overlap the execution of different actions in a program.

In general, an overlapping implementation of the call (C/S), written $(C//S)$ corresponds to the process

```
*[[$\overline{C} \longrightarrow C; S$]]
```

When S is a selection statement of the form [$B_1 \rightarrow S_1$▯$B_2 \rightarrow S_2$], the overlapping call $(C//S)$ can be written as:

```
    *[[$\overline{C} \longrightarrow C;$ [$B_1 \longrightarrow S_1$ ▯ $B_2 \longrightarrow S_2$]]]
▷
    *[[$\overline{C} \wedge B_1 \longrightarrow C; S_1$
      ▯$\overline{C} \wedge B_2 \longrightarrow C; S_2$
     ]]
```

The case of repetition is complicated. When S is the statement $*[B_1 \rightarrow S_1 ▯ B_2 \rightarrow S_2]$, the overlapping call $(C//S)$ is given by:

```
    *[[$\overline{C} \longrightarrow C;$ *[$B_1 \longrightarrow S_1$ ▯ $B_2 \longrightarrow S_2$]]]
▷
    *[[$\neg x \wedge \overline{C} \longrightarrow C; x\uparrow$
      ▯$x \wedge B_1 \longrightarrow S_1$
      ▯$x \wedge B_2 \longrightarrow S_2$
      ▯$x \wedge \neg B_1 \wedge \neg B_2 \longrightarrow x\downarrow$
     ]]
```

where x is a fresh variable. Since the communication on C completes before the repetition terminates, we have to introduce an additional variable that indicates the a communication on C had occurred.

4.1.3. Pipelining

Process decomposition, combined with the transformations shown above provides a systematic way to overlap the execution of actions that do not share variables and that operate on variables. However, processes also contain communication actions, and determining whether these can be overlapped with other actions in the process is a non-trivial task. To illustrate this, consider the following CHP program:

```
$P \;\equiv\;$ $X!; Y! \;\|\; Q$
$Q \;\equiv\;$ [$\overline{X} \longrightarrow X?; Y?; S_1$ ▯ $\overline{Y} \longrightarrow Y?; X?; S_2$ ]
```

We apply process decomposition to "$X!$; $Y!$" to obtain two processes:

```
$P_1 \;\equiv\;$ *[[$\overline{A} \longrightarrow X!; A?$]]
$P_2 \;\equiv\;$ $A!; Y!$
```

Now suppose we attempt to overlap the actions $X!$ and $Y!$ by replacing P_1 with $*[[\overline{A} \rightarrow A?; X!]]$. The resulting system is wrong, because the deterministic selection statement in process Q has changed to a non-deterministic one since $\overline{X}$ and $\overline{Y}$ can both be true. This problem also occurs if we introduce slack on channel A.

Example. Consider a simple process of the form

```
*[ $S_1; S_2$ ]
```

where we apply process decomposition to obtain:

$$\begin{array}{l} \quad *[\ S_1; X!\ ] \\ \|\ \ *[[\overline{X} \longrightarrow S_2; X?\ ]] \end{array}$$

Adding slack on channel X permits the execution of S_1 and S_2 to overlap. If X has slack 1, then the possible execution traces are of the form $S_1; (S_1 \| S_2); (S_1 \| S_2); ...$ ∎

A pipelined implementation of the CHP program $*[L?x; R!g(f(x))]$ is shown below:

$$*[\ L?x; R'!f(x)\ ]\ \|\ *[\ R'?y; R!g(y)\ ]$$

Here, R' is a new, intermediate channel that is introduced to communicate partial computations from one process to the next. The original function $g(f(x))$ has been broken down into the computation of $f(x)$ followed by $g(y)$, where $y = f(x)$. The key difference between the original process and the pipelined implementation is that the pipelined implementation permits $\mathbf{c}L? - \mathbf{c}R!$ to be 2. This transformation is equivalent to adding slack on the channel R, as is illustrated by the following sequence of transformations:

$$\begin{array}{l} \quad *[\ L?x; R!g(f(x))\ ] \\ \triangleright\ \{\ \textit{add slack on } R\ \} \\ \quad *[\ L?x; R'!g(f(x))\ ]\ \|\ *[\ R_?y; R!y\ ] \\ \triangleright\ \{\ \textit{distribute computation}\ \} \\ \quad *[\ L?x; R'!f(x)\ ]\ \|\ *[\ R'?y; R!g(y)\ ] \end{array}$$

Therefore, when pipelining a computation we face the same set of problems as when we attempt to add slack to a communication channel. While preserving correctness is difficult in general, we can prove when such transformations are valid by using the theory of *slack elastic programs*.

Slack elastic programs are those whose correctness is preserved when the slack on channels in the system is increased (cf. Section 2.15). Slack elasticity is a property of closed systems, since we must consider the process as well as its environment in order to determine if a system is slack elastic.

When a program is deterministic and does not probe any channels, the system is slack elastic. This is because all communication actions only depend on the data values that are transferred on channels—intuitively, all communication actions are controlled by local variables in processes. In this case, changing the slack of a channel does not affect what the system computes. The notion of what happens when the slack of a communication channel is increased is formalized in Theorem 2.9. Theorem 2.9 states that increasing the slack on a channel can change the behavior of a system only if it causes the introduction of non-deterministic choices within some process in the system. (This theorem is applicable only when examining systems in which processes do not share variables.) If we can show that we cannot introduce any non-determinism in the system, we are guaranteed that the system will not change its behavior when the slack on a channel is increased.

To analyze a component of a concurrent system, we introduced the concept of *locally slack elastic* systems. Given an open system $\mathcal{S}$, and a set of channels C belonging to the system $\mathcal{S}$, we define the concept of local slack elasticity of $\mathcal{S}_C$ as follows. We examine the parallel composition of $\mathcal{S}$ with any other system $\mathcal{S}'$ such that $\mathcal{S}\|\mathcal{S}'$ is closed. If adding slack to any channel in the closed system that isn't specified in the set C causes the system to fail only due to the introduction of non-determinism in $\mathcal{S}'$, $\mathcal{S}_C$ is said to be locally slack elastic. Once again, it should be clear that if $\mathcal{S}$ does not probe any channels, $\mathcal{S}_\emptyset$ is locally slack elastic. If we compose a collection of locally slack elastic systems to construct a closed system, the resulting system is locally slack elastic as well.

We state the following result which provides sufficient conditions for local slack elasticity:

Theorem 4.1. (*determinism*)

Let $\mathcal{S}$ be an open system in which the guards of selection statements are syntactically mutually exclusive and there are no probed channels. Then $\mathcal{S}_\emptyset$ is locally slack elastic.

In the following section, we see how to decompose a CHP program into smaller parts without the introduction of probes.

4.1.4. Process Decomposition Without Probes

In most examples,probed communication actions were used to select a particular action from a process. Instead of using probes, we can instead send control information to the process informing it what to do next. For instance, the process

```
$P \;\equiv\; *[\ S_1; x := x+1; S_2; x := 0; S_3; y := x\ ]$
```

can be decomposed into the following two processes:

```
$P \;\equiv\; *[\ S_1;\ C!\mathsf{inc}; S_2;\ C!\mathsf{reset}; S_3;\ C!\mathsf{read}; K?y\ ]$
  $\|$  $*[C?c;$
          $[c = \mathsf{inc} \longrightarrow x := x+1$
          $[] c = \mathsf{reset} \longrightarrow x := 0$
          $[] c = \mathsf{read} \longrightarrow K!x$
         ]]
```

This transformation is similar to process decomposition, but instead of using probes we use control information to select various actions.

Example. We used a recursive construction to design various stacks that consisted of a linear array of stack elements. The CHP for a lazy stack element was given by:

```
$b\uparrow;$
$*[[b \wedge \overline{push} \longrightarrow push?x; b\downarrow$
  $[] b \wedge \overline{pop} \longrightarrow get?x; pop!x$
```

```
    $[]\neg b \wedge \overline{push} \longrightarrow put!x; push?x$
    $[]\neg b \wedge \overline{pop} \longrightarrow pop!x; b\uparrow$
  ]]
```

Instead of using probed channels, we can read a control value indicating what action should be executed next. The following CHP is a rewriting of the stack element. The input control channel is L, and R is used to communicate with the stack element to the right.

```
  $b\uparrow$;
*[$L?c$; [$b \wedge c = \mathsf{push} \longrightarrow push?x; b\downarrow$
        $[] b \wedge c = \mathsf{pop} \longrightarrow R!\mathsf{pop}; get?x; pop!x$
        $[]\neg b \wedge c = \mathsf{push} \longrightarrow R!\mathsf{push}; put!x; push?x$
        $[]\neg b \wedge c = \mathsf{pop} \longrightarrow pop!x; b\uparrow$
        ]
 ]
```

▮

Using process decomposition without the introduction of probes makes it easy to introduce pipelining in a computation, since we can use results like Theorem 2.9 and 4.1.

4.2. Control Data Decomposition

The handshaking expansions and production rules we've been looking at involved processes that did not use variables that had many bits of data. Typically, the communication channels were dataless, or carried a single bit of data. In this section, we examine a systematic way to compile processes that manipulate variables that have a large number of bits.

The technique we study is called *control/data decomposition*, or *control/data separation* because it involves decomposing a process into two parts: the control part consisting of dataless communication actions, and the data part consisting of processes that manipulate multi-bit variables and whose actions are determined by the control part. This transformation is illustrated by the following simple example. Consider the process

```
*[ $L?x$;  $R!x$ ]
```

where x is a 32-bit variable. The *control part* of this process is obtained by eliminating all data communications from the process by replacing them with dataless communication actions on new channels. The resulting control process is:

```
*[ $L'$;  $R'$ ]
```

The *data part* is obtained by introducing new processes that perform the data communication

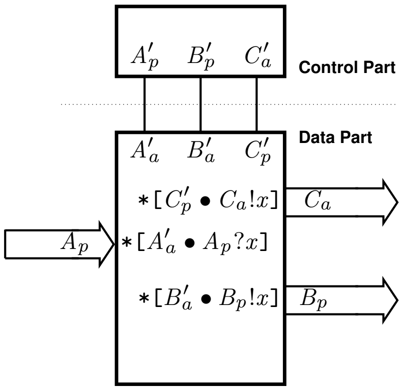


Figure 4.1. Process with three channels A, B, and C, after applying control/data decomposition. Subscripts indicate whether the port is active (a) or passive (p).

actions, controlled by the newly introduced control channels. In the example above, the data part is given by:

$$*[\ L' \bullet L?x\ ] \ \| \ *[\ R' \bullet R!x\ ]$$

where the variable x is *shared* between the two data processes. In general, control data decomposition splits a particular CHP program S is decomposed into two parts:

$$S \ \triangleright \ C_S \ \| \ D_S$$

C_S, the control part, is obtained from S by replacing all communication actions that exchange data with bare synchronization channels. D_S, the data part, consists of CHP processes of the form $*[X \bullet X']$, where X is a communication action from the original process S, and X' is the dataless communication channel that replaced it. Probes on communication channels from process S are replaced by probes on their respective dataless communication channels. For instance, the control part of

$$*[[\overline{L} \longrightarrow R!x; L?x]]$$

would be:

$$*[[\overline{L'} \longrightarrow R'; L']]$$

since L' was the dataless communication channel that replaced $L?x$.

When writing handshaking expansions for a particular process, we have to pick handshake protocols for communication actions. In particular, we have to determine whether a particular communication action is active or passive. Consider the example above where we replaced action $R!x$ with a dataless communication on R'. If R used a passive protocol, we select a passive protocol for R'; if R used an active protocol, we select an active protocol for R'. This applies to all newly introduced communication actions. In the example above, L is probed in the original CHP, and therefore L used a passive protocol. Our rule automatically ensures that L' uses a passive protocol, since L' was the channel that replaced $L?x$. Therefore, L' can be probed and we can compile the control part shown above without difficulty.

Example. Figure 4.1 shows the result of applying control data decomposition to a process that had three communication channels A, B, and C. A and B were passive ports (indicated by subscript p), and that C was an active port (indicated by subscript a). ∎

Suppose we replace a passive action $A?x$ with communication action A'. Since A was a passive port, A' in the control part must be passive. This implies that the A' port in the data part is active, since it is connected to a passive A' from the control part. Therefore, the data part $*[A' \bullet A?x]$ where A is passive *must* use an active protocol on A'. Therefore, a process that is in the data part is *always* of the form $*[C \bullet D]$ where C and D are two communication actions, and where one of them is active and the other is passive. This observation is illustrated by Figure 4.1. Therefore, there are four possible data part processes:

1. $*[\ C_a \bullet D_p?x\ ]$
2. $*[\ C_p \bullet D_a?x\ ]$
3. $*[\ C_a \bullet D_p!x\ ]$
4. $*[\ C_p \bullet D_a!x\ ]$

where the subscripts a and p indicate whether the port is active or passive.

4.2.1. Input On A Passive Port

If we are compiling a process with a passive input port, the data part is given by:

$*[\ C \bullet D?x\ ]$

where D is passive and C is active. We assume that the data on D is encoded using a dual rail encoding, and that D carries one bit of information.

The handshake protocol on channel C is given by:

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

where C is implemented using a four-phase active handshake on pair (co, ci). The handshake protocol for channel D is given by:

$[dti \vee dfi];\ do\uparrow;\ [\neg dti \wedge \neg dfi];\ do\downarrow$

where D is implemented using a four-phase passive handshake on variables (dti, dfi, do). In addition, we need to introduce the statement

$[dti \longrightarrow x\uparrow [\!] dfi \longrightarrow x\downarrow]$

that actually sets the variable x.

We first interleave the handshakes on channel C and D to implement the bullet between C and D. The interleaving we use is:

$$*[[\mathit{dti} \vee \mathit{dfi}]; \mathit{co}\uparrow; [\mathit{ci}]; \mathit{do}\uparrow; [\neg \mathit{dti} \wedge \neg \mathit{dfi}]; \mathit{co}\downarrow; [\neg \mathit{ci}]; \mathit{do}\downarrow]$$

In general, this is the only interleaving that results in a valid implementation of control data decomposition. First, observe that the variable *co* is used as the probe of channel D in the control part that would be connected to this particular data process. Therefore, *co*↑ cannot be executed until the probe of D is true; similarly, *co*↓ cannot be executed until the probe of D is false. Second, actions on *do* in original process are replaced by actions on *ci* in the control part. Therefore, we cannot raise *do* until *ci* is true, and we cannot lower *do* until *ci* is false. These conditions result in the reshuffling shown above.

To add the actions that set x, observe that we cannot set x until the control has notified the data part that this operation can proceed. The first point where we know that the control has reached the operation on data channel D is after *ci* becomes true. The final handshake protocol for $*[C \bullet D?x]$ is given by:

$$*[[\mathit{dti} \vee \mathit{dfi}]; \mathit{co}\uparrow; [\mathit{ci} \wedge \mathit{dti} \longrightarrow x\uparrow [\!] \mathit{ci} \wedge \mathit{dfi} \longrightarrow x\downarrow]; \mathit{do}\uparrow; [\neg \mathit{dti} \wedge \neg \mathit{dfi}]; \mathit{co}\downarrow; [\neg \mathit{ci}]; \mathit{do}\downarrow]$$

The production rules for this handshake are:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathit{dti} \vee \mathit{dfi} & \mapsto & \mathit{co}\uparrow & \mathit{ci} \wedge (\mathit{dti} \wedge x \vee \mathit{dfi} \wedge \neg x) & \mapsto & \mathit{do}\uparrow \\
\neg \mathit{dti} \wedge \neg \mathit{dfi} & \mapsto & \mathit{co}\downarrow & \neg \mathit{ci} & \mapsto & \mathit{do}\downarrow
\end{array}$$

$$\begin{array}{rcl}
\mathit{ci} \wedge \mathit{dti} & \mapsto & x\uparrow \\
\mathit{ci} \wedge \mathit{dfi} & \mapsto & x\downarrow
\end{array}$$

Instead of directly writing down production rules, we can decompose this into standard operators as follows. Notice that we can write the handshake as the parallel composition of two handshakes without changing the behavior of the handshaking expansion, as shown below.

$$\begin{array}{ll}
 & *[[\mathit{dti} \vee \mathit{dfi}]; \mathit{co}\uparrow; [\neg \mathit{dti} \wedge \neg \mathit{dfi}]; \mathit{co}\downarrow] \\
\| & *[[\mathit{ci} \wedge \mathit{dti} \longrightarrow x\uparrow; \mathit{do}\uparrow; [\neg \mathit{ci}]; \mathit{do}\downarrow \\
 & \;\; [\!] \mathit{ci} \wedge \mathit{dfi} \longrightarrow x\downarrow; \mathit{do}\uparrow; [\neg \mathit{ci}]; \mathit{do}\downarrow \\
 & \;\;]]
\end{array}$$

The second part of the handshake resembles the write part of the register process we compiled in the previous chapter. Indeed, we can further rewrite the handshaking expansion as the parallel composition of four parts, as shown below:

$$\begin{array}{ll}
 & *[[\mathit{dti} \vee \mathit{dfi}]; \mathit{co}\uparrow; [\neg \mathit{dti} \wedge \neg \mathit{dfi}]; \mathit{co}\downarrow] \\
\| & *[[\mathit{wti} \longrightarrow x\uparrow; \mathit{do}\uparrow; [\neg \mathit{wti}]; \mathit{do}\downarrow \\
 & \;\; [\!] \mathit{wfi} \longrightarrow x\downarrow; \mathit{do}\uparrow; [\neg \mathit{wfi}]; \mathit{do}\downarrow \\
 & \;\;]] \\
\| & *[[\mathit{ci} \wedge \mathit{dti}]; \mathit{wti}\uparrow; [\neg \mathit{ci}]; \mathit{wti}\downarrow] \\
\| & *[[\mathit{ci} \wedge \mathit{dfi}]; \mathit{wfi}\uparrow; [\neg \mathit{ci}]; \mathit{wfi}\downarrow]
\end{array}$$

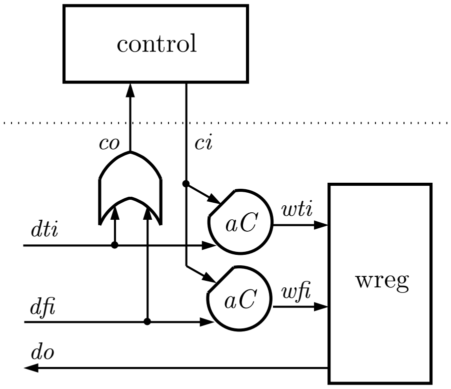


Figure 4.2. Compilation of an input on a passive port. "wreg" is the circuit corresponding to the write part of a register.

The first part can be compiled into production rules:

$$\begin{aligned} dti \lor dfi &\mapsto co\uparrow \\ \neg dti \land \neg dfi &\mapsto co\downarrow \end{aligned}$$

The last two parts can be compiled into:

$$\begin{aligned} ci \land dti &\mapsto wti\uparrow \qquad & ci \land dfi &\mapsto wfi\uparrow \\ \neg ci &\mapsto wti\downarrow \qquad & \neg ci &\mapsto wfi\downarrow \end{aligned}$$

Finally, the second part is the write part of a register, and the compilation can be obtained from Section 3.7. Figure 4.2 shows the circuit corresponding to reading an input on a passive port.

4.2.2. Input On An Active Port

If we are compiling a process with an active input port, the data part is given by:

$*[\ C \bullet D?x\ ]$

where D is active and C is passive. The handshake protocol on channel C is given by:

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

where C is implemented using a four-phase passive handshake on pair (co, ci). The handshake protocol for channel D is given by:

$do\uparrow; [dti \lor dfi]; do\downarrow; [\neg dti \land \neg dfi]$

where D is implemented using a four-phase active handshake on variables (dti, dfi, do). In addition, we need to introduce the statement

$[dti \longrightarrow x\uparrow [\!] dfi \longrightarrow x\downarrow]$

that actually sets the variable x.

We interleave these three parts to obtain the handshaking expansion for $*[C \bullet D?x]$. By arguments similar to those used for the case of a passive input, we must use the following interleaving:

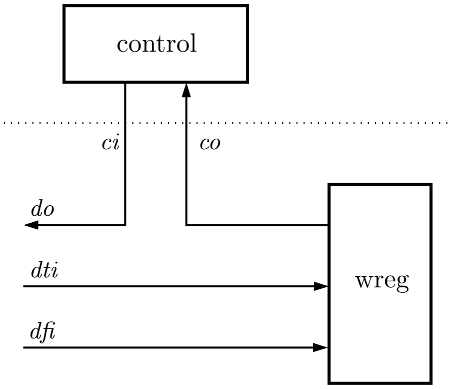


Figure 4.3. Compilation of an input on an active port. "wreg" is the circuit corresponding to the write part of a register.

$$*[[ci]; do\uparrow; [dti \longrightarrow x\uparrow [] dfi \longrightarrow x\downarrow]; co\uparrow; [\neg ci]; do\downarrow; [\neg dti \wedge \neg dfi]; co\downarrow]$$

Instead of directly writing down production rules, we can decompose this process into two parts whose concurrent composition is equivalent to the original handshake, as shown below.

```
    *[[$ci$]; $do\uparrow$; [$\neg ci$]; $do\downarrow$]
||  *[[$dti \longrightarrow x\uparrow$; $co\uparrow$; [$\neg dti$]; $co\downarrow$
     [] $dfi \longrightarrow x\downarrow$; $co\uparrow$; [$\neg dfi$]; $co\downarrow$
     ]]
```

The production rules for the first part are:

$$ci \mapsto do\uparrow$$
$$\neg ci \mapsto do\downarrow$$

and the second part is simply the write part of a register. The circuit is shown in Figure 4.3.

4.2.3. Output On A Passive Port

If we are compiling a process with a passive output port, the data part is given by:

$$*[\ C \bullet D!x\ ]$$

where D is passive and C is active. The handshake protocol on channel C is given by:

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

where C is implemented using a four-phase active handshake on pair (co, ci). The handshake protocol for channel D is given by:

$$[di]; [x \longrightarrow dto\uparrow [] \neg x \longrightarrow dfo\uparrow]; [\neg di]; dto\downarrow, dfo\downarrow$$

where D is implemented using a four-phase passive handshake on variables (di, dto, dfo), and x is the variable being transmitted on channel D.

To determine what interleavings are valid implementations of $*[C \bullet D!x]$, we make the following observations. First, observe that the variable co is used as the probe of channel D in the control part

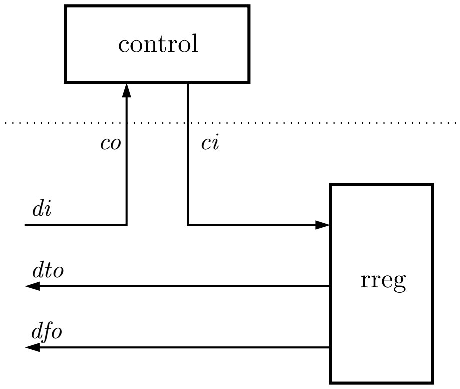


Figure 4.4. Compilation of an output on a passive port. "rreg" is the circuit corresponding to the read part of a register.

that would be connected to this particular data process. Therefore, $co\uparrow$ cannot be executed until di is true; similarly, $co\downarrow$ cannot be executed until di is false. Second, data cannot be transmitted along D until the control process permits the transmission to occur—i.e., until ci becomes true. These conditions directly lead to the following reshuffling:

$$*[[di];\, co\uparrow;\, [ci];\, [x \longrightarrow dto\uparrow \,\square\, \neg x \longrightarrow dfo\uparrow];\, [\neg di];\, co\downarrow;\, [\neg ci];\, dto\downarrow, dfo\downarrow]$$

We decompose this handshaking expansion into two concurrent parts, as shown below. The reader should verify that the concurrent composition of these two parts results in exactly the same set of behaviors as the original handshake.

```
    $*[[di];\, co\uparrow;\, [\neg di];\, co\downarrow]$
$\|$  $*[[ci \wedge x \longrightarrow dto\uparrow;\, [\neg ci];\, dto\downarrow$
      $\square\, ci \wedge \neg x \longrightarrow dfo\uparrow;\, [\neg ci];\, dfo\downarrow$
      $]]$
```

The production rules for the first part are given by

$$\begin{array}{rcl} di & \mapsto & co\uparrow \\ \neg di & \mapsto & co\downarrow \end{array}$$

and the second part is simply the read part of a register. The circuit is shown in Figure 4.4.

4.2.4. Output On An Active Port

If we are compiling a process with an active output port, the data part is given by:

$$*[\;\; C \bullet D!x \;\;]$$

where D is active and C is passive. The handshake protocol on channel C is given by:

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

where C is implemented using a four-phase passive handshake on pair (co, ci). The handshake protocol for channel D is given by:

$$[x \longrightarrow dto{\uparrow}\,[\!]\,\neg x \longrightarrow dfo{\uparrow}]; [di]; dto{\downarrow}, dfo{\downarrow}; [\neg di]$$

where D is implemented using a four-phase active handshake on variables (di, dto, dfo), and x is the variable being transmitted on channel D.

We interleave these two parts to obtain the handshaking expansion for $*[C \bullet D!x]$. By arguments similar to those used for the case of a passive output, we must use the following interleaving:

$$*[[ci]; [x \longrightarrow dto{\uparrow}\ \,[\!]\,\neg x \longrightarrow dfo{\uparrow}]; [di]; co{\uparrow}; [\neg ci]; dto{\downarrow}, dfo{\downarrow}; [\neg di]; co{\downarrow}]$$

We decompose this handshaking expansion into two concurrent parts, as shown below. The reader should verify that the concurrent composition of these two parts results in exactly the same set of behaviors as the original handshake.

$$\begin{array}{ll} & *[[di]; co{\uparrow}; [\neg di]; co{\downarrow}] \\ \| & *[[ci \wedge x \longrightarrow dto{\uparrow}; [\neg ci]; dto{\downarrow} \\ & \quad [\!]\, ci \wedge \neg x \longrightarrow dfo{\uparrow}; [\neg ci]; dfo{\downarrow} \\ & \quad]] \end{array}$$

Note that this is exactly the same handshake as the case of a passive output! Therefore, the circuit is the same as the one shown in Figure 4.4.

The output circuits for both passive and active ports are identical, whereas the circuit for a passive input port has more circuitry associated with it compared to the circuit for an active input port. Therefore, if we are interested in minimizing circuitry, it is preferable to have active input ports.

4.2.5. Multi-Bit Data Transmission

The compilation of the data part processes can be easily extended to situations when more than one bit of data is being sent or received. Some control signals that initiate operations will be broadcast to all the register bits. Signals from the data part that are used by the control will have to be combined into one control signal. The n-bit data part circuits look very similar to the one-bit case, with some additional circuitry.

4.2.5.1. Output Ports

The following handshaking expansion corresponds to an output on a passive port, where the data being transmitted consists of several bits.

$$*[[di]; co{\uparrow}; [ci]; (\| i :: [x_i \longrightarrow dto_i{\uparrow}\ \,[\!]\,\neg x_i \longrightarrow dfo_i{\uparrow}]); [\neg di]; co{\downarrow}; [\neg ci]; (\| i :: dto{\downarrow}, dfo{\downarrow})]$$

The only difference in the handshake is the part where we set and reset the output data rails dto and dfo; instead of setting one pair, we now have to set several pairs of output rails.

Instead of decomposing this handshake into two parts, we decompose it into $n+1$ parts, where n is the number of bits being transmitted on channel D. We obtain:

$$\begin{array}{ll}
 & *[[di];\, co\!\uparrow;\, [\neg di];\, co\!\downarrow] \\
\| & (\| \; i :: *[[ci \wedge x_i \longrightarrow dto_i\!\uparrow;\, [\neg ci];\, dto_i\!\downarrow \\
 & \qquad\quad \square\, ci \wedge \neg x_i \longrightarrow dfo_i\!\uparrow;\, [\neg ci];\, dfo_i\!\downarrow \\
 & \qquad\quad]])
\end{array}$$

The signal *ci* is shared by all the register bits. As before, the circuit for an n-bit output on an active port is identical.

4.2.5.2. Input on an Active Port

For an input on an active port, the handshaking expansion is given by:

$$*[[ci];\, do\!\uparrow;\, (\| \; i :: [dti_i \longrightarrow x_i\!\uparrow \square\, dfi_i \longrightarrow x_i\!\downarrow]);\, co\!\uparrow;\, [\neg ci];\, do\!\downarrow;\, [(\wedge i :: \neg dti \wedge \neg dfi)];\, co\!\downarrow]$$

Since we cannot permit each register process to set *co* directly as in the case of one-bit data transmission, we introduce auxiliary variables w_i, one for each bit being received. The decomposed handshaking expansion is given by:

$$\begin{array}{ll}
 & *[[ci];\, do\!\uparrow;\, [\neg ci];\, do\!\downarrow] \\
\| & (\| \; i :: *[[dti_i \longrightarrow x_i\!\uparrow;\, w_i\!\uparrow;\, [\neg dti_i];\, w_i\!\downarrow \\
 & \qquad\quad \square\, dfi_i \longrightarrow x_i\!\downarrow;\, w_i\!\uparrow;\, [\neg dfi_i];\, w_i\!\downarrow \\
 & \qquad\quad]]) \\
\| & *[[(\wedge i :: w_i)];\, co\!\uparrow;\, [(\wedge i :: \neg w_i)];\, co\!\downarrow]
\end{array}$$

The production rules for the last part of the decomposed handshaking expansion correspond to an n input C-element, where n is the number of bits being received by the input action. If n is large, the direct CMOS implementation of this C-element would be very inefficient. Note, however, that the inputs to this C-element—the w_i variables—have the property that they do not change until the output changes. Under these conditions, we can decompose the n input C-element into a tree of C-elements with fixed fanin. This structure is known as a *completion tree*, since it consists of a tree of C-elements and the output indicates that the write operation has completed.

4.2.5.3. Input on a Passive Port

For an input on a passive port, the handshaking expansion is given by:

$$\begin{array}{l}
*[[(\wedge i :: dti_i \vee dfi_i)];\, co\!\uparrow;\, (\| \; i :: [ci \wedge dti_i \longrightarrow x_i\!\uparrow \square\, ci \wedge dfi_i \longrightarrow x_i\!\downarrow]);\, do\!\uparrow; \\
\quad [(\wedge i :: \neg dti_i \wedge \neg dfi_i)];\, co\!\downarrow;\, [\neg ci];\, do\!\downarrow]
\end{array}$$

Following the transformations used earlier, we can implement the handshaking expansion as follows:

$$\begin{array}{ll}
 & *[[(\wedge i :: dti_i \vee dfi_i)];\, co\!\uparrow;\, [(\wedge i :: \neg dti \wedge \neg dfi)];\, co\!\downarrow] \\
\| & (\| \; i :: *[[wti_i \longrightarrow x_i\!\uparrow;\, do_i\!\uparrow;\, [\neg wti_i];\, do_i\!\downarrow \\
 & \qquad\quad \square\, wfi_i \longrightarrow x_i\!\downarrow;\, do_i\!\uparrow;\, [\neg wfi_i];\, do_i\!\downarrow \\
 & \qquad\quad]])
\end{array}$$

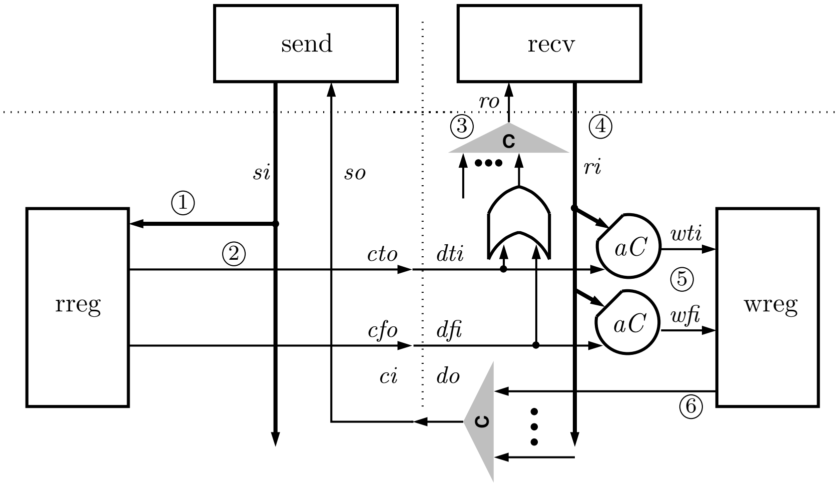


Figure 4.5. Passive input connected to an active output. The sequence of transitions that occur during data transmission is indicated by the numbers on the wires.

$$\| \quad (\| \; i :: *[[ci \wedge dti_i]; wti_i\uparrow; [\neg ci]; wti_i\downarrow])$$
$$\| \quad (\| \; i :: *[[ci \wedge dfi_i]; wfi_i\uparrow; [\neg ci]; wfi_i\downarrow])$$
$$\| \quad *[[(\wedge i :: do_i)]; do\uparrow; [(\wedge i :: \neg do_i)]; do\downarrow]$$

The first process can be transformed into an OR gate followed by a completion tree as follows:

$$(\| \; i :: *[[dti_i \vee dfi_i]; dv_i\uparrow; [\neg dti_i \wedge \neg dfi_i]; dv_i\downarrow])$$
$$\| \quad *[[(\wedge i :: dv_i)]; co\uparrow; [(\wedge i :: \neg dv_i)]; co\downarrow]$$

4.3. Putting The Pieces Together

Figure 4.5 shows the circuit one obtains by connecting a passive input port to an active output with one bit of data being transferred from sender to receiver. Figure 4.6 shows the circuit for an active input port and passive output port for a one-bit data channel. The thick lines are ones where the signal would be broadcast to n bits in the data part when n bit data is communicated from sender to receiver. The triangle with a “C” in it corresponds to the completion trees that would be required in the n-bit case. The dangling inputs (outputs) come from (go to) the corresponding inputs (outputs) that are part of the remaining bits.

Since the sender data part uses the same circuit for both active and passive protocols, it should be clear from Figure 4.5 and Figure 4.6 that using an active receiver (Figure 4.6) results in a smaller circuit than one where the receiver is passive. There is another significant advantage when using an active receiver. The handshake protocol for an active receiver only has one completion tree delay on it; the passive receiver has two completion tree delays on the handshake. This is an important

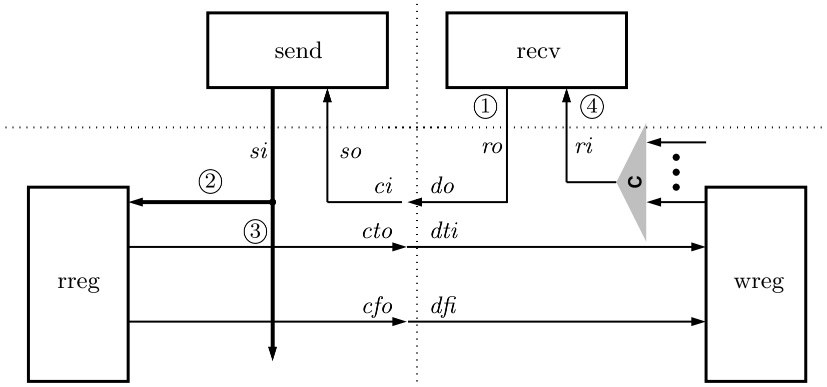


Figure 4.6. Active input connected to a passive output. The sequence of transitions that occur during data transmission is indicated by the numbers on the wires.

effect since it will limit the amount the circuit can be pipelined when designing datapath circuits that require many bits.

Finally, the send/recv boxes correspond to the control logic that remains after control/data decomposition. To provide a complete example, consider a simple FIFO $*[L?x; R!x]$ with L passive and R active. This process would be transformed into:

$$*[\ L'; R'\ ] \quad \| \quad *[\ L' \bullet L?x\ ] \quad \| \quad *[\ R' \bullet R!x\ ]$$

with the control using a passive handshake on L' and an active handshake on R'. If we implement L' with (ci, co) and R' with (di, do) in the control, the unreshuffled handshaking expansion for the control is:

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

We reshuffle $co\downarrow$ to the end of the handshake so we can re-use the control ciruits designed in the previous chapter. Note that for the sake of this example we are assuming that this reshuffling is valid (without checking it!). The reshuffled handshaking expansion is given by

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

The bubble-reshuffled production rules (from the previous chapter) for this control process are:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\neg\_x & \mapsto & co\uparrow & ci & \mapsto & \_x\downarrow \\
\_x \wedge \_di & \mapsto & co\downarrow & \neg\_di & \mapsto & \_x\uparrow \\
\\
\neg\_x \wedge \neg ci & \mapsto & do\uparrow \\
\_x & \mapsto & do\downarrow
\end{array}$$

Finally, connecting this control process to the datapath circuits just designed gives us the circuit shown in Figure 4.7. Note that the interface to the environment hasn't changed; namely, the

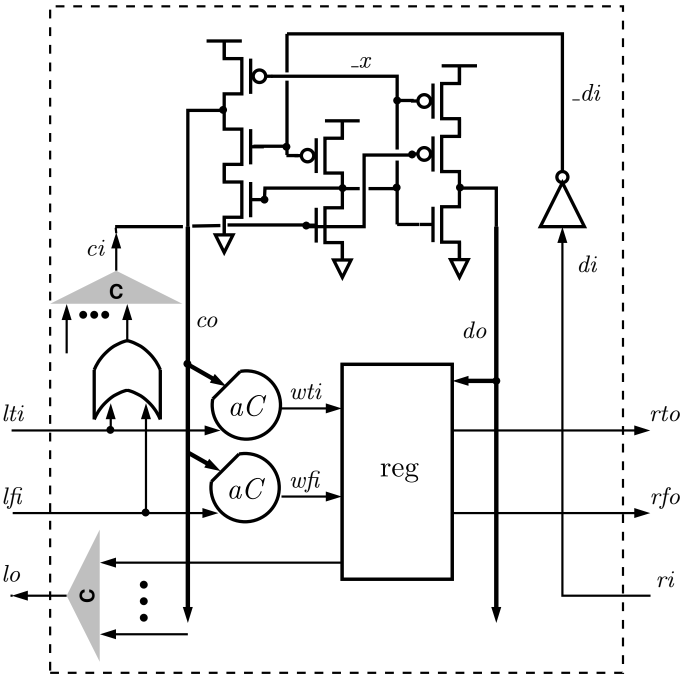


Figure 4.7. A simple FIFO process with control and data. Staticizers are omitted for clarity. The “reg” process is both the read and write part of a register.

interface still uses just the L and R channels. The control channels introduced by control/data decomposition are internal to the process.

4.4. Alternate Compilation: Passive Input Port

As we have seen, one might prefer to implement input ports using an active protocol since the resulting circuit is simpler. However, if the input port is probed, our convention dictates that the input port would be passive. In this section, we provide an alternate compilation of a process with a passive input port that avoids the double completion tree penalty.

Let D be a passive input port in process P_1, connected to the active output port C in process P_2. Furthermore, let D be controlled by channel R, and C be controlled by channel S. The data part for P_1 and P_2 that corresponds to ports C and D is given by:

$$*[\ R \bullet D?x\ ]\ \|\ *[\ S \bullet C!y\ ]$$

where D and S are passive, and R and C are active. Let’s examine the sequence of actions that occur when a communication is initiated by the active sender S. From the previous sections, we know that the compilation of these two data processes results in the following handshaking expansion:

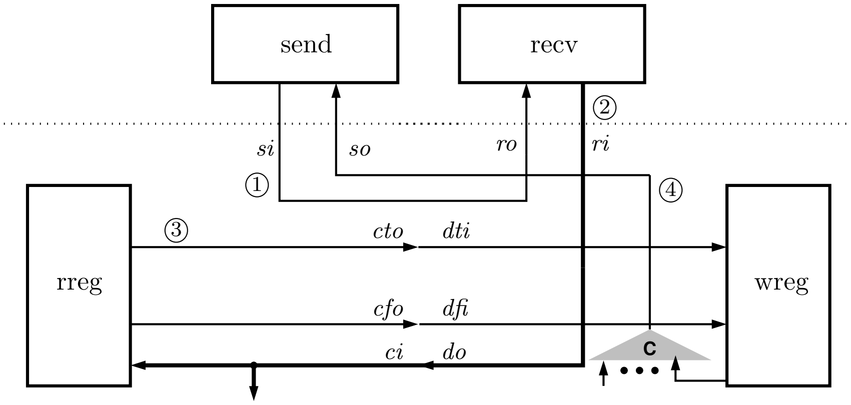


Figure 4.8. Alternate compilation of a passive input connected to an active output. The sequence of transitions that occur during data transmission is indicated by the numbers on the wires.

$$
\begin{aligned}
&*[[\mathit{si}]; [y \longrightarrow \mathit{cto}\uparrow [] \neg y \longrightarrow \mathit{cfo}\uparrow]; [\mathit{ci}]; \mathit{so}\uparrow; [\neg \mathit{si}]; \mathit{cto}\downarrow, \mathit{cfo}\downarrow; [\neg \mathit{ci}]; \mathit{so}\downarrow] \\
\| \; &*[[\mathit{dti} \vee \mathit{dfi}]; \mathit{ro}\uparrow; [\mathit{ri} \wedge \mathit{dti} \longrightarrow x\uparrow [] \mathit{ri} \wedge \mathit{dfi} \longrightarrow x\downarrow]; \mathit{do}\uparrow; [\neg \mathit{dti} \wedge \neg \mathit{dfi}]; \mathit{ro}\downarrow; [\neg \mathit{ri}]; \mathit{do}\downarrow]
\end{aligned}
$$

The standard compilation of this handshaking expansion is illustrated in Figure 4.5. The numbers on the wires indicate the sequence of operations that occur when data is transmitted from the sender to the receiver.

The sequence of operations is only required to ensure that: *ro* cannot become **true** until *si* becomes **true**; data is not transmitted until *si* becomes **true**; data is not received until *ri* becomes **true**. The first constraint ensures that the probe of R does not become **true** unless the sender has started the communication on D. The second and third constraints ensure that data transfer only occurs when permitted by the control processes. We can enforce this sequence using the following handshake:

$$
\begin{aligned}
&*[[\mathit{si}]; \mathit{ro}\uparrow; [\mathit{ci}]; [y \longrightarrow \mathit{cto}\uparrow [] \neg y \longrightarrow \mathit{cfo}\uparrow]; [\neg \mathit{si}]; \mathit{ro}\downarrow; [\neg \mathit{ci}]; \mathit{cto}\downarrow, \mathit{cfo}\downarrow] \\
\| \; &*[[\mathit{ri}]; \mathit{do}\uparrow; [\mathit{dti} \longrightarrow x\uparrow [] \mathit{dfi} \longrightarrow x\downarrow]; \mathit{so}\uparrow; [\neg \mathit{ri}]; \mathit{do}\downarrow; [\neg \mathit{dti} \wedge \neg \mathit{dfi}]; \mathit{so}\downarrow]
\end{aligned}
$$

The result is the circuit shown in Figure 4.8. The channel R can be probed in the control “recv,” and we still have the advantage of a simple circuit implementation.

Another compilation of a passive input connected to an active output is shown in Figure 4.9. Instead of sequencing the request to the “recv” process with the communication on channel D, this compilation attempts both of them in parallel. The net result is that we can eliminate the OR gate normally used to generate *ro*. However, the cost is that the asymmetric C-element in front of the write port to the register must be replaced with a full C-element since otherwise the downgoing transition on the data wires would not be acknowledged.

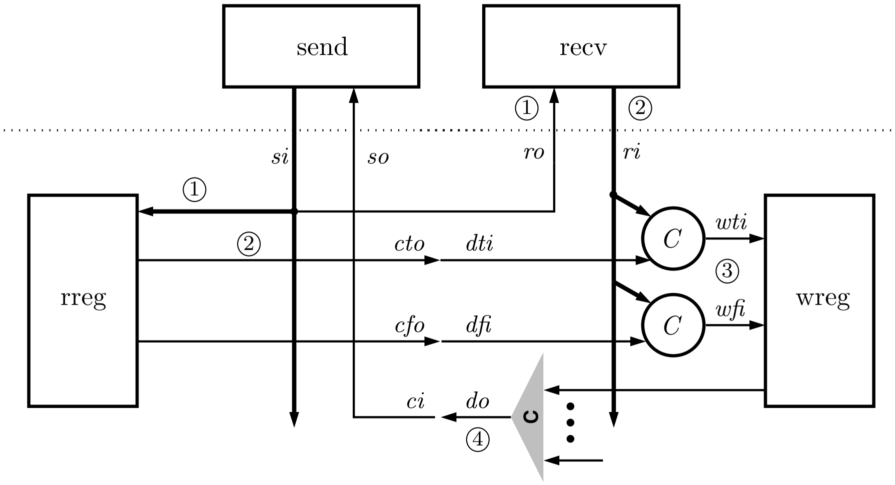


Figure 4.9. Alternate compilation of a passive input connected to an active output. The sequence of transitions that occur during data transmission is indicated by the numbers on the wires.

4.5. Shared Channels

The decompositions we examined all assumed that the variable x was shared among multiple processes, and that care had to be taken to ensure that access to x was controlled correctly. But what happens if we use control/data decomposition for a process like the one shown below?

$*[\ L?x;\ ...;\ L?y;\ ...\ ]$

The data part after control/data decomposition would be:

$*[\ Lx \bullet L?x\ ]\ \ \|\ \ *[\ Ly \bullet L?y\ ]$

The difference between this data part and what we have seen earlier is that the *channel* L is shared between two different processes.

In general, a shared channel may have a number of concurrent data parts that can send or receive data on it. Mutually exclusive access to the channel is guaranteed by the control. For instance, mutual exclusion on L in the example above is guaranteed by the fact that the control channels Lx and Ly originate in a single, sequential process. The shared channel L is an implementation of a bus. A discussion of bus implementations can be found in Chapter 5.

4.6. Projection*

In this section, we present a framework in which we can analyze a large class of program transformations that can be used for the introduction of pipelining. We present a method for the decomposition of programs into parts based on the notion of *projection*. The idea is inspired by examining the data-dependencies in the computation, and attempting to eliminate all unnecessary synchronization. In the simplest case, we take a program and syntactically project various variables into separate concurrent processes that are unsynchronized. For instance, we would decompose process

$$*[\ L?x; P?y;\ \ R!x; Q!y]$$

into

$$*[\ L?x; R!x\ ]\ \|\ *[\ P?y; Q!y\ ]$$

since the two parts of the original process are not data-dependent. Such transformations are not always semantics-preserving, and we examine the conditions under which we can apply them while preserving the correctness of the computation. The conditions amount to demonstrating that certain programs are *locally slack elastic*—a property which can be established for a relatively large class of programs.[23]

We would like to be able to decompose a process into parts and reduce the synchronization among parts that are not data-dependent. However, as was described above, reducing synchronization is not guaranteed to preserve the correctness of the original computation.

To address this issue, we adopt the following strategy. We begin by decomposing a process into parts, and keeping the parts tightly synchronized so as to preserve the semantics of the original computation. This decomposition is done via a *partition* function that systematically separates a CHP program fragment into two parts based on the structure of the program. The purpose of this part of the transformation is to syntactically divide the program fragment into two parts in a manner that guarantees correctness.

The second part of the transformation uses the concept of slack elasticity to introduce concurrency among the two parts that were decomposed by partitioning. We will loosen the synchronization constraints by assuming that we are designing our system by composing locally slack elastic components.

4.6.1. Partitions

A *partition* of a statement is a pair of statements such that their concurrent composition with a special *sequencer* process behaves exactly like the original statement, assuming that the two partitioned parts have infinite execution traces.

We introduce the program templates $LSync(\cdot)$ and $RSync(\cdot)$ to clarify the presentation of partitions. These template will be used to synchronize the two parts of the partition by using a "fork and join" synchronization mechanism when composed with the sequencer. These are defined as follows:

$$\begin{aligned} \mathit{LSync}(S) &\equiv\ \ C_1?;S;C_1? \\ \mathit{RSync}(S) &\equiv\ \ C_2?;S;C_2? \end{aligned}$$

where C_1 and C_2 are fresh channels. Given a partition of the form $\langle \mathit{LSync}(A), \mathit{RSync}(B)\rangle$, the concurrent composition of the two parts would be

$$C_1?;A;C_1? \ \| \ C_2?;B;C_2?$$

which, when composed with $C_1! \bullet C_2!;C_1! \bullet C_2!$, would implement a "fork and join" synchronization.

We define a function $\mathbf{part}(S)$ that maps statement S to a set of pairs, where each pair is a valid partition of S. This function is defined by structural induction on the body of S as follows:

- $\mathbf{part}(A) = \{\langle \mathit{Sync}(\mathbf{skip}), \mathit{Sync}(A)\rangle, \langle \mathit{Sync}(A), \mathit{Sync}(\mathbf{skip})\rangle\}$
  for any elementary statement A. We will define the partition function so that if $\langle p, q\rangle$ is in the set of valid partitions, so is $\langle q, p\rangle$ with *Lsync* and *Rsync* interchanged.
- $\mathbf{part}(S\|T) = \{\langle \mathit{Sync}(A), \mathit{Sync}(\mathbf{skip})\rangle, \langle \mathit{Sync}(\mathbf{skip}), \mathit{Sync}(A)\rangle, \langle \mathit{Sync}(S), \mathit{Sync}(T)\rangle,$
  $\langle \mathit{Sync}(T), \mathit{Sync}(S)\rangle\}$
  where A represents the statement $S\|T$. Parallel composition can be partitioned without introduction of any additional synchronization. Note that, for simplicity, parallel composition is one of the terminal symbols in our structural definition of partitions.
- $\mathbf{part}(S;T) = \{s,t \mid s \in \mathbf{part}(S) \wedge t \in \mathbf{part}(T) \triangleright \langle \mathbf{l}(s);\mathbf{l}(t), \mathbf{r}(s);\mathbf{r}(t)\rangle\}$
  The partition of the sequential composition of two statements is derived by examining all possible partitions of the two statements, and sequentially composing their components. The functions $\mathbf{l}(\cdot)$ and $\mathbf{r}(\cdot)$ are used to extract the left and right components of a pair. (An explanation of the set notation can be found in the appendix.)

Example. The partitions of $x := 0$ are

$$\{\langle \mathit{Sync}(x := 0), \mathit{Sync}(\mathbf{skip})\rangle, \langle \mathit{Sync}(\mathbf{skip}), \mathit{Sync}(x := 0)\rangle\}$$

∎

Example. The set of partitions of $A;B;C$ where A, B, C are elementary statements contains the following set:

$$\begin{aligned} \{&\langle \mathit{Sync}(A);\mathit{Sync}(B);\mathit{Sync}(C), \mathit{Sync}(\mathbf{skip});\mathit{Sync}(\mathbf{skip});\mathit{Sync}(\mathbf{skip})\rangle, \\ &\langle \mathit{Sync}(A);\mathit{Sync}(B);\mathit{Sync}(\mathbf{skip}), \mathit{Sync}(\mathbf{skip});\mathit{Sync}(\mathbf{skip});\mathit{Sync}(C)\rangle, \\ &\langle \mathit{Sync}(A);\mathit{Sync}(\mathbf{skip});\mathit{Sync}(C), \mathit{Sync}(\mathbf{skip});\mathit{Sync}(B);\mathit{Sync}(\mathbf{skip})\rangle, \\ &\langle \mathit{Sync}(A);\mathit{Sync}(\mathbf{skip});\mathit{Sync}(\mathbf{skip}), \mathit{Sync}(\mathbf{skip});\mathit{Sync}(B);\mathit{Sync}(C)\rangle\} \end{aligned}$$

The rest of the partitions can be obtained by the symmetric closure of the set (i.e., for each pair $\langle x, y\rangle$, add $\langle y, x\rangle$ to the set of partitions with *LSync* and *RSync* interchanged). Observe that for any pair $\langle x, y\rangle$ in the set of partitions, both components x and y of the partition locally preserve the ordering among actions in the original program. ∎

Partitions consist of two pieces of the original computation that, when composed in parallel with a sequencer process, precisely implement the original computation. We formalize this observation by the following theorem.

Theorem 4.2. (*partition*)

Let $E(\cdot)$ be a program template, and consider any pair $\langle L, R\rangle \in \mathbf{part}(P)$. Assume that in any deadlock-free execution of $E(P)$, action P is encountered infinitely often without any parallel duplication of P. Then,

$$E(L) \parallel *[\ R\ ] \parallel *[\ C_1! \bullet C_2!\ ] \quad \equiv \quad E(P)$$

Proof: (Sketch) Note that the requirement of infinite communication on channels C_1 and C_2 simply ensures that the process $*[C_1! \bullet C_2!]$ does not deadlock. If we permit process $*[C_1! \bullet C_2!]$ to deadlock, we can lift the infinite communication restriction.

Observe that L begins with $C_1?$ and R begins with $C_2?$—by definition of $\mathbf{part}(\cdot)$. Therefore, L cannot begin execution until R begins execution, and vice versa. Similarly, L cannot finish execution until R finishes (and vice versa), because they both end with $C_1?$ and $C_2?$ respectively. Therefore, it suffices to show that when $L \| R$ are composed with a process that repeatedly executes $C_1!\bullet C_2!$, they implement P. This can be shown by structural induction on the definition of $\mathbf{part}(\cdot)$.

Let $\langle L, R\rangle \in \mathbf{part}(P)$. The key observation is that the following properties hold: (a) L and R begin with $C_1?$ and $C_2?$ respectively; (b) L and R execute a matching number of communications on C_1 and C_2 (observe that this holds in the definition of $\mathbf{part}(\cdot)$ given above); (c) The environment executes $C_1!\bullet C_2!$, which enforces that the number of completed communication actions on channels C_1 and C_2 is equal.

The proof of equivalence is straightforward for the cases when P is an elementary statement or the parallel composition of two statements. We consider the case of sequential composition. Given $P = S; T$, we have $L = Ls; Lt$ and $R = Rs; Rt$ where Ls, Lt, Rs, Rt are obtained from the partitions of S and T by the appropriate substitution of $Sync(\cdot)$ actions with communications on C_1 and C_2. By induction, $Ls \| Rs$ implements S and $Lt \| Rt$ implements T (when composed with the process $*[C_1! \bullet C_2!]$). Since the number of C_1 actions in Ls equals the number of C_2 actions in Rs, we are guaranteed that the number of completed C_1 actions equals the number of completed C_2 actions at the semicolon between Ls and Lt—and at the semicolon between Rs and Rt. Since both Lt and Rt begin with a communication action on C_1 and C_2 respectively, they begin simultaneously—concluding the proof. □

Partitioning a process is a generalized form of process decomposition. In process decomposition, a CHP program was split into two parts by using the following transformation:

$$*[\ldots;\ S;\ \ldots] \ = \ *[\ldots;\ A;\ldots] \ \| \ *[[\overline{A}];S;A]$$

where A is a slack zero communication action. We can use partitioning to obtain a similar decomposition:

$$\begin{aligned} *[\ldots;\ S;\ \ldots] \ = \ & (*[\ldots;\ C_1?;C_1?;\ \ldots] \\ & \|*[\ C_2?;S;C_2?\ ] \\ & \|*[\ C_1! \bullet C_2!\ ]) \end{aligned}$$

We can replace $C_1?$ actions with $C_2!$ actions and eliminate the sequencer process, since this does not change any synchronization behavior. We obtain:

$$\begin{aligned} *[\ldots;\ S;\ \ldots] \ = \ & (*[\ldots;\ C_1!;C_1!;\ \ldots] \\ & \|*[\ C_1?;S;C_1?\ ]) \end{aligned}$$

(Indeed, when transformed into handshaking expansions, using a two phase protocol on channel C_1 would result in a reshuffled version of the system obtained through process decomposition.)

4.6.2. Projection

Given a statement S, we restrict our attention to *valid* partitions $\langle A, B\rangle \in \mathbf{part}(S)$—partitions such that A and B do not share any variables or communication ports. Disallowing shared variables permits us to use the theory of slack elastic systems to reason about partitions. Allowing the actions from A and B to overlap in any manner would not be permissible without this condition. For example, consider the process

$$x\uparrow;\ \ x\downarrow$$

One possible partition of it is shown below:

$$\langle \mathit{LSync}(x\uparrow); \mathit{LSync}(\mathbf{skip}),\ \ \mathit{RSync}(\mathbf{skip}); \mathit{RSync}(x\downarrow)\rangle$$

The "*Sync*" actions ensure that the two processes do not modify variable x simultaneously. If we attempted to remove the synchronizations, both $x\uparrow$ and $x\downarrow$ might execute concurrently, causing erroneous executions.

We introduce concurrency by adding slack on channels C_1 and C_2 that implement the "*Sync*" action. As we increase the slack on these channels, we loosen the synchronization constraints we have imposed through "*Sync*" operations. This transformation cannot be justified in general. We therefore restrict ourselves to considering systems such that the partitioned system (with the sequencer process) is locally slack elastic, and where channels C_1 and C_2 are slack elastic. The theorems stated earlier provide sufficient conditions that would ensure that the system is locally slack elastic.

We now let the slack on channels C_1 and C_2 be infinite. Notice that the sequencer process can run arbitrarily ahead of the the two partitions that it is synchronizing. As a result, the *LSync* and *RSync* actions no longer serve any purpose, and can be eliminated! The final result is a decomposition of the original process in which the two parts can execute concurrently.

Theorem 4.3. (*projection*)

Let $E(\cdot)$ be a program template, and let $\langle L, R\rangle$ be a valid partition of process P. Replace $LSync(A)$ by A in L (call this process L') and $RSync(B)$ by B in R (call this process R'). Assume that: (a) in any deadlock-free execution of $E(P)$, action P is encountered infinitely often without any parallel duplication of P; (b) $E(L) \| {*}[R] \| {*}[C_1! \bullet C_2!]$ is locally slack elastic; (c) $E(L)$ and R do not share variables. Then, $E(L') \| {*}[R']$ is a valid implementation of $E(P)$.

Proof: (Sketch) By Theorem 4.2, we are guaranteed that $E(L) \| {*}[R] \| {*}[C_1! \bullet C_2!]$ is equivalent to $E(P)$. By assumption (b), the system is locally slack elastic. Therefore, we can increase the slack on channels C_1 and C_2 without affecting the correctness of the entire system. When the slack on C_1 and C_2 is infinite, we can eliminate all communication actions on C_1 and C_2, because eliminating them does not affect the possible execution traces that could occur—concluding the proof. □

By the nature of this construction, we have a transformation that effectively *projects* a process onto two disjoint sets of variables.

Example. Consider the following process:

$${*}[\ L?x; R!x; P?y; Q!y\ ]$$

A valid partition of the body of the loop would be:

$$\langle LSync(L?x); LSync(R!x); LSync(\textbf{skip}); LSync(\textbf{skip}),$$
$$RSync(\textbf{skip}); RSync(\textbf{skip}); RSync(P?y); RSync(Q!y)\rangle$$

The two components of the partition do not share variables or ports. The system is locally slack elastic, because there are no selection statements. Therefore, we can apply Theorem 4.3 to argue that the original process is equivalent to:

$${*}[L?x; R!x; \textbf{skip}; \textbf{skip}]\ \|\ {*}[\textbf{skip}; \textbf{skip}; P?y; Q!y]$$

which is the same as:

$${*}[\ L?x; R!x\ ]\ \|\ {*}[\ P?y; Q!y\ ]$$

We can think of this as a transformation that syntactically projected the original process onto two sets of variables—$\{L?, R!, x\}$ and $\{P?, Q!, y\}$—to obtain the final result. ∎

4.6.3. Simple Pipelining

Consider the earlier example where we pipelined the computation of $g(f(x))$. The original program was

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

This program is equivalent to

```
*[ $L?x;\ \ y := f(x);\ \ R!g(y)$ ]
```

We can replace the assignment with communication actions using the communication axiom,[8] to obtain

```
*[ $L?x;\ \ (R'!f(x) \| R'?y);\ \ R!g(y)$ ]
```

We project the program onto the two disjoint sets $\{L?, x, R'!\}$ and $\{R'?, y, R!\}$ to obtain

```
*[ $L?x;\ \ R'!f(x)$ ] $\|$ *[ $R'?y;\ \ R!g(y)$ ]
```

which is the pipelined implementation. This example illustrates that the introduction of new variables and internal communication actions permits the systematic decomposition of a process into multiple parts that exchange data as required. The first step was to introduce variable y to hold the intermediate result $f(x)$. Next, the assignment to y was replaced with a communication action. Finally, we applied projection to obtain the pipelined implementation.

4.6.4. Data-dependent Synchronization

Consider the following program. It receives a bit on channel C, and depending on the value either discards the input received on channel A, or sends it out on channel X along with copying the value received on B to Y.

```
*[ $A?x;\ \ C?c;$
    [ $c = \mathbf{true} \longrightarrow\ \ B?y; X!x, Y!y$
    [] $c = \mathbf{false} \longrightarrow \mathbf{skip}$
    ]
 ]
```

Observe that the value communicated on channel Y in the first guard of the selection statement is received on B, and does not use the values c or x. We use the environment template

$E(P) \equiv$

```
*[ $A?x; C?c;$
    [ $c = \mathbf{true} \longrightarrow\ \ P$
    [] $c = \mathbf{false} \longrightarrow \mathbf{skip}$
    ]
    ]
```

We project $B?y; X!x, Y!y$ onto two disjoint sets $\{X!, x\}$ and $\{B?, Y!, y\}$; the two parts after projection are $X!x$ and $B?y; Y!y$ respectively. Since $E(X!x)$ does not share any variables with $B?y; Y!y$, we obtain the following decomposition:

```
*[ $A?x; C?c;$
   [ $c = \textbf{true} \longrightarrow X!x$ [] $c = \textbf{false} \longrightarrow \textbf{skip}$ ]
 ]
$\|$
*[$B?y; Y!y$]
```

Note that both processes are simpler to implement than the original process. Also, the actions on channels A and B are no longer synchronized. The second process has no internal selection statement, and does not even need to examine the value of variable c. Since our model of computation does not consider the relative order of actions on channels C and B, we can decompose the original computation into these two processes. Note that an environment where a process always attempts to send a message on channel B, but where C always receives a false value is ruled out because that would imply that the send action on B (in the environment) in the original computation was suspended forever—violating the absence of deadlock requirement on slack elastic programs.

4.6.5. Selections and Loops

The definition of partitions can be extended to handle selection statements and loops. When a selection statement is decomposed into two parts, we must ensure that we do not change the value of the guards of the selection statement. The definition of partitions for selection statements and loops is given below. We will use A to refer to the argument of **part** on the LHS for the remainder of this definition.

- $\mathbf{part}(\,[G_1 \wedge H_1 \rightarrow S_1 [] \cdots [] G_n \wedge H_n \rightarrow S_n]\,) =$

  $\{\langle \mathit{Sync}(A), \mathit{Sync}(\mathbf{skip})\rangle, \langle \mathit{Sync}(\mathbf{skip}), \mathit{Sync}(A)\rangle\} \cup$

  $\{s_1, \ldots, s_n \mid s_i \in \mathbf{part}(S_i) \triangleright$

  $\langle \mathit{Sync}(\mathbf{skip}); [G_1 \rightarrow \mathit{Sync}(\mathbf{skip}); \mathbf{l}(s_1) [] \cdots [] G_n \rightarrow \mathit{Sync}(\mathbf{skip}); \mathbf{l}(s_n)]\,,$

  $\mathit{Sync}(\mathbf{skip}); [H_1 \rightarrow \mathit{Sync}(\mathbf{skip}); \mathbf{r}(s_1) [] \cdots [] H_n \rightarrow \mathit{Sync}(\mathbf{skip}); \mathbf{r}(s_n)]\,\rangle\}$

  where $G_i \equiv H_i$, for all i. The first part of the expression is the trivial partition, where the entire statement is in one half of the partition. The rest of the definition consists of the case when we partition each statement in each guarded command. The guards can be partitioned as well; however, they can only be partitioned when they are equivalent in the two partitions, and equivalent to the guards used in the original program. Note that G_i and H_i can be interchanged freely. The partition of a selection statement is the union of all such conjunctive decompositions of the guards. The additional synchronization sequences the guard evaluation and the execution of the statement in the guarded command.

- $\mathbf{part}(*[G_1 \wedge H_1 \rightarrow S_1 \,[\!]\, \cdots \,[\!]\, G_n \wedge H_n \rightarrow S_n]) =$

$$\begin{array}{l}
\{\langle Sync(A), Sync(\mathbf{skip})\rangle, \langle Sync(\mathbf{skip}), Sync(A)\rangle\} \cup \\
\{s_1, \ldots, s_n \,|\, s_i \in \mathbf{part}(S_i) \rhd \\
\quad \langle Sync(\mathbf{skip}); *[G_1 \rightarrow Sync(\mathbf{skip}); \mathbf{l}(s_1); Sync(\mathbf{skip}) \\
\qquad\qquad\qquad [\!]\, \cdots \\
\qquad\qquad\qquad [\!]\, G_n \rightarrow Sync(\mathbf{skip}); \mathbf{l}(s_n); Sync(\mathbf{skip})], \\
\quad\; Sync(\mathbf{skip}); *[H_1 \rightarrow Sync(\mathbf{skip}); \mathbf{r}(s_1); Sync(\mathbf{skip}) \\
\qquad\qquad\qquad [\!]\, \cdots \\
\qquad\qquad\qquad [\!]\, H_n \rightarrow Sync(\mathbf{skip}); \mathbf{r}(s_n); Sync(\mathbf{skip})]\,\rangle\}
\end{array}$$

  where $G_i \equiv H_i$, for all i.

Non-deterministic loops and selection statements cannot be partitioned. This is because the implementation of a non-deterministic selection statement is free to pick any alternative that has a true guard. This implies that the two partitions might pick different alternatives since the choice is demonic,[6] thereby resulting in a system that would not be equivalent to the original program.

Example. Consider the following process. It receives inputs on channels A and B, and a control input on channel C. If the value received on C is **true**, the input on A is sent on channel X; otherwise, the input on B is sent on channel Y.

```
*[ $A?x; B?y; C?c;$
   [ $c = \mathbf{true} \longrightarrow X!x$
   [] $c = \mathbf{false} \longrightarrow Y!y$
   ]
 ]
```

We assume that the input on C is known early, and we would like to eliminate synchronization among actions A and B. To partition the selection statement, and to decouple the channel C from A and B we introduce variable ca and cb that is equivalent to c:

```
*[ $A?x; B?y; C?c; ca := c; cb := c;$
   [ $ca = \mathbf{true} \wedge cb = \mathbf{true} \longrightarrow X!x$
   [] $ca = \mathbf{false} \wedge cb = \mathbf{false} \longrightarrow Y!y$
   ]
 ]
```

The first step is to replace the assignment statements with communication actions, to obtain:

```
*[  $A?x; B?y; C?c; (Ca!c \| Ca?ca); (Cb!c \| Cb?cb);$
    [  $ca = \mathbf{true} \wedge cb = \mathbf{true} \longrightarrow X!x$
    [] $ca = \mathbf{false} \wedge cb = \mathbf{false} \longrightarrow Y!y$
    ]
 ]
```

Next, we project this process onto $\{C?, c, Ca!, Cb!\}$ and the rest of the ports and channels. The resulting system is:

```
*[  $A?x; B?y; Ca?ca; Cb?cb;$
    [  $ca = \mathbf{true} \wedge cb = \mathbf{true} \longrightarrow X!x$
    [] $ca = \mathbf{false} \wedge cb = \mathbf{false} \longrightarrow Y!y$
    ]
 ]
||
*[  $C?c; Ca!c; Cb!c$  ]
```

Finally, the first process can be projected onto $\{A?, x, Ca?, ca, X!\}$ and $\{B?, y, Cb?, cb, Y!\}$ to obtain:

```
*[  $A?x; Ca?ca;$
    [  $ca = \mathbf{true} \longrightarrow X!x$
    [] $ca = \mathbf{false} \longrightarrow \mathbf{skip}$
    ]
 ]
||
*[  $B?y; Cb?cb;$
    [  $cb = \mathbf{true} \longrightarrow \mathbf{skip}$
    [] $cb = \mathbf{false} \longrightarrow Y!y$
    ]
 ]
||
*[  $C?c; Ca!c; Cb!c$  ]
```

In the final program, we have decoupled channels A and B. Since the processes are all locally slack elastic, we can add slack on channels Ca and Cb to further decouple the control channel C from channels A and B. ▮

Quite often, we are faced with a program where we would like to partition a selection statement, but we cannot write the guards in the form required by our definition of **part**(·). Consider the selection statement shown below:

$$[G_1 \longrightarrow S_1 \;[\!]\; \cdots \;[\!]\, G_n \longrightarrow S_n]$$

We make the assumption that the guards of the selection statement are *stable*, i.e., the environment cannot change their value from true to false until the process executes some action that the environment can observe. (This is the case when the guards do not contain negated probes.) We introduce a fresh variable g as follows:

$$\begin{array}{l}[G_1 \longrightarrow g := 1 \;[\!]\; \cdots \;[\!]\, G_n \longrightarrow g := n];\\ [G_1 \longrightarrow S_1 \;[\!]\; \cdots \;[\!]\, G_n \longrightarrow S_n]\end{array}$$

Next, we strengthen the guards in the second statement to the following:

$$\begin{array}{l}[G_1 \longrightarrow g := 1 \;[\!]\; \cdots \;[\!]\, G_n \longrightarrow g := n];\\ [G_1 \wedge g = 1 \longrightarrow S_1 \;[\!]\; \cdots \;[\!]\, G_n \wedge g = n \longrightarrow S_n]\end{array}$$

We replace the assignment to g with communication actions, using a fresh slack zero channel G.

$$\begin{array}{l}[G_1 \longrightarrow G!1 \| G?g \;[\!]\; \cdots \;[\!]\, G_n \longrightarrow G!n \| G?g];\\ [G_1 \wedge g = 1 \longrightarrow S_1 \;[\!]\; \cdots \;[\!]\, G_n \wedge g = n \longrightarrow S_n]\end{array}$$

Observe that $G?g$ is common to all alternatives in the selection statement. Therefore, we can write:

$$\begin{array}{l}(G?g \| [G_1 \longrightarrow G!1 \;[\!]\; \cdots \;[\!]\, G_n \longrightarrow G!n]);\\ [G_1 \wedge g = 1 \longrightarrow S_1 \;[\!]\; \cdots \;[\!]\, G_n \wedge g = n \longrightarrow S_n]\end{array}$$

So far, we have not changed the computation performed by the original guarded command. This final form can be partitioned, because the guards of the selection statement are in the required form. We call this sequence of steps *control duplication*, as we have created a copy g of control flow information.

4.7. Case Study: The Writeback

We present the decomposition of the *writeback* unit from the asynchronous MIPS processor.[32] The MIPS Instruction Set Architecture (ISA) specifies that any exception caused by an instruction is *precise*.[11] A precise exception mechanism has to guarantee that that the instruction that caused the exception and all instructions following it until the first instruction of the exception handler do not modify any observable state of the processor. The observable state in the MIPS ISA consists of the memory, special purpose registers, and general purpose registers. The writeback unit coordinates this behavior, controlling when a particular instruction is permitted to modify the state of the processor. The sequential CHP program for the writeback unit is:

```
$WB \equiv$  $valid\uparrow$;
     *[  $UN?un, UZ?uz, VA?va, EPC?epc$;
         [$un = 0 \longrightarrow EX0?e$ ▯ $un = 1 \longrightarrow EX1?e$ ▯ $un = 2 \longrightarrow e := \textsf{none}$ ];
         [$uz = 1 \longrightarrow REG!(valid \wedge (e = \textsf{none}))$
         ▯$uz = 0 \wedge un = 1 \longrightarrow MEM!(valid \wedge (e = \textsf{none}))$
         ▯$uz = 0 \wedge un = 2 \longrightarrow MULT!(valid \wedge (e = \textsf{none}))$
         ];
         [$valid \wedge (e \neq \textsf{none}) \longrightarrow valid\downarrow, EX, CP0pc!epc, CP0e!e$
         ▯$va \longrightarrow valid\uparrow$
         ▯$\textbf{else} \longrightarrow \textbf{skip}$
         ]
      ]
```

The information received on various input channels is as follows:

- *UN*: where to read the exception information for the next instruction to be processed.
- *UZ*: 1 if the instruction writes its results to the register file, 0 otherwise.
- *VA*: 1 if the instruction is the first instruction of the exception handler, 0 otherwise.
- *EPC*: the program counter for the instruction.
- $EX0$, $EX1$: the type of exception from different execution units.

The writeback process is in two states: executing valid instructions (*valid* is true); canceling the results of instructions (*valid* is false). The variables *uz* and *va* are one bit wide; *un* is two bits wide; *epc* is a 32-bit quantity.

The writeback process reads the exception information from the appropriate channel. It sends permission to modify the state of the processor (or lack thereof) to the appropriate part of the processor (register file, memory unit, or multiplier/divider unit which has local registers). If an exception is detected, it notifies the *PCUNIT* (via *EX*) and sends the program counter and exception type to the CP0 on channels $CP0pc$ and $CP0e$ respectively. Finally, it updates the valid bit.[32,11] Note that the writeback process is locally slack elastic, because it does not probe any channels.

First, we decompose the computation of *valid* out of the writeback. The computation of valid depends on *va*, *valid*, and the value of *e*. Note that *va* is not used by any other part of the computation. However, the rest of the computation uses *valid*. We introduce a copy of valid, *valid*2, that is used by the rest of the program. To project the last selection statement (containing *epc* and *e*), we introduce a copy of *e* as well.

After introducing the appropriate copies of *valid* and *e*, and adding communication actions

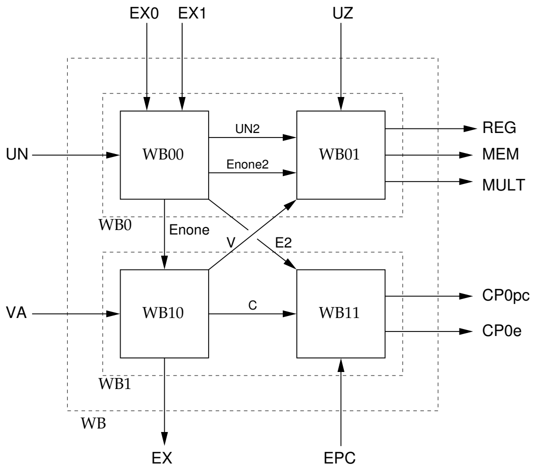


Figure 4.10. Decomposed version of the writeback.

(as in the examples earlier), we obtain:

```
$WB0 \equiv\ V?valid2;$
    $*[\ \ UN?un,\ UZ?uz;$
        $[un = 0 \longrightarrow EX0?e\ \ [\!]\ \ un = 1 \longrightarrow EX1?e\ \ [\!]\ \ un = 2 \longrightarrow e := \textsf{none}\ \ ];$
        $Enone!(e \neq \textsf{none}), E2!e,$
        $[uz = 1 \longrightarrow REG!(valid2 \wedge (e = \textsf{none}))$
        $[\!]uz = 0 \wedge un = 1 \longrightarrow MEM!(valid2 \wedge (e = \textsf{none}))$
        $[\!]uz = 0 \wedge un = 2 \longrightarrow MULT!(valid2 \wedge (e = \textsf{none}))$
        $];\ \ V?valid2$
    $]$
```

```
$WB1 \equiv\ valid\uparrow;\ V!valid;$
    $*[\ \ VA?va;\ EPC?epc;\ Enone?en;\ E2?e2;$
        $[valid \wedge en \longrightarrow valid\downarrow, EX, CP0pc!epc, CP0e!e2$
        $[\!]va \longrightarrow valid\uparrow$
        $[\!]\textbf{else} \longrightarrow \textbf{skip}$
        $];\ \ V!valid$
    $]$
```

We now focus on the second process. Observe that action $V!valid$ is performed on initialization, and after each iteration of the non-terminating outer loop in $WB1$. Therefore, it can be rewritten as:

```
$WB1 \equiv$  $valid\uparrow;$
    $*[$  $V!valid;\ VA?va;\ EPC?epc;\ Enone?en;\ E2?e2;$
        $[valid \wedge en \longrightarrow valid\downarrow, EX, CP0pc!epc, CP0e!e2$
        $[]\,va \longrightarrow valid\uparrow$
        $[]\,\textbf{else} \longrightarrow \textbf{skip}$
        $]$
    $]$
```

Since *epc* is a 32-bit quantity, we decompose it out into a separate process. We apply the control duplication steps shown at the end of the previous section in order to decompose the selection statement in $WB1$. We name the newly introduced channel C, and the new variable introduced gd. Applying projection once again, we obtain:

```
$WB10 \equiv$  $valid\uparrow;$
    $*[$  $V!valid;\ VA?va;\ Enone?en;$
        $[valid \wedge en \longrightarrow C!0, valid\downarrow, EX$
        $[]\,va \longrightarrow C!1, valid\uparrow$
        $[]\,\textbf{else} \longrightarrow C!2$
        $]$
    $]$
```

```
$WB11 \equiv *[$  $EPC?epc;\ E2?e2;\ C?gd;$
        $[gd = 0 \longrightarrow CP0pc!epc, CP0e!e2$
        $[]\,\textbf{else} \longrightarrow \textbf{skip}$
        $]$
    $]$
```

We now examine process $WB0$. We separate the process into two parts: one that computes the value of e, and the rest that uses the value of e. The part that computes the value of e depends on the value un, received on channel UN. The result of this projection is:

```
$WB00 \equiv *[$  $UN?un;$
        $[un = 0 \longrightarrow EX0?e \ [] \ un = 1 \longrightarrow EX1?e \ [] \ un = 2 \longrightarrow e := \textsf{none}\ ];$
        $Enone!(e \neq \textsf{none}), E2!e, Enone2!(e = \textsf{none}), UN2!un$
    $]$
```

```
$WB01 \equiv *[$  $V?valid2;\ UN?uz;\ Enone2?en2;\ UN2?un2;$
        $[uz = 1 \longrightarrow REG!(valid2 \wedge en2)$
        $[]\,uz = 0 \wedge un2 = 1 \longrightarrow MEM!(valid2 \wedge en2)$
```

```
    $[\!]\, uz = 0 \wedge un2 = 2 \longrightarrow MULT!(valid2 \wedge en2)$
    ]
  ]
```

By the projection theorem, the concurrent composition of $WB00$, $WB01$, $WB10$ and $WB11$ implement the original WB process. Figure 4.10 shows the final process decomposition, with the channels visible to the rest of the system as well as the internal channels introduced for the purposes of projection.

The final decomposition of the writeback is superior to the original program WB because each process in the final decomposition performs fewer actions in sequence, reducing the overall cycle time of the system. In addition, by further adjusting the slack on the different channels in the system (in particular, adding slack to channels $UN2$, $Enone2$, and $E2$) we can ensure that the pipelined implementation of the writeback operates at peak throughput.[5,42]

References

The material on projection-based decomposition is from a paper by Manohar[21] et al. The writeback was taken from the design of an asynchronous MIPS processor.[32]

Chapter 5.

Datapath Design

Datapath design raises issues that are different from and in some aspects more challenging than control circuitry. A datapath process tends to be replicated several times, since the same process is typically used for every bit in the datapath. In this chapter, we present techniques for the design of datapath circuitry that occurs in many asynchronous designs.

5.1. Function Blocks

In this section, we focus on function computation: given a value x, compute $f(x)$. We can describe this task with the following CHP program:

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

The first step in designing the production rules for this process is control data decomposition. Applying this transformation, we obtain:

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

$$\triangleright$$

$$*[\ L'; R'\ ] \parallel *[\ L' \bullet L?x\ ] \parallel *[\ R' \bullet R!f(x)\ ]$$

We focus on the implementation of process $*[R' \bullet R!f(x)]$, since this process is responsible for computing function f.

An active handshake protocol for $R!f(x)$, when R carries a single bit of information can be written as:

$$*[\ [f(x) \longrightarrow rto{\uparrow} \,\square\, \neg f(x) \longrightarrow rfo{\uparrow}]; [ri]; rto{\downarrow}, rfo{\downarrow}; [\neg ri]\ ]$$

When R transmits multiple bits, the parts of the handshake that change are the ones where we set and reset rto and rfo. We write the statement "$[f(x) \rightarrow rto{\uparrow} \,\square\, \neg f(x) \rightarrow rfo{\uparrow}]$" as $R := f(x)$ or simply $R \Uparrow$, indicating that we set the data rails of R to some data value that encodes the value of $f(x)$. We write the statement "$rto{\downarrow}, rfo{\downarrow}$" as $R \Downarrow$ to indicate that the data rails of channel R are set to the initial state. Using this notation, a standard handshake on channel R is given by:

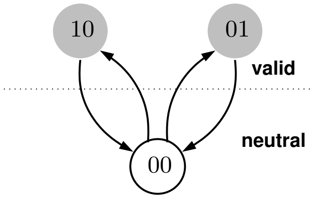


Figure 5.1. Dual-rail data encoding.

$*[\ R\Uparrow; [ri]; R\Downarrow; [\neg ri]\ ]$

Before we continue the compilation, we study different ways data values can be encoded on the data rails of a communication channel.

5.1.1. Delay-Insensitive Codes

Section 3.6 introduced dual-rail codes as a method to encode data values on wires. The important property of this code that makes it suitable for asynchronous communication is that the receiver can detect when valid data is being transmitted by the sender: when the code is "00," no data is being transmitted; when the code is "01" or "10," a one or a zero is being transmitted respectively. Before transmitting the next data value, the code returns to the "00" state. This is shown pictorally in Figure 5.1, with arrows depicting valid state transitions.

Codes that permit the receiver to detect when valid data is being transmitted independent of the delays in wires are said to be *delay-insensitive* codes. An important consequence of assuming wires have arbitrary delays on them is that the order in which wires are assigned values by the sender cannot be maintained on the receiver side.

Example. Consider the following scheme for encoding data. We use two Boolean-valued variables x and y, and set x to the data value that is to be transmitted. Once x has been set, y is set to "1." This scheme is not delay insensitive, because we cannot guarantee that the receiver will observe the correct value of x when it observes the change in y. ▮

We consider the problem of transmitting data in a delay-insensitive fashion using a vector of Boolean-valued variables. Given a vector $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$, the possible values the vector can take is given by the set $\mathcal{X} = \{0,1\}^n$. Let $\mathcal{B}$ be the set of data words that are to be transmitted. Any value being transmitted is encoded using a function $\mathcal{C}\colon \mathcal{B} \mapsto \mathcal{X}$ that maps data words to valid *code words*. The set $\mathcal{C}(\mathcal{B})$ is said to be the *set of valid values*, denoted $\mathcal{V}$. The coding function $\mathcal{C}$ must be chosen so that there exists a non-empty set $\mathcal{N} \subseteq \mathcal{X} - \mathcal{V}$ of *neutral values*. By definition, a code word in $\mathcal{X}$ cannot be both valid and neutral, since the sets $\mathcal{V}$ and $\mathcal{N}$ are disjoint.

Given a particular code word X, the predicate $v(X)$ denotes "X is a valid code word," which is the same as $X \in \mathcal{V}$. The predicate $n(X)$ denotes "X is a neutral code word," which is the same as $X \in \mathcal{N}$.

Example. For the dual-rail code shown in Figure 5.1, the set of valid code words $\mathcal{V} = \{01, 10\}$, and the set of neutral code words $\mathcal{N} = \{00\}$. ∎

The transmission of a data word w by the sender is the assignment of a valid code word $\mathcal{C}(w)$ to the set of wires. We can construct a communication protocol where the receiver detects the valid code word if the assignment implies that the wires change from a neutral to a valid value. A simple communication protocol can be described by the following handshaking expansion:

```
*[ $generate\ \ w; X \Uparrow;$ [$xi$]; $X \Downarrow;$ [$\neg xi$] ]
*[ [$v(X)$]; $read\ \ data; xi\uparrow;$ [$n(X)$]; $xi\downarrow$ ]
```

$X \Uparrow$ denotes the concurrent assignment of the wires encoding data such that the result sets X to a valid value, namely $\mathcal{C}(w)$, and $X \Downarrow$ denotes the concurrent assignment of the wires of X that set X to a neutral value.

Because the assignments to the wires used to communicate a code word are concurrent, and because there can be an arbitrary delay on the wires connecting the sender to the receiver, any transition from a neutral value to a valid value can go through several intermediate states. When executing the waits [$v(X)$] and [$n(X)$], the receiver can read several intermediate values for X. These values correspond to the cases when only a subset of the wires of X have changed. To avoid premature completion of the waits, we must ensure that none of the intermediate values generated during the transition from neutral to valid (also known as an *upward intermediate value*) is valid, and none of the intermediate values generated from the transition from valid to neutral (also known as a *downward intermediate value*) are neutral. A code that has this property is said to be *separable*.

Example. The dual-rail code is separable, because we change only one wire to go from either the valid state to the neutral state, or from the neutral state to the valid state. When transmitting n bits of data, the dual rail encodes each bit b_k using two wires xt_k and xf_k. The set of valid and neutral codes is given by:

$$n(X) = (\forall k : n : \neg xt_k \land \neg xf_k)$$

$$v(X) = (\forall k : n : xt_k \neq xf_k)$$

The code is separable because any valid code word differs from the neutral code word in n bit positions. Therefore, any upward intermediate value has fewer than n bit positions changed, and is therefore not a valid codeword. The code also has only one neutral value, so no downward intermediate value is neutral. ∎

Example. A commonly used delay-insensitive code is the *one-hot code*. For a data word consisting of n bits, the one-hot encoding for it uses 2^n bits with exactly one bit set to **true**. The position of the bit encodes the data value being transmitted. The valid and neutral codes for the one-hot code $(x_0, \ldots, x_{n-1})$ are given by:

$$n(X) = (\forall k : n : \neg x_k)$$
$$v(X) = (\exists i : n : x_i \wedge (\forall j : j \neq i : \neg x_j))$$

Since both $X \Uparrow$ and $X \Downarrow$ only set one data rail, the code is separable. ∎

Example. Both one-hot codes and dual-rail codes are examples of a special class of codes known as k-out-of-n codes. These codes use n bits to encode $\binom{n}{k}$ possible data values by setting any k out of n bits to one. The valid and neutral codes are given by:

$$n(X) = (\forall i : n : \neg x_i)$$
$$v(X) = ((Ni : n : x_i) = k)$$

where $(Ni : n : f(i))$ denotes the number of i for which $f(i)$ holds. The special class of codes where $k = \lfloor n/2 \rfloor$ are known as Sperner codes. ∎

If a code is separable, then the receiver can detect when the sender has sent a valid data value using the procotol described above. For practical reasons, it is preferable that the code be such that is is easy to translate between data values and code words, since such translations are necessary at the sender and receiver. In addition, the implementations of the waits $[n(X)]$ and $[v(X)]$ should be simple because the waits are part of the receiver circuitry.

An important property of the validity and neutrality tests used by delay insensitive communication protocols is that they are *stable* conditions: once the conditions become **true**, they remain **true** until the process containing the condition changes some variable. This follows from the fact that the codes are separable.

Codes that permit the tests $[n(X)]$ and $[v(X)]$ to be partitioned into smaller subtests are said to be *distributive*. Given a code X, a *subcode* of X is a word formed from a proper subset of the set of bits of X. A code is said to be distributive just when X can be divided into a set S of subcodes such that:

- $n(Y)$ and $v(Y)$ are defined for any $Y \in S$;
- $(\forall y : y \in S : n(Y)) = n(X)$
- $(\forall y : y \in S : v(Y)) = v(X)$

Example. The dual-rail code is distributive in the obvious way. For each subcode, we select any (non-zero) set of pairs (xt_k, xf_k), and ensure that the union of all the subcodes includes every

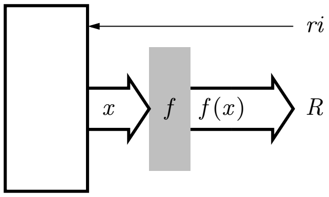


Figure 5.2. Function block compilation.

(xt_k, xf_k) pair. As opposed to this, one-hot codes are not distributive, and neither are k-out-of-n codes. ▮

Example. Berger codes consist of two parts: data bits and check bits. Given n bits to be transmitted, the code uses n data bits and $\lceil \lg(n+1) \rceil$ check bits. The code has one neutral value which corresponds to the case when all the bits are zero. A data word is transmitted by setting the data bits to the value to be sent, and setting the check bits to the number of zeros in the data word. A valid code word is one where the number of zeros in the data bits matches the number of zeros specified by the check bits. This code is separable but not distributive. ▮

5.1.2. Implementation of Function Blocks

We focus on the implementation of $*[R' \bullet R!f(x)]$ obtained after process decomposition. The handshaking expansion for this process is given by:

$$*[\ R\Uparrow; [ri]; R\Downarrow; [\neg ri]\ ]$$

where R is encoded using a delay-insensitive code. The receiver is given by:

$$*[[v(R)]; ri\uparrow; [n(R)]; ri\downarrow]$$

where $v(R)$ and $n(R)$ are the validity and neutrality conditions. Instead of attempting to compile the sender process directly, we decompose it into two parts: a process that behaves as an ordinary sender, and a process that performs the function computation specified by f. The sender is replaced by the concurrent composition of the two parts shown below.

$$\begin{aligned} \mathit{XMIT} &\equiv *[\ R'\Uparrow; [ri]; R'\Downarrow; [\neg ri]\ ] \\ \mathit{FBLK} &\equiv *[\ [v(R')]; R\Uparrow; [n(R')]; R\Downarrow\ ] \end{aligned}$$

where the data sent on R is a function of the data transmitted on R'. The data transmitted on R is transformed "on the wires" and sent on R', as shown in Figure 5.2.

Process *XMIT* performs a four-phase handshake on R' and ri, while the environment performs a four-phase handshake on R and ri. Process *FBLK* computes the necessary function f, and sets R using the value received on R'. To ensure that the handshake procotols are not violated, *FBLK* must ensure that $v(R)$ implies $v(R')$, and $n(R)$ implies $n(R')$. For the remainder of this section, we focus on the implementation a process of the form:

$$F \;\equiv\; *[\;\; [v(X)]\,;\, Y\Uparrow;\, [n(X)]\,;\, Y\Downarrow\;]$$

that corresponds to the intermediate *FBLK* process above. We attempt to decompose F into the concurrent composition of a number of processes that operate independently. In order to do so, we assume that both X and Y are encoded using a distributive code. In particular, we break up X and Y into the same number of subcodes that satisfy a *data-dependence* constraint. Given the subcodes Y_0, …, Y_n of output Y and the same number of subcodes X_0, …, X_n of input X, the choice of subcodes is said to satisfy the data-dependence constraint if all variables of X required to compute codeword Y_k are contained in X_k. Assuming this to be the case, we apply the following sequence of transformations.

The first transformation replaces the global waits by conjunctions of local waits on subcodes. In addition, it replaces the assignments $Y\Uparrow$ and $Y\Downarrow$ by assignments to subcodes of Y. In this case, we assume that the subcodes of Y are disjoint—they do not share any bits. The result is:

$$F \;\equiv\; *[\;\; [(\wedge k :: v(X_k))]\,;\,(\| \; k :: \; Y_k\Uparrow);\, [(\wedge k :: n(X_k))]\,;\,(\| \; k :: \; Y_k\Downarrow)\;]$$

where X_0, …, X_n are the subcodes of X, and Y_0, …, Y_n are the subcodes of Y. The correctness of this transformation is immediate from the definition of a distributive code.

The second transformation replaces the conjunction of the waits by the parallel composition of the individual waits. This transformation relies on the stability of the validity and neutrality tests. The result is:

$$F \;\equiv\; *[\;\; (\| \; k :: [v(X_k)]);(\|k :: \; Y_k\Uparrow);(\| \; k :: [n(X_k)]);(\|k :: \; Y_k\Downarrow)\;]$$

The third transformation eliminates the semicolons between the concurrent waits and the following concurrent assignments. The result is:

$$F \;\equiv\; *[\;\; (\| \; k :: [v(X_k)]\,;\, Y_k\Uparrow);(\| \; k :: [n(X_k)]\,;\, Y_k\Downarrow)\;]$$

We can apply this transformation because the number of subcodes used for the input X must match the number of subcodes used for the output Y, and for the following additional reasons. In this implementation of F, $Y_k\Uparrow$ can be executed before $v(X)$ holds, since it only waits for $v(X_k)$. However, the value set by the assignment $Y_k\Uparrow$ is still the same as before, because $Y_k\Uparrow$ waits for $v(X_k)$, which contains all variables necessary to compute Y_k. The environment guarantees that X will not be changed from valid to neutral until Y is valid; this is the same as all the subcodes of Y being valid, which guarantees that all subcodes of X are valid. Therefore, the handshake protocol on X is preserved.

The next transformation eliminates all the global synchronization points by replacing F with the following:

$$F \;\equiv\; (\| \; k :: *[\;\; [v(X_k)];\, Y_k \Uparrow;\, [n(X_k)];\, Y_k \Downarrow \;\;])$$

This transformation can be justified in a manner similar to the previous transformation; the sequencing we have eliminated is still being enforced by the environment, and therefore we need not re-implement it in process F.

The final implementation of F as the concurrent composition of several processes can be directly compiled into production rules. Suppose Y is encoded using dual-rail codes, and let the assignment $Y := f(X)$ be equivalent to:

$$(\| \; k :: [Bt_k \longrightarrow yf_k{\downarrow};\, yt_k{\uparrow} \;\; [] \;\; Bf_k \longrightarrow yt_k{\downarrow};\, yf_k{\uparrow} \;\;])$$

By construction, the predicates Bt_k and Bf_k only depend on variables in X_k. In this case, we can write F as:

$$\begin{aligned} F \;\equiv\; (\| \; k :: *[\,&[v(X_k) \wedge Bt_k \longrightarrow yt_k{\uparrow} \\ &[]v(X_k) \wedge Bf_k \longrightarrow yf_k{\uparrow} \\ &]; \\ &[n(X_k)];\, yt_k{\downarrow},\, yf_k{\downarrow} \\ &] \\ )& \end{aligned}$$

The production rules for this process are given by:

$$\begin{array}{lllllll} v(X_k) \wedge Bt_k & \mapsto & yt_k{\uparrow} & \qquad & n(X_k) & \mapsto & yt_k{\downarrow} \\ v(X_k) \wedge Bf_k & \mapsto & yf_k{\uparrow} & \qquad & n(X_k) & \mapsto & yf_k{\downarrow} \end{array}$$

Quite often the conditions Bt_k and Bf_k imply the validity condition $v(X_k)$. In this case, Boolean simplification of the guards results in the elimination of the explicit validity test.

Example. Assuming both X and Y are encoded using dual-rail codes, we compute the complement of X. The final function block process is given by:

$$*[[xt_k \vee xf_k];\, [xt_k \longrightarrow yf_k{\uparrow} \;\; [] \;\; xf_k \longrightarrow yt_k{\uparrow}];\, [\neg xt_k \wedge \neg xf_k];\, yt_k{\downarrow},\, yf_k{\downarrow}]$$

In this case, both xt_k and xf_k imply the validity of the kth input rail (X_k). Therefore, the production rules are given by:

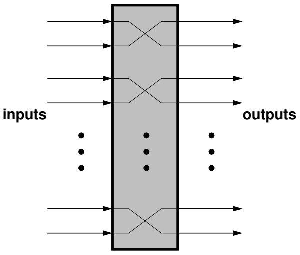


Figure 5.3. Function block: complement.

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
xt_k & \mapsto & yf_k\uparrow & xf_k & \mapsto & yt_k\uparrow \\
\neg xt_k & \mapsto & yf_k\downarrow & \neg xf_k & \mapsto & yt_k\downarrow
\end{array}
$$

and the resulting circuit is shown in Figure 5.3. ∎

Example. To compute the logical AND of two inputs, we design a function block with two bits as input per output bit. All codes are dual-rail, and output bit k is computed from inputs at_k, af_k, bt_k, and bf_k representing the two bits to be combined. The function block process is given by:

$$
\begin{array}{l}
*[\ \ [(at_k \vee af_k) \wedge (bt_k \vee bf_k)]\,;\, [at_k \wedge bt_k \longrightarrow yt_k\uparrow \;\square\; af_k \vee bf_k \longrightarrow yf_k\uparrow]\,; \\
\qquad [\neg at_k \wedge \neg af_k \wedge \neg bt_k \wedge \neg bf_k]\,;\, yt_k\downarrow, yf_k\downarrow \\
\quad]
\end{array}
$$

While the condition $(at_k \wedge bt_k)$ implies that the input is valid, $(af_k \vee bf_k)$ only checks for the validity of one of the inputs. Therefore, we must include the rest of the validity test in the production rules for $yf_k\uparrow$. The rules are given by:

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
at_k \wedge bt_k & \mapsto & yt_k\uparrow & at_k \wedge bf_k \vee af_k \wedge (bt_k \vee bf_k) & \mapsto & yf_k\uparrow \\
\neg at_k \wedge \neg bt_k & \mapsto & yt_k\downarrow & \neg at_k \wedge \neg af_k \wedge \neg bt_k \wedge \neg bf_k & \mapsto & yf_k\downarrow
\end{array}
$$

Note that we have also simplified the rules for $yt_k\downarrow$ by observing that the condition $\neg at_k \wedge \neg bt_k$ suffices for the neutrality test, because yt_k can only be **true** when both at_k and bt_k are **true**. ∎

5.1.3. An Asynchronous Adder

We present the function-block compilation of an asynchronous adder. The compilation is instructive, because it shows how careful design of a function block can improve its performance while simplifying the circuit.

The adder has two n-bit inputs a and b. To adopt the function block compilation strategy, we assume that the carry-in for each bit position is available as an n-bit input c. The function block generates two n-bit outputs: s, the sum, and d, the carry-out. The function block process for bit-position k is shown below:

$$
\begin{array}{l}
*[[(at \vee af) \wedge (bt \vee bf) \wedge (ct \vee cf)]\,; \\
\quad [at \wedge bt \vee (a \neq b) \wedge ct \longrightarrow dt\uparrow \;\square\; af \wedge bf \vee (a \neq b) \wedge cf \longrightarrow df\uparrow\ ], \\
\quad [ct \wedge (a = b) \vee cf \wedge (a \neq b) \longrightarrow st\uparrow \;\square\; cf \wedge (a = b) \vee ct \wedge (a \neq b) \longrightarrow sf\uparrow\ ]\,; \\
\quad [\neg at \wedge \neg af \wedge \neg bt \wedge \neg bf \wedge \neg ct \wedge \neg cf]\,;\, dt\downarrow, df\downarrow, st\downarrow, sf\downarrow \\
\ ]
\end{array}
$$

(We have dropped the subscript k for clarity.) Since the carry-out for one bit position should be the carry-in for the next bit-position, we can connect the d output of bit-position k to the c input

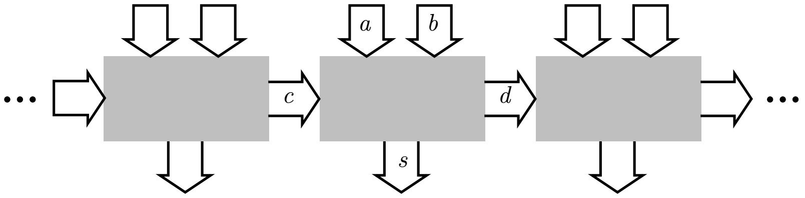


Figure 5.4. Function block implementation of adder.

for bit-position $k+1$. We can apply this transformation because the validity/neutrality of the s output implies that all the inputs—including c—are valid/neutral.

The adder structure is shown in Figure 5.4. From the HSE for the adder cell shown above, note that the adder does not produce any outputs until the c input is valid. Since the c input for bit-position $k+1$ is d output for bit-position k, the input to output latency of the adder is $\Theta(n)$ steps due to the linear chain of data-dependencies.

To improve the performance of the adder, we examine the environment for the adder. The environment is given by two processes that generate the a and b inputs, and a third process that reads the sum s, as shown below:

```
        $*[\ A\Uparrow;\ [ai];\ A\Downarrow;\ [\neg ai]\ ]\ \|\ *[\ B\Uparrow;\ [bi];\ B\Downarrow;\ [\neg bi]\ ]$
     $\|\ *[\ [v(S)];\ \ ai\uparrow, bi\uparrow;\ \ [n(S)];\ \ ai\downarrow, bi\downarrow\ ]$
```

The environment waits for all the bits of S to be valid before permitting the inputs A and B to change. Using this observation, we can improve the performance of the adder by distributing the validity test. We allow the adder cell to produce its d output early without all the inputs being valid. The result is shown below.

```
    $*[[at \wedge bt \vee (a \neq b) \wedge ct \longrightarrow dt\uparrow\ ▯ af \wedge bf \vee (a \neq b) \wedge cf \longrightarrow df\uparrow\ ],$
      $([(at \vee af) \wedge (bt \vee bf) \wedge (ct \vee cf)];$
        $[ct \wedge (a = b) \vee cf \wedge (a \neq b) \longrightarrow st\uparrow\ ▯ cf \wedge (a = b) \vee ct \wedge (a \neq b) \longrightarrow sf\uparrow\ ]);$
      $[\neg at \wedge \neg af \wedge \neg bt \wedge \neg bf \wedge \neg ct \wedge \neg cf];\ dt\downarrow, df\downarrow, st\downarrow, sf\downarrow$
     ]
```

In the case when $at \wedge bt$ holds or $af \wedge bf$ holds (i.e., when the inputs are equal), the output d is produced without waiting for c to be valid. Therefore, the time taken by the adder to set all the bits of the sum output is proportional to the longest sequence of consecutive input positions where the bits of a and b are not equal to each other. Analysis shows that this quantity is, on average (assuming a uniform input distribution), $O(\log n)$, where n is the number of input bits. Observe that the sum at position $k+1$ waits for the carry-out d at position k to be valid, thereby ensuring that all the inputs to every function block cell are valid before A and B can change. The cell at bit-position n is optimized so that it does not produce a d output.

The new adder cell is still slow in resetting its outputs back to the neutral state. This is once again because of the sequential dependence introduced by waiting for the c inputs to be neutral before resetting the d output. We can optimize this once again by distributing the neutrality test between the two outputs. The result is shown below.

$$
\begin{array}{l}
*\texttt{[}\,\texttt{[}at \wedge bt \vee (a \neq b) \wedge ct \longrightarrow dt\!\uparrow\ [\!]\, af \wedge bf \vee (a \neq b) \wedge cf \longrightarrow df\!\uparrow\ \texttt{]},\\
\quad (\texttt{[}(at \vee af) \wedge (bt \vee bf) \wedge (ct \vee cf)\texttt{]};\\
\quad \texttt{[}ct \wedge (a = b) \vee cf \wedge (a \neq b) \longrightarrow st\!\uparrow\ [\!]\, cf \wedge (a = b) \vee ct \wedge (a \neq b) \longrightarrow sf\!\uparrow\ \texttt{]});\\
\quad \texttt{[}\neg at \wedge \neg af \wedge \neg bt \wedge \neg bf \longrightarrow dt\!\downarrow, df\!\downarrow\texttt{]},\ \ \texttt{[}\neg ct \wedge \neg cf \longrightarrow st\!\downarrow, sf\!\downarrow\texttt{]}\\
\texttt{]}
\end{array}
$$

When the input a and b becomes neutral, every cell resets its d output. In the next step, all the cells will reset their s outputs because their c inputs would be neutral. Note how the reset phase for the sum output at bit-position $k+1$ waits for the neutrality of the d outputs at bit-position k, and therefore indirectly waits for the a and b inputs at bit-position k to be neutral. Also, we cannot use this particular implementation for the cell at position n since it does not have a carry-out d. The production rules for the cell are shown below.

$$
\begin{array}{rcl}
at \wedge bt \vee (at \vee bt) \wedge ct & \mapsto & dt\!\uparrow\\
\neg at \wedge \neg af \wedge \neg bt \wedge \neg bf & \mapsto & dt\!\downarrow\\
\\
af \wedge bf \vee (af \vee bf) \wedge cf & \mapsto & df\!\uparrow\\
\neg at \wedge \neg af \wedge \neg bt \wedge \neg bf & \mapsto & df\!\downarrow\\
\\
ct \wedge (at \wedge bt \vee af \wedge bf) \vee cf \wedge (at \wedge bf \vee af \wedge bt) & \mapsto & st\!\uparrow\\
\neg ct \wedge \neg cf & \mapsto & st\!\downarrow\\
\\
cf \wedge (at \wedge bt \vee af \wedge bf) \vee ct \wedge (at \wedge bf \vee af \wedge bt) & \mapsto & sf\!\uparrow\\
\neg ct \wedge \neg cf & \mapsto & sf\!\downarrow
\end{array}
$$

If data is encoded using dual-rail codes, the function-block compilation of a process yields production rules that are easily bubble-reshuffled. For example, the adder cell production rules are directly implementable in CMOS if we choose the inverted sense of all the outputs and introduce inverters to generate the uninverted outputs.

5.1.4. Alternative Compliations: Explicit Validity Check

As should be evident from the previous section, the function block implementation technique leads to circuits that may have a large number of transistors in series. These transistors are a direct consequence of the constraint of having $v(Y)$ implying that $v(X)$ holds, and $n(Y)$ implying that

$n(X)$ holds. In other words, the production rules corresponding to the handshaking expansion

$$F_k \;\equiv\; *\texttt{[[}v(X_k) \wedge Bt_k \longrightarrow yt_k\!\uparrow \;[\!]\, v(X_k) \wedge Bf_k \longrightarrow yf_k\!\uparrow\; \texttt{]};\texttt{[}n(X_k)\texttt{]}; yt_k\!\downarrow, yf_k\!\downarrow\; \texttt{]}$$

are given by

$$\begin{array}{lll} v(X_k) \wedge Bt_k \;\mapsto\; yt_k\!\uparrow & \qquad & n(X_k) \;\mapsto\; yt_k\!\downarrow \\ v(X_k) \wedge Bf_k \;\mapsto\; yf_k\!\uparrow & \qquad & n(X_k) \;\mapsto\; yf_k\!\downarrow \end{array}$$

and these rules have long series transistor chains when using delay-insensitive codes for X_k.

To eliminate these long series transistor chains, we can introduce an explicit state variable v_k as follows:

$$\begin{array}{ll} F_k \;\equiv & *\texttt{[[}v_k \wedge Bt_k \longrightarrow yt_k\!\uparrow \;[\!]\, v_k \wedge Bf_k \longrightarrow yf_k\!\uparrow\; \texttt{]};\texttt{[}\neg v_k\texttt{]}; yt_k\!\downarrow, yf_k\!\downarrow\texttt{]} \\ & \quad \| *\texttt{[[}v(X_k)\texttt{]}; v_k\!\uparrow; \texttt{[}n(X_k)\texttt{]}; v_k\!\downarrow\texttt{]} \end{array}$$

which leads to production rules:

$$\begin{array}{lll} v_k \wedge Bt_k \;\mapsto\; yt_k\!\uparrow & \qquad & \neg v_k \;\mapsto\; yt_k\!\downarrow \\ v_k \wedge Bf_k \;\mapsto\; yf_k\!\uparrow & \qquad & \neg v_k \;\mapsto\; yf_k\!\downarrow \end{array}$$

$$\begin{array}{l} v(X_k) \;\mapsto\; v_k\!\uparrow \\ n(X_k) \;\mapsto\; v_k\!\downarrow \end{array}$$

The reason this is an improvement is that, as discussed, the validity and neutrality tests are distributive, and can therefore be decomposed into a tree of gates. Also, if some gates in the validity test are common across different values of k, we can share those gates as well. Note, of course, that as in the adder example, we only need to ensure that *all* the outputs being valid implies that v_k is **true**; i.e., the check v_k need only appear on some of the outputs, not all of them.

A drawback of this method is that we have added to the number of sequential steps before the output becomes valid because we first have to set v_k before the output can change. Another problem arises because even if the condition Bt_k implies that the input is valid (i.e. $Bt_k \Rightarrow v(X_k)$), we still have to check v_k in the rule for $yt_k\!\uparrow$ otherwise the transition $v_k\!\uparrow$ would not be acknowledged resulting in an instability. Therefore, this transformation could *increase* the number of series transistors in some production rules.

5.1.5. Alternative Compliations: Forwarded Validity Check

To avoid both these problems, we would ideally like to not check v_k in the production rule for $yt_k\!\uparrow$ or $yf_k\!\uparrow$. We can do so by changing the output codeword produced by the function block. In other words, instead of producing a dual-rail output (yt_k, yf_k), we produce three output rails: (yt_k, yf_k, v_k). The validity test for this code is given by $(yt_k \vee yf_k) \wedge v_k$, and the neutrality is given by $\neg yt_k \wedge \neg yf_k \wedge \neg v_k$. This moves the problem of checking validity into the environment. With this transformation, we can use a number of different production rules:

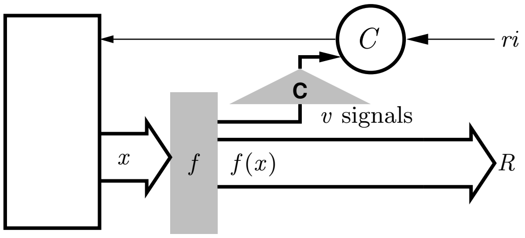


Figure 5.5. Modified function block compilation.

$$\begin{array}{rcl@{\qquad\qquad}rcl} Bt_k & \mapsto & yt_k\uparrow & v(X_k) & \mapsto & v_k\uparrow \\ Bf_k & \mapsto & yf_k\uparrow & n(X_k) & \mapsto & v_k\downarrow \\ \\ Ct_k & \mapsto & yt_k\downarrow \\ Cf_k & \mapsto & yf_k\downarrow \end{array}$$

We have quite a bit of freedom in picking the guards Ct_k and Cf_k. One choice would be to use $\neg Bt_k$ and $\neg Bf_k$ respectively, or a partial neutrality test. All that has to be guaranteed is that the outputs (yt_k, yf_k) will be reset if they were set by the upgoing phase of the handshake, and that they will not be reset until the inputs start becoming neutral.

The environment is now faced with the output dual-rail code *and* n additional validity signals. We can break this problem down into detecting two completions: one of the dual-rail code itself (as before), and another of the v_k signals. A simple way to handle this problem is to modify Figure 5.2 to Figure 5.5 as shown above. The outputs are partitioned into two components: Y (as before), and the new v_k signals. The v_k signals are all combined into a completion signal using a standard completion tree, while the Y signals are sent to the environment. The time taken by the completion tree shown in the figure will overlap with the time taken by the completion tree at the receiver, thereby minimizing the cycle time impact of this transformation.

5.1.6. Alternative Compliations: Merged Register/Function

The reason the validity and neutrality tests lead to a number of series transistors is because we have introduced new variables whose transitions need to be acknowledged. The compilation of process $*[R' \bullet R!f(x)]$ was replaced by the compilation of *XMIT* and a function block. Instead, we could compile this process directly without the introduction of the *XMIT* process.

If we follow the steps for the standard compilation of $*[C \bullet D!x]$, the only difference occurs in the process:

```
$*[[ci \wedge x \longrightarrow dto\uparrow; [\neg ci]; dto\downarrow$
  $[] ci \wedge \neg x \longrightarrow dfo\uparrow; [\neg ci]; dfo\downarrow$
```

```
    ]]
```

This normally turns into the sender part of a register. For the function block compilation, this process is replaced by:

```
    *[[$ci \wedge ft(x) \longrightarrow dto\uparrow$; [$\neg ci$]; $dto\downarrow$
     ▯$ci \wedge f\!f(x) \longrightarrow dfo\uparrow$; [$\neg ci$]; $dfo\downarrow$
     ]]
```

The production rules for this implementation follow from the production rules of a register. The rules are of the form:

$$\begin{array}{lcl@{\qquad\qquad}lcl} ci \wedge ft_k(x) & \mapsto & dto_k\uparrow & \neg ci & \mapsto & dto_k\downarrow \\ ci \wedge f\!f_k(x) & \mapsto & dfo_k\uparrow & \neg ci & \mapsto & dfo_k\downarrow \end{array}$$

While such an implemention is not amenable to composition (discussed in the next section), it is an efficient way to implement functions that do not need to be decomposed into component operations.

5.1.7. Composing Function Blocks

Computing a complex function with a single function block may lead to inefficient circuits with many transistors in series. However, computation can usually be partitioned into smaller function blocks that can be composed in various ways to compute the original function. This section discusses some function block composition topologies.

A linear composition of function blocks can be used to implement simple function composition. Given the task of computing $z := g(f(x))$, we can implement the function $g \circ f$ by taking the output of the function block f and connecting it to the input of the function block g. This is a valid function block implementation. To see this, let y be the intermediate result, i.e., $f(x)$. Then, we know that $v(z) \Rightarrow v(y)$ from the function block property for g, and that $v(y) \Rightarrow v(x)$ from the function block property for f. Using a similar argument for neutrality and the transitions from valid to neutral and neutral to valid shows that the function block property is preserved.

Consider the design of a simple up-shifter circuit. The shifter takes two inputs: a value to be shifted, and the amount by which the value is to be shifted. The CHP for the shifter is given by:

```
    *[ $A?x, C?a;\ \ B!(2^a x \ \ mod \ \ 2^n)$ ]
```

where n is the number of bits in the output. The function block implementation of this shifter would result in a complex circuit, since the output at bit-position k potentially depends on the inputs at bit-positions 0, 1, …, k depending on the value of a.

Instead of implementing a monolithic function block, we can decompose the function computation into several smaller function blocks that can be composed together to implement the original function. For the shifter, we decompose the function computation into k stages, where k is the number of bits used to represent a. Stage i performs a conditional shift by 2^i, for $0 \leq i < k$.

5.1.8. Quick Decision

5.1.9. State-Holding Function Blocks

$FREG \equiv *[[n(X)];\ Y := oldX;\ [v(X)];\ Y \Downarrow;\ oldX := X]$

5.2. Bus Design

Busses are a natural way to organize datapaths when high throughput is not required. A bus is a circuit that uses a shared set of wires to communicate among a collection of processes. At the CHP level, busses can arise naturally after control/data decomposition. The following shows control/data decomposition being applied to a CHP process:

$$*[\ L?x;\ \ ...;\ \ L?y;\ \ ...\ ]$$
$$\triangleright$$
$$*[\ Lx;\ \ ...;\ \ Ly;\ \ ...\ ]\ \ \|\ \ *[\ Lx \bullet L?x\ ]\ \ \|\ \ *[\ Ly \bullet L?y\ ]$$

The fact that L is shared among two datapath processes makes it a shared channel—and therefore, a bus. In this case, L is a shared input port. Similarly, one could have a shared output port as well. Figure 5.6 shows the general structure of a bus with n sender and m receiver processes that all share a communication channel. As the communication channel itself is shared, the production rules for the channel will include rules from a number of processes. When we consider the compilation of individual processes, we must therefore take this into account.

The bus shown in Figure 5.6 can be written as follows:

$$(\|\ k : n : *[\ S_k \bullet X!v_k\ ])\ \ \|\ \ (\|\ k : m : *[R_k \bullet X?w_k])$$

where the channel X is shared across all the processes. We have several choices in terms of active and passive protocols: X could have an active sender, passive receiver, or vice versa; S_k could be an active or a passive handshake; R_k could be an active or a passive handshake.

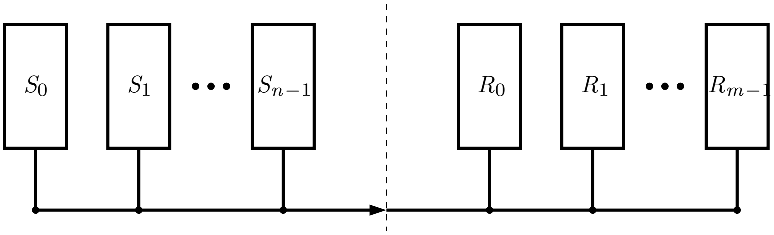


Figure 5.6. Bus with n senders and m receivers using a shared channel.

5.2.1. Passive Sender

We examine both options: a passive control handshake on the control channel, as well as an active control handshake. Note that in the former case, we will not be able to probe the bus from the sender. In both cases, the bus handshake ($X!v_k$) is given by:

$$[xi]\,;\,[vt_k \longrightarrow xto{\uparrow} \mathbin{[\!]} vf_k \longrightarrow xfo{\uparrow}]\,;\,[\neg xi]\,;\,xto{\downarrow}, xfo{\downarrow}$$

where (vt_k, vf_k) are private dual-rail variables, but the channel variables xto, xfo, and xi are shared.

Passive Control Handshake. The control handshake (S_k) is given by

$$[si_k]\,;\,so_k{\uparrow};\,[\neg si_k]\,;\,so_k{\downarrow}$$

The interleaved handshaking expansion for the bus process is given by:

$$*[[si_k \wedge xi]\,;\,[vt_k \longrightarrow xto{\uparrow} \mathbin{[\!]} vf_k \longrightarrow xfo{\uparrow}]\,;\,so_k{\uparrow};\,[\neg si_k \wedge \neg xi]\,;\,xto{\downarrow}, xfo{\downarrow};\,so_k{\downarrow}]$$

While it would be simple to turn this handshaking expansion into production rules, care must be taken because the x-variables are shared by multiple senders. In addition to checking the production rules, we must be sure that the operations on the bus variables shared among processes cannot possibly interfere with each other even at the handshaking expansion level. In the reshuffling shown above, $so_k{\downarrow}$ is the last action, and the x-variables are not modified after $so_k{\downarrow}$ (at least, not until the following bus request). Therefore, the control can guarantee mutual exclusion on access to the bus by waiting for $so_k{\downarrow}$ before beginning the next bus transaction. The production rules are given by:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
si_k \wedge xi \wedge vt_k & \mapsto & xto{\uparrow} & si_k \wedge (xto \vee xfo) & \mapsto & so_k{\uparrow} \\
\neg si_k \wedge \neg xi & \mapsto & xto{\downarrow} & \neg xto \wedge \neg xfo & \mapsto & so_k{\downarrow} \\
\\
si_k \wedge xi \wedge vf_k & \mapsto & xfo{\uparrow} \\
\neg si_k \wedge \neg xi & \mapsto & xfo{\downarrow}
\end{array}$$

Note that the signal $so_k{\uparrow}$ is guarded by si_k. We need this additional guard because xto and xfo are shared; simply using ($xto \vee xfo$) to set so_k high would cause so_k to go high when some other sender accessed the bus—an invalid transition.

When n sender processes are considered, the production rules are given by:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
(\vee k :: si_k \wedge vt_k) \wedge xi & \mapsto & xto{\uparrow} & (k :: si_k \wedge (xto \vee xfo) & \mapsto & so_k{\uparrow}) \\
(\wedge k :: \neg si_k) \wedge \neg xi & \mapsto & xto{\downarrow} & (k :: \neg xto \wedge \neg xfo & \mapsto & so_k{\downarrow}) \\
\\
(\vee k :: si_k \wedge vf_k) \wedge xi & \mapsto & xfo{\uparrow} \\
(\wedge k :: \neg si_k) \wedge \neg xi & \mapsto & xfo{\downarrow}
\end{array}$$

This is the circuit required for one bit of data; to extend the synthesis to N bits of data, the production rules for xto and xfo would be replicated per bit. Additionally, the rules for $so_k{\uparrow}$ and $so_k{\downarrow}$ would change to:

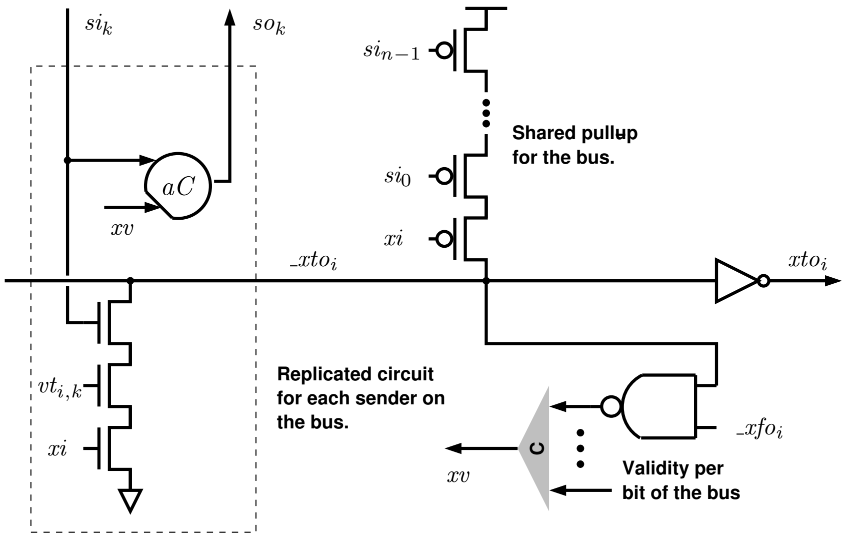

si_k
so_k
aC
xv
si_{n-1}
Shared pullup for the bus.
si_0
xi
$\_xto_i$
xto_i
$vt_{i,k}$
xi
Replicated circuit for each sender on the bus.
C
$\_xfo_i$
xv
Validity per bit of the bus

Figure 5.7. Sender circuit I, passive bus, passive control.

$$
\begin{array}{rclcrcl}
(k :: si_k \wedge xv & \mapsto & so_k\uparrow) & \qquad & (\wedge i : N :: (xto_i \vee xfo_i)) & \mapsto & xv\uparrow \\
(k :: \neg xv & \mapsto & so_k\downarrow) & \qquad & (\wedge i : N :: \neg xto_i \wedge \neg xfo_i) & \mapsto & xv\downarrow
\end{array}
$$

where xv is a new signal that checks that the bus data is valid. The circuit for the bus is shown in Figure 5.7, with $(vt_{i,k}, vf_{i,k})$ referring to the ith bit for the kth sender process.

There are several problems with this production rule set, as is evident from the circuit implementation. There are large series chains of transistors, and the shared bus wires have three series transistors driving them. Ideally, we would like to have a bus that has only a single pull-up transistor and at most two n-transistors in series for the pull-down.

The first step is to combine the xi and si_k together. An examination of the handshaking expansion shows that the waits on xi and si_k always appear in pairs. Therefore, we can rewrite the original HSE as follows:

$$
\begin{array}{ll}
 & *[[si_k \wedge xi]; [vt_k \longrightarrow xto\uparrow \,[]\, vf_k \longrightarrow xfo\uparrow]; so_k\uparrow; [\neg si_k \wedge \neg xi]; xto\downarrow, xfo\downarrow; so_k\downarrow] \\
\triangleright & \\
 & *[[si_k \wedge xi]; s_k\uparrow; [\neg si_k \wedge \neg xi]; s_k\downarrow] \\
\| & *[[s_k]; [vt_k \longrightarrow xto\uparrow \,[]\, vf_k \longrightarrow xfo\uparrow]; so_k\uparrow; [\neg s_k]; xto\downarrow, xfo\downarrow; so_k\downarrow]
\end{array}
$$

The bus production rules obtained by this transformation are shown below, and the corresponding circuit is shown in Figure 5.8.

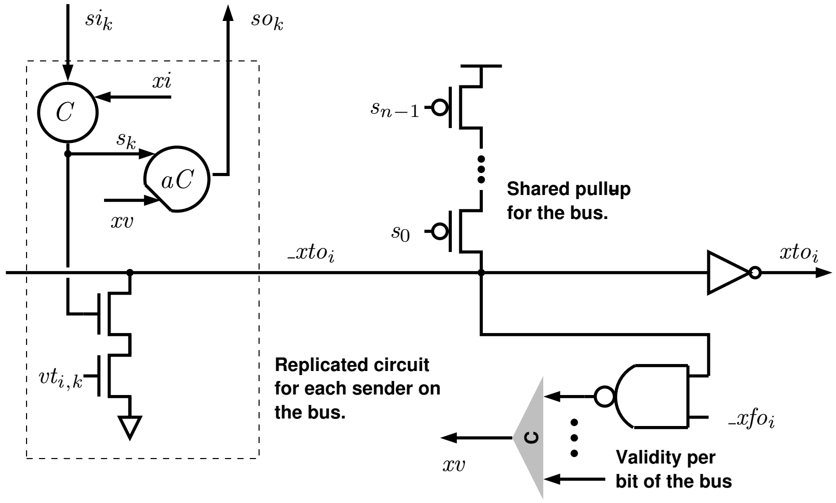

si_k
so_k
xi
C
s_k
aC
xv
$\_xto_i$
s_{n-1}
s_0
Shared pullup for the bus.
xto_i
$vt_{i,k}$
Replicated circuit for each sender on the bus.
$\_xfo_i$
C
xv
Validity per bit of the bus

Figure 5.8. Sender circuit II, passive bus, passive control.

$$\begin{array}{rclcrcl}
(\vee k :: s_k \wedge vt_k) & \mapsto & xto\uparrow & \qquad & (k :: s_k \wedge xv & \mapsto & so_k\uparrow) \\
(\wedge k :: \neg s_k) & \mapsto & xto\downarrow & & (k :: \neg xv & \mapsto & so_k\downarrow) \\
& & & & & & \\
(\vee k :: s_k \wedge vf_k) & \mapsto & xfo\uparrow & & (k :: si_k \wedge xi & \mapsto & s_k\uparrow) \\
(\wedge k :: \neg s_k) & \mapsto & xfo\downarrow & & (k :: \neg si_k \wedge \neg xi & \mapsto & s_k\downarrow)
\end{array}$$

$$\begin{array}{rcl}
(\wedge i : N :: (xto_i \vee xfo_i)) & \mapsto & xv\uparrow \\
(\wedge i : N :: \neg xto_i \wedge \neg xfo_i) & \mapsto & xv\downarrow
\end{array}$$

A problem with this bus reshuffling is the series transistor chain in the pull-up for the bus. In the first Caltech processor, this bus reshuffling was used with a passive pull-up (effectively a pseudo-NMOS style gate) instead of a series transistor chain. Another option is to introduce a new variable (sv) that checks that all the s_k signals are low and to use that variable to reset the bus. The modified handshaking expansion is:

$$*[[s_k];sv\uparrow;[vt_k \longrightarrow xto\uparrow \,[\,]\, vf_k \longrightarrow xfo\uparrow];so_k\uparrow;[\neg s_k];sv\downarrow;xto\downarrow,xfo\downarrow;so_k\downarrow]$$

Unfortunately, it should be clear that the production rule for $xto\uparrow$ would have to check si_k, sv, as well as vt_k because we have to wait for $sv\uparrow$ (shared) because of the order of actions in the HSE, while we must check si_k so that we know which sender should write the bus (the problem is caused by xto being shared). We are back to three transistors in series for the bus pull-down! Also, the reshuffling increases the latency from $si_k\uparrow$ to $xto\uparrow$ because we have to wait for $sv\uparrow$ before the bus is modified. Instead, we postpone the check for $sv\uparrow$ by using the following HSE:

$$*[[s_k];(sv\uparrow,[vt_k \longrightarrow xto\uparrow \,[\,]\, vf_k \longrightarrow xfo\uparrow]);so_k\uparrow;[\neg s_k];sv\downarrow;xto\downarrow,xfo\downarrow;so_k\downarrow]$$

We cannot postpone the check for $sv\uparrow$ further since $so_k\uparrow$ will cause s_k to change, which is used to

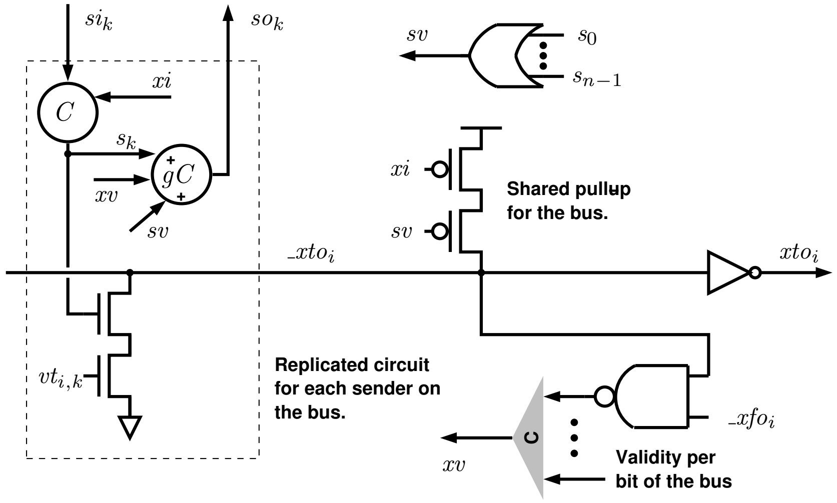


Figure 5.9. Sender circuit III, passive bus, passive control.

set sv. The production rules are:

$$\begin{array}{rcl@{\qquad}rcl}
(\vee k :: s_k) & \mapsto & sv\uparrow & (\wedge i : N :: (xto_i \vee xfo_i)) & \mapsto & xv\uparrow \\
(\wedge k :: \neg s_k) & \mapsto & sv\downarrow & (\wedge i : N :: \neg xto_i \wedge \neg xfo_i) & \mapsto & xv\downarrow \\
\\
(\vee k :: s_k \wedge vt_k) & \mapsto & xto\uparrow & (k :: sv \wedge s_k \wedge xv & \mapsto & so_k\uparrow) \\
\neg xi \wedge \neg sv & \mapsto & xto\downarrow & (k :: \neg xv & \mapsto & so_k\downarrow) \\
\\
(\vee k :: s_k \wedge vf_k) & \mapsto & xfo\uparrow & (k :: si_k \wedge xi & \mapsto & s_k\uparrow) \\
\neg xi \wedge \neg sv & \mapsto & xfo\downarrow & (k :: \neg si_k \wedge \neg xi & \mapsto & s_k\downarrow)
\end{array}$$

Notice that we have to check $\neg xi$ in the bus production rules to avoid interference between $xto\uparrow$ and $xto\downarrow$. The circuit for this bus reshuffling is shown in Figure 5.9.

If we are interested in optimizing area, the obvious target is the validity check xv. The bus validity is being checked twice: once in the sender, to generate xv, and once in the receiver to generate the xi signal. Can we use xi to replace the function of xv? To answer this question, we examine the HSE again:

$$*[[s_k];(sv\uparrow, [vt_k \longrightarrow xto\uparrow [] vf_k \longrightarrow xfo\uparrow]); so_k\uparrow; [\neg s_k]; sv\downarrow; xto\downarrow, xfo\downarrow; so_k\downarrow]$$

The xv check was introduced because $so_k\uparrow$ happened *after* the bus was set. Instead, we could execute $so_k\uparrow$ in parallel with the actions that set the bus. Finally, we can also reset the bus in parallel with setting $so_k\downarrow$. The modified HSE is:

$$*[[s_k];((sv\uparrow; so_k\uparrow), [vt_k \longrightarrow xto\uparrow [] vf_k \longrightarrow xfo\uparrow]); [\neg s_k]; ((sv\downarrow; xto\downarrow, xfo\downarrow), so_k\downarrow)]$$

Note that $so_k\downarrow$ is no longer the last action in the HSE, so we might have a problem with mutual exclusion on access to the bus. However, since $s_k\uparrow$ (for all k) waits for xi to be **true**, we can be

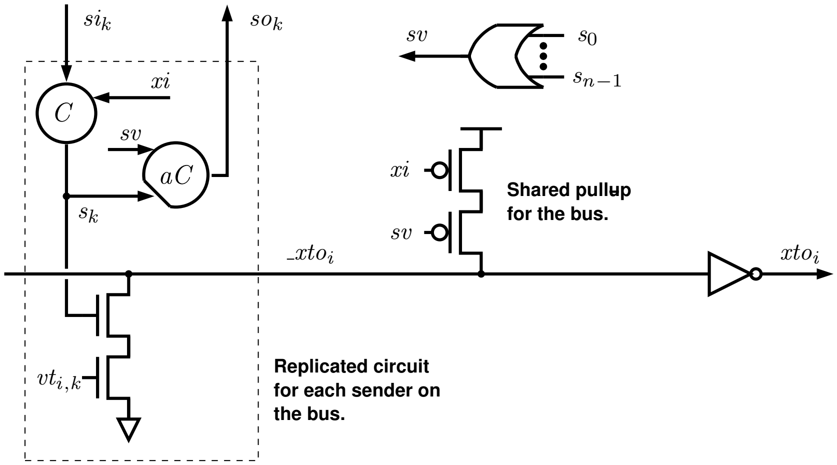


Figure 5.10. Sender circuit IV, passive bus, passive control.

sure that the bus has reset before the next bus transaction can begin. The new production rules (without xv) are:

$$\begin{array}{rcl@{\qquad}rcl}
(\vee k :: s_k) & \mapsto & sv\uparrow & (k :: sv \wedge s_k & \mapsto & so_k\uparrow) \\
(\wedge k :: \neg s_k) & \mapsto & sv\downarrow & (k :: \neg s_k & \mapsto & so_k\downarrow) \\
\\
(\vee k :: s_k \wedge vt_k) & \mapsto & xto\uparrow & (k :: si_k \wedge xi & \mapsto & s_k\uparrow) \\
\neg xi \wedge \neg sv & \mapsto & xto\downarrow & (k :: \neg si_k \wedge \neg xi & \mapsto & s_k\downarrow) \\
\\
(\vee k :: s_k \wedge vf_k) & \mapsto & xfo\uparrow \\
\neg xi \wedge \neg sv & \mapsto & xfo\downarrow
\end{array}$$

The circuit for this bus reshuffling is shown in Figure 5.10. Another possibility is to reorder the so_k and sv operations as follows:

$$*[[s_k];((so_k\uparrow;sv\uparrow),[vt_k \longrightarrow xto\uparrow \,[\!]\, vf_k \longrightarrow xfo\uparrow]);[\neg s_k];so_k\downarrow;sv\downarrow;xto\downarrow,xfo\downarrow]$$

In this reshuffling, sv is generated by an OR-gate that has the so_k signals as inputs. The production rules would be:

$$\begin{array}{rcl@{\qquad}rcl}
(\vee k :: so_k) & \mapsto & sv\uparrow & (k :: s_k & \mapsto & so_k\uparrow) \\
(\wedge k :: \neg so_k) & \mapsto & sv\downarrow & (k :: sv \wedge \neg s_k & \mapsto & so_k\downarrow) \\
\\
(\vee k :: s_k \wedge vt_k) & \mapsto & xto\uparrow & (k :: si_k \wedge xi & \mapsto & s_k\uparrow) \\
\neg xi \wedge \neg sv & \mapsto & xto\downarrow & (k :: \neg si_k \wedge \neg xi & \mapsto & s_k\downarrow) \\
\\
(\vee k :: s_k \wedge vf_k) & \mapsto & xfo\uparrow \\
\neg xi \wedge \neg sv & \mapsto & xfo\downarrow
\end{array}$$

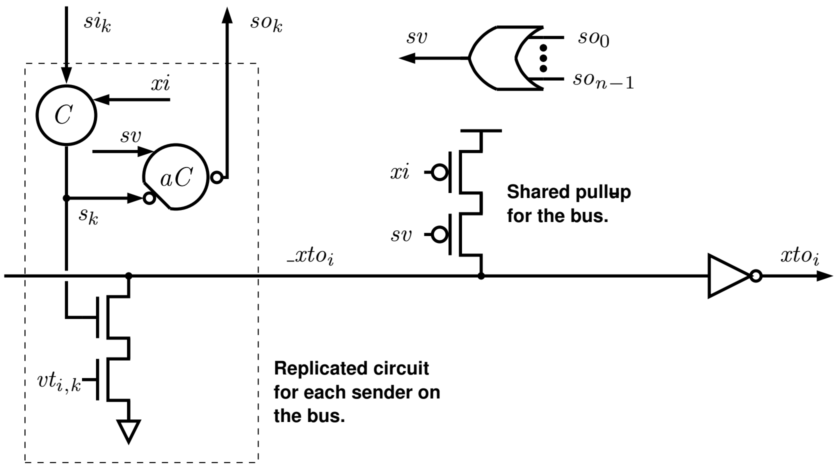


Figure 5.11. Sender circuit V, passive bus, passive control.

which result in the circuit in Figure 5.11. This reshuffling is better because we do not wait for $sv{\uparrow}$ before raising so_k; the acknowledge for the send control is quicker than in the circuit shown in Figure 5.10. We could also change the reshuffling so that we wait for $so_k{\uparrow}$ before setting the bus variables; the result of this would be that the selection transistor for the bus pull-down would be gated by so_k instead of s_k.

Active Control Handshake. If S_k uses an active protocol, the control handshake is

$so_k{\uparrow};\ [si_k];\ so_k{\downarrow};\ [\neg si_k]$

Unfortunately, S_k being active is problematic because the usual reshuffling

$*[[xi];\ so_k{\uparrow};\ [si_k];\ [vt_k \longrightarrow xto{\uparrow} \,[]\, vf_k \longrightarrow xfo{\uparrow}];\ [\neg xi];\ so_k{\downarrow};\ [\neg si_k];\ xto{\downarrow}, xfo{\downarrow}]$

cannot be used because xi is shared. If we have multiple senders, then all the so_k signals will go high in parallel once xi goes high. These transitions will not be acknowledged, because only one sender has permission to access the bus. In this case, it makes more sense to have a single, *shared* so signal that is a shared acknowledge signal from the bus to the control. The net effect is that the bus control channels S_0, …, S_{n-1} are implemented with $(n+1)$ wires, and resemble a single communication channel S that can send n different values encoded with a 1-of-n code. We discuss this new control interface to the busses in Section XXX.

5.2.2. Active Receiver

For an active receive on the bus, we have two choices for the control handshake: active or passive. In both cases, the bus handshake $(X?w_k)$ is given by:

$xo{\uparrow};\ [xti \longrightarrow wf_k{\downarrow}; wt_k{\uparrow} \,[]\, xfi \longrightarrow wt_k{\downarrow}; wf_k{\uparrow}];\ xo{\downarrow};\ [\neg xti \wedge \neg xfi]$

where (wt_k, wf_k) are the private dual-rail register variables, and xo, xti, and xfi are shared bus variables.

Active Control Handshake. If R_k uses an active protocol, the handshake for the control is given by:

$$ro_k{\uparrow};\ [ri_k];\ ro_k{\downarrow};\ [\neg ri_k]$$

Unfortunately, R_k being active is problematic for the same reasons that S_k being active is problematic. The net result is the implementation should use a shared ro signal, and n mutually exclusive ri_k signals. We discuss this new control interface to the busses in Section XXX.

Passive Control Handshake. The control handshake (R_k) is given by:

$$[ri_k];\ ro_k{\uparrow};\ [\neg ri_k];\ ro_k{\downarrow}$$

An interleaved handshake for the bus process is shown below.

$$*[[ri_k];\ xo{\uparrow};\ [xti \longrightarrow wf_k{\downarrow};\ wt_f{\uparrow} [\!] xfi \longrightarrow wt_k{\downarrow};\ wf_k{\uparrow}];\ ro_k{\uparrow};\ [\neg ri_k];\ xo{\downarrow};\ [\neg xti \wedge \neg xfi];\ ro_k{\downarrow}]$$

Instead of directly compiling this handshake, we adopt the same strategy as in control/data decomposition. We introduce an explicit register process. To enforce the selection of a specific register, we introduce a new variable r_k that guards the write to the register. The "register" process introduced is:

$$\begin{array}{l}
*[[r_k \wedge xti \longrightarrow wf_k{\downarrow};\ wt_k{\uparrow};\ wa_k{\uparrow};\ [\neg xti];\ wa_k{\downarrow} \\
\quad [\!]\, r_k \wedge xfi \longrightarrow wt_k{\downarrow};\ wf_k{\uparrow};\ wa_k{\uparrow};\ [\neg xfi];\ wa_k{\downarrow} \\
\ \ ]]
\end{array}$$

where the signal wa_k is the write acknowledge signal. The production rules for this process can be obtained from those of the register, and are given by:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
r_k \wedge xti & \mapsto & wf_k{\downarrow} & r_k \wedge (xti \wedge wt_k \vee xfi \wedge wf_k) & \mapsto & wa_k{\uparrow} \\
\neg xfi \wedge \neg wt_k & \mapsto & wf_k{\uparrow} & \neg xti \wedge \neg xfi & \mapsto & wa_k{\downarrow} \\
\\
r_k \wedge xfi & \mapsto & wt_k{\downarrow} \\
\neg xti \wedge \neg wf_k & \mapsto & wt_k{\uparrow}
\end{array}$$

Note the insertion of r_k to guard the write acknowledge signal as well as the write to the register. We do not need to check r_k before setting wf_k (or wt_k) high because in the cases when the register is not selected, those production rule firings are vacuous.

The bus handshake is given by

$$*[[ri_k];\ r_k{\uparrow};\ xo{\uparrow};\ [wa_k];\ ro_k{\uparrow};\ [\neg ri_k];\ r_k{\downarrow};\ xo{\downarrow};\ [\neg wa_k];\ ro_k{\downarrow}]$$

and the production rules for an individual process are given by:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
ri_k & \mapsto & r_k{\uparrow} & wa_k & \mapsto & ro_k{\uparrow} \\
\neg ri_k & \mapsto & r_k{\downarrow} & \neg wa_k & \mapsto & ro_k{\downarrow} \\
\\
r_k & \mapsto & xo{\uparrow} \\
\neg r_k & \mapsto & xo{\downarrow}
\end{array}$$

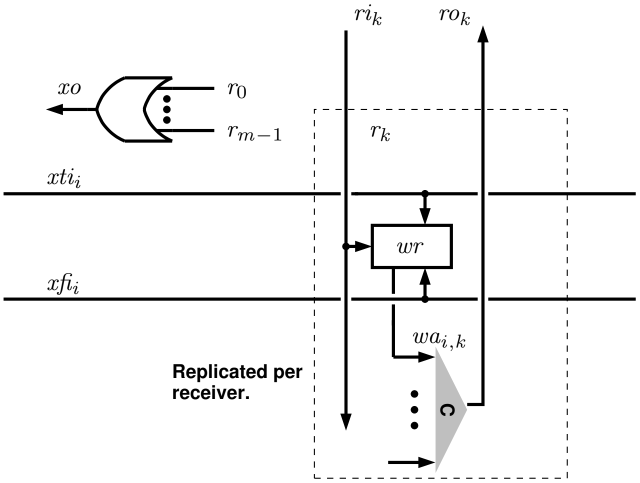


Figure 5.12. Receiver circuit I, active bus, passive control.

When these rules are generalized for m receivers, we obtain:

$$(k :: ri_k \;\mapsto\; r_k\uparrow) \qquad (k :: wa_k \;\mapsto\; ro_k\uparrow)$$
$$(k :: \neg ri_k \;\mapsto\; r_k\downarrow) \qquad (k :: \neg wa_k \;\mapsto\; ro_k\downarrow)$$

$$(\vee k :: r_k) \;\mapsto\; xo\uparrow$$
$$(\wedge k :: \neg r_k) \;\mapsto\; xo\downarrow$$

When the circuit is generalized to N bits in the datapath, the write acknowledge signals for each bit must be combined into a single write acknowledge signal per receiver using a completion tree. The generalized circuit is shown in Figure 5.12.

A problem with this circuit is that the control acknowledge ro_k waits for the bus before generating the acknowledge. This prevents any overlap between the control handshake and bus handshake. To eliminate this problem, we can change the resuffling to:

$$*[[ri_k]; r_k\uparrow; ro_k\uparrow; xo\uparrow; [wa_k \wedge \neg ri_k]; r_k\downarrow; ro_k\downarrow; xo\downarrow; [\neg wa_k]]$$

Unfortunately, there is a problem with using this reshuffling. The problem arises because $wa_k\downarrow$ is not checked until the *next* handshake on R_k. The guard for $wa_k\downarrow$ is $\neg xti \wedge \neg xfi$, and since $wa_k\downarrow$ is not checked before $ro_k\downarrow$ or $xo\downarrow$, the rule will be unstable. The problem is caused by the shared bus variables. We could change the reshuffling so that some transition occurs after $[\neg wa_k]$. Another option is to combine all the wa_k signals into a single wa using an m-way OR gate. Instead, we examine the option where we introduce a *shared* write acknowledge wa for the set of registers that are written by the bus. This has the added benefit of eliminating m completion trees (see Figure 5.12), replacing them with a single completion tree. The rules for the write acknowledge are:

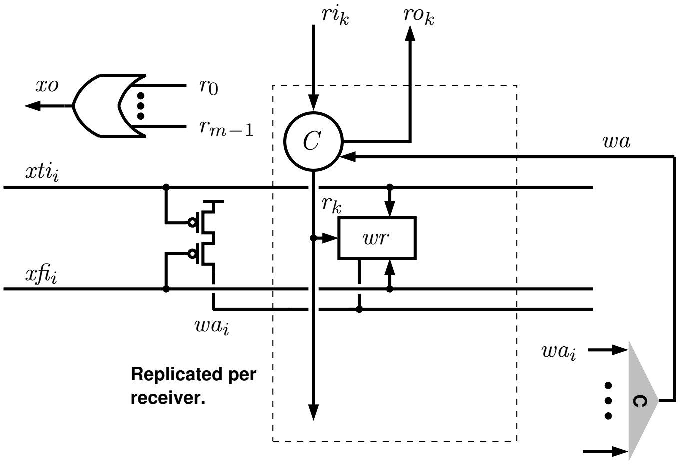


Figure 5.13. Receiver circuit II, active bus, passive control.

$$\begin{aligned}(\vee k :: r_k \wedge (xti \wedge wt_k \vee xfi \wedge wf_k)) &\mapsto wa\uparrow \\ \neg xti \wedge \neg xfi &\mapsto wa\uparrow\end{aligned}$$

This write acknowledge (per bit) is then combined into a single write acknowledge through a completion tree, and the bus control reshuffling is transformed into:

$$*[[ri_k]; r_k\uparrow; ro_k\uparrow; xo\uparrow; [wa \wedge \neg ri_k]; r_k\downarrow; ro_k\downarrow; xo\downarrow; [\neg wa]]$$

which is safe since wa is a shared signal and is checked by every bus control process before accessing the bus. The production rules are:

$$\begin{aligned}(k :: \neg wa \wedge ri_k &\mapsto r_k\uparrow) & (\vee k :: ro_k) &\mapsto xo\uparrow \\ (k :: wa \wedge \neg ri_k &\mapsto r_k\downarrow) & (\wedge k :: \neg ro_k) &\mapsto xo\downarrow\end{aligned}$$

$$\begin{aligned}(k :: r_k &\mapsto ro_k\uparrow) \\ (k :: \neg r_k &\mapsto ro_k\downarrow)\end{aligned}$$

The circuit for this production rule set is shown in Figure 5.13.

5.2.3. Active Sender

As an active control handshake for a sender bus that is active will lead to the same conclusions as examining an active control handshake in the case of a passive sender, we will simply examine reshuffings that use a passive control handshake.

The bus handshake ($X!v_k$) is given by:

$$[vt_k \longrightarrow xto\uparrow [] vf_k \longrightarrow xfo\uparrow]; [xi]; xto\downarrow, xfo\downarrow; [\neg xi]$$

and the control handshake is given by

$$[si_k]; so_k\uparrow; [\neg si_k]; so_k\downarrow$$

The reshuffled HSE is

$$*[[si_k]; [vt_k \longrightarrow xto\uparrow [] vf_k \longrightarrow xfo\uparrow]; [xi]; so_k\uparrow; [\neg si_k]; xto\downarrow; xfo\downarrow; [\neg xi]; so_k\downarrow]$$

This reshuffling matches the one used in standard control/data decomposition. The production rules are:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
si_k \wedge vt_k & \mapsto & xto\uparrow & \neg si_k & \mapsto & xto\downarrow \\
si_k \wedge vf_k & \mapsto & xfo\uparrow & \neg si_k & \mapsto & xfo\downarrow \\
 & & & & & \\
si_k \wedge xi & \mapsto & so_k\uparrow & \neg xi & \mapsto & so_k\downarrow
\end{array}$$

To generalize these rules, the pull-down for *xto* and *xfo* would have to check that all the si_k signals were **false**. The generalized rules are:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
(\vee k :: si_k \wedge vt_k) & \mapsto & xto\uparrow & (\wedge k :: \neg si_k) & \mapsto & xto\downarrow \\
(\vee k :: si_k \wedge vf_k) & \mapsto & xfo\uparrow & (\wedge k :: \neg si_k) & \mapsto & xfo\downarrow \\
 & & & & & \\
\multicolumn{3}{l}{(k :: si_k \wedge xi \ \mapsto\ so_k\uparrow)} & \multicolumn{3}{l}{(k :: \neg xi \ \mapsto\ so_k\downarrow)}
\end{array}$$

This particular reshuffling tightly synchronizes the sender control and data channel handshake, which is exactly what was required for control/data decomposition. However, in the case of a bus, decoupling part of the synchronization can improve performance.

HERE

$$*[[si]; [vt \longrightarrow xto\uparrow [] vf \longrightarrow xfo\uparrow]; [xi]; so\uparrow; xto\downarrow; [\neg xi \wedge \neg si]; so\downarrow\ ]$$

Production rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
si \wedge vt \wedge \neg xi & \mapsto & xto\uparrow & xi \wedge si & \mapsto & so\uparrow \\
so & \mapsto & xto\downarrow & \neg xi \wedge \neg si & \mapsto & so\downarrow
\end{array}$$

Hmm...

5.2.4. Passive Receiver

As an active control handshake for a receiver bus that is active will lead to the same conclusions as examining an active control handshake in the case of a passive receiver, we will simply examine reshufflings that use a passive control handshake.

$$\begin{array}{l}
*[[xti \wedge ri]; wti\uparrow; [\neg xti]; wti\downarrow] \quad \| \quad *[[xfi \wedge ri]; wfi\uparrow; [\neg xfi]; wfi\downarrow] \\
\| \\
*[[wa]; xo\uparrow; ro\uparrow; [\neg ri \wedge \neg wa]; xo\downarrow; ro\downarrow]
\end{array}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
wa & \mapsto & xo\uparrow & \neg wa \wedge \neg ri \wedge ro & \mapsto & xo\downarrow \\
wa \wedge xo & \mapsto & ro\uparrow & \neg xo & \mapsto & ro\downarrow
\end{array}$$

Another version, with shared write acknowledge:

$*[[wa \wedge ri]\,; xo{\uparrow}; ro{\uparrow};\ [\neg wa \wedge \neg ri]\,; xo{\downarrow}; ro{\downarrow}]$

$$\begin{array}{rcl@{\qquad\qquad}rcl} wa & \mapsto & xo{\uparrow} & \neg wa \wedge \neg ri \wedge ro & \mapsto & xo{\downarrow} \\ ri \wedge xo & \mapsto & ro{\uparrow} & \neg xo & \mapsto & ro{\downarrow} \end{array}$$

We consider the compilation of $*[C \bullet D?x]$ where D is a shared passive port and C is active. As usual, we assume that the data on D is encoded using a dual rail encoding, and that D carries one bit of information. The handshake protocol on channel C is given by:

$co{\uparrow};\ [ci]\,; co{\downarrow};\ [\neg ci]$

where C is implemented using a four-phase active handshake on pair (co, ci). The handshake protocol for channel D is given by:

$[dti \vee dfi]\,; do{\uparrow};\ [\neg dti \wedge \neg dfi]\,; do{\downarrow}$

where D is implemented using a four-phase passive handshake on variables (dti, dfi, do). In addition, we need to introduce the statement

$[dti \longrightarrow x{\uparrow} \,[\!]\, dfi \longrightarrow x{\downarrow}]$

that actually sets the variable x. The normal interleaving of the two handshakes used for control/data decomposition is given by:

$*[[dti \vee dfi]\,; co{\uparrow};\ [ci \wedge dti \longrightarrow x{\uparrow} \,[\!]\, ci \wedge dfi \longrightarrow x{\downarrow}]\,; do{\uparrow};\ [\neg dti \wedge \neg dfi]\,; co{\downarrow};\ [\neg ci]\,; do{\downarrow}]$

Unfortunately, this is not correct because the variables *dti* and *dfi* are shared. Therefore, multiple *co* signals could go high from different data part processes operating concurrently! Unfortunately, we cannot raise *co* before the wait for $(dti \vee dfi)$, because if D is being probed the probe would become **true** even without a communication being pending on D. The only way out of this dilemma is to introduce a new signal *dp* that is shared, and corresponds to the probe $\overline{D}$. This is the signal that must be inspected by the control if it probes D. We introduce:

$*[[dti \vee dfi]\,; dp{\uparrow};\ [\neg dti \wedge \neg dfi]\,; dp{\downarrow}]$

and completely eliminate variable *co*! The control process behaves as follows:

$[dp]\,; ci_k{\uparrow};\ [\neg dp]\,; ci_k{\downarrow}$

with different ci_k signals that correspond to different receive operations, and a single shared *dp* variable that serves as the acknowledge for every receive on the shared channel D. The data part can be implemented with the following handshake:

$*[[ci_k \wedge dti \longrightarrow x_k{\uparrow} \,[\!]\, ci_k \wedge dfi \longrightarrow x_k{\downarrow}]\,; do{\uparrow};\ [\neg ci_k]\,; do{\downarrow}]$

The production rules for this process contain the write part of the register as before. The signal *do* is generated by combining the individual write acknowledge signals from the registers using an OR-gate.

5.3. Register File

Blah blah blah.

References

The function block implementation is from a paper on datapath design by Martin.[30] The section on pipelined control/data decomposition borrows heavily from Manohar et al.[20,21]

Chapter 6.

Dataflow Synthesis

This chapter examines aggressive design techniques that enable highly pipelined implementations of asynchronous computations. We will discuss both circuit techniques for high-performance operation as well as CHP-level transformations that enable dataflow-style decomposition of asynchronous designs. The main theoretical tool we will use is the projection theorem (4.3).

6.1. Pipelined Function Computation

A multi-stage function block introduces multiple stages of computation from the inputs becoming valid to the outputs becoming valid. While this increases latency, it does not allow multiple data items to be processed in each function block stage. This set of transformations introduces slack on the output channels as discussed in Section 4.1.3. Given a function block of the form $f_{n-1}(\cdots(f_0(x))))$, we can decompose it into processes

$$(\| \; i :: *[\; X_i?x; X_{i+1}!f_i(x) \;])$$

This corresponds to traditional pipelining, where the datapath computation is broken up into multiple pipeline stages so as to improve the throughput of the system.

However, we face a general problem when communicating n bits of data: control/data decomposition makes it impossible to make the communication constant response time due to the introduction of an n-input completion tree. To avoid this problem, we can use additional process decomposition before applying control/data decomposition. These techniques for improving datapath performance are discussed in the following chapter.

6.2. Aggressive Datapath Design

In designing a delay-insensitive system, we face a problem when attempting to design datapaths where the quantities being manipulated are composed of a large number of bits. The problem is illustrated by examining the circuit implementation of the following program:

$$*[\ L?x;\ \ R!x\ ]$$

Before we send value x on channel R, we must be sure that all the bits used to represent x have been received on channel L. The circuit that waits for all the bits to have been received has a cycle time that is proportional to $\log N$, where N is the number of bits. As a result, as we increase the number of bits in x, the system throughput will decrease.

Instead, we examine an alternative implementation strategy. We implement channel L using an array of $\Theta(N)$ channels, where the individual channels have a fixed number of bits. As a result, we transform the program shown above into:

$$*[\ (\|i :: L[i]?x[i]);\ \ (\|i :: R[i]!x[i])\ ]$$

We have moved the performance problem from the implementation of the communication action on a channel to the implementation of the semicolon that separates the L and R actions. However, we observe that there is no data-dependency between channels $L[i]$ and $R[j]$ when $i \neq j$. For the rest of this section we assume that the system is locally slack elastic, so we can use projection-based transformations.

Applying projection to eliminate the synchronization, we obtain the following collection of N processes.

$$(\|i :: *[\ L[i]?x[i];\ \ R[i]!x[i\ ])$$

Observe that we now have $\Theta(N)$ *independent* processes, and increasing N will not affect the throughput of the system.

This transformation can be generalized into a technique for control-data decomposition. Standard techniques for the decomposition of a process into control and data consist of replacing actions $D!x$ and $D?x$ by Ds and Dr, and introducing processes

$$*[\ Ds \bullet D!x\ ]\ \|\ *[\ Dr \bullet D?x\ ]$$

If x is an N-bit binary integer, then once again the cycle time of communication actions on the D channels would be $\Theta(\log N)$.

Instead, we apply the following transformation. Let I_i be the possible channels that write to variable x, and let O_i be the possible channels that read from variable x. We replace actions $I_i?x$ with $C!(\mathbf{true}, i); D1$, and $O_i!x$ with $C!(\mathbf{false}, i); D1$. We introduce the following processes:

$$*[\ C?(b,k);\ \ [b \longrightarrow I_k?x\ \ [\!]\neg b \longrightarrow O_k!x];\ \ D2\ ]\ \|\ *[\ D2; D1\ ]$$

This transformation does not affect the correctness of the computation. By splitting up the input and output channels into an array of $\Theta(N)$ channels that carry a constant number of bits of data, we can transform the computation as follows:

```
   $(\|i :: *[\ C[i]?(b,k);\ \ [b \longrightarrow I_k[i]?x[i]\ \ []\neg b \longrightarrow O_k[i]!x[i]];D2[i]])$
$\|\ \ *[\ (\|\ i :: D2[i]);\ \ (\|\ i :: D1[i])\ ]$
```

and modify $C!(b,k); D1$ into $(\|i :: C[i]!(b,k)); (\|\ i :: D1[i])$.

The parallel control distribution $(\|i :: C[i]!(b,k))$ can be converted into a control distribution *tree*, where the value (b,k) is copied from one channel to $\Theta(N)$ leaves. We will call the control distribution tree *copytree*(R,C), where R is the root of the tree, and C is the array of channels above. The computation is equivalent to:

```
   $(\|i :: *[\ C[i]?(b,k);\ \ [b \longrightarrow I_k[i]?x[i]\ \ []\neg b \longrightarrow O_k[i]!x[i]];D2[i]])$
$\|\ \ *[\ (\|\ i :: D2[i]);\ \ (\|\ i :: D1[i])\ ]$
$\|\ \ copytree(R,C)$
```

where we replace $(\|i :: C[i]!(b,k))$ with $R!(b,k)$. Now, we are ready to introduce slack on channels $D1[i]$ and $D2[i]$. When we let the slack on these channels go to infinity, we are left with:

```
   $(\|i : *[\ C[i]?(b,k);\ \ [b \longrightarrow I_k[i]?x[i]\ \ []\neg b \longrightarrow O_k[i]!x[i]]\ \ ])$
$\|\ \ copytree(R,C)$
```

and the main control distribution turns into $R!(b,k)$. If we examine the net effect of this transformation, we observe that all semicolons that wait for $\Theta(N)$ actions to complete have been eliminated. Therefore, the throughput of the system no longer depends on N—a significant improvement. We have increased the latency of control distribution (although asymptotically the latency of the control distribution is still $\Theta(\log N)$, we have increased constant factors). However, we should not fail to observe that we have introduced communication on channel R; the number of bits sent on R is $\Theta(\log(|I|+|O|))$, where $|I|$ is the number of channels that write to variable x, and $|O|$ is the number of channels that read from variable x.

6.3. Memory Design

We apply the ideas from projection to design a pipelined memory. We begin the design with a sequential specification for the memory given by the following process:

```
$MEM(RW, A, DI, DO) \equiv *[\ RW?r,\ \ A?a;$
                             $[r \longrightarrow DO!mem[a]$
                             $[]\neg r \longrightarrow DI?d; mem[a] := d$
                             $]$
                           $]$
```

The memory has a control input channel *RW* that specifies whether the operation is a read or a write, an address channel A that specifies the address being read or written, and two data channels

DI and *DO* used to provide the data that is to be written to the memory or the data that is read from the memory.

6.3.1. Banking

The slow operation is the access to the "*mem*[·]" array. To speed up this operation, we cut the array into two halves. For simplicity, let the array size be 2^N. We can divide the memory into arrays *memH* and *memL* as follows:

```
$MEM_N(RW, A, DI, DO) \equiv *[\ RW?r,\ \ A?a;$
                              $[r \longrightarrow [a_{N-1} \longrightarrow DO!memH[a_{N-2..0}]$
                                         $\square \neg a_{N-1} \longrightarrow DO!memL[a_{N-2..0}]$
                                         $]$
                              $\square \neg r \longrightarrow DI?d;\ [a_{N-1} \longrightarrow memH[a_{N-2..0}] := d$
                                                  $\square \neg a_{N-1} \longrightarrow memL[a_{N-2..0}] := d$
                                                  $]$
                              $]$
                            $]$
```

The operations on *memH* and *memL* are read and write operations, and these can be implemented by two processes of type MEM_{N-1}. We implement this by introducing additional channels to communicate with these instances of MEM_{N-1} as follows:

$$MEM_N(RW, A, DI, DO) \equiv$$
$$BLK_N(RW, A, DI, DO, RWH, AH, DIH, DOH, RWL, AL, DIL, DOL)$$
$$\|\ \ MEM_{N-1}(RWH, AH, DIH, DOH)$$
$$\|\ \ MEM_{N-1}(RWL, AL, DIL, DOL)$$

```
$BLK_N(RW, A, DI, DO, RWH, AH, DIH, DOH, RWL, AL, DIL, DOL) \equiv$
     $*[\ RW?r,\ \ A?a;$
        $[r \longrightarrow [a_{N-1} \longrightarrow RWH!r, AH!a_{N-2..0}; DOH?d; DO!d$
                   $\square \neg a_{N-1} \longrightarrow RWL!r, AL!a_{N-2..0}; DOL?d; DO!d$
                   $]$
        $\square \neg r \longrightarrow DI?d;\ [a_{N-1} \longrightarrow RWH!r, AH!a_{N-2..0}, DIH!d$
                             $\square \neg a_{N-1} \longrightarrow RWL!r, AL!a_{N-2..0}, DIL!d$
                             $]$
        $]$
      $]$
```

In this transformation, we replaced operations on the *memH* and *memL* arrays with communication actions with the processes used to implement the arrays respectively.

Write operations to the two sub-arrays can now be overlapped, because the BLK process can process the next memory operation as soon as the DIH/DIL communication is finished. This communication can complete before thet memory has done the write. However, reads to seperate sections of the memory cannot be overlapped because the DO communication needs to data from DOH/DOL. To improve the concurrency, we use projection to extract communication actions DOH, DOL, and DO from the BLK process. We need to replicate the value of a_{N-1}, so we introduce a copy of it and send it to the projected process via channel C as follows:

```
$BLK_N \equiv$ *[ $RW?r,\ \ A?a;$
           [$r \longrightarrow C!a_{N-1},$ [$a_{N-1} \longrightarrow RWH!r, AH!a_{N-2..0}$
                              []$\neg a_{N-1} \longrightarrow RWL!r, AL!a_{N-2..0}$
                              ]
           []$\neg r \longrightarrow DI?d;$ [$a_{N-1} \longrightarrow RWH!r, AH!a_{N-2..0}, DIH!d$
                             []$\neg a_{N-1} \longrightarrow RWL!r, AL!a_{N-2..0}, DIL!d$
                             ]
           ]
         ]
       || *[ $C?b;$  [$b \longrightarrow DOH?d; DO!d$ []$\neg b \longrightarrow DOL?d; DO!d$] ]
```

The two parts of the outer selection statement resemble each other. To make them almost identical, we extract the DI, DIH, and DIL actions as well by introducing another copy of a_{N-1} on control channel D:

```
$BLK_N \equiv$ *[ $RW?r,\ \ A?a;$
           [$r \longrightarrow C!a_{N-1},$ [$a_{N-1} \longrightarrow RWH!r, AH!a_{N-2..0}$
                              []$\neg a_{N-1} \longrightarrow RWL!r, AL!a_{N-2..0}$
                              ]
           []$\neg r \longrightarrow D!a_{N-1},$ [$a_{N-1} \longrightarrow RWH!r, AH!a_{N-2..0}$
                              []$\neg a_{N-1} \longrightarrow RWL!r, AL!a_{N-2..0}$
                              ]
           ]
         ]
       || *[ $C?b;$  [$b \longrightarrow DOH?d; DO!d$ []$\neg b \longrightarrow DOL?d; DO!d$ ] ]
       || *[ $D?b, DI?d;$  [$b \longrightarrow DIH!d$ []$\neg b \longrightarrow DIL!d$ ] ]
```

Finally, note that we can rewrite the first CHP process in BLK_N as follows:

```
*[ $RW?r,\ \ A?a;$
```

```
      $[r \longrightarrow C!a_{N-1} ▯ \neg r \longrightarrow D!a_{N-1}]$,
      $[a_{N-1} \longrightarrow RWH!r, AH!a_{N-2..0}$  $▯ \neg a_{N-1} \longrightarrow RWL!r, AL!a_{N-2..0}$ ]
   ]
```

If the address input is provided using two separate channels, one containing the most significant bit *Amsb* and one containing the rest of the bits *Arest*, this can be re-written as follows:

```
   *[ $RW?r$,  $Amsb?am$,  $Arest?ar$;
      $[r \longrightarrow C!am ▯ \neg r \longrightarrow D!am]$,  $[am \longrightarrow RWH!r, AH!ar ▯ \neg am \longrightarrow RWL!r, AL!ar]$
   ]
```

which can be further decomposed using projection into:

```
   *[ $RW?r, Amsb?am; AC!am$;
      $[r \longrightarrow C!am ▯ \neg r \longrightarrow D!am]$, $[am \longrightarrow RWH!r ▯ \neg am \longrightarrow RWL!r]$
   ]
   ||
   *[ $Arest?ar, AC?am$; $[am \longrightarrow AH!ar ▯ \neg am \longrightarrow AL!ar]$ ]
```

The full implementation of BLK_N is given by:

```
   *[ $RW?r, Amsb?am; AC!am$;
      $[r \longrightarrow C!am ▯ \neg r \longrightarrow D!am]$, $[am \longrightarrow RWH!r ▯ \neg am \longrightarrow RWL!r]$
   ]
   ||  *[ $Arest?ar, AC?am$; $[am \longrightarrow AH!ar$  $▯ \neg am \longrightarrow AL!ar$ ] ]
   ||  *[ $C?b$;  $[b \longrightarrow DOH?d$  $▯ \neg b \longrightarrow DOL?d$ ]$; DO!d$ ]
   ||  *[ $D?b, DI?d$;  $[b \longrightarrow DIH!d$  $▯ \neg b \longrightarrow DIL!d$ ] ]
```

6.3.2. Line Buffers

Another approach to speeding up access to the memory is to amortize the cost of the "*mem*[·]" operation over multiple accesses. Instead of *mem* being a one-dimensional array of size 2^N, suppose we think of it as being an array of (small) arrays, each array being of size 2^k, and *mem* being an array with 2^{N-k} entries. The modified CHP is shown below:

```
$MEM_{N,k}(RW, A, DI, DO) \equiv$ *[ $RW?r$,  $A?a$;
                              $[r \longrightarrow x := mem[a_{N-1..k}]$;  $DO!x[a_{k-1..0}]$
                              $▯ \neg r \longrightarrow DI?d; x := mem[a_{N-1..k}]; x[a_{k-1..0}] := d$;
                                          $mem[a_{N-1..k}] := x$
                              ]
                           ]
```

6.4. Common Reshufflings

$PCHB \equiv *[[Re \wedge L]; R\Uparrow; Le{\downarrow}; [\neg Re]; R\Downarrow; [\neg L]; Le{\uparrow}]$

$WCHB \equiv *[[Re \wedge L]; R\Uparrow; Le{\downarrow}; [\neg Re \wedge \neg L]; R\Downarrow; Le{\uparrow}]$

$PCFB \equiv *[[Re \wedge L]; R\Uparrow; Le{\downarrow}; en{\downarrow}; ([\neg Re]; R\Downarrow), ([\neg L]; Le{\uparrow}); en{\uparrow}]$

$PCEHB \equiv *[[Re]; en{\uparrow}; [L]; R\Uparrow; Le{\downarrow}; [\neg Re]; en{\downarrow}; R\Downarrow; [\neg L]; Le{\uparrow}]$

$PCHBA \equiv *[[Re \wedge L]; R\Uparrow; Le{\downarrow}; [\neg Re]; R\Downarrow, ([\neg L]; Le{\uparrow})]$

Blah blah.

6.4.1. Split and Merge

The "split" function block is given by:

$*[[v(X)]; [c0 \longrightarrow Y0\Uparrow \;[]\; c1 \longrightarrow Y1\Uparrow]; [n(X) \wedge n(c)]; Y0\Downarrow, Y1\Downarrow]$

The "merge" fblock is:

$*[[c0 \wedge v(X0) \longrightarrow Y\Uparrow \;[]\; c1 \wedge v(X1) \longrightarrow Y\Uparrow]; [n(c) \wedge n(X0) \wedge n(X1)]; Y\Downarrow]$

6.5. Register Files: Pipelined Mutual Exclusion

A register file can be thought of as a number of registers that use shared read and write ports that are implemented by buses. In principle, we can already implement a register file given the circuits we know how to design. This section presents some new techniques for implementing certain features that are important when optimizing the control performance of register files. The technique is general, and is referred to as *pipelined mutual exclusion*.

Include TR here...

Chapter 7.

Metastability

Jean Buridan, a fourteenth century French philosopher posed the following paradox: if a dog was equidistant to two sources food[†], wouldn't it starve to death because it has no reason to pick one source of food over the other? The problem arises because the dog has to make a discrete decision—what food to eat—based on a continuous input—the initial position of the dog. What Buridan proposed was that such a decision cannot be made in a bounded amount of time. The problem is that the dog is in a state of unstable equilibrium—known as a *metastable state*—and may remain there for an arbitrary amount of time thereby starving.

We can restate this same paradox as the following problem in the context of circuit design. The two sources of food represent the two logic values, **false** (logic 0) and **true** (logic 1). These two logic values are represented by voltages *GND* and *Vdd* respectively. Given an intermediate voltage value between *Vdd* and *GND*, does it represent logic 0 or logic 1? Once again, we have to make a discrete—the logic value—based on a continuous input—the voltage.

A traditional manifestation of this particular problem in circuit design arises from a clocked processor's interrupt-handling mechanism. Computers interact with external devices by means of interrupts. An interrupt is an input that can arrive at any time, and a clocked processor must decide if an interrupt arrived during a particular clock cycle. The problem arises because the interrupt signal is not synchronized with the clock. The clocked storage element used to store the external asynchronous input (known as a *synchronizer*) may not reliably store a logic 0 or logic 1 because the asynchronous input might change in a way that violates timing requirements of the storage element. In such circumstances, the device is said to exhibit *synchronization failure*,

[†] This paradox was re-stated as the paradox of Buridan's ass, where an ass is placed between two equidistant bales of hay.

because the synchronizer failed to store a logic 0 or a logic 1.

An interesting observation is that in each of the examples considered above, the problem is not one of making a "right" decision; rather, the problem is one of making the decision itself! In the case of the dog, it does not matter what food the dog chooses to eat.

Logic designers long denied the existence of synchronization failure. Many schemes were proposed to "solve" the problem, but they only reduced the probability of synchronization failure or moved the problem to another part of the system. While this phenomenon was discovered by several engineers in the 1960s, the work done by Chancey and Molnar[4] provided a convincing demonstration of the existence of the problem.

7.1. Arbitration

Consider the following CHP program:

```
*[[$\overline{X} \longrightarrow X?x$
  |$\overline{Y} \longrightarrow Y?x$
  ];
  $Z!x$
 ]
```

While earlier chapters covered the compilation of CHP programs into production rules, we did not study the compilation of processes with non-deterministic selections. In the program shown above, when $\overline{X}$ and $\overline{Y}$ are both **true** we have to pick one of them and execute the appropriate branch of the selection statement. To compile processes of this type, we introduce a special process known as an *arbiter*.

The arbiter problem is formalized as follows: Given two Boolean-valued variables x and y that could be either **false** or **true**, design a circuit with two Boolean-valued outputs u and v that determine which of the two inputs are **true**. The arbiter can be described by the following handshaking expansion:

```
*[[$a \longrightarrow u\uparrow$; [$\neg a$]; $u\downarrow$
  |$b \longrightarrow v\uparrow$; [$\neg b$]; $v\downarrow$
 ]]
```

The arbiter does a handshake on (a, u) and (b, v), and as such could be described as the CHP program:

```
*[[$\overline{A} \longrightarrow A$
  |$\overline{B} \longrightarrow B$
 ]]
```

where A is implemented by wires (a, u), and B is implemented by wires (b, v). As an attempt at

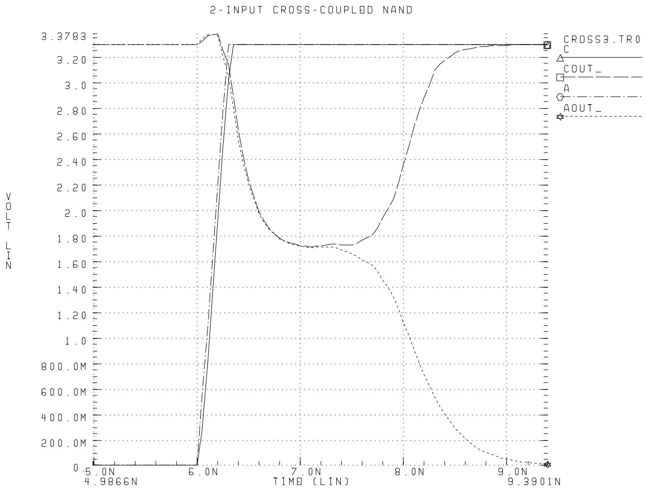


Figure 7.1. Behavior of two cross-coupled NAND gates.

writing production rules, we write down the rules for the HSE assuming the selection is deterministic. The rules are given by:

$$\begin{array}{rcl@{\qquad\qquad}rcl} a \wedge \neg v & \mapsto & u\uparrow & b \wedge \neg u & \mapsto & v\uparrow \\ \neg a \vee v & \mapsto & u\downarrow & \neg b \vee u & \mapsto & v\downarrow \end{array}$$

To make the circuit directly implementable, we invert the sense of variables u and v, and obtain:

$$\begin{array}{rcl@{\qquad\qquad}rcl} a \wedge \_v & \mapsto & \_u\downarrow & b \wedge \_u & \mapsto & \_v\downarrow \\ \neg a \vee \neg\_v & \mapsto & \_u\uparrow & \neg b \vee \neg\_u & \mapsto & \_v\uparrow \end{array}$$

This circuit is a pair of cross-coupled NAND gates. We now examine the following question: what happens if both a and b go up at the same time? When both a and b are **true**, the circuit can be reduced to

$$\begin{array}{rcl@{\qquad\qquad}rcl} \_v & \mapsto & \_u\downarrow & \_u & \mapsto & \_v\downarrow \\ \neg\_v & \mapsto & \_u\uparrow & \neg\_u & \mapsto & \_v\uparrow \end{array}$$

which corresponds to a pair of cross-coupled inverters. This circuit can be analyzed using transistor physics to determine the behavior of signals $\_u$ and $\_v$. The HSPICE simulation of this circuit is shown in Figure 7.1.

Analysis reveals that the signals $\_u$ and $\_v$ will separate eventually. If $P(t)$ is the probability that the difference between the voltages of $\_u$ and $\_v$ is less than a fixed amount (say $Vdd/2$), we can show that $P(t) \propto e^{-t/\tau_0}$. The likelihood that it takes at least time t for the cross-coupled NAND gates to "decide" which output should be low decays exponentially with t. If $P(t) = e^{-t/\tau_0}/\tau_0$, then the average arbitration time is given by $\int_0^\infty te^{-t/\tau_0}/\tau_0\, dt = \tau_0$, which is a constant.

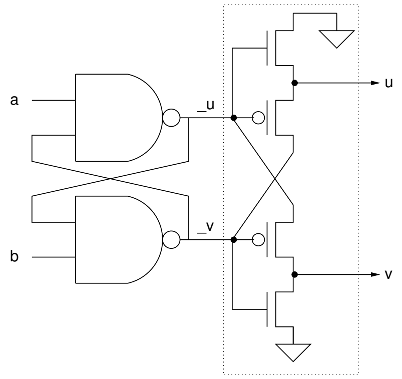


Figure 7.2. Arbiter, showing cross-coupled NAND gates and output filter.

Since we are connecting the arbiter to an asynchronous circuit, we can wait until the arbiter has determined which of $\_u$ or $\_v$. In other words, we can wait until the signals $\_u$ and $\_v$ separate before proceeding since on average this time interval is simply τ_0. However, we do not want the intermediate values of $\_u$ and $\_v$ to be directly connected to other parts of the circuit, because different gates could have different threshold voltages, and we can only be sure of the final values of $\_u$ and $\_v$ after the signals separate. Therefore, we connect the outputs of the cross-coupled NAND gates to a filter circuit that waits for the signals to be separated by at least a threshold voltage before its output changes. The cross-coupled NAND gate combined with the filter is our implementation of the arbiter, and is shown in Figure 7.2. The circuit symbol for the arbiter is shown in Figure 7.3.

7.1.1. Implementing Mutual Exclusion

As a first example of using the arbiter, consider the following CHP program.

```
*[[$\overline{A} \longrightarrow X \bullet A$
  |$\overline{B} \longrightarrow Y \bullet B$
 ]]
```

The purpose of this process is to implement a non-deterministic choice between two channels A and B. The output channels X and Y are used to indicate which input channel was selected by the process. Given a non-deterministic choice between two channels, this process can be used to

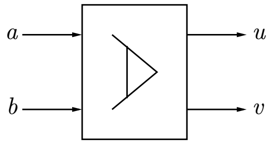


Figure 7.3. Circuit symbol for arbiter.

turn the non-deterministic choice into a deterministic one. We shall see examples of this in the following section.

Since A and B are probed, we implement them with a passive protocol. We implement X and Y with an active protocol, since we expect to connect this process to one that probes X and Y. The handshaking expansion for this process is given by:

```
$*[[ai \longrightarrow xo\uparrow; [xi]; ao\uparrow; [\neg ai]; xo\downarrow; [\neg xi]; ao\downarrow$
  $| bi \longrightarrow yo\uparrow; [yi]; bo\uparrow; [\neg bi]; yo\downarrow; [\neg yi]; bo\downarrow$
 ]]
```

Note that we have used a reshuffling that ensures that both halves of the handshake are tightly coupled (cf. Section 3.5.1). As the next step, we would like to introduce an explicit arbiter process thereby factoring out the non-determinism from the process. In order to do so, we introduce variables u and v that correspond to the output of the arbiter. Since these variables are set and reset whenever the arbiter input changes, we place the changes on u and v next to the changes in ai and bi. The resulting HSE is shown below.

```
$*[[ai \longrightarrow u\uparrow; [u]; xo\uparrow; [xi]; ao\uparrow; [\neg ai]; u\downarrow; [\neg u]; xo\downarrow; [\neg xi]; ao\downarrow$
  $| bi \longrightarrow v\uparrow; [v]; yo\uparrow; [yi]; bo\uparrow; [\neg bi]; v\downarrow; [\neg v]; yo\downarrow; [\neg yi]; bo\downarrow$
 ]]
```

The next step is to decompose the arbiter out of the handshaking expansion. This step resembles the transformations we applied in Section 4.2, where we broke apart handshaking expansions into concurrent parts. This transformation is known as *process factorization*. The rules for process factorization can be summarized as: (a) Each output is in exactly one process; (b) Split two actions if the semicolon between them is superfluous. Detailed semantic definitions of process factorization can be found in van der Goot's thesis.[7] After process factorization, we obtain the following handshaking expansion:

```
 $*[[ai \longrightarrow u\uparrow; [\neg ai]; u\downarrow$
   $| bi \longrightarrow v\uparrow; [\neg bi]; v\downarrow$
  ]]
$\|$
 $*[[u \longrightarrow xo\uparrow; [xi]; ao\uparrow; [\neg u]; xo\downarrow; [\neg xi]; ao\downarrow$
   $[] v \longrightarrow yo\uparrow; [yi]; bo\uparrow; [\neg v]; yo\downarrow; [\neg yi]; bo\downarrow$
  ]]
```

The first part corresponds to an arbiter, while the second part can be compiled using standard techniques that yield the following production rules:

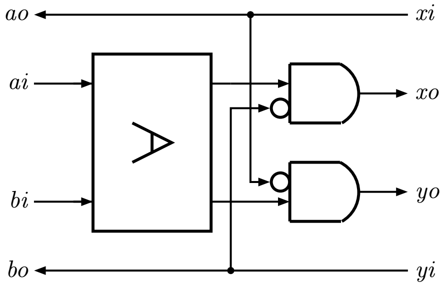


Figure 7.4. Compilation of basic mutual exclusion.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\neg bo \wedge u & \mapsto & xo\uparrow & \neg ao \wedge v & \mapsto & yo\uparrow \\
bo \vee \neg u & \mapsto & xo\downarrow & ao \vee \neg v & \mapsto & yo\downarrow \\
\\
xi & \mapsto & ao\uparrow & \neg yi & \mapsto & bo\downarrow \\
\neg xi & \mapsto & ao\downarrow & yi & \mapsto & bo\uparrow
\end{array}$$

The final circuit is shown in Figure 7.4.

7.1.2. Arbitration With Multiplexing

A common use of arbitration is to implement mutually exclusive access to a shared resource. Often the shared resource is of the form ($S/resource$) because it has been factored out of the original computation using process decomposition, and therefore the resource is controlled by channel S. Since we do not want to introduce shared channels, we introduce a special process that controls access to the shared resource. This process is shown below:

```
*[[$\overline{A} \longrightarrow S; A$
  |$\overline{B} \longrightarrow S; B$
 ]]
```

Channels A and B are passive since they are probed, and channel S is active because it is used to control the shared resource. The first step in the compilation is to replace the non-deterministic selection between A and B with a deterministic selection. This is achieved by the introduction of a mutual exclusion process compiled in the previous section. The result is:

```
 *[[$\overline{A} \longrightarrow P \bullet A$
   |$\overline{B} \longrightarrow Q \bullet B$
  ]]
$\|$
 *[[$\overline{P} \longrightarrow S; P$
   []$\overline{Q} \longrightarrow S; Q$
```

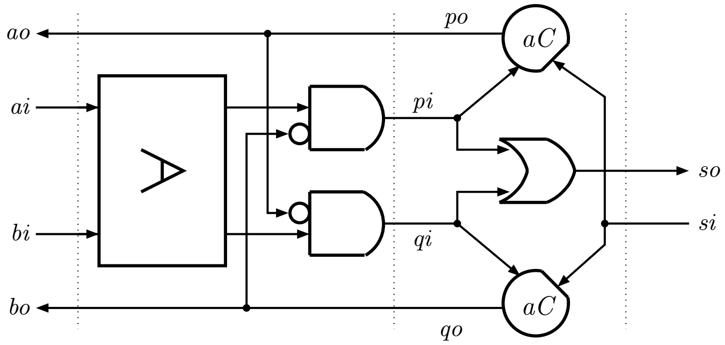


Figure 7.5. Arbitration with multiplexing.

```
    ]]
```

The handshaking expansion for the second process is given by:

```
    $*[[pi \longrightarrow so\uparrow; [si]; po\uparrow; [\neg pi]; so\downarrow; [\neg si]; po\downarrow$
      $[] qi \longrightarrow so\uparrow; [si]; qo\uparrow; [\neg qi]; so\downarrow; [\neg si]; qo\downarrow$
    ]]
```

and we can transform this directly into production rules to obtain:

$$\begin{array}{rcl@{\qquad\qquad}rcl} pi \vee qi & \mapsto & so\uparrow & si \wedge qi & \mapsto & qo\uparrow \\ \neg pi \wedge \neg qi & \mapsto & so\downarrow & \neg si & \mapsto & qo\downarrow \end{array}$$

$$\begin{array}{rcl} si \wedge pi & \mapsto & po\uparrow \\ \neg si & \mapsto & po\downarrow \end{array}$$

The result of the compilation is shown in Figure 7.5.

7.2. Evaluating A Probe

In this section we consider another type of non-deterministic process. The process involves negated probes, and is shown below.

```
    $*[[\overline{X} \wedge \overline{E} \longrightarrow E!\textbf{true}, X$
      $| \neg\overline{X} \wedge \overline{E} \longrightarrow E!\textbf{false}$
    ]]
```

This process can be used to determine the value of a probe by sending a **true** or **false** value on channel E. However, when the probe is **true**, the process completes the communication on the pending channel. Assuming the channels are passive, we get the following handshaking expansion:

```
    $*[[Xi \wedge Ei \longrightarrow Eto\uparrow; [\neg Ei]; Eto\downarrow; Xo\uparrow; [\neg Xi]; Xo\downarrow$
      $| \neg Xi \wedge Ei \longrightarrow Efo\uparrow; [\neg Ei]; Efo\downarrow$
    ]]
```

We use the fact that the CMOS implementation of a two-way arbiter is weakly fair. We eliminate

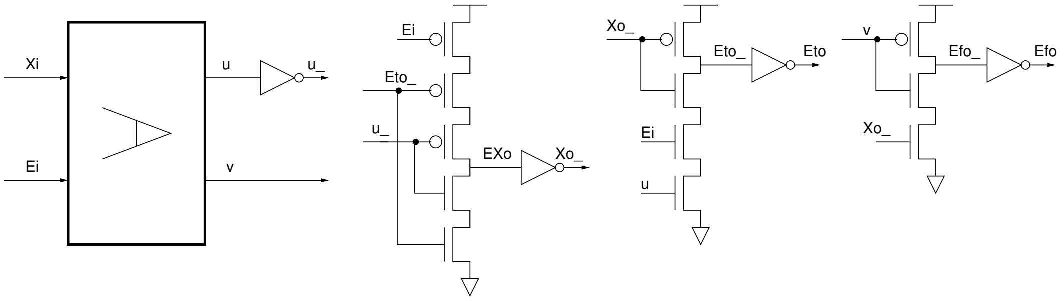


Figure 7.6. Evaluating the probe of a channel.

the check for $\neg Xi$ in the handshaking expansion for the second guarded command. The reason we can eliminate this check is that if Xi is true, the first alternative in the selection will execute eventually. The new handshaking expansion given by:

```
$*[[Xi \longrightarrow [Ei]; Eto\uparrow; [\neg Ei]; Eto\downarrow; Xo\uparrow; [\neg Xi]; Xo\downarrow$
  $| Ei \longrightarrow Efo\uparrow; [\neg Ei]; Efo\downarrow$
 $]]$
```

As before, we introduce variables u and v to model the arbitration that is required by the above handshaking expansion.

```
$*[[Xi \longrightarrow u\uparrow; [u]; [Ei]; Eto\uparrow; [\neg Ei]; Eto\downarrow; Xo\uparrow; [\neg Xi]; u\downarrow; [\neg u]; Xo\downarrow$
  $| Ei \longrightarrow v\uparrow; [v]; Efo\uparrow; [\neg Ei]; v\downarrow; [\neg v]; Efo\downarrow$
 $]]$
```

Applying process factorization, we obtain:

```
$*[[u \longrightarrow [Ei]; Eto\uparrow; [\neg Ei]; Eto\downarrow; Xo\uparrow; [\neg u]; Xo\downarrow$
  $▯ v \longrightarrow Efo\uparrow; [\neg v]; Efo\downarrow$
 $]]$
$\|$
$*[[Xi \longrightarrow u\uparrow; [\neg Xi]; u\downarrow$
  $| Ei \longrightarrow v\uparrow; [\neg Ei]; v\downarrow$
 $]]$
```

When reshuffling the first part of this process, we need to be careful because Ei changes in both guards of the selection statement, and Ei is an input to the arbiter. To ensure that the arbiter behaves correctly, we must ensure that Xi remains stable high until Ei is guaranteed to be low. Otherwise, the outputs of the arbiter could change in the middle of the handshaking expansion thereby leading to instability. The reshuffled HSE is given below.

```
$*[[u \longrightarrow [Ei]; Eto\uparrow; [\neg Ei]; Xo\uparrow; Eto\downarrow; [\neg u]; Xo\downarrow$
  $▯ v \longrightarrow Efo\uparrow; [\neg v]; Efo\downarrow$
```

```
    ]]
```

The production rules generated from this HSE are:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
u \wedge \_Xo \wedge Ei & \mapsto & \_Eto\downarrow & Xo & \mapsto & \_Xo\downarrow \\
\neg\_Xo & \mapsto & \_Eto\uparrow & \neg Xo & \mapsto & \_Xo\uparrow
\end{array}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\_Xo \wedge v & \mapsto & Efo\downarrow & u & \mapsto & \_u\downarrow \\
\neg v & \mapsto & Efo\uparrow & \neg u & \mapsto & \_u\uparrow
\end{array}$$

$$\begin{array}{rcl}
\neg\_u \wedge \neg\_Eto \wedge \neg Ei & \mapsto & Xo\uparrow \\
\_u \wedge \_Eto & \mapsto & Xo\downarrow
\end{array}$$

and the final circuit is shown in Figure 7.6.

7.3. Synchronizers

7.4. High Fanin Arbitration

7.5. Connecting Clocked and Asynchronous Systems

7.6. Case Study: Fair Arbitration with Unfair Arbiters

References

The output filter circuit of the arbiter is due to Seitz.[37] The process that evaluates the probe of a channel was designed by Manohar.[20] The synchronizer circuits were designed by Mika Nyström, Rajit Manohar, and Andrew Lines. The material on pipeline synchronization is due to Seizovic.[38]

Chapter 8.

Performance Analysis

8.1. Event-Rule Systems

8.2. Extended Event-Rule Systems

8.3. Pipeline Dynamics

8.4. Slack Matching

References

All the material in this chapter is a result of Steve Burns's seminal work on timing analysis.[3]

Chapter 9.

Limitations to Delay-Insensitivity

Various models of CMOS circuits are used to hide the electrical properties of transistors that would otherwise complicate the design process. These models typically assume that voltages represent Boolean values, and that a transistor can be thought of as a switch. Delay-insensitive circuit design assumes that the correct operation of a circuit is independent of the delay in operators and wires. In this chapter we show that the class of circuits that are entirely delay-insensitive is quite limited.

Quasi-delay-insensitive circuit design assumes that both operators and wires can take an arbitrary time to switch, except for certain wires that form isochronic forks. In this chapter we will examine the limits to implementing QDI circuits by determining the class of computations that are QDI.

9.1. Limitations to Purely Delay-Insensitive Circuits

Section 3.8.1 discussed some properties that were necessary for a circuit to be delay-insensitive (DI). In particular, any DI computation exhibited the acknowledgment property, restated below:

> **Acknowledgment Theorem.** *For a production rule set to be stable and non-interfering, every transition in the system that is followed by another transition that changes the same variable must be acknowledged.*

Figure 9.1 shows operators for variables x, y, and z. The variable x is an input to the operators for y and z. Since the production rule for x is stable, it satisfies the acknowledgment theorem and the output of a gate *must be acknowledged* by the output of some gate where x is an input.

In a truly DI circuit, adding arbitrary delay to any wire or gate in the computation does not change the behavior of the circuit. By the discussion in Section 3.8.1, it follows that *all* forks in a

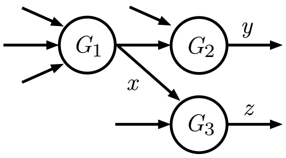


Figure 9.1. Three operators, with branches from x to y and z.

DI circuit are non-isochronic. In particular, a transition on x in Figure 9.1 must be acknowledged by a transition on *both* y and z. It follows that all up-going and down-going transitions on x must be acknowledged by transitions on y and z.

9.1.1. Unique Successor Sets

Recall the successor relation imposed by the computation, discussed in Section 3.8.1. Let $\langle t, k\rangle$ denote the kth occurrence of transition t, where a transition is $v\uparrow$ or $v\downarrow$ for a Boolean-valued variable v. Then we write $\langle t, k\rangle \prec \langle t', k'\rangle$ just when the kth occurrence of transition t is guaranteed to be executed before the k'th occurrence of transition t'.

Given a particular event $\langle t, k\rangle$, event $\langle t', k'\rangle$ is said to be its *immediate successor* if there is no event e satisfying $\langle t, k\rangle \prec e \prec \langle t', k'\rangle$. The *successor set* of a particular event is the set of variables that are modified by its immediate successors.

Example. If the immediate successors of $\langle x\uparrow, 0\rangle$ are $\langle y\downarrow, 1\rangle$ and $\langle z\uparrow, 2\rangle$, then the successor set of $\langle x\uparrow, 0\rangle$ is $\{y, z\}$ since those are the variables that are modified by the events that are its immediate successor. ▮

In a DI circuit of the form shown in Figure 9.1, it follows that the successor set of any transition of x must be $\{y, z\}$ because both y and z must acknowledge x, and they are clearly immediate successors of any x transition.

A computation is said to have the *unique successor set* (USS) property if every non-final transition on a variable has the same successor set.

Example. If the immediate successor of $\langle x\uparrow, 0\rangle$ is $\langle y\uparrow, 1\rangle$ and the immediate successor of $\langle x\downarrow, 0\rangle$ is $\langle z\uparrow, 1\rangle$, then the computation does not satisfy the USS property since the successor set of $\langle x\uparrow, 0\rangle$ is $\{y\}$, and the successor set of $\langle x\downarrow, 0\rangle$ is $\{z\}$. ▮

Theorem 9.1. (*USS Theorem*)

Any set of computations of a DI circuit has the USS property.

This result is a direct consequence of the discussion above. Since every transition in a DI circuit must be acknowledged by the outputs of all gates that use the variable, the successor sets of all non-final transitions on a particular variable are equal—which is precisely the property stated by Theorem 9.1.

Consider a computation that satisfies the USS property, and consider a variable y that occurs in the computation. If we examine the same computation but we pretend that the variable y is hidden, the computation still satisfies the USS property. This is because the successor relation is modified so that relations of the form $e \prec \langle y{\uparrow}, k\rangle$ are replaced with $e \prec e'$ where e' is an immediate successor of $\langle y{\uparrow}, k\rangle$. If the original computation satisfied the USS property, then the new computation will still satisfy it. This is stated formally below.

Theorem 9.2. (*Projection Theorem for the USS Property*)

If a set of computations has the USS property, the projection on a subset of variables has the USS property.

The combination of Theorem 9.1 and Theorem 9.2 is sufficient to prove that several computations cannot be implemented with circuits that are purely DI.

Example. The following handshaking expansion describes a simple one-place buffer.

$$\begin{array}{ll} & *[\,[xi]\,;xo{\uparrow};\,[\neg xi]\,;xo{\downarrow};\,yo{\uparrow};\,[yi]\,;yo{\downarrow};\,[\neg yi]\,] \\ \| & *[xi{\uparrow};\,[xo]\,;xi{\downarrow};\,[\neg xo]\,] \\ \| & *[\,[yo]\,;yi{\uparrow};\,[\neg yo]\,;yi{\downarrow}] \end{array}$$

If we project this computation onto variables $\{xo, yo\}$, the behavior of this buffer is captured by the following handshaking expansion:

$$*[xo{\uparrow};\,xo{\downarrow};\,yo{\uparrow};\,yo{\downarrow}]$$

This sequence of operations does not satisfy the USS property because the successor set of transitions of the type $xo{\uparrow}$ is $\{xo\}$, while the successor set of transitions of the type $xo{\downarrow}$ is $\{yo\}$. ▮

Example. Consider the following handshaking expansion, which describes the behavior of a two-input OR gate whose inputs are not both high simultaneously.

$$*[\;a{\uparrow};\,c{\uparrow};\,a{\downarrow};\,c{\downarrow};\,b{\uparrow};\,c{\uparrow};\,b{\downarrow};\,c{\downarrow}\;]$$

This computation does not satisfy the USS property, and therefore cannot be DI. ▮

9.1.2. The C-Element Theorem

It might come as a surprise to the reader that even simple circuits like certain buffers and OR gates cannot possibly have a DI implementation. In this section we show that the class of entirely DI circuits is very limited.

Given a computation that satisfies the USS property, how do individual gates in the computation behave? Consider a particular gate with multiple inputs i_0, ..., i_{n-1} and output o. If we project the computation onto the variables that are inputs to the gate and the gate output itself, the computation still satisfies the USS property (by Theorem 9.2). We also know that any transition on i_k is acknowledged by o for the original circuit to be DI. If we project the computation on i_k and o, it must be of the form

$$*[\ i_k := \neg i_k; o := \neg o\ ]$$

for the computation to satisfy the USS property. If multiple inputs change, then they must *all change* before o changes, otherwise different transitions on o would have different successor sets. This implies that the behavior of the computation can be given by the following handshaking expansion

$$*[\ (\| \, k :: i_k := \neg i_k); o := \neg o\ ]$$

for the USS property to be satisfied. This HSE corresponds to the behavior of a C-element! We formalize this conclusion in the following Theorem.

Theorem 9.3. (*C-element Theorem*)

Given a DI circuit in which each variable has at least three transitions, the circuit can be constructed with only C elements.

Therefore, almost every circuit of interest will not be entirely delay-insensitive because otherwise it could be constructed with just C-elements.

The key difference between circuits that are purely delay-insensitive and QDI circuits is that QDI circuits permit the presence of isochronic forks. Therefore, they need not satisfy the USS property because a transition on a variable need only be acknowledged by the output of one of gates that use the variable, not all the gates.

9.2. QDI Circuits

A circuit is said to be quasi-delay-insensitive if its correct operation is independent of the delays of gates and wires, except for certain wires that form isochronic forks.

We assume that the only way a QDI circuit can malfunction is if the output of any gate in the circuit glitches. If all gates are hazard-free, then we consider the circuit to be QDI. (An error in the design of a circuit may produce a QDI circuit that implements a different specification!) Stability and non-interference were the properties we used to ensure that a QDI circuit functioned correctly. The following Theorem shows that these properties are precisely the ones needed for correct operation under the QDI model.

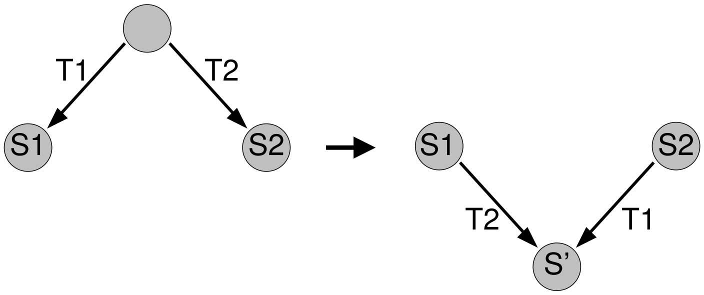


Figure 9.2. Strongly confluent computation.

Theorem 9.4. (*quasi-delay-insensitivity*)

A circuit is QDI if and only if the production rule set describing it is stable and non-interfering.

Proof: Suppose the production rule set is unstable. Then, there exists a gate represented by the production rules $B^+ \mapsto z\uparrow$ and $B^- \mapsto z\downarrow$ with an unstable production rule. Without loss of generality, there is a state in which $\neg z \wedge B^+$ holds, which is followed by a state in which $\neg z \wedge \neg B^+$ holds before z changes. Therefore, the output of the gate can glitch, which implies that the circuit is not QDI. If the production rule set is non-interfering, there can be a short-circuit.

Suppose the production rule set is stable and non-interfering. Consider a gate $B^+ \mapsto z\uparrow$ and $B^- \mapsto z\downarrow$. From stability, we know that if $\neg z \wedge B^+$ holds, then B^+ remains true until z changes. In other words, we cannot have a state in which $\neg z \wedge \neg B^+$ holds before z changes. Similarly, the transition $z\downarrow$ is also hazard-free, implying that the gate is hazard-free. Since every gate is hazard-free, the circuit is QDI. □

9.2.1. Confluence, Determinism, and Arbiters

In this section we examine some of the consequences of stability and non-interference, the two properties that characterize QDI computations. We use the following definition of strong confluence[16] to characterize QDI computations.

Definition 9.5. (*strong confluence*)

Let t_1 and t_2 be two transitions that can fire in state s. Let s_1 be the state obtained by firing t_1 in s, and s_2 be the state obtained by firing t_2 in s. The computation is said to be strongly confluent, if t_1 can fire in state s_2 and t_2 can fire in state s_1, and both alternatives lead to the same final state. (cf. Figure 9.2)

Theorem 9.6. (*strong confluence*)

A computation can be described by a stable, non-interfering production rule set if and only if it is strongly confluent.

Proof: Let $G_1 \mapsto t_1$ and $G_2 \mapsto t_2$ be two production rules that have effective firings in state s, i.e., $s \Rightarrow G_1 \wedge G_2 \wedge \neg R(t_1) \wedge \neg R(t_2)$. Now, t_1 cannot make G_2 false, since that would make the production rule unstable. Therefore, after t_1 fires, $G_2 \mapsto t_2$ can still fire. Similarly, $G_1 \mapsto t_1$ can fire after t_2 changes as well. Since all transitions are elementary assignments, the final state does not depend on the order of the two firings. Therefore, the computation is strongly confluent.

Conversely, suppose a computation is strongly confluent. For each transition t, define $G(t)$ as the disjunction of all the states in which the transition has an effective firing. Then we claim that the production rule set $\{G(t) \mapsto t \mid t \textit{ is a transition}\}$ is a stable, non-interfering production rule set that describes the computation. Let $G(t) \mapsto t$ be a production rule that has an effective firing in some state s. Firing any other production rule cannot disable t, since that would violate strong confluence. This implies that the rule $G(t) \mapsto t$ is stable. Since $G(t) \mapsto t$ is an arbitrary production rule, the production rule set is stable. The production rule set is non-interfering since both $x\uparrow$ and $x\downarrow$ cannot have an effective firing in a state s, which implies that $G(x\uparrow) \wedge G(x\downarrow) \equiv \textbf{false}$. Finally, the production rule set correctly describes the computation since, by construction, a transition is enabled in the production rule set if and only if the transition had an effective firing in the original computation. □

Theorem 9.6 does not directly rule out self-invalidating production rules. However, these rules can be systematically eliminated by the introduction of new variables. Let one of $B^+ \mapsto x\uparrow$ and $B^- \mapsto x\downarrow$ be a self-invalidating rule. We can replace these rules with the following (y is fresh):

$$\begin{array}{lllcrll} B^+ & \mapsto & y\uparrow & \qquad & y & \mapsto & x\uparrow \\ B^- & \mapsto & y\downarrow & \qquad & \neg y & \mapsto & x\downarrow \end{array}$$

These rules are no longer self-invalidating since y is a fresh variable. They also do not change the result of the computation. Therefore, ruling out self-invalidating rules does not restrict the computation in any way. (The rules $y \mapsto x\uparrow$ and $\neg y \mapsto x\downarrow$ are implemented with two inverters.)

Consider any strongly confluent computation. Suppose we identify all the effective firings that can take place at a particular point in the execution and artificially prevent any other production rule from firing. Then, no matter which path was taken by the computation, the final result would be the same. This observation holds at any point in the computation. We conclude that a strongly confluent computation is essentially deterministic. Therefore, a QDI computation will always be deterministic.

An arbiter with inputs ai and bi, and outputs u and v is described by the handshaking expansion

```
*[[$ai \longrightarrow u\uparrow$; [$\neg ai$]; $u\downarrow$
  | $bi \longrightarrow v\uparrow$; [$\neg bi$]; $v\downarrow$
```

```
        ]]
```

where the thin bar for the selection denotes arbitration. There is no QDI implementation of this circuit because a computation that uses an arbiter cannot be strongly confluent. In the state in which $ai \wedge bi$ holds, both $u\uparrow$ and $v\uparrow$ can fire. However, after $u\uparrow$ fires, $v\uparrow$ can no longer fire. This implies that when designing an arbiter, we have to consider the electrical behavior of transistor circuits.

9.3. Compilation of a Turing machine

In this section we demonstrate that QDI circuits can be used to compute any computable function by constructing a QDI bounded-tape Turing machine. Since any computation can only use a finite amount of memory (since any physical implementation of a computation can only use finite resources), this demonstrates that restricting the design space to QDI circuits does not limit the class of functions that can be computed. The implementation we propose does not include any inverters on a single branch of a fork, and therefore doesn't contain inverters on the branch of an isochronic fork. Hence, the implementation is QDI (and therefore, also speed-independent).

Let $TM = \langle S, K, \delta \rangle$ be a Turing machine with a semi-infinite tape where S and K are positive integers, and δ is a function

$$\delta \colon \{q_0, \ldots, q_S\} \times \{\sigma_0, \ldots, \sigma_K\} \to \{q_0, \ldots, q_S\} \times \{\sigma_0, \ldots, \sigma_K, L, R\}$$

satisfying $\delta(q_1, \sigma_k) = (q_1, \sigma_k)$ for all k. Using s to denote the state, array a to denote the tape, and p to denote the head position, a CHP program that describes such a Turing machine is:

```
$TM \equiv$  $s := q_0; p := 0;$
      *[ $(s, d) := \delta(s, a[p]);$
           [ $d = L \longrightarrow p := p - 1$ ▯ $d = R \longrightarrow p := p + 1$ ▯ $else \longrightarrow a[p] := d$ ]
       ]
```

The Turing machine is initialized in state zero with its head located at position zero on the tape. It uses δ, commonly referred to as the *next move* function to determine the action it should take. It then updates the tape appropriately, and continues the computation. In the rest of this section, we omit the assignments $s := q_0$ and $p := 0$, since they can be performed by appropriately initializing the circuit on reset.

9.3.1. Process Decomposition

The computation of the Turing machine proceeds in two steps. Using the current value on the tape and the current value of the state, the next state and action is computed. Following this, the tape is appropriately updated. We therefore decompose the Turing machine into two parts, one for each distinct step of the computation. Using variable c to denote the current value of the symbol on the tape, we obtain:

```
$TM1 \equiv *[\;\; Tout?c;\;\; (s,d) := \delta(s,c);\;\; Tin!d \;\;]$
$TM2 \equiv *[\;\; Tout!a[p];\;\; Tin?d;$
        $[\;\; d = L \longrightarrow p := p-1 \;\;[]\;\; d = R \longrightarrow p := p+1 \;\;[]\;\; else \longrightarrow a[p] := d \;\;]$
    $]$
$TM \equiv \;\; TM1 \;\|\; TM2$
```

The computation $(s,d) := \delta(s,c)$ reads and writes the same variable s. Since we cannot perform this operation without making a temporary copy of s, we make this copy explicit by introducing a state buffer. Using $(X?)$ as an abbreviation for the value received on X, we obtain:

```
$next \equiv *[\;\; (s,d) := \delta(Sout?, Tout?);\;\; Sin!s, Tin!d \;\;]$
$statebuf \equiv *[\;\; Sout!x; Sin?x \;\;]$
$TM1 \equiv \;\; next \;\|\; statebuf$
```

We do not decompose $TM1$ furthur, since both *next* and *statebuf* can easily be transformed into a circuit (see below). For the remainder of this section, we concentrate on $TM2$.

A computation resembles a function block when it is of the form "read inputs", followed by "produce outputs." This computation has a standard QDI implementation, as discussed in Section 5.1. We make $TM2$ resemble this form by introducing a tape buffer process.

```
$fulltape \equiv *[\;\; Tin?d;$
        $[\;\; d = L \longrightarrow p := p-1 \;\;[]\;\; d = R \longrightarrow p := p+1 \;\;[]\;\; else \longrightarrow a[p] := d \;\;];$
        $TTout!a[p]$
    $]$
$tapebuf \equiv *[\;\; Tout!x; TTout?x \;\;]$
$TM2 \equiv \;\; fulltape \;\|\; tapebuf$
```

On receiving an input on *Tin*, *fulltape* updates its current state by either changing the value of p or changing the value of $a[p]$. Finally, the value of $a[p]$ is read. Since reading $a[p]$ and modifying p and $a[p]$ happen in sequence, we can implement this sequencing once and write *fulltape* as a reactive process. We split the tape into the tape control, which sequences the read and update operations, and the tape itself.

```
$tapecontrol \equiv *[\;\; LTin!(Tin?); TTout!(L?) \;\;]$
```

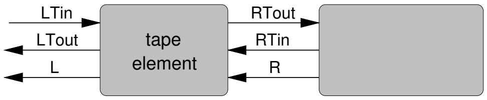


Figure 9.3. Decomposition of tape into an array of tape elements.

```
$tape \equiv$ *[[$\overline{LTin} \longrightarrow LTin?d$;
                    [$d = L \longrightarrow p := p - 1$ ▯ $d = R \longrightarrow p := p + 1$ ▯ $else \longrightarrow a[p] := d$]
      ▯$\overline{L} \longrightarrow L!a[p]$
     ]]
```

$fulltape \equiv \; tapecontrol \; \| \; tape$

To implement *tape*, we consider the tape to be the concurrent composition of a linear array of tape elements as shown in Figure 9.3. Each tape element maintains information about its position relative to the tape head. Its current state s is l if the element is to the left of the head position, r if the element is to the right of the head position, and t otherwise. At any point, the tape state (from left to right) can be described by the regular expression l^+tr^*, where the length of the expression is the size of the tape. (The presence of a leading "l" is guaranteed by the program of a Turing machine with a semi-infinite tape.)

Each tape element contains a register that stores the symbol in the tape element. Apart from this register, each tape element must maintain its state, s. The operation of a tape element (given its current state) can be described as follows:

$s = \texttt{l}$. If a "move left" action is to be performed, it is communicated to the rest of the tape. The new state is the state of the first process in the rest of the tape. If a write, read, or a "move right" action is performed, this action is communicated to the rest of the tape, and the state remains unchanged.

$s = \texttt{r}$. A "move left", read, and write action can never happen. If a "move right" action occurs, the new state is `t`.

$s = \texttt{t}$. A "move left" action results in state `r`. A "move right" action results in state `l`, and this action is communicated to the rest of the tape so as to move the head to the right. A read/write action is sent to the register.

Since the next state of a tape element can depend on the state of the first element in the rest of the tape, we introduce channels *RTin* and *LTout* that communicate this information.

Once again, since the tape reads and modifies its own state, we introduce a buffer to make the copy explicit. The CHP description of the tape element is:

```
$tapeelem \equiv$ *[[$\overline{LTin} \longrightarrow LTin?d, STin?s$;
```

```
                  $[d = L \longrightarrow [s = \texttt{l} \longrightarrow RTout!d; RTin?s; LTout!\texttt{l}$
                                 $[]s = \texttt{t} \longrightarrow LTout!\texttt{t}; s := \texttt{r}$
                                 $]$
                  $[]d = R \longrightarrow [s = \texttt{l} \longrightarrow RTout!d; RTin?; LTout!s$
                                  $[]s = \texttt{t} \longrightarrow RTout!d; RTin?; LTout!s; s := \texttt{l}$
                                  $[]s = \texttt{r} \longrightarrow LTout!s; s := \texttt{t}$
                                  $]$
                  $[]else \longrightarrow [s = \texttt{l} \longrightarrow RTout!d; RTin?; LTout!s$
                               $[]s = \texttt{t} \longrightarrow GW!d; LTout!s$
                               $]$
                  $];\ \ STout!s$
              $[]\overline{L} \longrightarrow [s = \texttt{l} \longrightarrow L!(R?)$
                      $[]s = \texttt{t} \longrightarrow L!(GR?)$
                      $]$
              $]]$
$tapereg \equiv *[[\overline{GR} \longrightarrow GR!x$
              $[]\overline{GW} \longrightarrow GW?x$
             $]]$
$tapestate \equiv *[STin!x; STout?x]$
$tapeelement \equiv\ \ tapeelem\ \parallel\ tapereg\ \parallel\ tapestate$
```

Since we need an additional channel *LTout* to perform correct state updates, we modify *tapecontrol*

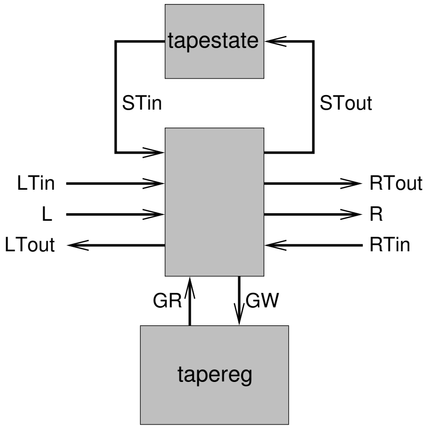


Figure 9.4. Decomposition of a tape element.

to the following process:

$tapecontrol \equiv *[\;\; LTin!(Tin?);\, TTout!(L?), LTout? \;\;]$

The complete tape element decomposition is shown in $^{Figure9\triangleright 4}$.

In the following sections, we compile the Turing machine into production rules. The compilation will proceed by translating each CHP process into handshaking expansions and, finally, into directly implementable production rules.

9.3.2. Compilation of *TM1*

The handshaking expansion for $TM1$ is straightforward. We use the output on *Sin* and *Tin* as the acknowledge for *Sout* and *Tout*, and the input on *Sout* and *Tout* as the request for data on *Sin* and *Tin*. We have:

$next \equiv *[[v(Sout) \wedge v(Tout)];\, Sin{\Uparrow}, Tin{\Uparrow};\, [n(Sout) \wedge n(Tout)];\, Sin{\Downarrow}, Tin{\Downarrow}]$

$statebuf \equiv *[Sout := a;\, [v(Sin)];\, a := Sin; Sout{\Downarrow};\, [n(Sin)]]$

where the functions v and n encode the validity and neutrality test respectively. We provide the production rules corresponding to a one-bit version of *statebuf*. This construction can be easily generalized to n-bits.

$$\begin{array}{ll} statebuf1bit \equiv *[& [\neg Sint \wedge \neg Sinf];\;\; [at \longrightarrow Soutt{\uparrow} \;[]\; af \longrightarrow Soutf{\uparrow}];\\ & [Sint \longrightarrow af{\downarrow}; at{\uparrow} \;[]\; Sinf \longrightarrow at{\downarrow}; af{\uparrow}];\;\; Soutt{\downarrow}, Soutf{\downarrow} \;\;] \end{array}$$

The production rules are:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\neg Sint \wedge \neg Sinf \wedge \neg af & \mapsto & Soutt{\uparrow} & at \vee Sint & \mapsto & af{\downarrow}\\
\neg Sint \wedge \neg Sinf \wedge \neg at & \mapsto & Soutf{\uparrow} & \neg Sinf \wedge \neg af & \mapsto & at{\uparrow}\\
\\
(Sint \wedge at) \vee (Sinf \wedge af) & \mapsto & Soutt{\downarrow} & af \vee Sinf & \mapsto & at{\downarrow}\\
(Sint \wedge at) \vee (Sinf \wedge af) & \mapsto & Soutf{\downarrow} & \neg Sint \wedge \neg at & \mapsto & af{\uparrow}
\end{array}$$

Notice that the compilation of this buffer completes the compilation of *tapebuf*, and part of *tapeelement* as well. The rest of $TM1$ can be compiled using the standard *function block* compilation technique, and depends on the next move function δ. The block diagram of $TM1$ is shown in Figure 9.5.

9.3.3. Compilation of the tape

To simplify the handshaking expansion for the tape, we use the input on *LTin* as a request for data on *LTout*, and the output on *LTout* as the acknowledge for channel *LTin*. The protocol used on L is the usual four-phase handshake.

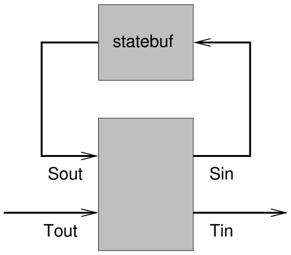


Figure 9.5. Compilation of *TM1*.

Compiling tapecontrol. The handshaking expansion for *tapecontrol* is given by:

$$\begin{array}{l} *[[v(Tin)]; LTin := Tin; [v(LTout)]; LTin\Downarrow; [n(LTout)]; \\ \quad Lo\uparrow; [v(Li)]; TTout := Li; [n(Tin)]; Lo\downarrow; [n(Li)]; TTout\Downarrow] \end{array}$$

We use *process factorization* to split the handshaking expansion into the following two concurrent processes.

$$\begin{array}{l} *[[v(Tin)]; LTin := Tin; [v(LTout)]; LTin\Downarrow; [n(LTout)]; Lo\uparrow; [n(Tin)]; Lo\downarrow] \\ \| \\ *[[v(Li)]; TTout := Li; [n(Li)]; TTout\Downarrow] \end{array}$$

The second process can be translated into a number of wires. We compile the first process for one-bit data. This construction can be easily generalized for n-bits of data. We introduce a state variables st and sf to remove indistinguishable states. (We could have just used one variable, but the resulting production rule set would need a number of extra inverters to make it directly implementable.) The resulting handshaking expansion is:

$$\begin{array}{l} *[[Tint \longrightarrow LTint\uparrow \;[]\; Tinf \longrightarrow LTinf\uparrow];\;\; [LToutt \vee LToutf]; sf\downarrow; st\uparrow; LTint\downarrow, LTinf\downarrow; \\ \quad [\neg LToutt \wedge \neg LToutf]; Lo\uparrow; st\downarrow; sf\uparrow; [\neg Tint \wedge \neg Tinf]; Lo\downarrow] \end{array}$$

The production rules corresponding to this handshaking expansion are:

$$\begin{array}{rclcrcl} \neg Tint\_ \wedge \neg st \wedge \neg Lo & \mapsto & LTint\uparrow & \qquad & LToutt \vee LToutf & \mapsto & sf\downarrow \\ \neg Tinf\_ \wedge \neg st \wedge \neg Lo & \mapsto & LTinf\uparrow & & \neg LToutt \wedge \neg LToutf \wedge \neg st & \mapsto & sf\uparrow \\ \\ st & \mapsto & LTint\downarrow & & \neg LToutt \wedge \neg LToutf \wedge \neg sf & \mapsto & Lo\uparrow \\ st & \mapsto & LTinf\downarrow & & Tint\_ \wedge Tinf\_ \wedge sf & \mapsto & Lo\downarrow \\ \\ \neg sf \wedge \neg Lo & \mapsto & st\uparrow & & & & \\ Lo & \mapsto & st\downarrow & & & & \end{array}$$

Notice that we have used the inverted sense of Tin ($Tin\_$) in this production rule set. We have to use an inverter to generate this signal from Tin.

Compiling a register. Compilation of the register used in *tapelement* is very simple. We compile a one-bit dual-railed register; the construction can be easily extended to n-bits. The handshaking expansion for the register is:

```
$*[[GRi \wedge xt \longrightarrow GRto\uparrow; [\neg GRi]; GRto\downarrow$
  $[] GRi \wedge xf \longrightarrow GRfo\uparrow; [\neg GRi]; GRfo\downarrow$
  $[] GWti \longrightarrow xf\downarrow; xt\uparrow; GWo\uparrow; [\neg GWti]; GWo\downarrow$
  $[] GWfi \longrightarrow xt\downarrow; xf\uparrow; GWo\uparrow; [\neg GWfi]; GWo\downarrow$
 $]]$
```

Since the register is accessed in a manner which guarantees mutual exclusion between GR and GW, the production rules for this process are given by:

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\neg xf \wedge \neg GWfi & \mapsto & xt\uparrow & \neg xt \wedge \neg GRi_{\_} & \mapsto & GRfo\uparrow \\
xf \vee GWfi & \mapsto & xt\downarrow & GRi_{\_} & \mapsto & GRfo\downarrow \\
\\
\neg xt \wedge \neg GWti & \mapsto & xf\uparrow & (xf \wedge GWfi) \vee (xt \wedge GWti) & \mapsto & GWo_{\_}\downarrow \\
xt \vee GWti & \mapsto & xf\downarrow & \neg GWfi \wedge \neg GWti & \mapsto & GWo_{\_}\uparrow \\
\\
\neg xf \wedge \neg GRi_{\_} & \mapsto & GRto\uparrow & & & \\
GRi_{\_} & \mapsto & GRto\downarrow & & &
\end{array}
$$

The production rule set uses the inverted sense of GRi ($GRi_{\_}$) and generates the inverted sense of GWo ($GWo_{\_}$). Since $GWo_{\_}$ is an output, we can safely invert it to generate GWo.

Compiling the tape element control. The tape element control can send d to the right or to the register, and conditionally compute the new state. The handshaking expansion for the tape element control can be written as follows:

```
$tapeelem1 \equiv *[[v(LTin) \wedge v(STin)];\ \ [R \longrightarrow RTout\Uparrow [] \neg R \longrightarrow skip], [W \longrightarrow GWo\Uparrow [] \neg W \longrightarrow skip], I\Uparrow$
                 $[((R \wedge v(RTout)) \vee (W \wedge GWi) \vee (\neg R \wedge \neg W)) \wedge v(I)]; STout\Uparrow, LTout\Uparrow;$
                 $[n(LTin) \wedge n(STin)]; RTout\Downarrow, GWo\Downarrow, I\Downarrow$
                 $[n(RTout) \wedge \neg GWi \wedge n(I)]; STout\Downarrow, LTout\Downarrow]$
```

```
$read \equiv *[[Li \wedge s = \mathtt{l} \longrightarrow Ro\uparrow; [v(Ri)]; Lo\Uparrow; [\neg Li]; Ro\downarrow; [n(Ri)]; Lo\Downarrow$
          $[] Li \wedge s = \mathtt{t} \longrightarrow GRo\uparrow; [v(GRi)]; Lo\Uparrow; [\neg Li]; GRo\downarrow; [n(GRi)]; Lo\Downarrow$
          $]]$
```

We can decompose the tape element control into the read and write part since the environment guarantees that they do not overlap. We use process factorization on *tapeelem*1 to decompose it into the following two processes, using channel I to communicate information needed by the other process.

$$\begin{aligned}
&\mathit{change} \equiv *[[v(\mathit{LTin}) \wedge v(\mathit{STin})];\, [R \longrightarrow \mathit{RTout}{\Uparrow} [\!]\, \neg R \longrightarrow \mathit{skip}],\, [W \longrightarrow \mathit{GWo}{\Uparrow} [\!]\, \neg W \longrightarrow \mathit{skip}],\, I{\Uparrow};\\
&\qquad\qquad [n(\mathit{LTin}) \wedge n(\mathit{STin})];\, \mathit{RTout}{\Downarrow}, \mathit{GWo}{\Downarrow}, I{\Downarrow}\ ]
\end{aligned}$$

$$\begin{aligned}
&\mathit{right} \equiv *[[v(I)];\, [((R \wedge v(\mathit{RTout})) \vee (W \wedge \mathit{GWi})) \vee (\neg W \wedge \neg R)];\, \mathit{LTout}{\Uparrow}, \mathit{STout}{\Uparrow};\\
&\qquad\qquad [n(I) \wedge n(\mathit{RTout}) \wedge \neg \mathit{GWi}];\, \mathit{LTout}{\Downarrow}, \mathit{STout}{\Downarrow}]
\end{aligned}$$

We use two dual-railed bits (st_0, sf_0) and (st_1, sf_1) to encode s. The different states are:

$(s = \mathtt{l}) \equiv sf_0 \wedge sf_1$

$(s = \mathtt{t}) \equiv sf_0 \wedge st_1$

$(s = \mathtt{r}) \equiv st_0 \wedge sf_1$

The value received on *LTin* is encoded using $2+2\lceil \log_2 K+1 \rceil$ wires. The encoding has the following meaning:

LTin_0 is true just when $d = L$.

LTin_1 is true just when $d = R$.

$\mathit{LTin}_2..\mathit{LTin}_{1+2\lceil \log_2 K+1 \rceil}$ are used to specify the dual-rail encoding of the symbol to be printed.

The information sent on channel I is used to determine the new state and acknowledge. This channel is encoded using 13 wires as follows:

$I_0..I_3$ contain the old state, which is used to generate the acknowledge.

$I_4..I_7$ contain the new state, if computable without information from *RTin*.

I_8 is true if the new state should be computed from the information received from *RTin*.

$I_9..I_{10}$ contains the dual-rail encoding of whether or not a communication was initiated on *RTout*.

$I_{11}..I_{12}$ contains the dual-rail encoding of whether or not a communication was initiated on GW.

(This encoding is not meant to be efficient in any way!) The predicates R and W are given by:

$R \equiv (s = \mathtt{l}) \vee (s = \mathtt{t} \wedge \mathit{LTin} = R)$

$W \equiv (\mathit{LTin} \neq L \wedge \mathit{LTin} \neq R) \wedge (s = \mathtt{t})$

Since both *tapeelem*1 and *right* are function blocks, they can be compiled using the standard technique outlined in Section 5.1. We compile *read* assuming a one-bit dual-railed input on L; this compilation can be easily generalized to n-bits, for any n. The handshaking expansion for *read* is:

$$\begin{aligned}
&*[[\mathit{Li} \wedge sf_0 \wedge sf_1 \longrightarrow \mathit{Ro}{\uparrow};\, [\mathit{Rti} \longrightarrow \mathit{Lto}{\uparrow} [\!]\, \mathit{Rfi} \longrightarrow \mathit{Lfo}{\uparrow}];\, [\neg \mathit{Li}];\, \mathit{Ro}{\downarrow};\\
&\qquad\qquad [\neg \mathit{Rti} \wedge \neg \mathit{Rfi}];\, \mathit{Lto}{\downarrow}, \mathit{Lfo}{\downarrow}\\
&\quad [\!]\, \mathit{Li} \wedge sf_0 \wedge st_1 \longrightarrow \mathit{GRo}{\uparrow};\, [\mathit{GRti} \longrightarrow \mathit{Lto}{\uparrow} [\!]\, \mathit{GRfi} \longrightarrow \mathit{Lfo}{\uparrow}];\, [\neg \mathit{Li}];\, \mathit{GRo}{\downarrow};\\
&\qquad\qquad [\neg \mathit{GRti} \wedge \neg \mathit{GRfi}];\, \mathit{Lto}{\downarrow}, \mathit{Lfo}{\downarrow}\\
&\ ]]
\end{aligned}$$

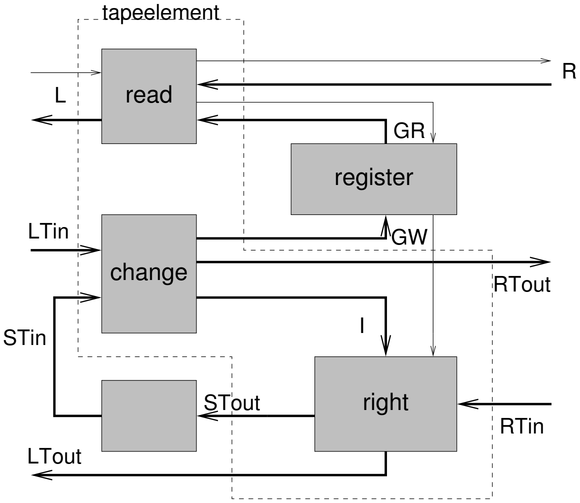


Figure 9.6. Compilation of *tapeelement* as a number of function blocks.

The production rules are:

$$\begin{array}{llll} Li \wedge sf_0 \wedge sf_1 & \mapsto \; Ro\_\!\downarrow & \neg Li & \mapsto \; Ro\_\!\uparrow \\ Li \wedge sf_0 \wedge st_1 & \mapsto \; GRo\_\!\downarrow & \neg Li & \mapsto \; GRo\_\!\uparrow \\ & & & \\ Rti \vee GRti & \mapsto \; Lto\_\!\downarrow & \neg Rti \wedge \neg Rfi \wedge \neg GRti \wedge \neg GRfi & \mapsto \; Lto\_\!\uparrow \\ Rfi \vee GRfi & \mapsto \; Lfo\_\!\downarrow & \neg Rti \wedge \neg Rfi \wedge \neg GRti \wedge \neg GRfi & \mapsto \; Lfo\_\!\uparrow \end{array}$$

Note that we have generated the inverted signals $Lto\_$, $Lfo\_$, $Ro\_$ and $GRo\_$. Since these are output variables, and are not used by any operators in this process, we can safely invert them to generate Lto, Lfo, Ro, and GRo. The entire *tapeelement* with the state buffer is shown in Figure 9.6.

9.3.4. Putting the pieces together

Each component given above is QDI. Some components require the inverted senses of certain signals as input. As noted in the previous section, the introduction of inverters can compromise the QDI of a circuit.

Observe that the inverted senses of signals are required by *tapecontrol* and the register. However, the inputs to these processes are not forked to any other process or operator! As a result, we can safely insert inverters between the processes to obtain a QDI Turing machine (cf. Figure 9.7). By the construction given above, we can state that

Theorem 9.7. (*Turing-completeness*)

Any bounded-tape Turing-computable function can be implemented using a QDI circuit.

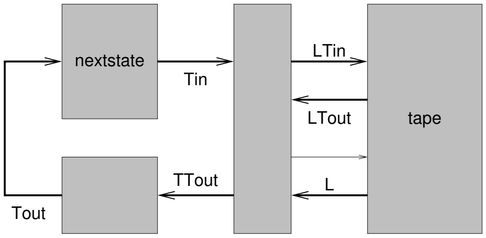


Figure 9.7. A complete Turing machine.

References

The material on the limitations to delay-insensitive design is taken from a paper by Martin,[26] and the material on QDI circuits is taken from a paper by Manohar and Martin.[22]

Appendix A.

The First Asynchronous Microprocessor

A.1. Decomposition Corresponding to the First Processor

Sequential CHP:

```
*[$(i, pc) := (imem[pc], pc + 1)$;
  [$offset(i.op) \longrightarrow (offset, pc) := imem[pc], pc + 1$
  []$\neg offset(i.op) \longrightarrow$ **skip**
  ];
  [$alu(i.op) \longrightarrow (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
  []$ld(i.op) \longrightarrow reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
  []$st(i.op) \longrightarrow dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
  []$ldx(i.op) \longrightarrow reg[i.z] := dmem[offset + reg[i.y]]$
  []$stx(i.op) \longrightarrow dmem[offset + reg[i.y]] := reg[i.z]$
  []$lda(i.op) \longrightarrow reg[i.z] := offset + reg[i.y]$
  []$stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]$
  []$jmp(i.op) \longrightarrow pc := reg[i.y]$
  []$brch(i.op) \longrightarrow$ [$cond(f, i.cc) \longrightarrow pc := pc + offset$
                   []$\neg cond(f, i.cc) \longrightarrow$ **skip**
                   ]
 ]]
```

First decomposition: use process decomposition to break up the CHP into a part that fetches instructions and a part that executes them.

```
*[$(i, pc) := (imem[pc], pc + 1)$;
```

```
        [$offset(i.op) \longrightarrow (offset, pc) := imem[pc], pc + 1$
        []$\neg offset(i.op) \longrightarrow$ **skip**
        ]; $E$ ]
  ||
      *[[$\overline{E}$];
        [$alu(i.op) \longrightarrow (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
        []$ld(i.op) \longrightarrow reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
        []$st(i.op) \longrightarrow dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
        []$ldx(i.op) \longrightarrow reg[i.z] := dmem[offset + reg[i.y]]$
        []$stx(i.op) \longrightarrow dmem[offset + reg[i.y]] := reg[i.z]$
        []$lda(i.op) \longrightarrow reg[i.z] := offset + reg[i.y]$
        []$stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]$
        []$jmp(i.op) \longrightarrow pc := reg[i.y]$
        []$brch(i.op) \longrightarrow$ [$cond(f, i.cc) \longrightarrow pc := pc + offset$
                              []$\neg cond(f, i.cc) \longrightarrow$ **skip**
                              ]
        ];$E$
       ]
```

Make a copy of i so that we can overlap fetching the next instruction with part of the execution. Split E into E_1 and E_2, where E_2 indicates when instruction fetch can proceed.

```
      *[$(i, pc) := (imem[pc], pc + 1)$;
        [$offset(i.op) \longrightarrow (offset, pc) := imem[pc], pc + 1$
        []$\neg offset(i.op) \longrightarrow$ **skip**
        ];
        $E_1!i; E_2$
       ]
  ||
      *[$E_1?i$;
        [$alu(i.op) \longrightarrow (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
        []$ld(i.op) \longrightarrow reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
        []$st(i.op) \longrightarrow dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
        []$ldx(i.op) \longrightarrow reg[i.z] := dmem[offset + reg[i.y]]$
        []$stx(i.op) \longrightarrow dmem[offset + reg[i.y]] := reg[i.z]$
        []$lda(i.op) \longrightarrow reg[i.z] := offset + reg[i.y]$
```

```
    []$stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]$
    []$jmp(i.op) \longrightarrow pc := reg[i.y]$
    []$brch(i.op) \longrightarrow$ [$cond(f, i.cc) \longrightarrow pc := pc + offset$
                         []$\neg cond(f, i.cc) \longrightarrow \mathbf{skip}$
                         ]
    ]; $E_2$
   ]
```

Complete E_2 as early as possible (look at operations on local variables only).

```
   *[$(i, pc) := (imem[pc], pc + 1)$;
     [$offset(i.op) \longrightarrow (offset, pc) := imem[pc], pc + 1$
     []$\neg offset(i.op) \longrightarrow \mathbf{skip}$
     ];
     $E_1!i; E_2$
    ]
||
   *[$E_1?i$;
     [$alu(i.op) \longrightarrow E_2; (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
     []$ld(i.op) \longrightarrow E_2; reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
     []$st(i.op) \longrightarrow E_2; dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
     []$ldx(i.op) \longrightarrow reg[i.z] := dmem[offset + reg[i.y]]; E_2$
     []$stx(i.op) \longrightarrow dmem[offset + reg[i.y]] := reg[i.z]; E_2$
     []$lda(i.op) \longrightarrow reg[i.z] := offset + reg[i.y]; E_2$
     []$stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]; E_2$
     []$jmp(i.op) \longrightarrow pc := reg[i.y]; E_2$
     []$brch(i.op) \longrightarrow$ [$cond(f, i.cc) \longrightarrow pc := pc + offset$
                          []$\neg cond(f, i.cc) \longrightarrow \mathbf{skip}$
                          ]; $E_2$
    ]]
```

Move access to the *imem* array into a separate process. This process can be easily compiled into a circuit to talk to off-chip unpipelined SRAM. The execution block remains unchanged.

```
   *[[$\overline{ID}$]; $ID!imem[pc]$]
||
   *[$ID?i; pc := pc + 1$;
     [$offset(i.op) \longrightarrow ID?offset; pc := pc + 1$
```

```
      []$\neg offset(i.op) \longrightarrow \mathbf{skip}$
      ];
      $E_1!i; E_2$
    ]
```

Move operations on *pc* into a separate process, without changing the process that accesses *imem*.

```
    *[[$\overline{PCI} \longrightarrow pc := pc + 1; PCI$
      []$\overline{PCA} \longrightarrow pc := pc + offset; PCA$
      []$\overline{X} \longrightarrow X!pc$
      []$\overline{Y} \longrightarrow Y?pc$
     ]]
||
    *[$ID?i; PCI$
      [$offset(i.op) \longrightarrow ID?offset; PCI$
      []$\neg offset(i.op) \longrightarrow \mathbf{skip}$
      ];
      $E_1!i; E_2$
     ]
||
    *[$E_1?i;$
      [$alu(i.op) \longrightarrow E_2; (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
      []$ld(i.op) \longrightarrow E_2; reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
      []$st(i.op) \longrightarrow E_2; dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
      []$ldx(i.op) \longrightarrow reg[i.z] := dmem[offset + reg[i.y]]; E_2$
      []$stx(i.op) \longrightarrow dmem[offset + reg[i.y]] := reg[i.z]; E_2$
      []$lda(i.op) \longrightarrow reg[i.z] := offset + reg[i.y]; E_2$
      []$stpc(i.op) \longrightarrow X?x; reg[i.z] := x + reg[i.x]; E_2$
      []$jmp(i.op) \longrightarrow Y!reg[i.y]; E_2$
      []$brch(i.op) \longrightarrow$ [$cond(f, i.cc) \longrightarrow PCA$
                          []$\neg cond(f, i.cc) \longrightarrow \mathbf{skip}$
                          ]; $E_2$
     ]]
```

To overlap the *pc* increment with instruction fetch, we split the *PCI* operation into two parts: PCI_1 and PCI_2. We apply a similar transformation to *PCA*, to obtain:

```
    *[[$\overline{PCI_1} \longrightarrow PCI_1; y := pc + 1; PCI_2; pc := y$
      []$\overline{PCA_1} \longrightarrow PCA_1; y := pc + \mathit{offset}; PCA_2; pc := y$
      []$\overline{X} \longrightarrow X!pc$
      []$\overline{Y} \longrightarrow Y?pc$
     ]]
$\|$
    *[$PCI_1; ID?i; PCI_2;$
      [$\mathit{offset}(i.op) \longrightarrow PCI_1; ID?\mathit{offset}; PCI_2$
      []$\neg\mathit{offset}(i.op) \longrightarrow$ skip
      ];
      $E_1!i; E_2$
     ]
$\|$
    *[$E_1?i;$
      [$alu(i.op) \longrightarrow E_2; (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
      []$ld(i.op) \longrightarrow E_2; reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
      []$st(i.op) \longrightarrow E_2; dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
      []$ldx(i.op) \longrightarrow reg[i.z] := dmem[\mathit{offset} + reg[i.y]]; E_2$
      []$stx(i.op) \longrightarrow dmem[\mathit{offset} + reg[i.y]] := reg[i.z]; E_2$
      []$lda(i.op) \longrightarrow reg[i.z] := \mathit{offset} + reg[i.y]; E_2$
      []$stpc(i.op) \longrightarrow X?x; reg[i.z] := x + reg[i.x]; E_2$
      []$jmp(i.op) \longrightarrow Y!reg[i.y]; E_2$
      []$brch(i.op) \longrightarrow$ [$cond(f, i.cc) \longrightarrow PCA_1; PCA_2$
                             []$\neg cond(f, i.cc) \longrightarrow$ skip
                             ]; $E_2$
     ]]
```

The reason this works is because when *pc* is being read by the *imem*[*pc*] operation, we know that we are between PCI_1 and PCI_2.

Create two execution units, one for the alu operations and the other for memory ops.

```
    *[[$\overline{AC} \longrightarrow AC?op; (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f); AC_2$
     []$\overline{F} \longrightarrow F!f$
     ]]
$\|$
    *[[$\overline{MC_1} \longrightarrow reg[i.z] := dmem[reg[i.x] + reg[i.y]]; MC_1$
```

```
          ▯$\overline{MC_2} \longrightarrow dmem[reg[i.x] + reg[i.y]] := reg[i.z]; MC_2$
          ▯$\overline{MC_3} \longrightarrow reg[i.z] := dmem[offset + reg[i.y]]; MC_3$
          ▯$\overline{MC_4} \longrightarrow dmem[offset + reg[i.y]] := reg[i.z]; MC_4$
          ▯$\overline{MC_5} \longrightarrow reg[i.z] := offset + reg[i.y]; MC_5$
          ]]
      $\|$
         *[$E_1?i;$
           [$alu(i.op) \longrightarrow E_2; AC!i.op; AC_2$
           ▯$ld(i.op) \longrightarrow E_2; MC_1$
           ▯$st(i.op) \longrightarrow E_2; MC_2$
           ▯$ldx(i.op) \longrightarrow MC_3; E_2$
           ▯$stx(i.op) \longrightarrow MC_4; E_2$
           ▯$lda(i.op) \longrightarrow MC_5; E_2$
           ▯$stpc(i.op) \longrightarrow X?x; reg[i.z] := x + reg[i.x]; E_2$
           ▯$jmp(i.op) \longrightarrow Y!reg[i.y]; E_2$
           ▯$brch(i.op) \longrightarrow F?f;$ [$cond(f, i.cc) \longrightarrow PCA_1; PCA_2$
                                        ▯$\neg cond(f, i.cc) \longrightarrow \textbf{skip}$
                                        ];$E_2$
          ]]
```

Eliminate shared register file, and introduce busses. Use *Xpc* and *Ypc* to select the *X* and *Y* communications in the *pc*-process.

```
$PCADD \equiv$ *[[$\overline{PCI_1} \longrightarrow PCI_1; y := pc + 1; PCI_2; pc := y$
              ▯$\overline{PCA_1} \longrightarrow PCA_1; y := pc + offset; PCA_2; pc := y$
              ▯$\overline{Xpc} \longrightarrow X!pc \bullet Xpc$
              ▯$\overline{Ypc} \longrightarrow Y?pc \bullet Ypc$
             ]]

$EXEC \equiv$ *[$E_1?i;$
            [$alu(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet AC!i.op; Zs$
            ▯$ld(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet MC_1; Zs$
            ▯$st(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet MC_2 \bullet ZWs$
            ▯$ldx(i.op) \longrightarrow Ys \bullet MC_3; Zs; E_2$
            ▯$stx(i.op) \longrightarrow Ys \bullet MC_4 \bullet ZWs; E_2$
            ▯$lda(i.op) \longrightarrow Ys \bullet MC_5; E_2$
            ▯$stpc(i.op) \longrightarrow Xpc \bullet Ys \bullet AC!add; Zs; E_2$
```

```
        $[] jmp(i.op) \longrightarrow Ypc \bullet Ys; E_2$
        $[] brch(i.op) \longrightarrow F?f;$ [$cond(f, i.cc) \longrightarrow PCA_1; PCA_2$
                                  $[] \neg cond(f, i.cc) \longrightarrow$ skip
                                  ]; $E_2$
      ]]
```

```
$REG \equiv$ (  *[[$\overline{Xs} \longrightarrow X!reg$[$i.x$] $\bullet\ Xs$]]
         $\|$ *[[$\overline{Ys} \longrightarrow Y!reg$[$i.y$] $\bullet\ Ys$]]
         $\|$ *[[$\overline{Zs} \longrightarrow Z?reg$[$i.z$] $\bullet\ Zs$]]
         $\|$ *[[$\overline{ZWs} \longrightarrow ZM!reg$[$i.z$] $\bullet\ ZWs$]]
         )
```

```
$ALU \equiv$ *[[$\overline{AC} \longrightarrow AC?op \bullet X?x \bullet Y?y; (z, f) := aluf(x, y, op, f); Z!z$
         $[] \overline{F} \longrightarrow F!f$
        ]]
```

```
$MU \equiv$ *[[$\overline{MC_1} \longrightarrow X?x \bullet Y?y \bullet MC_1; z := dmem$[$x + y$]$; Z!z$
        $[] \overline{MC_2} \longrightarrow X?x \bullet Y?y \bullet MC_2 \bullet ZM?w; dmem$[$x + y$] $:= w$
        $[] \overline{MC_3} \longrightarrow x := offset; Y?y \bullet MC_3; z := dmem$[$x + y$]$; Z!z$
        $[] \overline{MC_4} \longrightarrow x := offset; Y?y \bullet MC_2 \bullet ZM?w; dmem$[$x + y$] $:= w$
        $[] \overline{MC_5} \longrightarrow X?x \bullet Y?y \bullet MC_5; z := x + y; Z!z$
       ]]
```

Communicate *offset* value on shared bus X, collapse MC_1 and MC_3 into one channel, MC_2 and MC_4 into one channel. Renumber MC channels.

```
$PCADD \equiv$ *[[$\overline{PCI_1} \longrightarrow PCI_1; y := pc + 1; PCI_2; pc := y$
            $[] \overline{PCA_1} \longrightarrow PCA_1; y := pc + offset; PCA_2; pc := y$
            $[] \overline{Xpc} \longrightarrow X!pc \bullet Xpc$
            $[] \overline{Ypc} \longrightarrow Y?pc \bullet Ypc$
           ]]
        $\|$  *[[$\overline{Xof} \longrightarrow X!offset \bullet Xof$]]
```

```
$EXEC \equiv$ *[$E_1?i;$
        [$alu(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet AC!i.op; Zs$
        $[] ld(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet MC_1; Zs$
        $[] st(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet MC_2 \bullet ZWs$
        $[] ldx(i.op) \longrightarrow Xof \bullet Ys \bullet MC_1; Zs; E_2$
        $[] stx(i.op) \longrightarrow Xof \bullet Ys \bullet MC_2 \bullet ZWs; E_2$
        $[]\, lda(i.op) \longrightarrow Ys \bullet MC_3; E_2$
        $[]\, stpc(i.op) \longrightarrow Xpc \bullet Ys \bullet AC!add; Zs; E_2$
        $[]\, jmp(i.op) \longrightarrow Ypc \bullet Ys; E_2$
        $[]\, brch(i.op) \longrightarrow F?f; [cond(f, i.cc) \longrightarrow PCA_1; PCA_2$
                                   $[]\, \neg cond(f, i.cc) \longrightarrow \mathbf{skip}$
                                   $]; E_2$
      ]]
```

```
$MU \equiv *[[\overline{MC_1} \longrightarrow X?x \bullet Y?y \bullet MC_1; z := dmem[x+y]; Z!z$
      $[]\, \overline{MC_2} \longrightarrow X?x \bullet Y?y \bullet MC_2 \bullet ZM?w; dmem[x+y] := w$
      $[]\, \overline{MC_3} \longrightarrow X?x \bullet Y?y \bullet MC_5; z := x+y; Z!z$
     ]]
```

Use *ZM* as a bidirectional bus between the memory unit and register file. Call the *Zs* selection *ZAs* for alu select.

```
$REG \equiv ($  $*[[\overline{Xs} \longrightarrow X!reg[i.x] \bullet Xs]]$
       $\|\, *[[\overline{Ys} \longrightarrow Y!reg[i.y] \bullet Ys]]$
       $\|\, *[[\overline{ZAs} \longrightarrow Z?reg[i.z] \bullet ZAs]]$
       $\|\, *[[\overline{ZWs} \longrightarrow ZM!reg[i.z] \bullet ZWs]]$
       $\|\, *[[\overline{ZRs} \longrightarrow ZM?reg[i.z] \bullet ZRs]]$
       $)$
```

```
$EXEC \equiv *[E_1?i;$
        $[alu(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet AC!i.op; ZAs$
        $[]\, ld(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet MC_1; ZRs$
        $[]\, st(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet MC_2 \bullet ZWs$
        $[]\, ldx(i.op) \longrightarrow Xof \bullet Ys \bullet MC_1; ZRs; E_2$
        $[]\, stx(i.op) \longrightarrow Xof \bullet Ys \bullet MC_2 \bullet ZWs; E_2$
        $[]\, lda(i.op) \longrightarrow Ys \bullet MC_3; E_2$
        $[]\, stpc(i.op) \longrightarrow Xpc \bullet Ys \bullet AC!add; ZAs; E_2$
        $[]\, jmp(i.op) \longrightarrow Ypc \bullet Ys; E_2$
        $[]\, brch(i.op) \longrightarrow F?f; [cond(f, i.cc) \longrightarrow PCA_1; PCA_2$
                                   $[]\, \neg cond(f, i.cc) \longrightarrow \mathbf{skip}$
                                   $]; E_2$
      ]]
```

```
$MU \equiv *[[\overline{MC_1} \longrightarrow X?x \bullet Y?y \bullet MC_1; z := dmem[x+y]; ZM!z$
```

```
    ▯$\overline{MC_2} \longrightarrow X?x \bullet Y?y \bullet MC_2 \bullet ZM?w; dmem[x+y] := w$
    ▯$\overline{MC_3} \longrightarrow X?x \bullet Y?y \bullet MC_5; z := x+y; ZM!z$
   ]]
```

Split register access into a process per register, and introduce Boolean b_k to allow the write selection to complete early.

```
$REG \equiv (\| \; k ::$  $*[[\overline{Xs} \wedge i.x = k \wedge \neg b_k \longrightarrow X!r_k \bullet Xs]]$
             $\| \; *[[\overline{Ys} \wedge i.y = k \wedge \neg b_k \longrightarrow Y!r_k \bullet Ys]]$
             $\| \; *[[\overline{ZAs} \wedge i.z = k \wedge \neg b_k \longrightarrow b_k\uparrow; ZAs; Z?r_k; b_k\downarrow]]$
             $\| \; *[[\overline{ZWs} \wedge i.z = k \wedge \neg b_k \longrightarrow ZM!r_k \bullet ZWs]]$
             $\| \; *[[\overline{ZRs} \wedge i.z = k \wedge \neg b_k \longrightarrow b_k\uparrow; ZRs; ZM?r_k; b_k\downarrow]]$
       )
$EXEC \equiv *[E_1?i;$
           $[alu(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet AC!i.op \bullet ZAs$
           ▯$ld(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet MC_1 \bullet ZRs$
           ▯$st(i.op) \longrightarrow E_2; Xs \bullet Ys \bullet MC_2 \bullet ZWs$
           ▯$ldx(i.op) \longrightarrow Xof \bullet Ys \bullet MC_1 \bullet ZRs; E_2$
           ▯$stx(i.op) \longrightarrow Xof \bullet Ys \bullet MC_2 \bullet ZWs; E_2$
           ▯$lda(i.op) \longrightarrow Ys \bullet MC_3; E_2$
           ▯$stpc(i.op) \longrightarrow Xpc \bullet Ys \bullet AC!add \bullet ZAs; E_2$
           ▯$jmp(i.op) \longrightarrow Ypc \bullet Ys; E_2$
           ▯$brch(i.op) \longrightarrow F?f; [cond(f, i.cc) \longrightarrow PCA_1; PCA_2$
                                ▯$\neg cond(f, i.cc) \longrightarrow$ **skip**
                                $]; E_2$
    ]]
```

A.2. New Decomposition

We begin with the stage when E_1 and E_2 are introduced:

```
$*[(i, pc) := (imem[pc], pc+1);$
  $[offset(i.op) \longrightarrow (offset, pc) := imem[pc], pc+1$
  ▯$\neg offset(i.op) \longrightarrow$ **skip**
  $];$
  $E_1!i; E_2$
 ]
```

```
$\|$
    $*[E_1?i;$
        $[alu(i.op) \longrightarrow E_2; (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
        $▯ld(i.op) \longrightarrow E_2; reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
        $▯st(i.op) \longrightarrow E_2; dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
        $▯ldx(i.op) \longrightarrow reg[i.z] := dmem[offset + reg[i.y]]; E_2$
        $▯stx(i.op) \longrightarrow dmem[offset + reg[i.y]] := reg[i.z]; E_2$
        $▯lda(i.op) \longrightarrow reg[i.z] := offset + reg[i.y]; E_2$
        $▯stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]; E_2$
        $▯jmp(i.op) \longrightarrow pc := reg[i.y]; E_2$
        $▯brch(i.op) \longrightarrow [cond(f, i.cc) \longrightarrow pc := pc + offset$
                                   $▯\neg cond(f, i.cc) \longrightarrow \textbf{skip}$
                                   $]; E_2$
     ]]
```

We remove sharing on *offset* by sending it over to the execution process:

```
    $*[(i, pc) := (imem[pc], pc + 1);$
        $[offset(i.op) \longrightarrow (offset, pc) := imem[pc], pc + 1;$
        $▯\neg offset(i.op) \longrightarrow \textbf{skip}$
        $];$
        $E_1!(i, offset); E_2$
     ]
$\|$
    $*[E_1?(i, offset);$
        $[alu(i.op) \longrightarrow E_2; (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
        $▯ld(i.op) \longrightarrow E_2; reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
        $▯st(i.op) \longrightarrow E_2; dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
        $▯ldx(i.op) \longrightarrow reg[i.z] := dmem[offset + reg[i.y]]; E_2$
        $▯stx(i.op) \longrightarrow dmem[offset + reg[i.y]] := reg[i.z]; E_2$
        $▯lda(i.op) \longrightarrow reg[i.z] := offset + reg[i.y]; E_2$
        $▯stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]; E_2$
        $▯jmp(i.op) \longrightarrow pc := reg[i.y]; E_2$
        $▯brch(i.op) \longrightarrow [cond(f, i.cc) \longrightarrow pc := pc + offset$
                                   $▯\neg cond(f, i.cc) \longrightarrow \textbf{skip}$
                                   $]; E_2$
    ]]
```

Now we complete E_2 early in additional actions since the only variable that is shared is pc:

```
    *[$E_1?(i, offset)$;
      [$alu(i.op) \longrightarrow E_2; (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
      []$ld(i.op) \longrightarrow E_2; reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
      []$st(i.op) \longrightarrow E_2; dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
      []$ldx(i.op) \longrightarrow E_2; reg[i.z] := dmem[offset + reg[i.y]]$
      []$stx(i.op) \longrightarrow E_2; dmem[offset + reg[i.y]] := reg[i.z]$
      []$lda(i.op) \longrightarrow E_2; reg[i.z] := offset + reg[i.y]$
      []$stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]; E_2$
      []$jmp(i.op) \longrightarrow pc := reg[i.y]; E_2$
      []$brch(i.op) \longrightarrow$ [$cond(f, i.cc) \longrightarrow pc := pc + offset$
                         []$\neg cond(f, i.cc) \longrightarrow$ **skip**
                         ]; $E_2$
    ]]
```

Now we have instructions where we send $offset$ even if we don't use it. To change this, we can conditionally send $offset$:

```
    *[$(i, pc) := (imem[pc], pc + 1)$;
      [$offset(i.op) \longrightarrow (offset, pc) := (imem[pc], pc + 1); E_1!i; E_1!offset$
      []$\neg offset(i.op) \longrightarrow E_1!i$
      ];
      $E_2$
     ]
  ||
    *[$E_1?i$;
      [$alu(i.op) \longrightarrow E_2; (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
      []$ld(i.op) \longrightarrow E_2; reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
      []$st(i.op) \longrightarrow E_2; dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
      []$ldx(i.op) \longrightarrow E_1?offset; E_2; reg[i.z] := dmem[offset + reg[i.y]]$
      []$stx(i.op) \longrightarrow E_1?offset; E_2; dmem[offset + reg[i.y]] := reg[i.z]$
      []$lda(i.op) \longrightarrow E_1?offset; E_2; reg[i.z] := offset + reg[i.y]$
      []$stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]; E_2$
      []$jmp(i.op) \longrightarrow pc := reg[i.y]; E_2$
      []$brch(i.op) \longrightarrow E_1?offset$; [$cond(f, i.cc) \longrightarrow pc := pc + offset$
```

```
                         [] $\neg cond(f, i.cc) \longrightarrow \textbf{skip}$
                         ]; $E_2$
      ]]
```

In the case when $\mathit{offset}(i.op)$ is true, $E_1!i$ is postponed until the second word is fetched from instruction memory. If we complete $E_1!i$ early, then the execution unit can start decoding the instruction early. However, note that when $\mathit{offset}(i.op)$ is **true**, *pc* is being modified by the fetch until the second E_1 communication. However, note that the only instruction that uses both *pc* and *offset* is a branch; and that instuction waits for *offset* before reading or writing *pc*. Therefore, we can transform the CHP to:

```
      *[$(i, pc) := (imem[pc], pc + 1); E_1!i;$
        [$\mathit{offset}(i.op) \longrightarrow (\mathit{offset}, pc) := (imem[pc], pc + 1); E_1!\mathit{offset}$
        [] $\neg \mathit{offset}(i.op) \longrightarrow \textbf{skip}$
        ];
        $E_2$
       ]
  ||
      *[$E_1?i;$
        [$alu(i.op) \longrightarrow E_2; (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
        [] $ld(i.op) \longrightarrow E_2; reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
        [] $st(i.op) \longrightarrow E_2; dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
        [] $ldx(i.op) \longrightarrow E_1?\mathit{offset}; E_2; reg[i.z] := dmem[\mathit{offset} + reg[i.y]]$
        [] $stx(i.op) \longrightarrow E_1?\mathit{offset}; E_2; dmem[\mathit{offset} + reg[i.y]] := reg[i.z]$
        [] $lda(i.op) \longrightarrow E_1?\mathit{offset}; E_2; reg[i.z] := \mathit{offset} + reg[i.y]$
        [] $stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]; E_2$
        [] $jmp(i.op) \longrightarrow pc := reg[i.y]; E_2$
        [] $brch(i.op) \longrightarrow$ [$cond(f, i.cc) \longrightarrow E_1?\mathit{offset}; pc := pc + \mathit{offset}$
                            [] $\neg cond(f, i.cc) \longrightarrow E_1?\mathit{offset}$
                            ]; $E_2$
      ]]
```

This transformation now overlaps the instruction decode (implementing the guards of the selection statement) with instruction fetching, as well as overlaps the condition code calculation for branches with fetching the offset.

Finally, observe that since we have eliminated sharing i and *offset*, there are number of cases when the E_2 synchronization action is unnecessary. In fact, we only require E_2 in three cases: *stpc*, *jmp*, and *brch*. We make E_2 conditional to obtain:

```
    $*[(i, pc) := (imem[pc], pc + 1); E_1!i;$
      $([offset(i.op) \longrightarrow (offset, pc) := (imem[pc], pc + 1); E_1!offset$
       $[]\neg offset(i.op) \longrightarrow \textbf{skip}$
       $],$
       $[stpc(i.op) \lor jmp(i.op) \lor brch(i.op) \longrightarrow E_2 \;\; [] \;\; \textbf{else} \longrightarrow \textbf{skip} \;\; ]$
      $)$
     $]$
$\|$
    $*[E_1?i;$
      $[alu(i.op) \longrightarrow (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
      $[]ld(i.op) \longrightarrow reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
      $[]st(i.op) \longrightarrow dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
      $[]ldx(i.op) \longrightarrow E_1?offset; reg[i.z] := dmem[offset + reg[i.y]]$
      $[]stx(i.op) \longrightarrow E_1?offset; dmem[offset + reg[i.y]] := reg[i.z]$
      $[]lda(i.op) \longrightarrow E_1?offset; reg[i.z] := offset + reg[i.y]$
      $[]stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]; E_2$
      $[]jmp(i.op) \longrightarrow pc := reg[i.y]; E_2$
      $[]brch(i.op) \longrightarrow [cond(f, i.cc) \longrightarrow E_1?offset; pc := pc + offset$
                                  $[]\neg cond(f, i.cc) \longrightarrow E_1?offset$
                                  $]; E_2$
     $]]$
```

Finally, we can eliminate *offset* from the fetch part and re-use i:

```
    $*[(i, pc) := (imem[pc], pc + 1); op := i.op, E_1!i;$
      $([offset(op) \longrightarrow (i, pc) := (imem[pc], pc + 1); E_1!i$
       $[]\neg offset(op) \longrightarrow \textbf{skip}$
       $],$
       $[stpc(op) \lor jmp(op) \lor brch(op) \longrightarrow E_2 \;\; [] \;\; \textbf{else} \longrightarrow \textbf{skip} \;\; ]$
      $)$
     $]$
$\|$
    $*[E_1?i;$
      $[alu(i.op) \longrightarrow (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f)$
      $[]ld(i.op) \longrightarrow reg[i.z] := dmem[reg[i.x] + reg[i.y]]$
```

```
 []$st(i.op) \longrightarrow dmem[reg[i.x] + reg[i.y]] := reg[i.z]$
 []$ldx(i.op) \longrightarrow E_1?\mathit{offset}; reg[i.z] := dmem[\mathit{offset} + reg[i.y]]$
 []$stx(i.op) \longrightarrow E_1?\mathit{offset}; dmem[\mathit{offset} + reg[i.y]] := reg[i.z]$
 []$lda(i.op) \longrightarrow E_1?\mathit{offset}; reg[i.z] := \mathit{offset} + reg[i.y]$
 []$stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x]; E_2$
 []$jmp(i.op) \longrightarrow pc := reg[i.y]; E_2$
 []$brch(i.op) \longrightarrow$ [$cond(f, i.cc) \longrightarrow E_1?\mathit{offset}; pc := pc + \mathit{offset}$
                   []$\neg cond(f, i.cc) \longrightarrow E_1?\mathit{offset}$
                   ]$; E_2$
]]
```

References

1. Kees van Berkel. Single-Track Handshake Signalling. *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996.
2. A.W. Burks, H.H. Goldstein, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Institute for Advanced Study, Princeton, N.J., June 1946.
3. Steven M. Burns and Alain J. Martin. Performance analysis and optimization of asynchronous circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*, pp. 71–86, 1991.
4. T. J. Chancey and C. E. Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on COmputers*, **22**(4):421–422, April 1973.
5. Uri V. Cummings, Andrew M. Lines, and Alain J. Martin. An asynchronous pipelined lattice-structure filter. In *Proceedings of the First International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 126–133, November 1994.
6. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
7. Marcel van der Goot. The Semantics of VLSI Synthesis. Ph.D. thesis CS-TR-95-08, California Institute of Technology, 1996.
8. C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666–677, 1978.
9. C.A.R. Hoare. A model for communicating sequential processes. Technical monograph PRG-22, Oxford University, 1981.
10. Kevin S. van Horn. *An Approach to Concurrent Semantics Using Complete Traces*. M.S. thesis 5236:TR:86, California Institute of Technology, 1986.

11. G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

12. Leslie Lamport. Buridan's Principle. Technical Note, Digital Equipment Corporation Systems Research Center, October 1984.

13. Tak-Kwan Lee. *A General Approach to Performance Analysis and Optimization of Asynchronous Circuits*. Ph.D. thesis, California Institute of Technology, 1995.

14. Tak-Kwan Lee and Alain J. Martin. Extended event-rule systems and performance analysis of asynchronous systems. In *TAU'95, ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1995.

15. Tak-Kwan Lee. Man page for `bubble`. Caltech Asynchronous Synthesis Tools, 1993.

16. J. van Leeuwen. *Handbook of Theoretical Computer Science, Volume B: Formal models and semantics.* MIT Press, 1990.

17. F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan-Kaufmann, 1992.

18. Andrew Matthew Lines. Pipelined Asynchronous Circuits. M.S. thesis, California Institute of Technology, 1996.

19. Rajit Manohar. An Analysis of Handshaking Expansions. Submitted to *Conference on Advanced Research in VLSI*.

20. Rajit Manohar. *The Impact of Asynchrony on Computer Architecture*. Ph.D. thesis, CS-TR-98-12, California Institute of Technology, July 1998

21. Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. Projection: A Synthesis Technique for Concurrent Systems. *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1999.

22. Rajit Manohar and Alain J. Martin. Quasi-delay-insensitive circuits are Turing-complete. Invited article, *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996. Available as Caltech technical report CS-TR-95-11, November 1995.

23. Rajit Manohar and Alain J. Martin. Slack Elasticity in Concurrent Computing. *Proceedings of the Fourth International Conference on the Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pp. 272–285, Springer-Verlag, June 1998.

24. Alain J. Martin. An Axiomatic definition of synchronization primitives. *Acta Informatica*, **16**:219–235, 1981.

25. Alain J. Martin. *Synthesis of Asynchronous VLSI Circuits.* Caltech Computer Science Technical Report CS-TR-93-28, 1993.

26. Alain J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conference on Advanced Research in VLSI*, 1990.

27. Alain J. Martin. The Probe: An addition to communication primitives. *Information Processing Letters*, **20**:125–130, 1985.

28. Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4), 1986.

29. Alain J. Martin. A Program Transformation Approach to Asynchronous VLSI Design.

30. Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in Systems Design*, **1**(1):119–139, 1992.

31. Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351–373, MIT Press, 1991.

32. Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri V. Cummings, and Tak-Kwan Lee. The Design of an Asynchronous MIPS R3000. *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

33. Mary Ann Maher and Carver Mead. Fine Points of Transistor Physics. Appendix B in *Analog VLSI and Neural Systems*, by Carver Mead, Addison-Wesley, 1989.

34. Raymond E. Miller. *Switching Theory*, Volumes 1 and 2. John Wiley and Sons, 1965.

35. Jayadev Misra and K. Mani Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, **SE-7**(4):417–426, July 1981.

36. Charles L. Seitz (editor). *Proceedings of the Caltech Conference on Very Large Scale Integration*, 1979.

37. Charles L. Seitz. System Timing. Chapter 7 in *Introduction to VLSI Systems*, by Carver Mead and Lynn Conway, Addison-Wesley, 1979.

38. Jakov N. Seizovic. Pipeline Synchronization. In *Proceedings of the First International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87-96, November 1994.

39. Jan L.A. van de Snepscheut. *Trace theory and VLSI design.* Lecture Notes in Computer Science 200, Springer-Verlag, 1985.

40. S.M. Sze. *Physics of Semiconductor Devices*, John Wiley and Sons, 1981.

41. José A. Tierno, Rajit Manohar, and Alain J. Martin. The Energy and Entropy of VLSI Computations. *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996.

42. Ted Eugene Williams. Self-timed Rings and their Application to Division. Ph.D. thesis, Computer Systems Laboratory, Stanford University, May 1991.

43. S. Winograd. On the Time Required to Perform Addition. *Journal of the Association of Computing Machinery*, **12**(2):277–285, April 1965.