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 \( x := \text{true} \) to \( x \uparrow \) and \( x := \text{false} \) 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 \( \{ rs \neq rd \land rt \neq rd \} \), 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.

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

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

\[
regfmt \equiv \text{struct} \{ \\
\text{int op: 6;} \\
\text{int rs: 5;} \\
\text{int rt: 5;} \\
\text{int rd: 5;} \\
\text{int sa: 5;} \\
\text{int func: 6;}
\}
\]

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

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.

\[
[ \begin{array}{l} G_1 \rightarrow S_1 \quad G_2 \rightarrow S_2 \quad \cdots \quad G_n \rightarrow S_n \end{array} ]
\]

\( G_1, G_2, \ldots, G_n \) are Boolean-valued expressions called guards, and \( S_1, S_2, \ldots, S_n \) 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 \( x \).

\[
[ \begin{array}{l} x \geq 0 \rightarrow y := x \quad \text{or} \quad x < 0 \rightarrow y := -x \end{array} ]
\]

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

\[
[ \begin{array}{l} G_1 \rightarrow S_1 \quad \text{or} \quad G_2 \rightarrow S_2 \quad \cdots \quad \text{or} \quad G_n \rightarrow S_n \end{array} ]
\]

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.

\[ \text{true} \rightarrow x \uparrow \mid \text{true} \rightarrow x \downarrow \]

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.

\[ x \leq y \rightarrow \text{min}, \text{max} := x, y \mid y \leq x \rightarrow \text{min}, \text{max} := y, x \]

The program fragment shown above sets $\text{min}$ to the minimum of $x$ and $y$, and $\text{max}$ 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 $[G \to \text{skip}]$, 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.

\[ \ast [ G_1 	o S_1 \mid G_2 	o S_2 \mid \ldots \mid G_n 	o S_n ] \]

$G_1, G_2, \ldots, G_n$ are Boolean-valued expressions called guards, and $S_1, S_2, \ldots, S_n$ 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.

\[ i := 0; \ast [ i < 10 \to S; i := i + 1 ] \]

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 \land Y > 0\}
\begin{align*}
x, y & := X, Y; \\
* [x > y & \rightarrow x := x - y] \\
\lnot x < y & \rightarrow y := y - x \\
\} \{x = \text{gcd}(X, Y)\}
\end{align*}

If multiple guards can be true, we use the *non-deterministic repetition* statement shown below.
\[ 
* [G_1 \rightarrow S_1 \mid G_2 \rightarrow S_2 \mid \cdots \mid G_n \rightarrow S_n ]
\]

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 \(*[\text{true} \rightarrow S]\), corresponds to this behavior and can be read as “repeat \(S\) forever.”

2.6. Replication Construct

The syntactic replication construct is shown below:
\[
\langle \text{op} \ i : n..m : S(i) \rangle
\]

\(i\) 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:
\[
S(n) \text{ op } S(n+1) \text{ op } \cdots \text{ op } S(m)
\]

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 \textbf{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 \textbf{true}. Assuming all the \(x_i\) variables are false initially, we write a decoder as the following selection:
\[
\{[\forall i : 1..N : y = i \rightarrow x_i]\}
\]
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. \( \{ x = 0 \} \ (x := 1 \ || \ y := 2) \)
2. \( \{ x = 0 \} \ (x := 1 \ || \ x := 2) \)
3. \( \{ x = 0 \} \ (x := x + 1 \ || \ x := 3) \)

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 $x = 1 \lor x = 2$ 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.

\[
\begin{align*}
\{ x = 0 \} & \quad x := 3; r := x; r := r + 1; x := r \quad \{ x = 4 \} \\
\{ x = 0 \} & \quad r := x; x := 3; r := r + 1; x := r \quad \{ x = 1 \} \\
\{ x = 0 \} & \quad r := x; r := r + 1; x := 3; x := r \quad \{ x = 1 \} \\
\{ x = 0 \} & \quad r := x; r := r + 1; x := r; x := 3 \quad \{ x = 3 \}
\end{align*}
\]

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. $\{ \neg x \land \neg y \} \quad x \uparrow \parallel \quad y := x$

5. $\{ \neg x \land \neg y \} \quad x \uparrow \parallel \quad [x]; y := x$

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 $x↑$ 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. $\{¬x \land ¬y\} \left( [x↑; x↓] \parallel [y↑; y↓] \right)$
7. $\{¬x \land ¬y\} \left( [x↑; x↓] \parallel [x↑; y↑; y↓] \right)$

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:

$x↑; x↓; x↑; x↓; \ldots$
$y↑; y↓; y↑; y↓; \ldots$

The first interleaving only has actions from process $*[x↑; x↓]$, and the second only has actions from $*[y↑; y↓]$. 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 $*[x↑; x↓]$ 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 $\parallel$ is weakly fair. We sometimes denote $S_1\parallel S_2$ 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 \( c.X \) which tells us how often the command has executed. More formally, the counter \( c.X \) 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 \( c.A - c.B \). Two commands \( A \) and \( B \) are said to be *synchronized* if the difference \( c.A - c.B \) is bounded.

**Example.** Consider the following three CHP processes.

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

In process 1, the difference \( c.A - c.B \) 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 \( 0 \leq c.A - c.B \leq 1 \).

\[
\]

Similarly, in process 3 we have \( -1 \leq c.A - c.B \leq 1 \). 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 \( c.A - c.B \) 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 \( c.A - c.B \) 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\(^\dagger\)

\[
0 \leq c.S - c.R
\]

The maximum difference \( c.S - c.R \) 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

\(^\dagger\) \( c.R \) can be greater than \( c.S \) 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:

\[ cS = cR \]

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,

\[(S!e|R?v) \equiv v := e\] .

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:

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

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 \( cS = cR \) 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 \( qX \) to reason about a blocked action. The variable \( qX \) 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, ¬qS ∨ ¬qR 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 S, has the following two properties:

\[ S \Rightarrow qR \]
\[ qR \Rightarrow \diamond S \]

The first property states that if the probe of S is true, then qR must be true as well—i.e., there is a pending R-action. The second property states that if qR 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 qS as the number of S-actions that are blocked. If the slack of the channel is k, the safety property for the channel is:

\[ 0 \leq cS - cR \leq k \]

The quantity k is denoted kS. The progress requirement is:

\[ (qS = 0 \lor qR = 0) \land (qS > 0 \Rightarrow cS - cR = k) \land (qR > 0 \Rightarrow cS = cR) \]

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:

\[ \overline{S} \Rightarrow (cS - cR < k \lor qR > 0) \land (cS - cR < k \lor qR > 0) \Rightarrow \diamond \overline{S} \]
\[ \overline{R} \Rightarrow (cS > cR \lor qS > 0) \land (cS > cR \lor qS > 0) \Rightarrow \diamond \overline{R} \]

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 qR 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 cR = cS). 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 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 S become false. However, when 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.

\textbf{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, $qR$ 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:

$$\overline{S} \rightarrow \ldots \ S!0$$

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:

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

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 $cS_1 = cS_2$. 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

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

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

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>bullet, simultaneous composition</td>
</tr>
<tr>
<td>;</td>
<td>comma, parallel composition</td>
</tr>
<tr>
<td>;</td>
<td>semicolon, sequential composition</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

$$\ast [[X \rightarrow X?; Z!a]$$

$$| Y \rightarrow Y?; Z!a ]$$

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:

$$\ast [[X \rightarrow Z!(X?)]$$

$$| Y \rightarrow Z!(Y?) ]$$

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 $cX + cY = cZ$.

One problem with the merge process is that it is possible for it to never receive a value on port $X$, even if $X$ 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:

$$\ast [[X \rightarrow Z!(X?); [Y \rightarrow Z!(Y?) | \neg Y \rightarrow \text{skip}]]$$

$$| X \rightarrow Z!(X?); [X \rightarrow Z!(X?) | \neg X \rightarrow \text{skip} ]$$

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 $X$ becomes true, it remains true, and the selection statement $[\overline{X} \rightarrow Z!(X?) | \overline{X} \rightarrow \text{skip}]$ will eventually have only one true guard. An examination of the process shows that once $X$ 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 $[\overline{Y} \rightarrow Z!(Y?) | \overline{Y} \rightarrow \text{skip}]$ 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 $\overline{Y}$, and find it to be true (because $Y$ was false). Since the false value of the probe is unstable, when we evaluate the guard $Y$ 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:

$$M_1 \equiv *[\overline{X} \rightarrow Z!(X?) \mid \overline{X} \rightarrow \text{skip}]; [\overline{Y} \rightarrow Z!(Y?) \mid \overline{Y} \rightarrow \text{skip}]$$

A problem with the fair merge $M_1$ is that it “busy waits”—if both $X$ and $Y$ 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 $X$ or $Y$ to be true. The resulting merge is shown below:

$$M_2 \equiv *[\overline{X} \lor \overline{Y}]; [\overline{X} \rightarrow Z!(X?) \mid \overline{X} \rightarrow \text{skip}]; [\overline{Y} \rightarrow Z!(Y?) \mid \overline{Y} \rightarrow \text{skip}]$$

Depending on the environment, $M_1$ may be more or less efficient than $M_2$. If the environment is very active and we find that the condition $\overline{X} \lor \overline{Y}$ 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.

$$*[\{i : 0..9 : \overline{X}_i \rightarrow X_i!a; Z!a\};]$$

A generalized fair merge is given by:
### 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:

$$\star[(\forall i : 0..9 : X_i) \mid (i : 0..9 : [X_i \rightarrow Z!(X_i?) \mid \neg X_i \rightarrow \text{skip})]$$

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:

$$\star[ \ X?a; \ Y!a, Z!a \ ]$$

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

$$\star[ \ X?a; \ Y!a; \ X?a; Z!a \ ]$$

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:

$$\star[ \! C?c, X?a; \! [c \rightarrow Y!a \! \mid \neg c \rightarrow 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:

$$\star[ \! C?c; \! [c \rightarrow X?a \! \mid \neg c \rightarrow 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 \( C_1, C_2, \ldots, C_n : E \), where \( C_1, \ldots, C_n \) are the input ports, and \( E \) is an expression. Channels from the list \( C_1, \ldots, C_n \) used in \( E \) evaluate to the value of the pending input. The probe expression is \textit{false} if \( C_i \) is \textit{false}, for any \( i \). Otherwise it evaluates to \( E \). Given this extended syntax, we can define \( X \) to be an abbreviation for \( X: \text{true} \). Using this new syntax, we can write the controlled split without using any auxiliary storage as \(*[ [C \cdot \lnot C \rightarrow Y!(X?), C?|\lnot C \rightarrow Z!(X?), C?]] \). Note the use of \( C? \) to complete the communication action without the use of any variable. The controlled merge is \(*[ [C \cdot C \rightarrow Z!(X?), C?|\lnot C \rightarrow Z!(Y?), C?]] \).

2.11.5. Reactive Process Structure

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

\[
*[ [G_1 \rightarrow S_1 \]
\[ \| G_2 \rightarrow S_2 \]
\[ \| \ldots \]
\[ \| G_n \rightarrow S_n \]
\]

Operationally, this process structure can be described as follows: “repeatedly execute the following: wait for some \( G_i \) to be \textit{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 \textit{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:

- \textit{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:

\[
\begin{align*}
n &:= 0; \\
\{ \text{push} \land n < N \rightarrow \text{push}!x[n]; n := n + 1 \\
\text{pop} \land n > 0 \rightarrow \text{pop}?x[n-1]; n := n - 1 \}
\end{align*}
\]

The process maintains the property that the elements stored in the stack are \(x[0], x[1], \ldots, x[n-1]\), 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 \(x[0], \ldots, x[n-1]\) 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:

\[
\begin{align*}
\{ \text{push} \rightarrow \text{push}!x \\
\text{pop} \rightarrow \text{pop}?x
\}
\end{align*}
\]

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:

\[
\begin{align*}
\{ \text{stack is empty} \} \text{push}!x; \\
\{ \text{stack is full} \} \text{pop}?!x
\end{align*}
\]

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

\[
\begin{align*}
\text{push} & \rightarrow \text{push} ? x \{ \text{full} \} \\
\text{pop} & \rightarrow \text{get} ? x; \text{pop} ! x \{ \text{empty} \}
\end{align*}
\]

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:

\[
\begin{align*}
\text{push} & \rightarrow \text{put} ! x; \text{push} ? x \{ \text{full} \} \\
\text{pop} & \rightarrow \text{pop} ! x \{ \text{empty} \}
\end{align*}
\]

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...
A neater coding of this same program is shown below:

\[
\star \begin{cases}
  \{ \text{empty} \} & \{ \text{push} \} \rightarrow \text{push}?x; \{ \text{pop} \} \rightarrow \text{get}?x;
  \\
  \{ \text{full} \} & \{ \text{push} \} \rightarrow \text{put}!x; \{ \text{pop} \} \rightarrow \text{pop}!x;
\end{cases}
\]

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:

\[
\begin{align*}
&\text{full}\downarrow; \\
&\text{push} \land \neg\text{full} \rightarrow \text{push}\?x; \text{full}\uparrow \\
&\text{pop} \land \neg\text{full} \rightarrow \text{pop}!(x, \text{false}) \\
&\text{push} \land \text{full} \rightarrow \text{put}!x; \text{push}\?x \\
&\text{pop} \land \text{full} \rightarrow \text{pop}!(x, \text{true}); \text{get}?(x, \text{full})
\end{align*}
\]

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:

\[
\begin{align*}
&\text{full}\downarrow; \\
&\text{push} \land \neg\text{full} \rightarrow \text{push}\?x; \text{full}\uparrow \\
&\text{pop} \land \neg\text{full} \rightarrow \text{pop}!(x, \text{false}) \\
&\text{push} \land \text{full} \rightarrow \text{push}\?y; \text{put}!x; x := y \\
&\text{pop} \land \text{full} \rightarrow \text{pop}!(x, \text{true}); \text{get}?(x, \text{full})
\end{align*}
\]

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.

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 2.6. Binary Tree FIFO.](image)

![Figure 2.7. Serial-to-parallel Converter.](image)
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 \( T(x \uparrow) = O(1) \). 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.,

\[ T(x := E) = O(1) + O(T(E)) \]

where \( T(E) \) is the time taken to evaluate the expression \( E \). The time \( T(E) \) 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,

\[
T(S_1 \parallel \cdots \parallel S_N) = \max_i T(S_i)
\]

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 \( \max_i T(S_i) \) as before, and the time taken by the concurrent composition of the \( T \)-statements is given by \( \max_i T(T_i) \). So the combined statement takes at least time \( \max_i T(S_i) + \max_i T(T_i) \). 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 \textit{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:

\[
T((S_1 \parallel \cdots \parallel S_N); (T_1 \parallel \cdots \parallel T_M)) = \max_i T(S_i) + \max_i T(T_i) + O(\log M + \log N)
\]

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

\[
T(C!e) = T(e) + O(1)
\]

\[
T(C?v) = O(1)
\]

Consider the deterministic selection statement \([G_1 \rightarrow S_1 \cdots \cdots G_N \rightarrow S_N]\) where we find \( G_i \equiv \text{true} \). 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 \text{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 \textbf{true}, we have:

\[
T([G_1 \rightarrow S_1 \quad \cdots \quad G_N \rightarrow S_N]) = \max_i T(G_i) + T(S_i) + O(\log N)
\]

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:

\[
\begin{align*}
[i = 0] & \rightarrow x[0] := v \\
[i = 1] & \rightarrow x[1] := v \\
& \cdots \\
[i = N - 1] & \rightarrow x[N - 1] := v
\end{align*}
\]

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 \textit{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 \textit{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 \textit{pop} operation for the eager stack is \( O(1) \).

A CHP program is said to exhibit \textit{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 \textit{push} and \textit{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.
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 \( \tau \) 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 \( \tau \). 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:

\[
* \{ L?x; \ Ain!x, \ Bin!x \} \parallel * \{ Aout?x, \ Bout?y; \ R!x \}
\]

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

\[
* \{ L?x; \ Ain!x; \ L?x; \ Bin!x \} \parallel * \{ Aout?x; \ R!x; \ Bout?x; \ R!x \}
\]

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 \( PB \).

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 \( S_1, \ldots, S_N \), then

\[ \mathcal{E}(S_1 \parallel \cdots \parallel S_N) = \sum_i \mathcal{E}(S_i) \]

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

The energy index of \( (S_1 \parallel \cdots \parallel S_N); (T_1 \parallel \cdots \parallel T_M) \) 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,

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

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

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

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

\[
\begin{aligned}
&\neg i_0 \rightarrow \neg i_1 \rightarrow \ldots \neg i_{N-1} \rightarrow x[0] := v \\
&\quad \Box i_1 \rightarrow \ldots \\
&\quad \Box i_{N-1} \rightarrow x[1] := v
\end{aligned}
\]

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:

\[
\begin{aligned}
&\ast [ \ A?x, B?y; \ C!(x + y) \ ]
\end{aligned}
\]

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

\[
0 \leq c_A - c_C \leq 1
\]

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

\[
0 \leq c_A - c_C \leq \log N
\]