Chapter 3

Source Language

The most important feature of a description language is that it must be able to describe naturally the types of applications that interest the user. Basic features essential to the systems being described must be expressible in the source language, and should be expressed in a natural and convenient way. While it is usually possible to describe the same system using a variety of different languages, it is often the case that one set of language constructs can express the desired feature more naturally than another. In this case, interesting systems are those which can eventually be built as VLSI circuits. This eventual target requires that the language features included in the source language may be somewhat different than the standard features of a programming language targeted towards producing code for a general purpose computer.

One reason that the target of circuits requires a novel choice of source language is that circuits are inherently concurrent at a very low level. Even in a small circuit, whether implemented as an integrated circuit, or as a collection of higher level circuit modules, there may be a great many parts of the circuit active at the same time. In addition, many applications of VLSI are of systems that are concurrent at a higher level. It is therefore essential that the source language allow the programmer explicit control over concurrent as well as sequential computation. Most traditional concurrent programming languages allow such explicit control, but also include various distractions such as complicated data domains, a choice of variable binding constructs, recursion, buffered communication channels for message passing, and process spawning and deletion. An asynchronous circuit, in contrast, consists of a fixed set of modules communicating over a fixed set of wires. These wires may be used as simple handshaking wires or as unbuffered channels for communication between circuit modules. Any buffering desired in the communication channels must be included explicitly, and must, because it is implemented in actual hardware, be of finite depth. A physical circuit cannot replicate itself to provide for recursion, or provide extra hardware for a new process spawned at run time. The wires cannot move to reconfigure a hard-wired communication topology, Control over the concurrent aspects of a system may also help control costs in the target circuit. One cost is the added hardware required for parallel execution. Replicating parts of the circuit can increase execution speed at the cost of increased circuit area. If the control of concurrency is explicit, this allows the system designer to exchange increased circuit speed due to parallel execution for the extra hardware that such parallel execution requires by modifying the source program. If the target circuit area is at a premium, then sequential execution on a single hardware component may be required.

Communication is another costly part of VLSI circuits because input/output pins are a very "expensive" resource. Explicit control over communication in the source language can help the designer control these costs. The result is that the source language must allow for explicit control over concurrency and communication. At the same time, given this application, the language must not be confounded with features that are difficult to implement in hardware.

3.1 Concurrent Communicating Programs

A recurring problem with creating systems that execute concurrently is finding a way to express the concurrent nature of the system in a natural way. Many existing languages depend on the existence of a uniformly accessible shared memory provided by a conventional computer. This can require complicated schemes to ensure that there are no problems with concurrent access to common memory locations.

Another way to think of a concurrent system is to imagine a collection of processes, all running concurrently, and with their own private memories, that cooperate to perform some task. The processes cannot share memory locations and instead cooperate by sending messages to each other to pass information about the state of the system. This model for cooperating processes was proposed by Hoare in [35] and is known as Communicating Sequential Processes, or CSP [35, 37]. The communication primitive used in CSP is a synchronizing model. The sending process must suspend execution until the receiving process is able to receive the data being sent. At this point, the two processes synchronize and the communication takes place as a single atomic action. Systems that operate in this manner are a nicely intuitive model for designing concurrent systems without the complications of imposing non-interference or disjointness requirements on memory references. In addition, CSP is a standard language for theoretical investigations into concurrent systems [8, 9, 1, 34]. This allows many important theoretical results to be applied to systems that are designed using the same philosophy.

Designing systems of concurrent communicating processes is especially appealing when the ultimate target is parallel hardware. Whether this hardware is a multiprocessor system of some sort, or a collection of circuit modules assembled on a single integrated circuit, it is natural to imagine distributing the communicating processes amongst the parallel hardware.

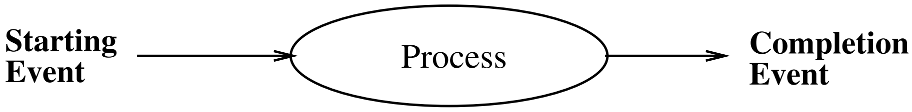

Figure 3.1: A Generic OCCAM Process

3.2 OCCAM

The source language used for the system described in the following chapters is a variant of CSP known as OCCAM [42, 59]. OCCAM is a programming language based on CSP, but with some different features and syntax. It was designed to run on the Inmos Transputer [43, 39] which had a strong influence on its design. It was designed to support concurrent applications in which many parts of a system operate separately but with interaction. Control over concurrent and sequential aspects of the computation, as well as the communication between the processes, is explicit. In particular, OCCAM is useful for describing small-grain processes communicating over fixed, synchronized links called channels. The exact form of those communication channels was influenced by the communication hardware available on the Transputer. OCCAM requires that the channels that define the communication paths between the processes, as well as the processes themselves, be fixed at compile time.

Although OCCAM was intended to describe software processes running on a Transputer, it is also a very useful notation for describing hardware processes. When used in this way, an OCCAM process can be viewed as a black box that begins execution after some event signals it to start, and then reports completion by producing some ending event. These events could be signals in the software environment running the OCCAM program, or electrical events in the eventual hardware implementation. An abstract representation of an OCCAM process is shown in Figure 3.1. These processes may be connected in a variety of ways and embedded in larger black boxes to construct larger systems using the same type of starting and stopping control events. By allowing these sets of processes to be viewed from the outside as a single process, OCCAM can capture the hierarchical structure of a system.

The variant of OCCAM used here departs from the standard language in several ways. Arrays of variables and replicated commands are omitted, as are commands that deal with resource allocation and priority such as *pri-alt*, *placed-par*, *pri-par*, and *allocate*. Commands that deal with timing such as *now* and *after* are also omitted. A weaker version of *pri-alt* called *lex-alt* is added to the command set of basic OCCAM.

3.2.1 Primitive Processes

As in CSP, input and output are fundamental language constructs in OCCAM. OCCAM adds assignment to variables as a third primitive process, and OCCAM programs are constructed using these three primitive processes. Communication between processes over channels is synchronized between the two participating processes but not dependent on any global signal. If a process tries to output a value to a channel, it must wait before the communication can take place until some other process tries to input a value from that same channel. Likewise for an input, which must wait for a matching output. There is no automatic buffering in the channels themselves. If no variable is mentioned in an input statement, and no expression in the corresponding output statement, then no value is passed and the communication is reduced to synchronization of the two processes. Note that although a synchronization of this sort is written as an input and an output process communicating, it is actually a symmetric operation and it makes no difference which process is executing which communication primitive. This style of communication maps well to the method in which circuit modules communicate over wires. In addition, the fact that communication must be specified explicitly is useful in controlling the relatively expensive communication costs that exist at the circuit level. Specifically, the three primitive processes are:

Input The input process accepts a value from an input channel. The process must synchronize with the output process providing the value before communication takes place. The value received from the channel is stored in a variable. If no data is passed this input reduces to simple synchronization of the two processes.

Output The output process sends the value of some expression through a channel to an input process. The two processes must synchronize before information is communicated. If no expression is mentioned then the output process reduces to synchronization.

Assignment The result of evaluating an expression is assigned as the value of some variable.

3.2.2 Processes Constructs

Primitive processes may be combined to create more complex processes using a set of process constructs. These constructs are described in more detail below.

Sequential Construct - *Seq* Each of the component processes is executed in turn. The sequential construct terminates when the last component process has finished. This is the standard sequential statement execution model used in standard non-concurrent programming languages.

Parallel Construct - *Par* The component processes are executed concurrently. The construct terminates when all of the component processes have finished. Two component processes in a parallel construct may communicate using a shared channel. This is the only method OCCAM process may use to share data. Any variables that are shared by multiple processes running concurrently may only be read; their value may not be changed by either assignment or input primitives.

Conditional Construct - *If* The components of an *If* construct are processes paired with Boolean expressions. These components are similar to familiar guarded commands where each command has an associated guard to indicate that the command may be executed [22]. These pairs are considered in the order in which they appear in the code. The Boolean expression associated with each process is tested and the first component whose expression evaluates to *True* executes. When that process completes, the entire construct finishes. At most one component process will execute, and if no Boolean expressions are true the construct terminates with no other effect.

Alternative Construct - *Alt* Component processes are guarded by both a Boolean expression and a guard process. The guard process may be an input process which is ready to execute if its partner output process is ready to execute, or the *skip* process which is always ready to execute and then simply terminates with no other effect. The component process is not available for execution unless both the Boolean expression is *True* and the guard process is ready to execute. An alternative construct waits until at least one of the guarded commands is ready to execute and then selects one of the available processes for execution. Exactly one of the ready processes is eventually selected and executed. The construct finishes when the selected process finishes.

Lexical Alternative Construct - *Lex-Alt* Similar to a plain Alternative construct except that rather than selecting any of the guarded processes which are valid, the guards of the component processes are checked in the same order that they are written in the program text. This is similar to the operation of the Conditional construct except that if, after the first pass through the components, no guards are found to be valid, the construct repeats the guard checking process until one becomes valid and a process may be executed.

Repetitive Construct - *While* The single component process is guarded with a Boolean expression. The component process is executed repeatedly until the expression evaluates to *False*, after which the construct terminates. If the conditional expression is initially false then the process is not executed and the construct terminates with no other effect.

3.2.3 Named Processes

OCCAM allows processes to be named so that they can be used as macros in further processes. The text of these named macros will be substituted for all occurrences of that name in subsequent processes. The effect of this substitution is to make a copy of the named macro process and to execute it in place of the substitution. No recursion is allowed in OCCAM macros.

The macro definition includes the formal parameters for that macro. These must match the actual parameters in the call to the macro. The macro may contain an identifier which is the same as an identifier substituted as an actual parameter. In this case, the actual parameter refers to the identifier in use at the point in the program where the substitution takes place.

An identifier which is used, but not declared, in a macro is known as a free identifier. A free identifier may be the same as an identifier at use in the program where the substitution takes place. In this case, the free identifier is the identifier in use at the point in the program where the macro is declared.

3.3 A LISP-Like Syntax for OCCAM

The syntax used for the example in Figure 1.1 of Chapter 1 was based on the original syntax of OCCAM [42]. The actual syntax used to write OCCAM programs for evaluation by this translation system is also quite similar to that original syntax, but uses a structure similar to that of LISP [93] for program notation. This syntax was designed mostly to facilitate the implementation of the prototype compiler which is written in LISP. One main feature of this new syntax is that parentheses are used for block structuring instead of indentation alone as in the standard OCCAM syntax. In addition to the added parentheses, expressions are written in prefix notation rather than infix notation, and primitive processes and named procedure calls are expressed as lists in the manner of LISP. This LISP-like syntax of the language is shown in Figure 3.2. Optional items are enclosed in square brackets and braces denote items that may be repeated zero or more times.

The only data type available in this dialect of OCCAM is a simple word consisting of a fixed number of bits. Variables and channels may be defined with any number of bits, with a *Subseq* operator used to access contiguous subparts of a variable or channel. The number of bits in a variable is specified by giving the variable width in angle brackets after its definition. If no width is given, the variable is assumed to be a single-bit Boolean value. The width of the data path of a channel is specified in the same way. If a channel is used for synchronization only with no data being passed it must have a width of zero. Note that although there is an explicit *Block* process used to create a new computational block, *Seq* and *Par* processes implicitly create a new block with possible variable and channel declarations.

Even though OCCAM is a language designed to support concurrent applications, the sequential construct *Seq* is used in enough situations to warrant a simple shortcut in the notation. For convenience a *Seq* construct with no local declarations may be written as a sequence of processes with no *Seq* keyword.

```
Program       =  Process
Process       =  Primitive | Construct
Primitive     =  Assignment | Input | Output | Skip
Assignment    =  (Set  Variable Expression )
Input         =  (?   Channel [ Variable ] )
Output        =  (!   Channel [ Expression ] )
Skip          =  (Skip)
Construct     =  (Seq  Decl {Process} ) | (Par  Decl {Process} )
                 | (Alt  {Guarded-Proc} ) | (Lex-Alt  {Guarded-Proc} )
                 | (If  {Cond-Proc} )
                 | (While  Cond Process ) | (While  Cond Implicit-Seq )
                 | (Block  Decl Process ) | (Block  Decl Implicit-Seq )
                 | ( macro-name { var-name | chan-name} )
Implicit-Seq  =  {Process}
Guarded-Proc  =  ( Guard Process) | ( Guard Implicit-Seq )
Guard         =  (  Cond Input ) | ( Cond Skip )
Cond-Proc     =  ( Cond Process) | ( Cond Implicit-Seq )
Decl          =  ( {Decl-Part} )
Decl-Part     =  (Chan  {chan-name [ < int > ]} )
                 | (Var  {var-name [ < int > ] } )
Variable      =  var-name | (Subseq var-name int int )
Channel       =  chan-name | (Subseq chan-name int int )
Expression    =  Variable | ( gate-name { Expression })
Cond          =  Boolean-expression
Macro-Def     =  (Process macro-name Decl Process )
                 | (Process macro-name Decl Implicit-Seq )
```

Figure 3.2: Source Language Syntax

```
PROC buffer (CHAN A B)          – Define a simple buffer
  WHILE TRUE                    – Repeat forever
    VAR temp:                   – Define a temporary variable
    SEQ                         – Sequential composition
      A ? temp                  – Get value from Channel A, store in temp
      B ! temp                  – Send value in temp to channel B

CHAN input, output, mid:        – Define some channels
PAR                             – Parallel Composition
  buffer(input, mid)            – of two single-place
  buffer(mid, output)           – buffer macros
```

Figure 3.3: Traditional OCCAM Code for a Two-Place FIFO Buffer

```
(Process buffer ((Chan A B))    ; Define a simple buffer
  (While True                   ; Repeat forever
    (Seq ((Var temp<8>))        ; Sequential composition
      (? A temp)                ; Get value from channel A, store in temp
      (! B temp))))             ; Send value in temp to channel B

(Block ((Chan input<8> output<8>))
  (Par ((Chan mid<8>))          ; Parallel composition
    (buffer input mid)          ; of two single-place
    (buffer mid output))        ; buffer macros
```

Figure 3.4: LISP-like OCCAM Code for a Two Place Fifo Buffer

3.4 Example Code

3.4.1 A FIFO Buffer

As an example of using this syntax to write OCCAM processes, the code for a simple first-in first-out (FIFO) buffer is shown in Figure 3.4 in both original OCCAM notation and the new LISP-like syntax. The *Process* construct is used to define a macro named *buffer* with two channel arguments. The body of the *buffer* macro consists of a non-terminating *While* loop. The *Seq* construct declares an 8-bit wide local variable and performs an input and output primitive in sequence.

Instantiation of this macro will result in a process that accepts input on its *A* channel, stores the value in a local variable, then outputs that value on its *B* channel. This process is

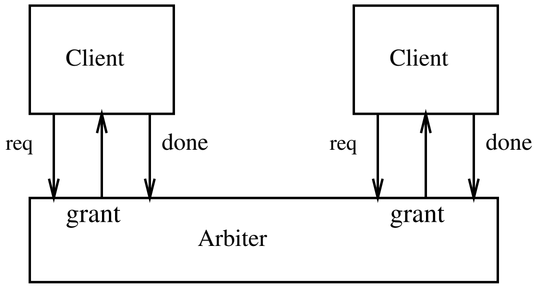

Figure 3.5: A Simple Two Input Arbiter

repeated forever and implements a buffer that can store a single 8-bit value.

Two instances of this macro can be combined in a *Par* construct to implement a FIFO buffer that can buffer two 8-bit words. A local channel is defined in the parallel construct, and the *buffer* macros are connected so that the output channel of the first buffer and the input channel of the second buffer are connected. Note that this channel is local to the parallel construct and hidden from the external interface of the *Block* construct. The interface with a single input channel and single output channel is used both for the single place buffer and two place buffer. The only observable difference at the interface is that the second buffer can store two words before it must output a word.

The circuit resulting from the translation of the FIFO buffer program in Figure 3.4 will be described in Chapter 5.

3.4.2 Distributed Mutual Exclusion

The next example considers a process that implements mutual exclusion for a set of processes. Whenever more than one independent process desires exclusive access to a shared resource, some sort of arbitration must be used. The purpose of an arbiter is to make sure that only one of the competing processes has access to the shared resource at any time [83]. A simple arbiter grants access to the shared resource to exactly one of two contenders at a time. The contending processes communicate with the arbiter using three channels as shown in Figure 3.5. Each contending process may signal on its *Req* line that it wants access to the resource. The arbiter responds on the *Grant* line to exactly one of the contending processes indicating that the process has access and may use the resource. When that process is finished it releases the arbiter with a signal on the *Done* line so that the arbiter may consider other requests.

Rather than have each of the processes communicate with a single arbiter process, it is sometimes desirable to distribute the arbitration amongst the contending processes [53]. The example shown here is based on the algorithm used by Martin in [55]. Other algorithms are explored, along with their resulting circuits, in Chapter 5.

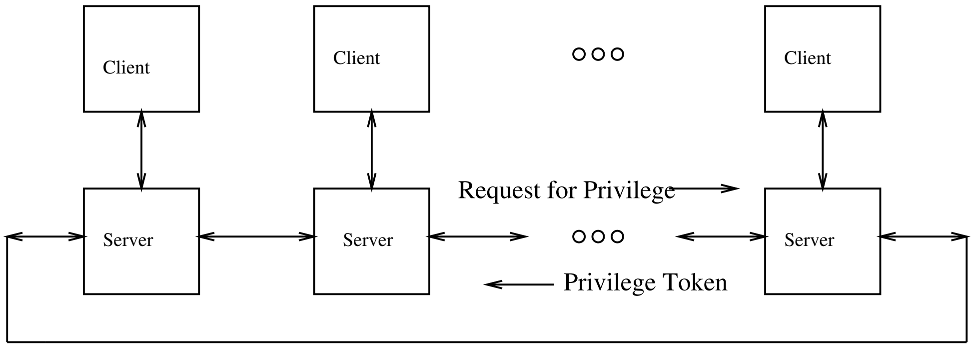

Figure 3.6: An Arbiter Distributed in Multiple Server Processes

In this algorithm, each contending process, called a *client*, communicates with a local process called a *server* that grants access to the shared resource as shown in Figure 3.6. Note that lines in this figure may denote multiple communication channels. There is a single server process associated with each client process. While the client processes are independent and need not communicate with each other, the server processes must communicate with each other. For this example, the servers are connected in a ring. A token known as the *privilege* is passed around this ring by the servers so that at any time exactly one of the server processes holds the privilege. Only the server that holds the privilege may grant access to the shared resource to its client. The servers pass requests for the privilege around the ring to their right and receive the token from their right.

An example of using this dialect of OCCAM and this LISP-like syntax is a process to implement this algorithm for distributed mutual exclusion. The code to implement each server process is shown in Figure 3.7 with the code fragment from the client process that communicates with the server shown in Figure 3.8. This algorithm for implementing distributed mutual exclusion has been implemented by Martin as a self-timed circuit by compiling a program written an a variant of CSP using a somewhat different technique than presented here [55].

The code for the server is shown as a macro process named *server*. The process takes three channels as arguments: the *Serv* channel is connected to the client process, and the *Right* and *Left* channels communicate with the other servers as shown in Figure 3.6. The body of the macro uses a single Boolean local variable to hold the value of the privilege and then executes a non-terminating loop. Note that exactly one of the servers in the ring must be modified such that its privilege variable is true after a global clear signal.

The body of the loop is an *Alt* construct that uses guarded input channels to choose whether to accept communication from the client on the *Serv* channel or from the server on the left over the *Left* channel. The guards on the input channels indicate whether there is a pending communication on that channel. Notice that the code contains two consecutive outputs to the right channel. Because no expression is mentioned, no data is

```
(Process server ((Chan Serv Right Left))
  (Block ((Var priv))            ; Define local variable.
    (While True                  ; Repeat forever.
      (Alt                       ; choose an alternative
        ((True (? Serv))         ; Request from client.
          (If ((not priv)        ; Not privileged.
              (! Right)          ; Ask for the privilege.
              (! Right)))        ; Got the privilege.
          (? Serv)               ; Grant access to client.
          (? Serv)               ; Client is finished.
          (Set priv True))       ; Remember privilege.
        ((True (? Left))         ; Request for privilege.
          (If ((not priv)        ; Not privileged.
              (! Right)          ; Ask for the privilege.
              (! Right)))        ; Got the privilege.
          (? Left)               ; Pass privilege left.
          (Set priv False))))))  ; Forget privilege.
```

Figure 3.7: Code for Distributed Mutex Server

```
  ⋮
(! Serv)             ;request access
(! Serv)             ;access is granted when this communication
                     ; synchronizes with the server
  ...                ;use resource
(! Serv)             ;signal completion
  ⋮
```

Figure 3.8: Code in Client that Communicates with Server

passed and this communication reduces to synchronization. Note that synchronization is a symmetric form of communication. Because no data is passed, it makes no difference which process executes an input and which process executes an output. The only function of the communication is to synchronize the two participating processes. The fact that OCCAM channels are defined to be uni-directional dictates that the communicating processes must agree on the "direction" of the communication, but for simple synchronization, this choice may be made arbitrarily.

The first synchronization signals to the server on the right that the current server wants the privilege. The second synchronization means that the server on the right has passed the privilege to this server and so this server now owns the privilege. This synchronization corresponds to the two inputs on the *Left* channel in the other alternative body. The same argument explains the multiple output synchronizations on the *Serv* channel. Recall that OCCAM semantics require unbuffered synchronization. When one process sends a synchronization signal it must wait for the other process to reply.

This example is only one of many different ways that distributed mutual exclusion can be implemented. A variety of different versions of this basic scheme, along with their circuit implementations, are presented in Chapter 5.