Circa: A Hardware Description Language

by

Peter Somers

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
the Department of Computer Science

Winnipeg, Manitoba

ISBN 0-315-34009-6

CIRCA: A HARDWARE DESCRIPTION LANGUAGE

BY

PETER SOMERS

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

ACKNOWLEDGEMENTS

Many thanks are due to my thesis advisor, Dr. Michael Miller, and to Gerhard Dueck and Chris Carson for their helpful suggestions and criticisms.

In addition, a debt of gratitude is owed to the Natural Science and Engineering Research Council for their financial support during my research.

ABSTRACT

Hardware and software design have many similarities. In this thesis, these similarities are examined and the technique of abstraction, common in software development, is applied to hardware design and synthesis.

In the spirit of software languages, hardware description languages are seen as a way of assisting the hardware design process. The design can be verified by simulation and synthesized by silicon compilers.

A hardware description language, Circa, is proposed. The implementation of an interpreter or simulator is discussed in detail. Some ideas for silicon compilation of Circa programmes are included.

CONTENTS

LIST OF FIGURES

Chapter I
INTRODUCTION

A number of formalisms are used in the hardware design process, for example: boolean algebras, state diagrams, stick diagrams [Mead80], and timing charts. The one attribute they have in common is that they hide certain details of the hardware structure such as chip structure, electrical characteristics, etc. In other words, hardware formalisms raise the level of abstraction at which the designer works. This aids in design synthesis and in hardware description to other designers or to an automated synthesis system.

These traditional formalisms are fine for designing small circuits, but as circuit complexity increases they are glaringly deficient. Why is this so? It is because their abilities for abstraction are limited to one level only. This is termed horizontal abstraction.

A more general approach, vertical abstraction, allows the designer to substitute details at any level by an "abstraction". By successive applications of this process, the designer can raise the level at which he works.

Hardware description languages (HDLs) allow vertical abstraction. HDLs give the designer a methodology for developing large hardware systems. This methodology is analogous to the software design philosophy and is discussed in subsequent chapters. HDLs also provide a vehicle for describing circuits to other designers in addition to circuit compilers and simulators.

The goal of this thesis is to present Circa, a hardware description language, and some of the ideas that led to its development.

The second chapter describes the hardware design process at a very abstract level. It attempts to tie together the notions of software engineering with hardware design.

In the third chapter, the concept of a hardware description language is explained.

Our HDL is based on a software language called Occam [Inmos84]. Occam is a language oriented towards parallel processing. Chapter four examines the shortcomings of the Occam language for hardware description.

A tutorial to the Circa hardware description language is presented in chapter five.

Circuit compilation using Circa is the subject of chapter six. A microprocessor tailored to the execution of Occam programmes called the Transputer was built by Inmos [Inmos85]. Unlike the Occam/Transputer relationship, the circuit compiler proposed generates hardware tailored to the execution of a single (and unique) Circa programme.

Chapter seven looks at the implementation of the Circa interpreter on a UNIX† computer system.

Finally, chapter eight presents conclusions from this work and looks at some areas for further research.

† UNIX is a trademark of A T & T Bell Laboratories.

Chapter II

THE DESIGN PROCESS

In order to understand the problem, what type of hardware description language is required, we need to better understand the hardware design process.

There are two major approaches to designing hardware just as there are in designing software: bottom-up and top-down [Wirth74].

Bottom-up design involves taking components and assembling them together into larger components. This is done without any prior knowledge of the rest of the design. These larger parts are put together to create even larger components. This process continues until we have a component whose behaviour matches the initial specifications. There are problems with this approach. Bottom-up design often imposes restrictions on a final design or gives an inefficient fit to the optimal solution.

Most designers of large systems use the top-down approach (see figure 1). Illustrated is the step-wise interlevel refinement from of an algorithm to the transistor level [Laws77]. Not evident from the diagram is the intralevel refinement that must take place. Each level, while appearing somewhat flat here, actually contains many sublevels.

Let us examine microprocessor design as an example. Typical software compilers automate the refinement from the high level language to the instruction set architecture level. Most hardware designers tend to restrict their activities to levels below the instruction set architecture. There have been attempts to automate the design process at these levels [Suss81]. Ideally, an algorithm would be translated to a high level language programme and through the lower layers until a complete circuit is realized.

In fact, it is really irrelevant which layers are implemented in hardware or software. The designer may move the hardware/software boundary up or down depending on his constraints (ie: speed, cost, chip yield, market, etc.).

The RISC (Reduced Instruction Set Computer) versus the CISC (Complex Instruction Set Computer) debate is just one such example of a shifting hardware/software boundary [Patt82] [Ungar84] [Rowen86] as shown in figure 2. Complex operations done in hardware on a CISC would be done in software on a RISC. The RISC philosophy is to place simple, frequently used instructions in hardware. Complex operations are simulated by subprogrammes. What we have here is a higher level virtual machine being emulated by a RISC. RISC advocates avoid microcode as it slows down the instruction cycle of the machine [Radin82].

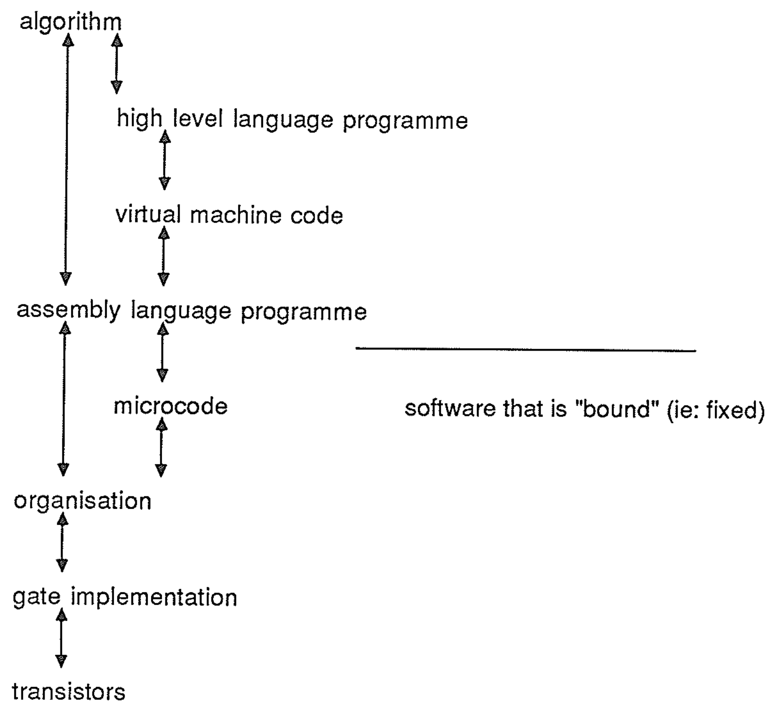

algorithm
high level language programme
virtual machine code
assembly language programme
microcode
software that is "bound" (ie: fixed)
organisation
gate implementation
transistors

Figure 1: Algorithm to Transistor Refinement

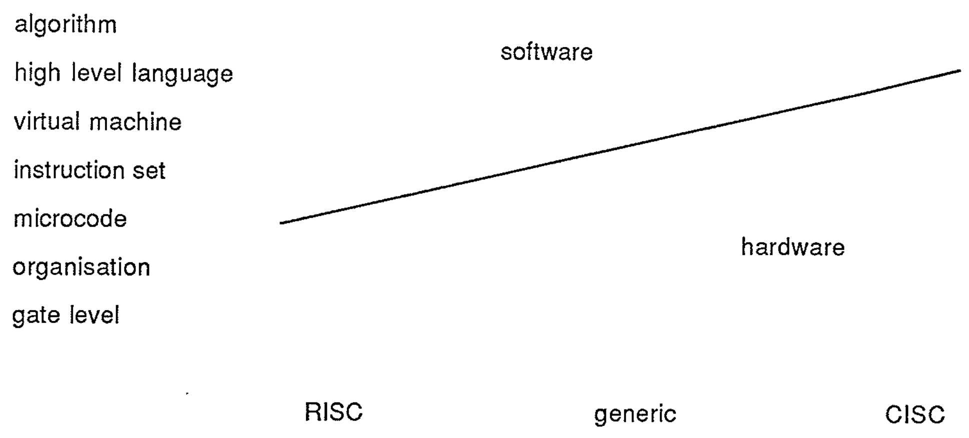


Figure 2: Software/Hardware Partitioning

Constraints that have influenced RISC designers are design time, a small silicon area to work with and a need to have more reliable (bug-free?) hardware. RISC designers also cite an increase in performance as an advantage.

If we look at another example of hardware design, say a traffic light controller, we can see that there are a number of different implementation alternatives. One could use a register and combinational logic. By replacing the combinational logic with a ROM, we would have moved up one level of abstraction with the creation of a horizontal microcodable machine. Moving up even higher still is an assembly language programme and a single chip microcomputer. Although each implementation is different, the external behaviour is still the same.

Even though the software and hardware implementations are expressed in different forms, they are fundamentally the same. Our hardware is a sequential machine which is composed of combinational logic and a state. Imperative software gives its state by the contents of the variables and the programme counter. The analogue of combinational logic is the instruction set of a computer. Memory locations are modified, output is produced and the programme counter is adjusted according to the flow of control. The whole point of an instruction set is to alter the state of the machine and its outputs.

Noting the close relationship that hardware has with software, it would seem desirable to have the same or at least a similar formalism or model for describing both hardware and software. We know that we can certainly describe the behaviour of hardware in software. But could the structure of hardware precipitate out of the formalism?

It is to the designer's advantage not to bind an algorithm until the last possible moment. The designer should be allowed to experiment or, indeed, to just change his mind as to what should be committed in hardware. Another advantage of a universal model is that the compiler itself could make judgements regarding hardware/software partitioning. The compiler could design or create several alternatives and based on a list of constraints given by the designer, analyse those alternatives and return to the user with the best possible solution.

Chapter III

HARDWARE DESCRIPTION LANGUAGES

Hardware can be described in various levels of abstraction: behavioural, functional, structural, logical and, at the lowest level, physical.

At the behavioural and functional levels, processes are interconnected without regard as to their actual implementation or structure [Lewin81]. Strictly speaking, the behavioural level describes the behaviour of a system in its entirety. If the system is broken into submodules, even though each submodule may be given at the behavioural level, the system is considered to be described at the functional level. These two definitions tend to be blurred as most behavioural languages allow some sort of partitioning, thus placing them under the functional language category as well. For simplicity, we will classify functional hardware description languages as behavioural languages. Traditional HDLs of this class have been Register Transfer Level (RTL) languages and have had only qualified success [Kato83].

At the structural level, a system is described in terms of abstract components and their interconnections [Lewin81]. The difficulty with structural and indeed lower level descriptions is that the entire system must be described.

Submodules must be given in terms of lower level components and interconnections; in other words, the submodules must be told not only what to do, but how to do it. This is not the case for the behavioural level. A behavioural language becomes part of the step-wise refinement approach rather than an after thought. Nevertheless, a structural language can help the designer to more readily visualise the circuit that will be synthesized.

A system described at the logic level is given in terms of physically realisable, primitive logic functions (for example nand gates) and their interconnections. In general, designers should avoid designing hardware at this level. It is too expensive for a sizable system due to the number of devices involved. It is also unnecessary with design automation.

Finally, the physical level consists of a description of a digital system in terms of the final implementation technology. An example is that of silicon masks for an integrated circuit. It is far too time consuming and error prone to work at this level.

Only in a small number of specialised cases is manual design necessary either at the logic or physical level. These usually involve power or speed constraints.

Typical design mechanics are given in figure 3. The hardware description language is passed to Simulator A for verification of the design. Simulator A gives an approximate picture of behaviour of the hardware; but what it lacks in parametric accuracy it makes up for in speed. Once the description is found to be suitable, it is passed to the silicon compiler which generates a low level HDL. The Low Level description is at the transistor level. It can be passed to Simulator B for a very accurate, but very slow simulation or to the Mask Generator which produces the masks necessary for construction of the chip (if it is to be implemented in silicon).

The layout that is generated by the Mask Generator may be used by Simulator C to produce a still better picture of the hardware. This would allow the designer to examine the device physics in more detail since it is at this level that the actual device construction is known.

If one has enough confidence in the silicon compiler, simulators B and C could be ignored. A working chip would be the indicator of success.

The designer should not have to know any details of the low level HDL. The designer's work should be contained within the higher level HDL. The HDL should support the design process: the step-wise refinement of the behavioural description.

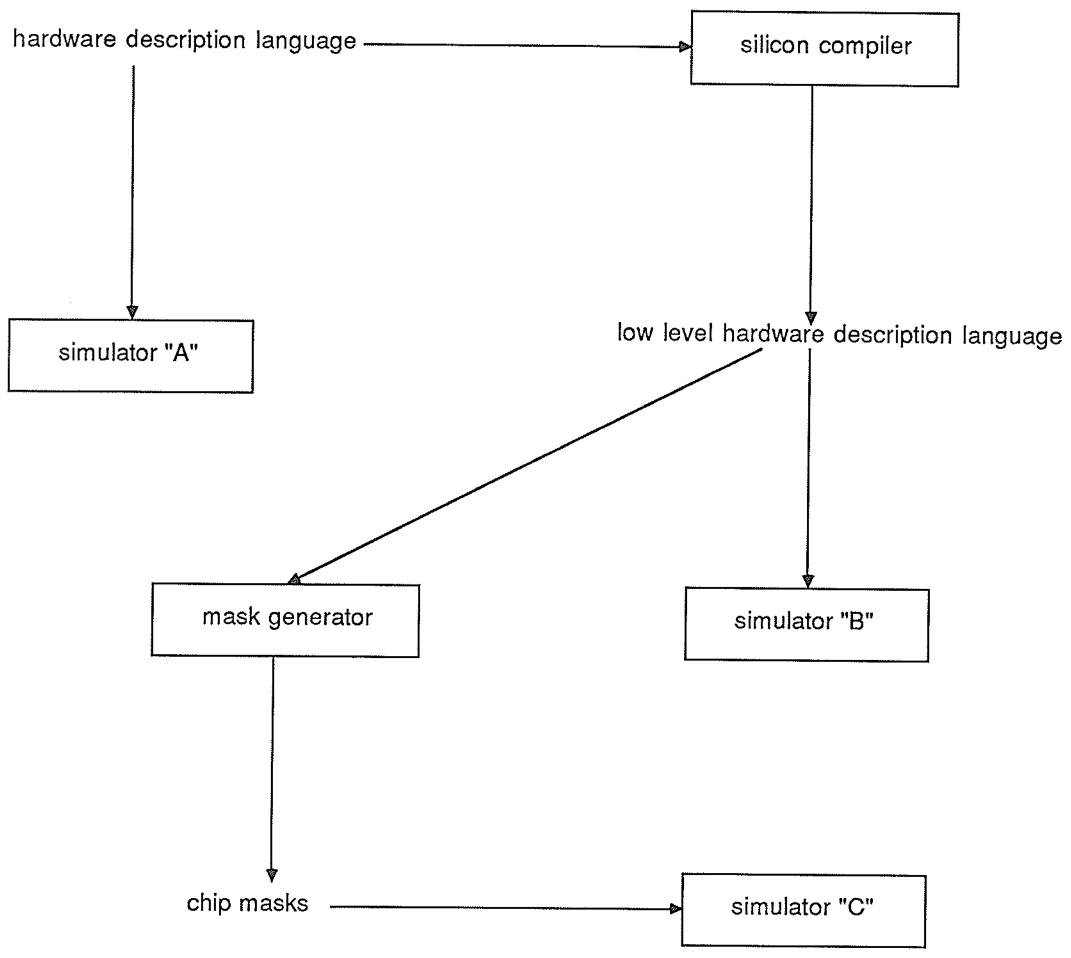


Figure 3: Design Mechanics

Chapter IV

LANGUAGE REQUIREMENTS

From the previous chapter we can see that our hardware description language should be a behavioural language. There must be an easy way of specifying components and interconnections. Subordinate components should be given either as a behavioural model or as smaller parts interconnected together; the latter more likely as the design nears completion. We should not have to finish the design in order to verify it. Only a behavioural language offers this kind of rapid prototyping so important to designers. To reiterate, we are looking for a design tool in the broadest sense.

The language should be as simple and as consistent as possible. This will make it easy to learn and to use. It should be modular; code and type definitions should be sharable between system descriptions.

The HDL should provide a model which is software/hardware independent. The designer must be able to move back and forth between software and hardware without any translation in terms of the model.

An ideal model is that of the process. A process is a sequence of events occuring concurrently with other events. Every level in the design, from high level software to the lowest hardware entity, can be modeled as a process. Is this possible? The following list illustrates that it is not only possible, but quite natural.

LEVEL	EXAMPLE
high level language	real time O.S.
instruction set	hardware scheduling (a simple example is an interrupt vector)
system level	each chip/subsystem is modeled as a process
organisational level	pipeline
gate	each gate is modeled as a simple process

Not only do we want a process model, but we want to make the parallelism as explicit as possible. The circuit compiler should be given enough information to allow the designer control over the fundamental nature of the circuit.

Since we will have a collection of processes, it is assumed that they will be communicating with each other. A communication model must be developed. Several possibilities exist: monitors, message passing via a channel and message passing directly to a process, for example.

The monitor concept turns out to be inadequate [Harl85]. It is less natural on machines without a common store, since the monitor provides only exclusive access to shared store: nothing else. It also reduces parallelism by forcing the calling task to wait while processing inside the monitor takes place. Now if very little processing takes place, then the monitor is analogous to the channel, so we might as well use a channel.

Message passing mirrors the actions of hardware. However, sending messages directly to processes (or receiving from processes) can cause problems when a library of generalised programmes are to be created. To avoid explicit naming of the target or the source of the message, messages are sent over channels (analogous to wires in hardware). Channels in the sender are linked to channels in the receiving process. This would be done as processes are created.

It would also be an advantage if our HDL was an extension of an existing software language. This would certainly reduce the learning curve and make it more natural to software people. This is not as simple as it may seem. Most languages are oriented towards single-threaded execution. Those languages which do support processes often do so in the most cumbersome of ways. A good example of this is the UNIX operating system's fork and join functions [Blair85].

However, two languages, namely CSP (Communicating Sequential Processes) [Hoare78] and Occam [Inmos84] provide suitable support for processes. Occam, a derivative of CSP, was designed primarily for multiprocessing applications.

CSP has a few problems that should be noted.

First, it passes messages to processes directly; no channels are used. The explicit naming of processes makes it difficult to construct libraries of general programmes.

Second is the vague definition of process termination and failure [Fay84].

Both Occam and CSP require processes to be synchronised when communicating with each other. This type of message protocol is, unfortunately, not always exhibited by hardware. Sometimes a device sends a message and then carries on with the next sequence of actions; it does not wait to ensure that any of the potential receivers has actually collected the message.

The last shortcoming CSP shares with Occam,† that is the lack of suitable data types. The basic data type required by the HDL is the bit. Bits need to be combined to form other types (for example: the type INTEGER). Bits need to take on other values besides zero and one: values such as unstable, tri-state, and unknown are necessary. Occam does not provide any support for this.

† It should be noted that Inmos has recently extended Occam (called Occam 2) giving it a more extensive set of data types [Poun86].

The language Circa, presented in the next chapter, is a derivative of Occam and as a result Circa shares much of Occam's philosophy. However, there are differences. The rigid format of Occam was liberalised, but Circa still uses the concept of indentation of subordinate processes to illustrate the process hierarchy and flow of control. A few new operators and types were introduced to better support hardware description. The communication mechanism of Occam was generalised - the handshaking protocol was augmented with an asynchronous capability.

Chapter V

THE LANGUAGE CIRCA

5.1 A TUTORIAL

This chapter presents an informal examination of the language Circa: a hardware description language developed by the author. The syntax is given in Appendix A.

The basic building block of Circa is the process. There are several primitive processes in Circa: input, output and assignment.

An input process looks like:

```
channel ? variable
```

A value is transferred from the channel into the variable. An input process will wait until a new value is placed onto the channel by some output process. It is quite acceptable to have more than one process waiting for input from the same channel. This one to many relationship works exactly the same way as a one to one message transfer. If we wished to disregard the value, we could rewrite the input process:

```
channel ? ()
```

To do the above, but not requiring the input process to wait for a new value, we would write:

```
channel ?? variable
```

Whatever happens to be in the channel at the time will be placed into the variable.

An output process is used to put a value onto a channel. It is denoted by:

```
channel ! expression
```

If more than one process attempts to place a value onto a channel at the same time, that channel will contain an undefined value. One can issue signals by:

```
channel ! ()
```

Signals do not involve any transfer of data, but are useful in process synchronisation.

The above form of the output command forces process synchronisation. The process will wait until there is a process ready to receive the data: that is, until a process requests input (in any form) from the channel. If the designer wishes to have the command terminate as soon as it has placed a value on a channel, he should use:

```
channel !! value
```

Assignment works exactly the same as in most other languages. The notation was designed to reflect this:

```
variable := expression
```

Expressions are constructed in a similar manner. The following is a list of operators:

OPERATOR	MEANING
/	integer division
*	multiplication
\	remainder
&&	logical and
<>	not equals (logical)
||	logical or
&	bitwise and
><	bitwise exclusive or
|	bitwise or
+	addition
-	subtraction
,	concatenation
>=	greater than or equals (logical)
<=	less than or equals (logical)
>	greater than (logical)
<	less than (logical)

Expressions are evaluated from left to right, except for expressions within parenthesis (these are done first). In Circa, all operators have the same precedence.

Values can be any number of bits in length. Concatenation combines two values, of n amd m bits in length, into a single value with n+m bits. The bitslice operation extracts bits from a value. Variables, channels and constants consist of a set of bits numbered 0 (least significant bit) to "n" (there would be "n"+1 bits). To select the lowest order bit, we would write: variable{0} .

Other primitive processes include: the wait and skip. Skip is a do nothing process and is defined simply by the keyword SKIP. The wait is used to cause a delay; WAIT n causes the process to be suspended for "n" time units.

The processes mentioned so far are useless unless they can be combined together. To assemble commands to form larger, more powerful processes, constructors are used. There are five constructors in Circa: SEQ, PAR, IF, ALT and WHILE.

The SEQ constructor causes its constituent processes or commands† to be executed sequentially. It is formed by:

```
SEQ
   command
   command
   ....
   command
```

To execute processes in parallel, Circa provides the PAR constructor. The form is similar to that of SEQ:

```
PAR
    command
    command
    ....
    command
```

The PAR command terminates when all of the subcommands have terminated.

† The word "command" is used as a synonym for "process".

The alternative (ALT) constructor waits until one of its guarded processes is ready. A guarded process is considered to be ready if its guard (Dijkstra's guards [Dijk75]) is ready (if the guard is a process) or has a non-zero value in the case of an expression. The ready process is then executed. When the process is finished, the ALT constructor terminates. If more than one guarded process is ready to execute, the first one to be encountered textually is chosen. Simple guards may be expressions, SKIP commands, WAITs or input commands. A guarded command has the form:

```
guard
   command
```

Guards can be cascaded together to form more complex and more useful structures. For example:

```
a = 0
   ch ? a
      chx ! 3
```

The value 3 will be placed onto channel "chx" if the variable "a" is equal to 0 and if there is a value sent over channel "ch" (the value is inserted into "a" overwriting the old value of "a" used in the comparison). This of course assumes that there are no other ready guarded processes before this one (textually) in an ALT.

Normally an ALT will wait until one of its guards becomes ready before it can execute a guarded command and then terminate. Sometimes we might want to exit the ALT if none of the guards are ready. We can do this by adding a SKIP as a guard.

The IF evaluates its expression guards sequentially. When an expression is found to yield a non-zero result, the guarded command is then executed. Once this has finished, the IF terminates. To place the maximum of two variables "a" and "b" into a variable "c", we would write:

```
IF
   a > b
      c := a
   a <= b
      c := b
```

We could omit the last test, since it must be true if a > b is false. We merely substitute a non-zero constant (a "1" in our case) for the true expression. This saves us from having to evaluate it during execution time. Rewriting the last example, we get:

```
IF
   a > b
      c := a
   1
      c := b
```

The WHILE constructor works exactly like while loops in traditional languages. The body of the WHILE loop is a single process. Since infinite loops are common in hardware descriptions, we can omit the expression to denote this. To output numbers from 1 to 10 to channel "x", the following would suffice:

```
SEQ
  count := 1
  WHILE count < 10
    SEQ
      x ! count
      count := count + 1
```

Up until now, we have not looked at declarations: at how variables and channels are defined. Variables and channels are declared by giving a name and a type name:

```
name :: type name
```

This type name is user defined. It refers to the actual type definition. In Circa, each type definition is stored in its own file. This enhances reusability of the variable/ channel definitions. To create a variable type definition for a byte, we would write:

```
Byte :: VAR { 8 }
```

The number 8 in this example indicates the number of bits in variable declared with the type "Byte". A channel is a little more complicated; a byte wide channel type might be given by:

```
ByteChan :: CHAN { 8 } FALL 2 RISE 3
```

The RISE and FALL keywords deserve a small mention here. Bits that make up a channel may change from 0 to 1 (rising edge) or from 1 to 0 (falling edge). The expression immediately following the FALL keyword indicates the number of time units that elapse before the 1 to 0 changes are seen on the output side of the channel. In this case, the delay will be at least 2 time units. If the bits that change go from 0 to 1, the delay will be 3 time units as indicated by the expression following the RISE. This allows the designer to simulate rise and fall times of logic devices as well as propagational delays through wiring. To create an instance of this kind of channel, we would write:

```
        Fred :: ByteChan
```

in the programme text.

Of course, if we wanted a byte wide channel with a delay of 4 time units† for a rising edge and a delay of 3 time units for a falling edge we could create another type definition. But this is very wasteful since most of the type definition is useful; only two numbers need to be changed. This brings us to variant types.

Variant types are types with parameters. So to redeclare our byte channel, we would write:

```
        ByteChan(f;r) :: CHAN {8} FALL f RISE r
```

Now to reallocate Fred: Fred :: ByteChan(3;4) or Fred :: ByteChan(2;3) depending upon the delays required.

† Note that Circa does not have any concept of absolute time: the units are arbitrary.

Circa provides symbolic references to constants. In Pascal, they are called constant identifiers. Circa's symbolic constants may refer to an array of constants, however, not just a single value. Each constant appears on a separate line after the constant header. Given

```
HowLong :: CONS
                3
```

we could allocate: WaitTime :: HowLong . WaitTime now refers to the value 3 and can be used anywhere the constant 3 can.

A declaration for a symbolic constant, variable or a channel is prefixed to a particular constructor. The scope of the identifier created in the declaration is that constructor. Scope is hierarchical and is formed around the process hierarchy. In this way, the designer can specify which processes are to have access to which values.

An important aspect of Circa is that of process abstraction. A process is created by giving a name and a programme which defines what the process will look like. Programs in Circa are merely an extension of the type concept used earlier for variables, etc.

A programme resides in its own file as do all type definitions. It is defined in a similar fashion as well:

```
programme name :: PROC (channel list)
                   process
```

The channel list consists of channel declarations separated by semicolons. These channels are external channels and allow intermodule communication. They are linked together with other channels during process creation. For example: to join the following processes together

```
        Pump :: PROC(x :: ChanY)
            some process here

        Tank :: PROC(w :: ChanY)
            some process here
```

we could use a superordinate process:

```
        hose :: ChanY:
        PAR
          gasPump(hose) :: Pump
          carTank(hose) :: Tank
```

A real world analogy of the previous set of processes is given in figure 4.

Array definition is the only major subject not yet covered. Circa has only single dimensional arrays. We can, however, have arrays of channels, variables, constants or even processes. An array is defined as follows:

```
        name [subrange] :: type
```

So to declare a variable array "M" with 5 elements:

```
        M[1..5] :: Number   or  M[0..4] :: Number
```

or whatever.

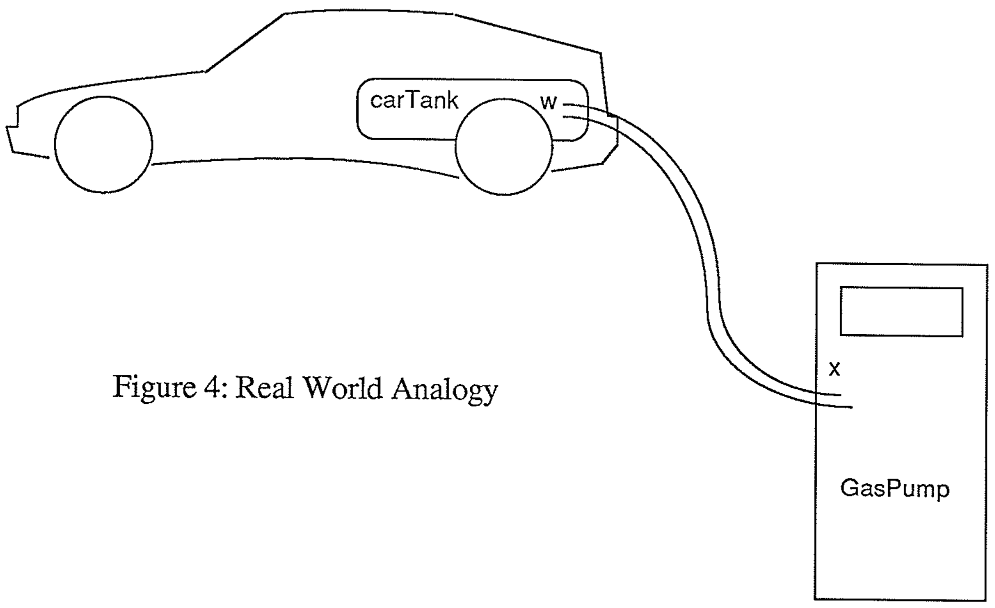


Figure 4: Real World Analogy

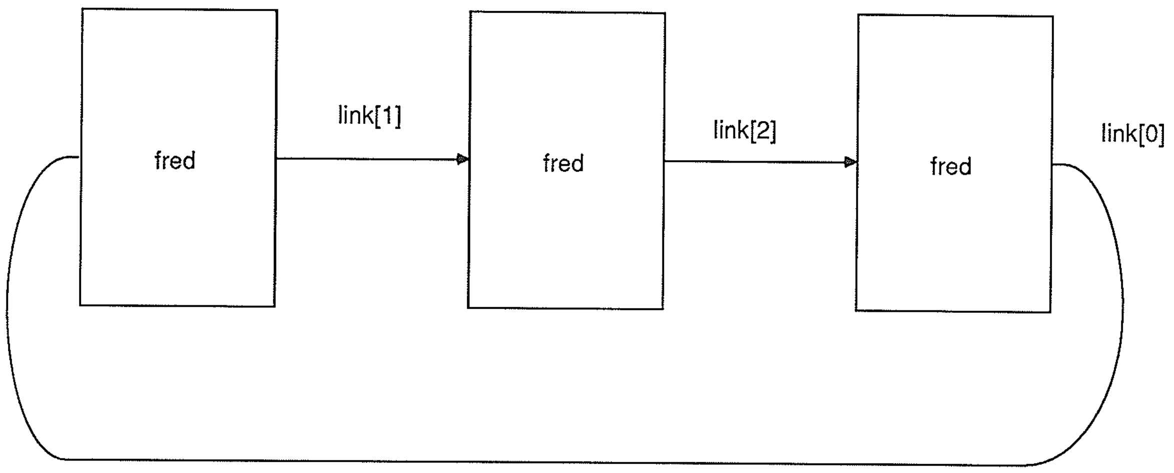


Figure 5: Ring of Processes

We can declare an array of processes in exactly the same way. Named processes or process abstractions are quite straightforward, but what about arrays of constructors which do not have any names? After the constructor name, a size (as above) can be given. So

```
PAR [1..3]
   A
   B
```

where “A” and “B” are two unspecified processes, is equivalent to

```
PAR
   A
   B
   A
   B
   A
   B
```

In the size specification, a symbolic constant can be given. This symbol is called the replication identifier. The value of this identifier coincides with the subscript of the particular element. For example:

```
link[0..2] :: SomeChannel:
PAR [j :: 0..2]
  fred(link[j]; link[(j+1)\2]) :: WhoKnowsWhat

WhoKnowsWhat :: PROC (in :: SomeChannel; out :: SomeChannel)
  some process here
```

is the same as:

```
link[0..2] :: SomeChannel:
PAR
  fred(link[0]; link[1]) :: WhoKnowsWhat
  fred(link[1]; link[2]) :: WhoKnowsWhat
  fred(link[2]; link[0]) :: WhoKnowsWhat
```

In this example, a ring of processes (see figure 5) is created. Each process communicates with its neighbours through two channels: one for input and the other for output.

Replicators make the specification of large arrays of similar processes trivial. As above, we can use the replicator identifier to vary the declaration somewhat.

It should be pointed out that Circa keywords, like Occam keywords, must be in uppercase. In the interests of a uniform programming style, it is recommended that type names begin capitalised and instance names do not.

5.2 AN EXAMPLE

To illustrate the language Circa we will consider a simple traffic light example. A similar example can be found in [Mead80].

There is an intersection (figure 6) we wish to control using four sets of traffic lights. The lights work in a straightforward way: first red, then green, followed by yellow and then back to red. Lights on opposite sides of the intersection go through this sequence at exactly the same time. The set of lights, say on the south side, are out of phase with the lights on the east side. The length of time the light is red, green or yellow is fixed. A more intelligent traffic controller might monitor traffic flow or adjust the lights depending on the time of day.

Let us sketch out the overall design in pseudo-Circa:

```
Traffic
  PAR
    lights facing east-west
    lights facing north-south

Lights
  WHILE
    SEQ
      green
      yellow
      red
```

We can describe the behaviour of the lights in more detail by using a simple timing diagram in figure 7.

Now to create the main process (Traffic). The controller must have two external channels: one to pass the value to the east and west light stands and the other for the north and south light stands.

```
Traffic :: PROC(eastwest :: LightVal; northsouth :: LightVal)
  yellowTime :: WarningTime:
  greenTime :: GoTime:
  PAR
    a(eastwest) :: Light(0)
    b(northsouth) :: Light(greenTime + yellowTime)
```

Processes "a" and "b" are both of the same type: "Light". We decided to make "Light" a variant programme. The variant is the amount of skew we want the light to have. "GoTime" and "WarningTime" are the times when the light is to be green and yellow respectively. These constants are referred to within the programme as "greenTime" and "yellowTime". They were made symbolic constants because at this point in the design their value is unknown and indeed irrelevant so

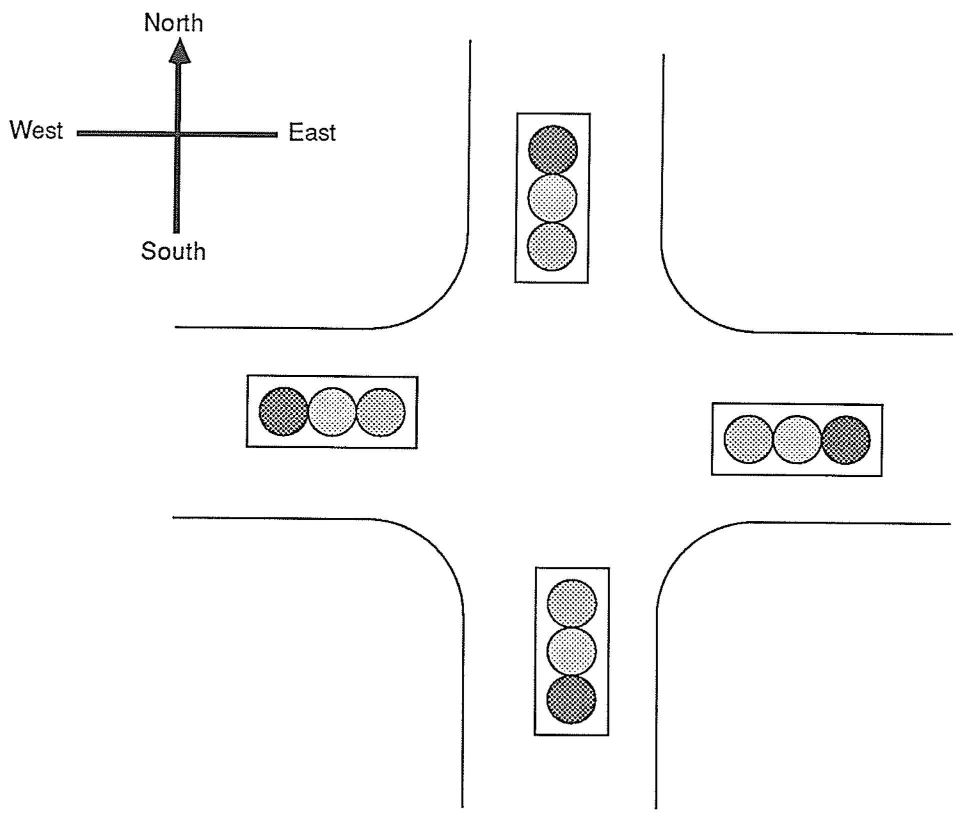


Figure 6: Street Intersection

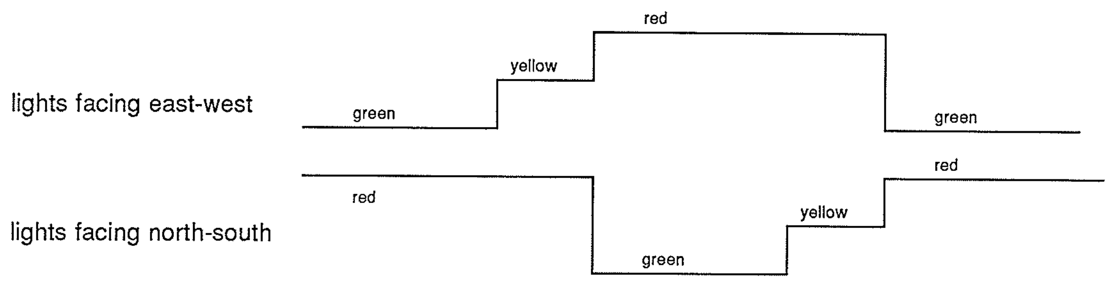


Figure 7: Light Timing

we will put off giving them values until the very last possible moment.

The “Light” process is essentially an infinite loop cycling through the light sequence.

```
Light(skew) :: PROC (mode :: LightVal)
  SEQ
    WAIT skew
    yellowTime :: WarningTime:
    greenTime :: GoTime:
    WHILE
      SEQ
        mode ! 0            -- green light
        WAIT greenTime
        mode ! 1            -- yellow light
        WAIT yellowTime
        mode ! 2            -- red light
        WAIT greenTime + yellowTime
```

Note that when the lights are first turned on (after maintenance or installation) the east-west lights will actually be on before the north-south lights. This anomaly may only last a minute and is not a critical issue in this example, but there may be systems where it is.

So far our channels have been undefined. We have given them a name: “LightVal”, but nothing more. Since there are three values that the lights can take on: green, red and yellow, we will need at least two bits to represent them. That means that the channels used to carry those values must be at least two bits wide.

```
LightVal :: CHAN { 2 } RISE 0 FALL  0
```

In Circa the designer must determine the channel width. An enumerated type concept, which would allow the compiler to make those decisions, does not exist in Circa.

To fill in the rest of the picture, we need to define the symbolic constants “GoTime” and “WarningTime”.

```
WarningTime :: CONS
   2

GoTime :: CONS
   16
```

These would be in separate files of course.

There is alternative to this design. Rather than having a process control the entire phase for a set of lights, we could have a process control only half a phase for both sets of lights.

```
Traffic :: PROC(eastwest :: LightVal; northsouth :: LightVal)
  WHILE
    SEQ
      first(northsouth; eastwest) :: Phase
      second(eastwest; northsouth) :: Phase
```

The first phase will have the north-south lights turn green then yellow while the east-west lights remain red. During the second phase the reverse will be true; this is reflected in the channel argument list of “first” and “second”. Let us have a look at “Phase”.

```
Phase :: PROC(goLight :: LightVal; stopLight :: LightVal)
  greenTime :: GoTime:
  yellowTime :: WarningTime:
  SEQ
    stopLight ! 2    -- red colour
    goLight ! 0      -- green colour
    WAIT greenTime
    goLight ! 1      -- yellow colour
    WAIT yellowTime
```

Now this does not look like anything would be gained by partitioning the problem in this way. Let us suppose that a more intelligent traffic light controller is desired. As traffic increases, the lights are to change more quickly. During the first phase, if more than "n" cars pass through the intersection travelling on the north-south route before the normal phase (defined in the previous example) has finished, the lights are to change immediately. The same applies to the east-west route during the second phase.

To accomplish this, we need some way of measuring traffic. Two sensors are introduced into the system: one for north-south traffic, then other for east-west. When a north (or south) bound car drives over the north-south sensor, a signal is sent along that line to our controller.

To simplify processing in the "Phase" module, we create a new module called "CheckCars". "CheckCars" receives the signals from the sensors. When the number of signals reach a certain limit, a "traffic is busy" signal is sent to "Phase". "Phase" must terminate its part of the cycle as soon as it receives one of these "traffic is busy" notifications. If it does not receive any signal from "CheckCars", then the phase terminates normally.

```
Phase :: PROC(goCar::Sensor;goLight::LightVal;stopLight::LightVal)
  yellowTime :: WarningTime:
  greenTime :: GoTime:
  SEQ
    stopLight ! 2       -- red colour
    goLight   ! 0       -- green colour
    isBusy :: Signal:
    stop :: Signal:
    PAR
      busy(goCar; isBusy; stop) :: CheckCars
      ALT
        WAIT greenTime
          stop ! ()     -- sends a signal to busy to terminate
        isBusy ? ()
          SKIP
    goLight ! 1         -- yellow colour
    WAIT yellowTime


CheckCars :: PROC(goCar :: Sensor; busy :: Signal; stop :: Signal)
  ok :: Bit:
  count :: Short:
  SEQ
    ok := 1
    count := 0
    WHILE ok && (count < 10)
      ALT
        stop ? ()
          ok := 0
        goCar ? ()
          count := count+1
    IF
      ok
        busy ! ()
```

The miscellaneous declarations are:

```
Sensor :: CHAN {1} RISE 0 FALL 0

Signal :: CHAN {1} RISE 0 FALL 0

Bit :: VAR {1}

Short :: VAR {16}
```

Naturally the sensors must go out to the external world, therefore they must be included in the mainline header.

```
Traffic :: PROC(nsc::Sensor;ewc::Sensor;nsl::LightVal;ewl::LightVal)
  WHILE
    SEQ
      first(nsc; nsl; ewl) :: Phase
      second(ewc; ewl; nsl) :: Phase
```

We have abbreviated the channel names for lack of room in the header.

To simulate this system, we would simulate the programme "Traffic"; all other definitions would be automatically read in from their respective files.

It is quite evident from these simple examples how Circa can implement a set of communicating processes. Also shown is Circa's ability to aid the designer in the top-down design process.

An Occam programme to implement this would be very similar to the above Circa programme. The Occam implementation is weak given the limited choice for variable and channel sizes. Occam channels are not capable of supporting channel delays. The latter is not significant in this case since the delays are 0, but for many problems delays are of great importance.

Chapter VI

AUTOMATED SYNTHESIS

6.1 SILICON COMPILERS

Currently, microprocessors with 250 000 [Inmos85] and 275 000 [ElAy85] transistors are available. It has been suggested that circuits will have on the order of 10 million transistors by 1990 and 250 million by the year 2000 [Solo86]. At first glance, one might think this freedom would make the system designer's job easier, but in fact, it is precisely the thing which makes VLSI design more difficult [Deny85]. The detail and number of design alternatives are overwhelming.

In this chapter, we outline a silicon compiler in the hopes of addressing this problem. The term "silicon compiler" was first coined by [Johan79] and there is some debate concerning what actually constitutes a silicon compiler [Gajski85]. The silicon compiler suggested takes a programme written in Circa and translates it into a low level hardware description (perhaps chip masks).

The key element of this circuit compiler is the notion of modularisation. Modules appear like black boxes from the outside; internal details are hidden. Higher level modules are assembled together from lower level ones using a set of rules derived from the definition of the Circa language. The language primitives, which are at the lowest level, map into pre-defined pieces of hardware. This is similar to software compilation where primitives such as operators or subroutine calls are implemented by pre-defined sets of machine instructions.

The abstraction mechanism of the silicon compiler helps the designer manage detail. The designer is not concerned with implementation details at lower levels, only with their external behaviour.

By using a high level language, chip design becomes available to a larger group of potential designers. It is no longer necessary to have an intimate knowledge of device physics to build circuits.

Assuming a valid design, the compiler will generate the masks † for a working chip. Since the primitive modules and the assembly mechanism have been verified, the chip will be free of implementation errors.

The hardware description language is technology independent. Should it be necessary to convert to a new device technology, only the mask generation portion of the compiler would have to be rewritten. This situation is analogous to

† Masks are typically described in a simple layout language such as CIF (Caltech Intermediate Form) [Mead80].

a portable software language that is transferred to a new machine.

Finally, a uniform software based design methodology offers a vehicle for organising designs and exchanging designs between interested groups.

6.2 THE CIRCA MODEL

A standard computer organisation is the von Neumann machine. It is typified by a single computational unit and a small set of registers located in the processor. The processor communicates with a memory through a bus. The memory bus is usually only a word in width.

The primary complaint with this style of computer organisation is the serial nature of the system. Computation takes place in one part of the computer, but the data, upon which the operations are to be performed, reside in another part. The only link between the two is the aforementioned bus. This link tends to serialise processing as only one word of data (or instruction) can pass during a bus cycle. This situation gives rise to the term “von Neumann bottleneck” [Back78].

Computer designers have suggested building computers with arrays or trees of processors each with its own local store [Mead80]. By splitting up a programme among a number of processors greater speeds should be obtained. This is true if the algorithm can be partitioned so as to minimise inter-process communication and to maximise concurrency.

Now this is not to say that all computer systems should be designed this way. The von Neumann architecture will be around for a while; we are very adept at programming them and hierarchically organised machines are still as yet unproven.

An interesting parallel exists between the organisation of a computer system and the internals of the processor itself. A microprogrammed processor can be thought of as a miniature computer system. There is the "von Neumann bottleneck" (the data paths) connecting the processing unit (the ALU) to the memory (the register set). Perhaps we can use the new approach to systems design in processor design.

This is the approach that we have taken with the Circa silicon compiler. The low level primitives of the Circa language are implemented by small, simple processors. Circa constructors are used to assemble these processors.

The Circa language provides a time framework in which processes are executed. What we mean by this is that there is a definite ordering of events, but no fixed times at which these events must occur.

We have taken this liberal approach in the timing between the elementary or primitive processors. Each processor is a sequential machine and therefore has a clock. However, the clock is hidden inside the primitive processor. The processors communicate between each other in an asynchronous manner using a set of control lines. Since interprocess communication is independent of a clock, the clocks inside the processors need not be the same. A two-phase non-overlapping processor clock can be generated by the circuit in figure 8 (from [Mead80]).

A set of signals, described in figure 9, is used to control the actions of the processor. RESET is used, usually during power up, to reset the processor back to the known initial state. The initial state of the processor is an idle state where nothing happens. To start the processor up so that it can perform its function, a pulse is sent along the ENABLE control line. When the processor accomplishes its task, it sends a pulse over the DONE line to another processor which is waiting. The Mealy state graph in figure 10 illustrates the control sequence of a processor.

The processors implementing Circa primitives are assembled together to form larger, higher level processors as specified by the Circa constructors. These high level processors look exactly like their primitive counterparts from an external perspective. High level processors do not really exist in silicon; it is only a concept to make the transition from Circa to masks easier.

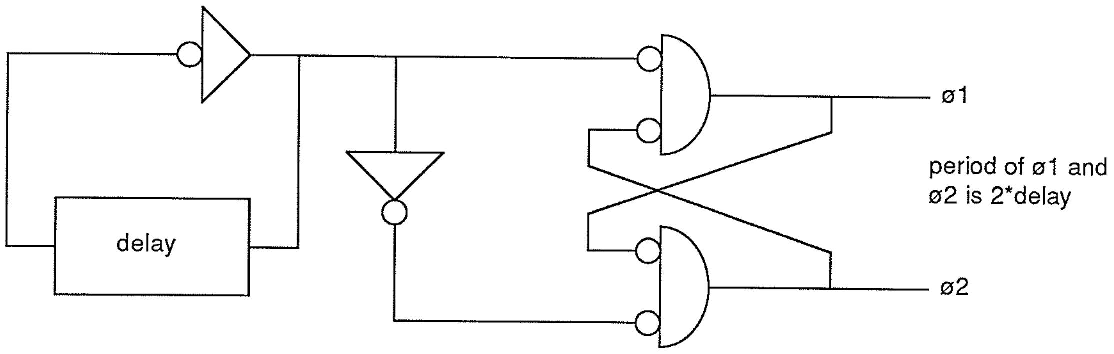


Figure 8: Clock Generation

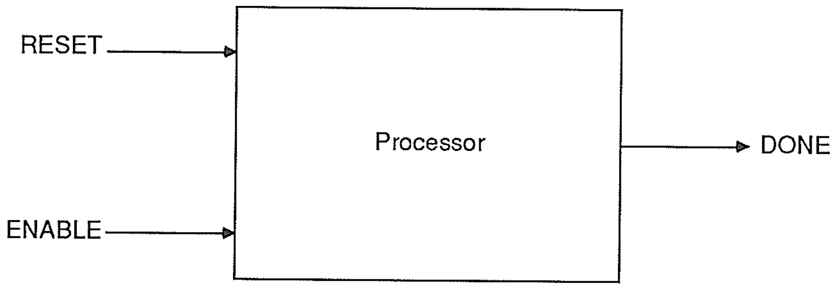


Figure 9: Processor Control Signals

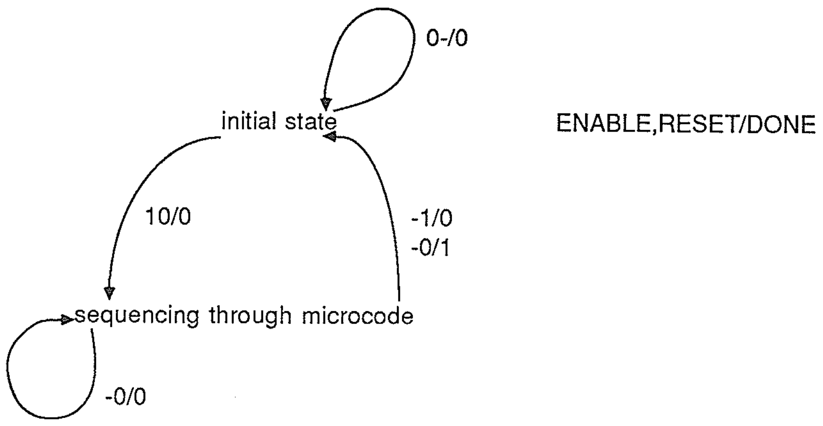


Figure 10: Processor Control Sequence

6.3 PRIMITIVES

6.3.1 Primitive Processors

Input, output, assignment and wait are primitive processes in the Circa language. They are implemented as primitive processors. Primitive processors are microcoded finite state machines.

Expressions which appear in the WHILE constructor and as guards in the IF and ALT constructors are also implemented as microcoded processors. Expression processors deviate from the model in only one aspect: there are two separate DONE signals leaving the module. One is used to signal a processor if the expression is evaluated to be a one; the other indicates a zero result.

Note that expressions in construct replicators and variant parameters can only be constant expressions. These expressions can be evaluated at compile time and therefore are not implemented as primitive processors.

Although SKIP is a primitive process in Circa, it is not implemented as a sequential machine simply because it does not do anything. It is a dummy processor with the ENABLE control line passing though to the DONE control line without any modification.

6.3.2 Arrays

When faced with an array in the Circa language, the silicon compiler has two choices: to implement the Circa array as a real memory array (large ROM or RAM) or to split up the array into its individual elements.

In this section, we will deal with the criteria that Circa uses for making this implementation decision.

For channels, there is no decision to make. Channel subscripts must be evaluated at compile time; they must be constant expressions. Consequently, channels are always implemented as individual elements. It complicates the hardware considerably if channels are chosen dynamically from a memory array.

With variables and constants there is some flexibility. Let us consider the following example:

```
a[0..2]::Byte:
SEQ [i::0..2]
  a[i] := i
```

In this case, the variable "a" is subscripted using only constant expressions. "a" could be implemented as a three byte RAM with a bus that is shared between the three assignment processes or as three individual one byte RAMs (with three individual buses: one for each assignment processor). There does not seem to be any obvious advantage to one approach over the other.

Suppose we were to perform the operations in parallel.

```
a[0..2]::Byte:
PAR [i::0..2]
  a[i] := i
```

If "a" were to be implemented as a three byte RAM, we would have a problem. Although we are modifying a different element in each assignment command, there is only a single bus. The data bus certainly cannot be shared: nor can the address bus (an address bus is required when a memory array is accessed in order to select the appropriate element). There are two alternatives here. A bus arbitration scheme could be used. But this would serialise access and would therefore make the PAR behave like a SEQ constructor in this case. The other possibility is to implement the array as a series of individual elements and to allow only constant expressions as subscripts. The latter approach was adopted. Note that this only applies to parallel access; for sequential access, none of these problems appear.

Constant arrays also follow this scheme.

6.3.3 Memory Access

We will examine variable access first.

A variable is accessed by a processor over an asynchronous bus. The bus control signals (figure 11) are patterned after the asynchronous bus interface of the 68000 processor [Moto82].

The flow charts in figure 12 and timing diagram (in figure 13) illustrate the bus protocol. Each stage of the read or write cycle corresponds to one micro cycle in the state machine.

Constants are usually constructed out of ROMs and have the same interface logic as variables. Depending on the choice of microcode instruction format, it may be possible to encode individual constant values in the instruction itself (immediate values).

The behaviour of channels differs radically from that of constants or variables. Channels can cause a processor to wait indefinitely; constants and variables have a known upper bound. The upper bound is a function of the memory access time.

A channel has its own memory and is dual-ported. There is an input and output port. Each port has its own set of asynchronous bus logic (figure 14).

The idea behind the channel is quite straightforward. Processors communicate between each other using channels. Processor synchronisation is accomplished through the bus control signals.

For example, if we write a value to an active channel (using a "!"), the bus cycle will complete as soon as a processor on the input side of the channel is ready to read a

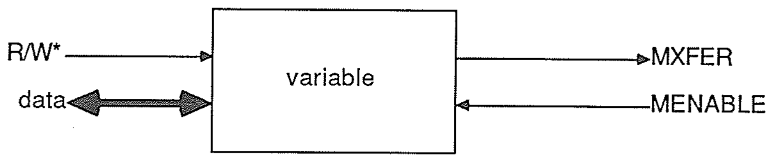


Figure 11: Bus Interface

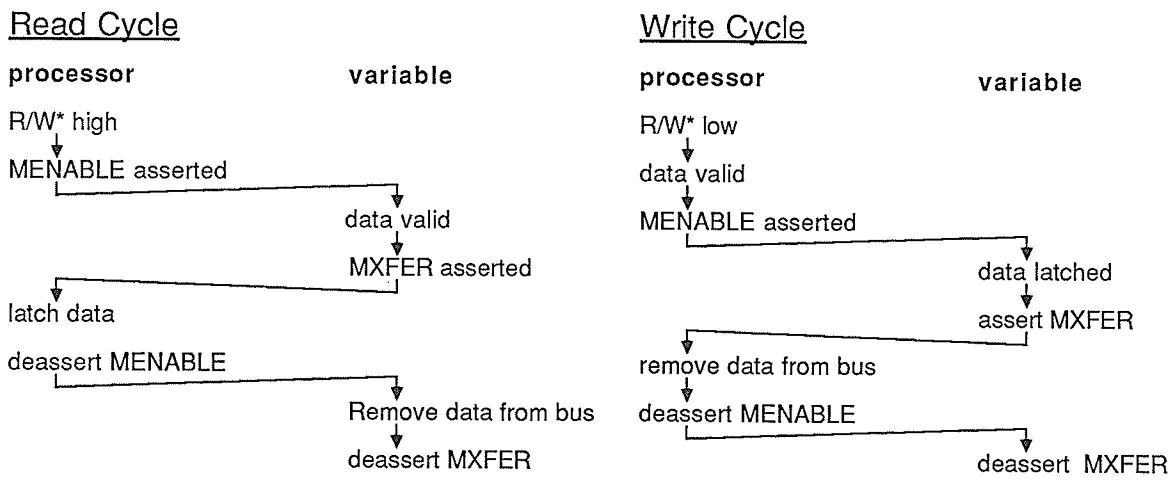


Figure 12: Read and Write Cycles

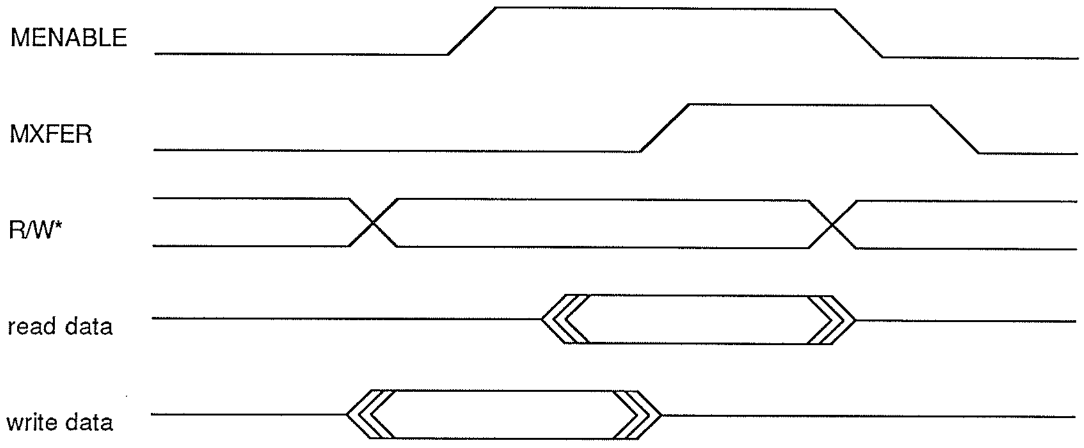


Figure 13: Bus Timing

value. The bus cycle could complete immediately if a processor is already waiting.

If a value is sent to a passive channel (we issued a "!!"), then the bus cycle completes immediately, regardless of the status of the input side of the channel. Any processors waiting on the input side are woken up. This is done by the completion of the bus cycle (figure 15).

With each channel bit is a "new bit" which is set when data is sent to the channel and reset when data is read out. This is used by the input side of the channel. When a processor issues an input command from an active channel (a "?"), the new bit is examined. If the new bit is set, then the bus cycle completes. Suppose the new bit is clear, the processor will wait. As soon as the new bit is set (by data being placed into the channel), the bus cycle on the input side resumes.

If data is requested from a passive channel, the new bit is ignored and the bus cycle completes without waiting.

What happens when more than one processor issues an input command from the same channel at the same time? The MENABLE signals from each processor on the input side of the channel are ORed together. This means that the bus cycle on the input side of the channel will finish when the slowest processor is finished.

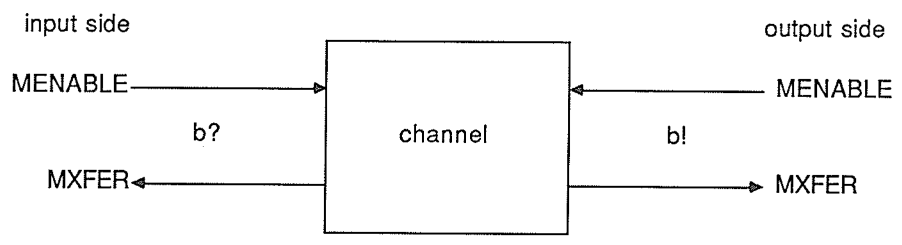


Figure 14: Channel Interface

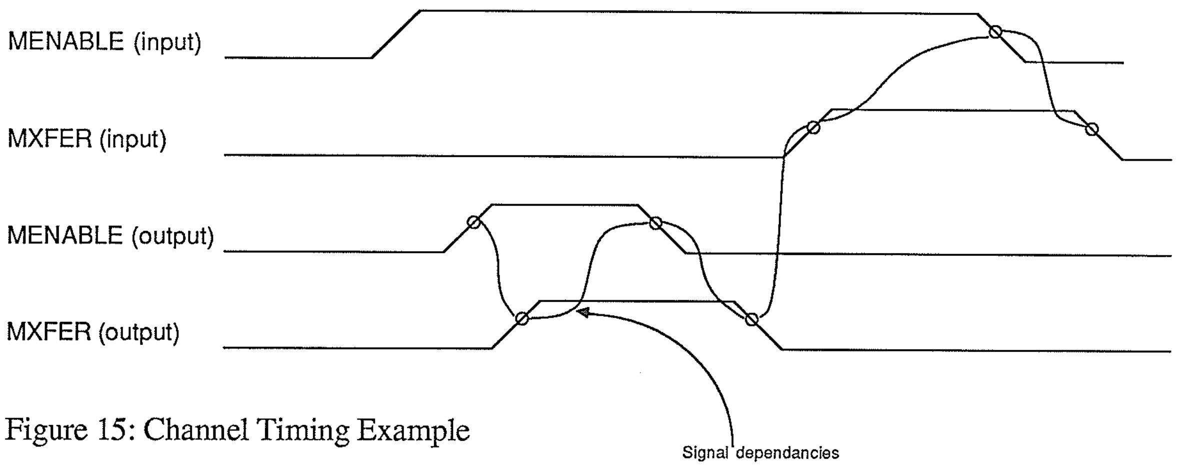


Figure 15: Channel Timing Example

6.4 HIGH LEVEL PROCESSORS

Processors can be formed into a hierarchical structure of higher level processors. They are formed in such a way as to implement the appropriate Circa constructor.

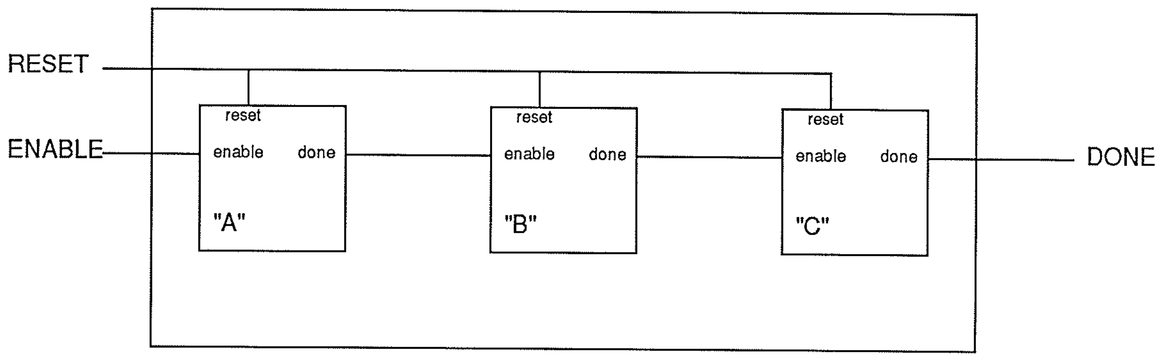
The following sections outline the Circa constructors and their corresponding implementations. Note that the graphical representations of Circa constructors do not necessarily imply suitable layout.

6.4.1 Sequential Constructor

Subprocesses of the SEQ are formed into a pipeline-like structure (figure 16). It is not a true pipeline, however, as only one stage can be active at any one time.

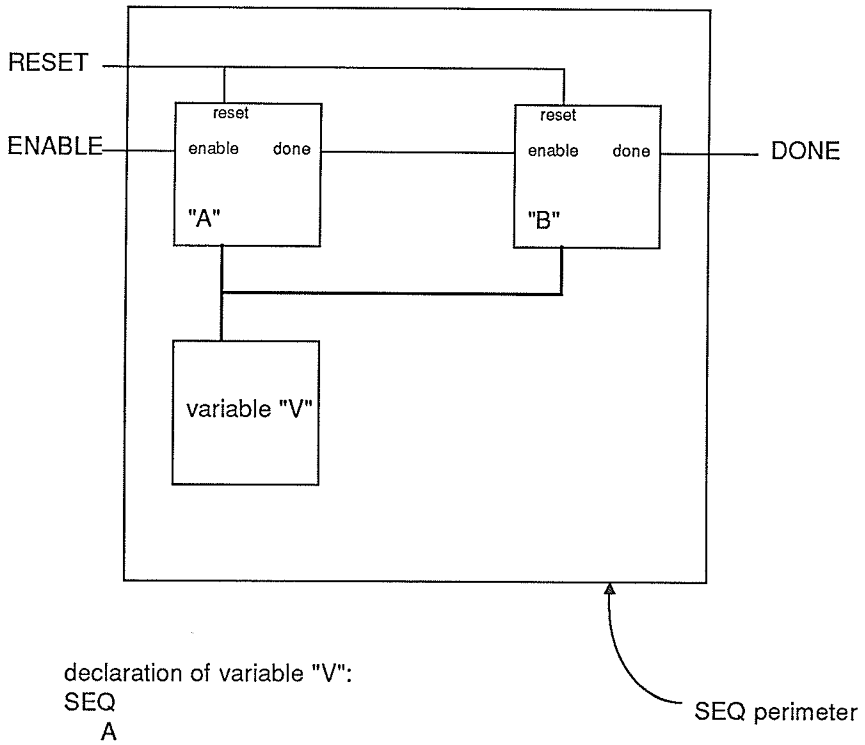
If a variable were to be declared so that its scope encompassed only this SEQ constructor, the variable could be located within the SEQ perimeter (figure 17).

As an optimisation feature, a SEQ containing only sequential constructs (IFs, WHILEs, other SEQs or primitive processes) could be compacted into a single processor. No performance degradation would take place and chip space would be saved.



```
SEQ
  A
  B
  C
```

Figure 16: Hardware Implementation of the Sequential Constructor



```
declaration of variable "V":
SEQ
  A
  B
```

Figure 17: A Variable in the SEQ Machine

6.4.2 While Constructor

Figure 18 describes the implementation of the WHILE.

It may be possible to include this construct when compacting sequential constructs as the previous section suggests. However, our microcode must be able to handle conditional execution.

If the expression is omitted in Circa (an infinite loop), the result would be the structure in figure 19.

6.4.3 If Constructor

An IF construct evaluates its guards sequentially. Concurrent evaluation of guards is only done by the ALT construct. The structure is given in figure 20. The IF guard must be an expression.

6.4.4 Parallel Constructor

The PAR definition states that a PAR terminates when the slowest of its components terminates. We use latches to hold the DONE signals of its components. When all of the latches are set, the PAR is finished and a signal is therefore sent. The latches are then reset. See figure 21 for details.

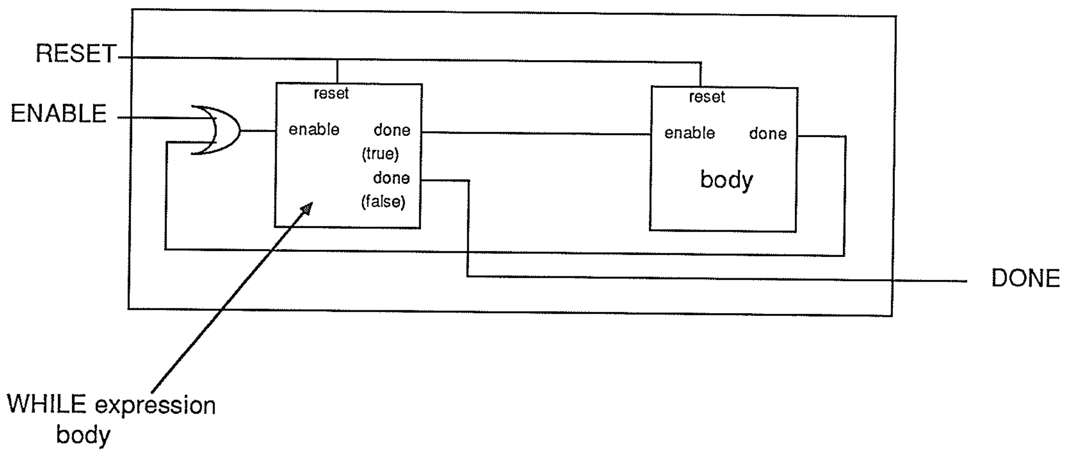


Figure 18: Hardware Implementation of the WHILE Constructor

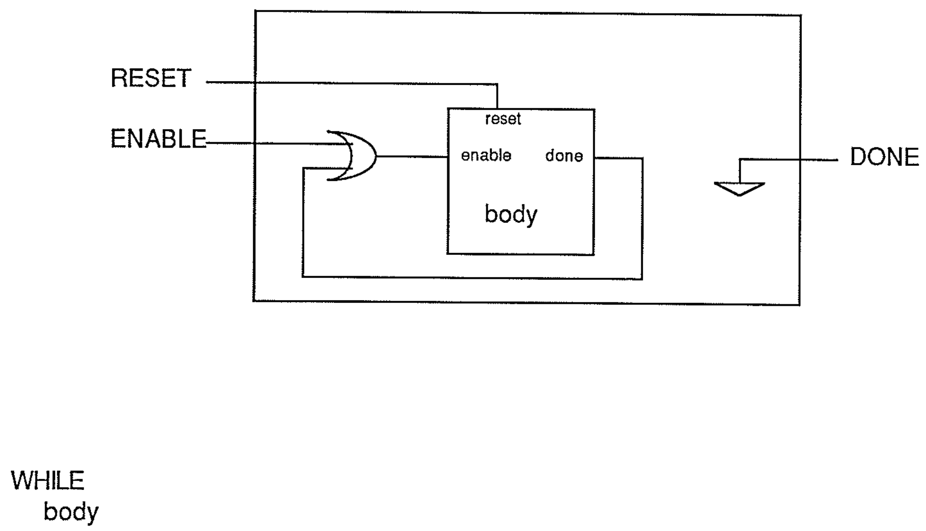
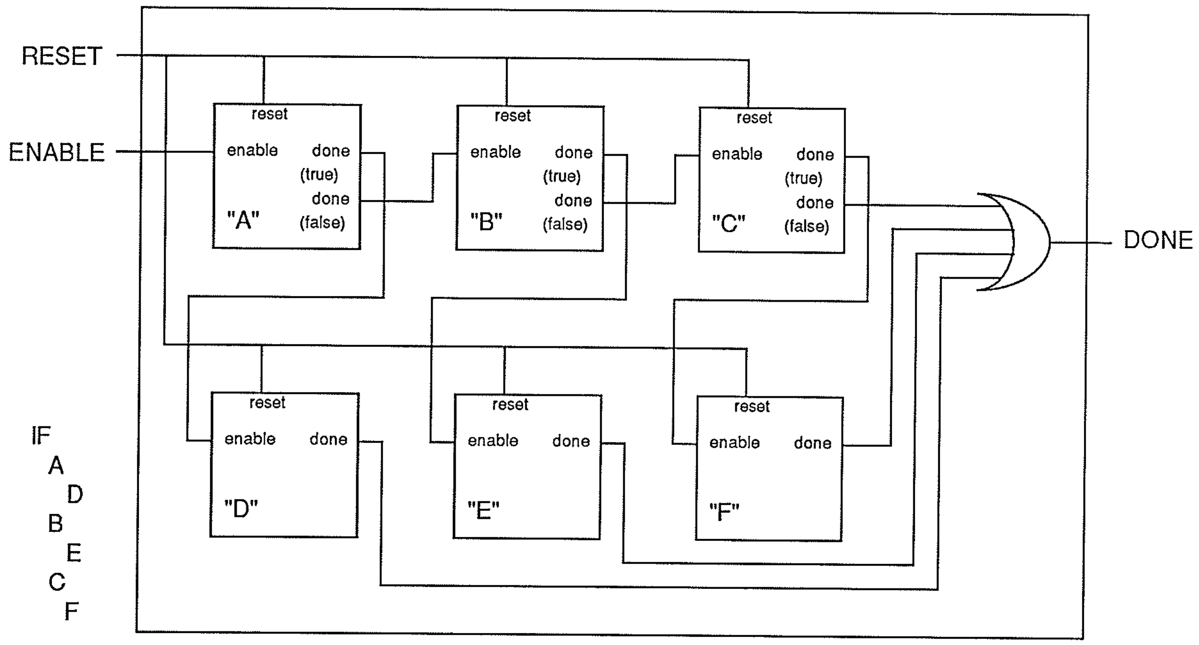
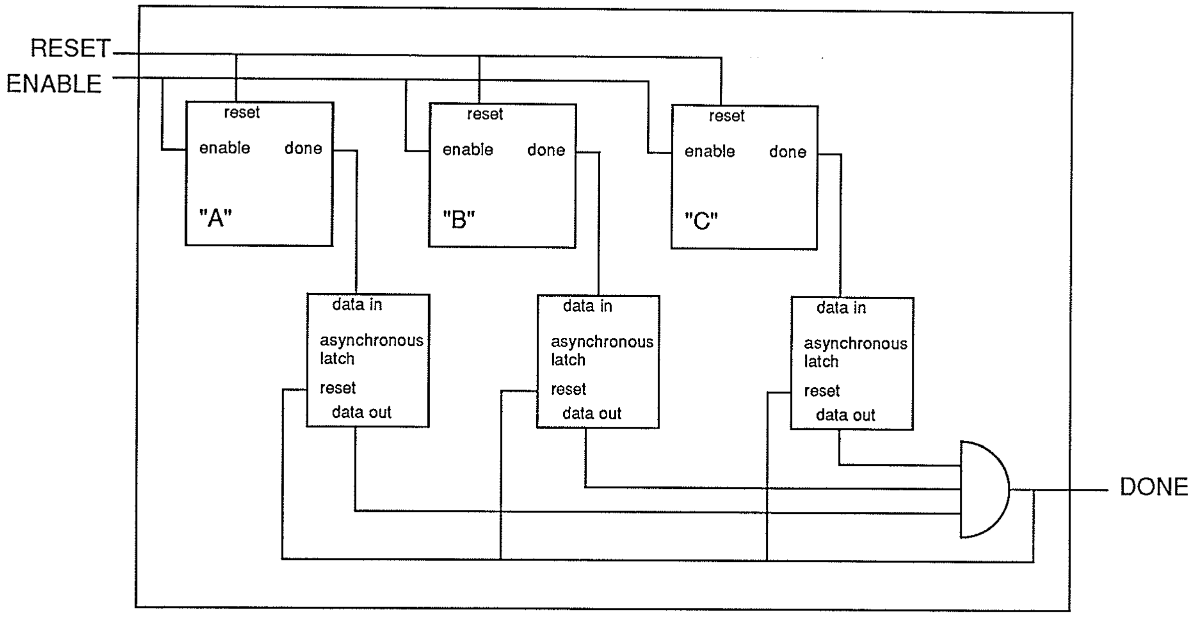


Figure 19: An Implementation of an Infinite Loop



```
IF
  A
    D
  B
    E
  C
    F
```

Figure 20: Hardware Implementation of the IF Constructor



```
PAR
  A
  B
  C
```

Figure 21: Hardware Implementation of the Parallel Constructor

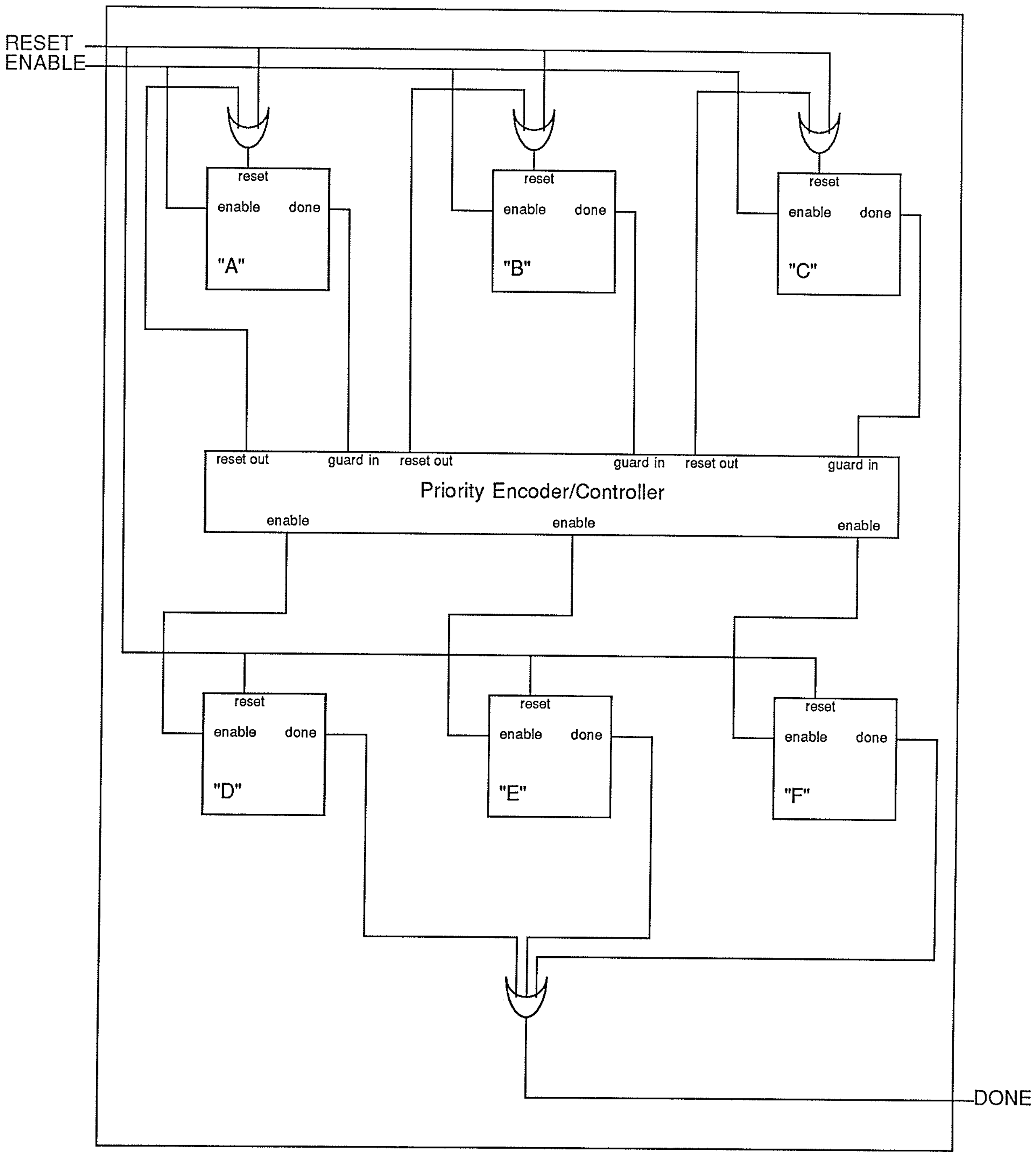
6.4.5 <u>Alternative Constructor</u>

The ALT guards are evaluated concurrently. The first guard to finish gets to execute its guarded command. The priority encoder (see figure 22) allows only the first signal to get through. The guards have a priority: “A” is the highest and “C” the lowest. This is determined by their placement in the ALT text. The priority encoder does more than just enabling a particular processor: it also resets the guards.

It is important to realize that the speed of the guards plays an important part in determining which guarded process is executed. The difference between the time when the conditions for a guard to become ready (such as receiving an input signal) and the time when the guard actually responds with a pulse on the DONE wire is called the reaction time. Therefore, guards with small reaction times will have their guarded processes executed in the case of races with slower guards. The SKIP guard has the fastest reaction time.

6.5 <u>AN EXAMPLE</u>

In chapter five, we discussed an intelligent traffic light controller. In this section, we shall apply the ideas in silicon compilation to the implementation of such a controller.



```
ALT
  A
    D
  B
    E
  C
    F
```

Figure 22: Hardware Implementation of the Alternative Constructor

The controller mainline was given by the following:

```
Traffic :: PROC(nsc::Sensor;ewc::Sensor;nsl::LightVal;ewl::LightVal)
  WHILE
    SEQ
      first(nsc; nsl; ewl) :: Phase
      second(ewc; ewl; nsl) :: Phase
```

An implementation of this process is given in figure 23.

Note that the module "Phase" is not defined yet (in figure 23), so it appears as a black box. This is how the compiler works: a top-down decomposition of Circa processes.

The input side of the "nsc" and "ewc" channels and the output side of the "nsl" and "ewl" channels are connected to the chip pads.

Given the definition of "Phase" and its subordinate module "CheckCars":

```
Phase :: PROC(goCar::Sensor;goLight::LightVal;stopLight::LightVal)
  yellowTime :: WarningTime:
  greenTime :: GoTime:
  SEQ
    stopLight ! 2       -- red colour
    goLight   ! 0       -- green colour
    isBusy :: Signal:
    stop :: Signal:
    PAR
      busy(goCar; isBusy; stop) :: CheckCars
      ALT
        WAIT greenTime
          stop ! ()     -- sends a signal to busy to terminate
        isBusy ? ()
          SKIP
    goLight ! 1         -- yellow colour
    WAIT yellowTime
```

```
CheckCars  ::  PROC(goCar :: Sensor; busy :: Signal; stop :: Signal)
  ok :: Bit:
  count :: Short:
  SEQ
    ok := 1
    count := 0
    WHILE ok && (count < 10)
      ALT
        stop ? ()
          ok := 0
        goCar ? ()
          count := count+1
    IF
      ok
        busy ! ()
```

Figure 24 shows the result of compiling the module “Phase”. The structure of “CheckCars” is illustrated in figure 25.

6.6 CONCLUSION

This is only one possible model for silicon compilation using Circa. Similar ideas regarding integrated circuit design have been proposed by [Hayes83].

One objection to this approach is its inefficient use of chip area. This arises from the duplication of processing units and the potentially large number of buses. However, this objection is becoming less important as circuit densities increase.

There are some interesting advantages to this particular model.

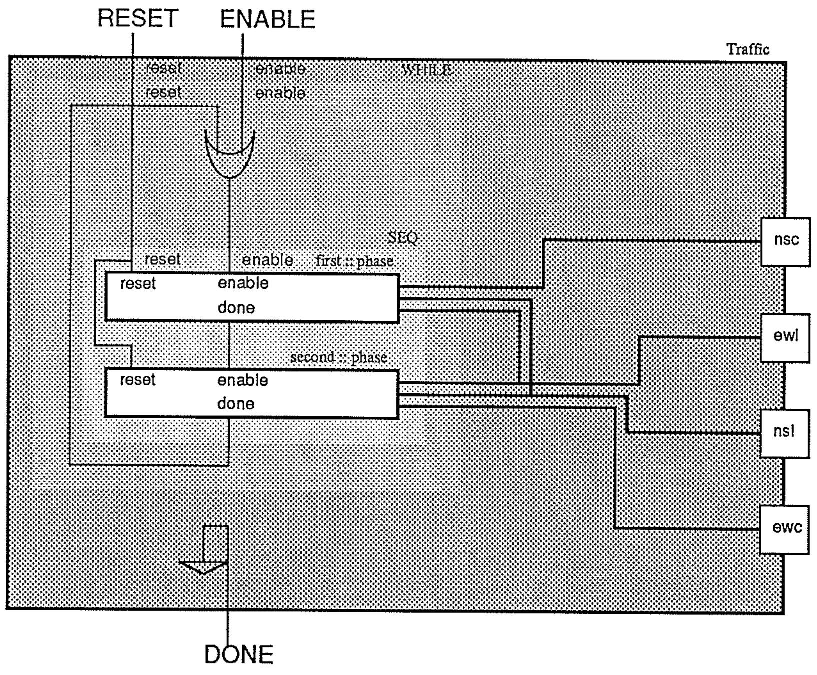


Figure 23: Hardware Implementation of "Traffic"

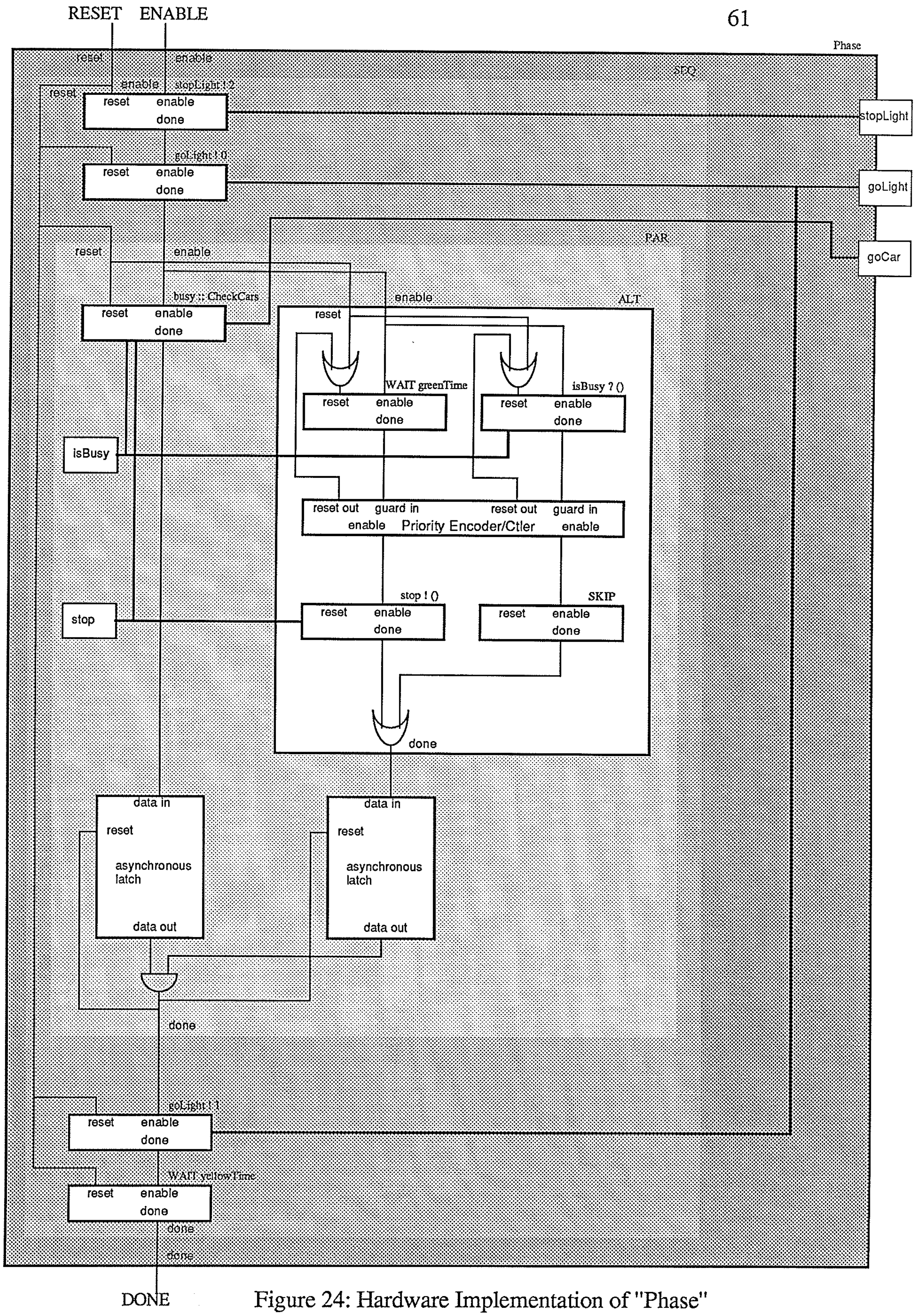


Figure 24: Hardware Implementation of "Phase"

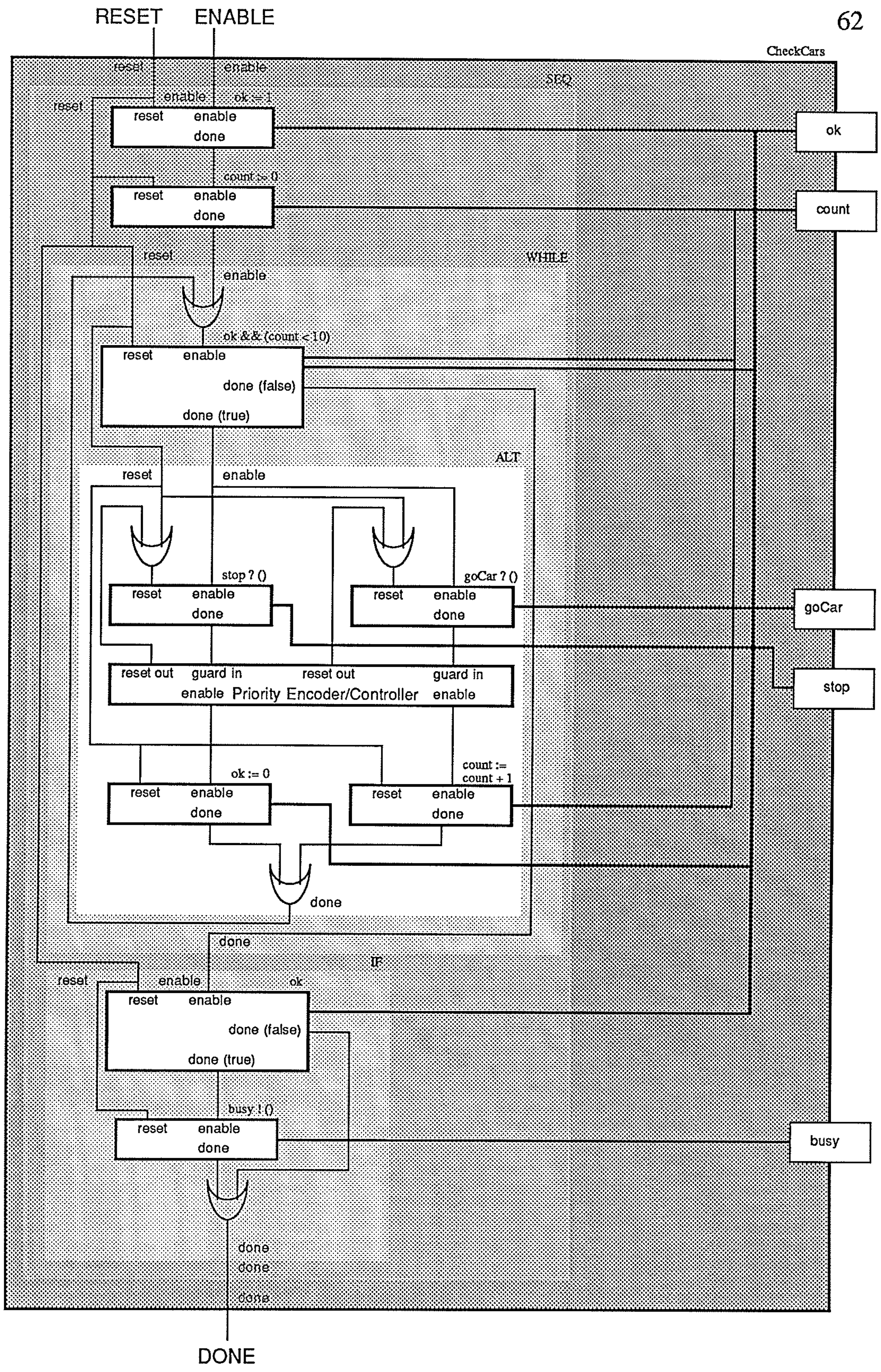


Figure 25: Hardware Implementation of "CheckCars"

The problem of clock distribution within the chip virtually disappears since the organisation is based on asynchronous processing units. This also means that global processing (processing between processors) takes place at the fastest possible speed.

Due to the localised processing and memory, still greater speeds can be obtained. This is attributed to the tendency towards small buses. Traditional organisations usually have large buses which slow up processing. Large buses have large capacitances which cause signals to travel slower [Mead80].

Chapter VII

THE IMPLEMENTATION OF THE CIRCA INTERPRETER

7.1 INTRODUCTION

The Circa interpreter is an event-driven simulator. This means that the processing of data occurs only when it has to; that is, when messages are sent from one process to another or when a process proceeds to its next state. The word simulator is used as a synonym for interpreter in our discussions.

The simulator is written in C and runs under the UNIX operating system. The simulator is called from the shell and is ready to execute a Circa process typed in from the terminal. When the process terminates, the Circa interpreter returns control back to the shell. The process would usually be a call to the top level process. To aid the Circa programmer, a source language trace facility is provided.

The discussion of the workings of the simulator is broken down into four areas: analysis of the source text, process management, expression evaluation and channel implementation.

7.2 ANALYSIS OF THE SOURCE

When a type is encountered (a declaration or an abstraction), the type definition is read in from a file and interpreted. The result depends upon the type. If it was a variable or a channel, attributes will be the result. If the type was a constant, then a list of constants (or perhaps only one) will be the result. Finally, if the type was a procedure, it will be executed.

Let us examine the mechanism that gathers the source code. First of all, the source code that has already been gathered into the Circa system resides in a list: each element being one type definition. Each element consists of a tree of source lines (see figure 26). The depth of a line in the tree is determined by its indentation in the original source. For each line, there is a list of tokens each representing some elementary syntactic unit (for example an identifier) from the source.

The programme which recognises tokens from the source file was generated by UNIX's LEX [Lesk75]. LEX takes a file consisting of regular expressions and produces a programme which recognises them. It is called by a higher level programme which inserts the tokens into the appropriate place in the source tree.

The source is not compiled into any type of low level form. There would be little advantage in doing this. We would be replacing the tokens with another form which would require as much processing to interpret it as did the original. Furthermore, the source code is kept around to be printed during traces or when an error occurs.

Each line also contains the column position and the line number in the file where the code starts so that when the piece of code is printed, the user knows exactly the place where the line came from.

Once a type definition has been "compiled" into the source tree form, subsequent accesses to that type need only consult the source list.

7.3 PROCESS MANAGEMENT

To create a process, a process control structure is formed. Its function is to keep track of the state of the process. The state includes information such as the major and minor states, the stack and the source pointer.

The major and minor state indicate to the Circa interpreter exactly what the process is doing. Possible major states are: process is active, inactive, waiting for a time event or waiting for some input/output event to occur. Minor states further define the actions of the process. For example, if a process is active (a major state), it may be any of three minor states: initial, executing and finished. During the "initial" minor state, the process may be inter-

Some example text from file "fred":

```
...
PAR
   SEQ [i :: 1..3]
      b[i] := i
      c[i] ! i+1
  IF
      num = 0
         flag ! 0
      num > 0
         flag ! 1
      num < 0
         flag ! 0-1
```

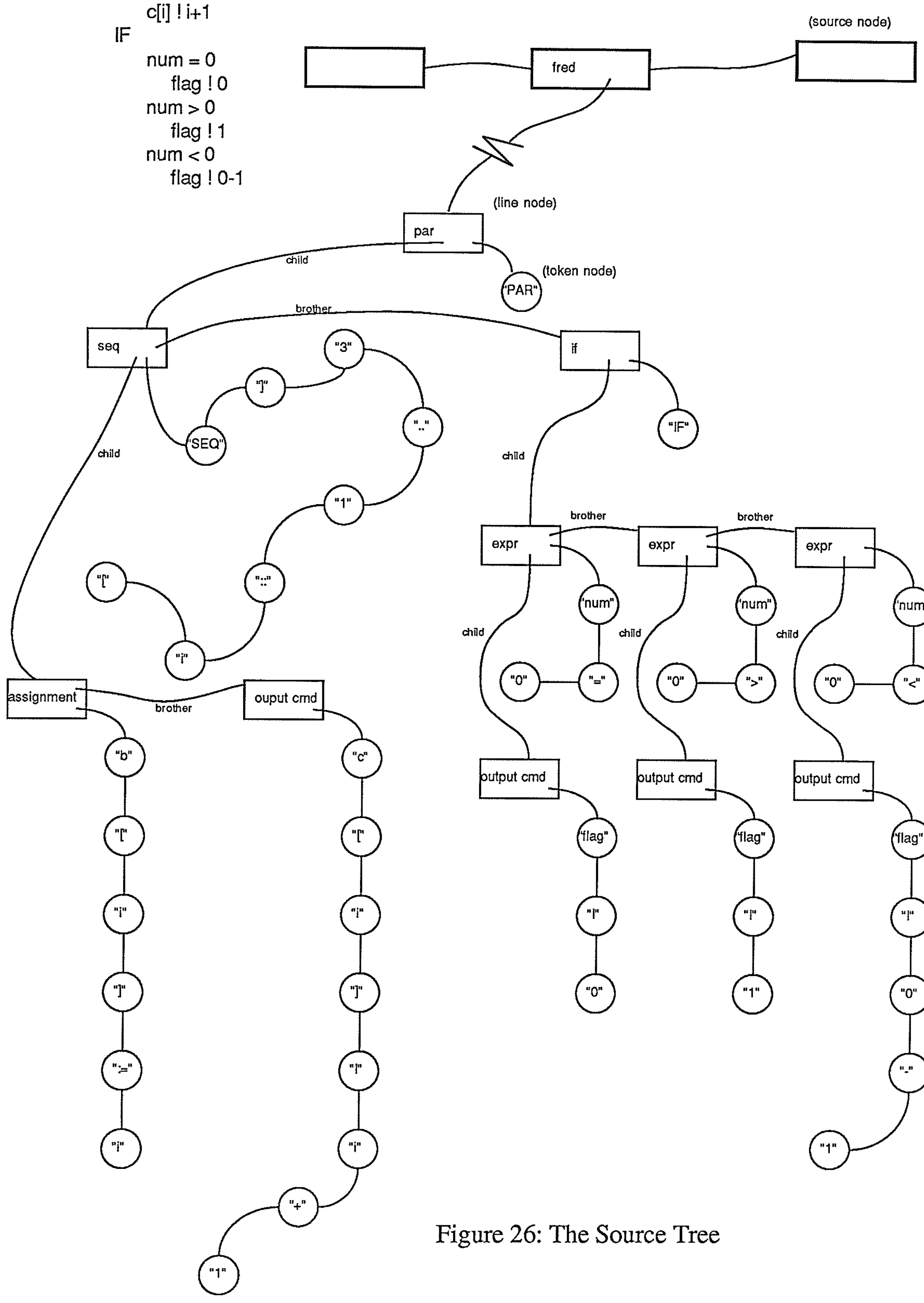


Figure 26: The Source Tree

preting the replicator if it is a constructor. While in the "executing" minor state, a SEQ constructor fetches the next substatement to be executed. When a process is ready to return control to the father process, it is finished and enters the "finished" state.

While in many of the above major states, the process is placed into a queue. The only exception to this is the inactive state; the process does not belong to any queue. Processes are picked from the active queue for execution. When the active queue has been exhausted, the current time is set to the time of the process at the beginning of the time queue. Processes on the time queue which have their time field equal to that of the global time counter will be activated (placed on the time queue). The time queue is in ascending order by time so that only the first part of the queue needs to be examined. Processes are placed onto the time queue if they are waiting for some time event to occur (after issuing a WAIT for example). A process waiting for an I/O event to occur is placed onto the input or output queue of a channel.

The stack is usually reserved for primitive processes but can be used by any process that requires the evaluation of expressions. Normally, processes point to tokens at the beginning of a line (also the beginning of a Circa statement) and when finished have an empty stack. Expressions can deviate from this. If some process, say an assignment, wishes to evaluate an expression, it creates a separate process to do this. The source pointer points to the token in the line currently being executed by the father process (the assignment). The expression is evaluated and the result is placed on the father's stack. The father's source pointer is also updated to point to the token following the expression just evaluated. The expression process no longer exists and the assignment process continues on.

7.4 EVALUATION OF EXPRESSIONS

The evaluation of expressions is quite straightforward. This is due primarily to the uniformity of precedence between operators. The only part that requires explanation is the identifier search. When an identifier is encountered in the expression, it is searched for in the environment list of the current process. If it is not found, the search continues in the father process' environment list. The search proceeds, moving up the process hierarchy until the identifier is found or a process header is reached.

The expression evaluation module executes the expression in the same manner as if it were parsed by a recursive descent parser (see Appendix A for parse rules). A nested expression, for example, would be evaluated in a subprocess of the orginal expression process.

7.5 CHANNEL IMPLEMENTATION AND ENVIRONMENT LISTS

Channels, constants and variables are organised as an array of pointers, each of which points to an individual bit structure. Channel and variable bit structures not only contain the current value for the bit, but also the process that last modified it and the time when the change occured. This allows us to introduce "undefined" values. Values become undefined when two or more processes attempt to modify a variable, for example, at the same time.

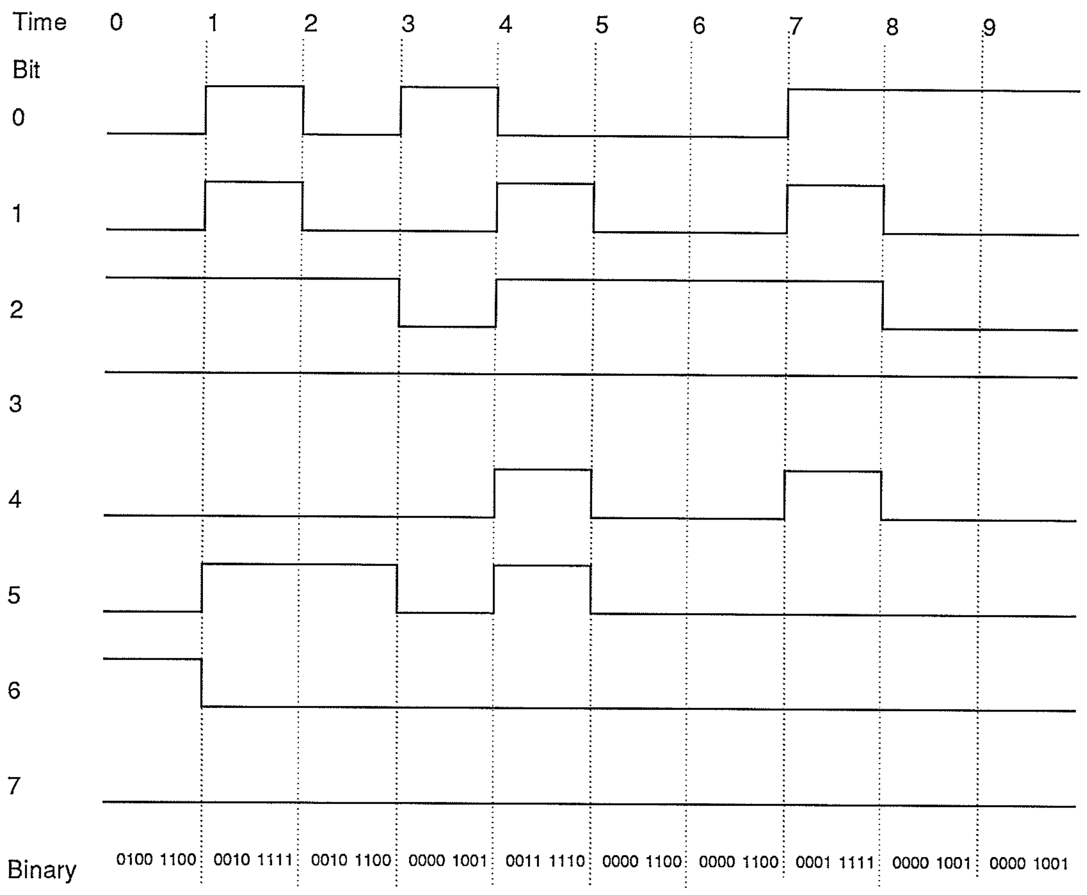
Any time a declaration is encountered, the statement is evaluated and the result (some variable, constant or channel) is placed in the environment list of the following process.

When a variable is subscripted or bitsliced, a new set of bit pointers are created. However, the bit pointers still point to their old bit structures. This is also the case for constants and channels.

Channels are a little bit more complex than the reader has been led to believe up until this point. Each channel bit actually contains a process control structure; this is used to implement channel delays. The following pseudo-code illustrates the channel mechanics:

```
initially:

        if a "!" or a "!!" command was issued
                if channel process is in WAIT TIME state or
                there is a process in the channel output queue
                        channel new value := UNDEFINED
                else
                        channel new value := value sent
                        put channel process onto time queue for
                        appropriate delay
                add sending process onto output queue

        if a "?" command was issued
                if new data ("new" bit set)
                        return channel old data
                        activate processes on output queue
                else
                        place receiving process onto channel's input
                        queue

        if a "??" command was issued
                return channel old data
                activate processes on output queue


after channel delay time event occurs:

        look through output queue for processes that issued a
        "!!" command; if any found, activate them

        if there are any processes waiting on the channel input queue
                activate them
                activate all processes on the output queue
        channel old value := channel new value
```

Channels specified in a Circa command typed in by the user during simulation are assumed to be external channels. External channels are really just UNIX files and therefore allow the user to provide external stimuli to his model or to collect simulation results. The file format is the same for input files as it is for output files. Each line contains two numbers; the first number is the value to be sent (or received), while the second number denotes the duration of the signal (see figure 27).



The resulting external channel file would be:

```
#4c  0
#2f  1
#2c  2
9    3
#3e  4
#c   5
#1f  7
9    8
```

Figure 27: External Channels

Chapter VIII

CONCLUSIONS AND FURTHER WORK

8.1 EVOLUTION OF THE CIRCA LANGUAGE

One of the principal aspects of designing hardware using an HDL is the ability to verify the design. The verification is accomplished through simulation. Often this is done before it is passed to a silicon compiler. MacPitts, a circuit synthesis system, uses this approach [South83].

The development of Circa was an evolutionary process. Early on, structural hardware description languages were considered; they were soon discarded in favour of a behavioural language as the latter's advantages became apparent. As we discussed in previous chapters, Occam was chosen as the basis for Circa because of its ability to deal with processes.

Of course, we were not satisfied with Occam. The communications protocol was among the most significant deviations from the Occam definition, but by no means the only one.

One of the early (and perhaps most superficial) complaints of Occam was its rigid format. Occam uses two spaces for each level of indentation. The initial solution was to place the body of a constructor within a C-like begin-end block: that is, enclose the constructor body within a "{"-"}" pair. Although this allowed complete freedom for the Circa programmer, rarely was it used. Circa programmers writing in a reasonable manner tended to format the source text much like Occam programmes. The result was code that appeared very much like Occam with additional "{"-"}" clutter.

A compromise was taken allowing some freedom while maintaining the clean format of Occam. This took the form of relative positioning of statements. Subordinate statements have a greater indentation than their superiors. This is the only rule and seems to work quite well.

More fundamental "improvements" were considered, but were once again discarded in favour of Occam's definition. An IF constructor was initially omitted since it was deemed to be unnecessary. An alternative constructor with expressions as the only guards would be functionally equivalent. This is not quite the case when compiling to hardware. The IF is evaluated sequentially, while the ALT is a parallel construct. Thus the IF was included.

Occam's WHILE was chosen as the loop construct over a repeat loop despite the repeat's many advantages [Buhr85]. The WHILE construct has one main advantage; it is much easier to implement in hardware.

A type concept was introduced. Occam has only two types, WORD and BYTE, which are not sufficient for the task at hand. The size of each variable or channel element must be determined by the system designer. As well, the type mechanism was extended to encompass externally defined constants and procedure abstractions.

Each type consists of three components: the type name, the actual type (channel, variable, constant or procedure) and the type attributes. The type attributes consist of a procedure body, a list of constants, or a channel or variable description.

The type mechanism allows the programmer to hide some of the details regarding channels, variables, etc. By including procedure abstractions under types, we are able to use variant parameters with procedures in a clean, elegant way.

Other differences between Circa and Occam include additional operators and the uniform treatment of constructor replicators and subranges.

There is, of course, room for improvement in Circa. Monadic operators (negation specifically) would be convenient. Caution should be taken, however, not to fall into the trap of featurism that plague many of the conventional languages (PL/1 for example).

One improvement would be to alleviate the problem Circa programmers have with the limited room for the procedure header (it must fit on one line). The space restriction acts as an incentive to construct headers with only a few parameters. This tends to make procedures easier to work with, since their interface is simpler and more comprehensible.

Each so-called improvement must be judged according to the language philosophy; in Circa's case: keep it simple.

8.2 APPLYING CIRCA TO HARDWARE DESIGN

The vehicle through which the technique of abstraction has been utilised is the hardware description language, Circa. We have looked at how Circa was successfully applied to the problems of circuit synthesis and verification.

While a simulator has been implemented, a silicon compiler has not. The silicon compiler holds a great deal of promise: more so than ever before. Together with technological improvements, building chips will become as easy and as commonplace as programming.

Appendix A

THE SYNTAX OF CIRCA

The syntax of the Circa language is described using Wirth's version of BNF [Wirth77].

```
type_definition = id “(” id {“;” id} “)” “::” type_body.

type_body       = “CONS” expr {expr}
                | “VAR” bitslice
                | “CHAN” bitslice chan_delay
                | “PROC” “(” parm_list “)” process.

chan_delay      = “FALL” expr “RISE” expr
                | “RISE” expr “FALL” expr.

parm_list       = item_declaration {“;” item_declaration}.

process         = item_declaration “:” process
                | construct
                | abstraction
                | simple_command.

construct       = “WHILE” expr process
                | “PAR” construct_body
                | “SEQ” construct_body
                | “IF” expr_body
                | “ALT” guarded_body.

construct_body  = [size] process {process}.

expr_body       = [size] expr process {expr process}.

guarded_body    = [size] guard process {guard process}.

abstraction     = id [size] “(” channel {“;” channel} “)” “::” type.

item_declaration = id [size] “::” type.

size            = “[” [id “::”] subrange “]”.

type            = id “(” expr {“;” expr} “)”.

guard           = simple_guard {simple_guard}.

simple_guard    = expr
                | wait_command
```

```
                  | input_command.

simple_command    = variable ":=" expr
                  | wait_command
                  | input_command
                  | output_command
                  | "SKIP".

input_command     = channel "?" variable
                  | channel "?" "()"
                  | channel "?" variable
                  | channel "?" "()".

output_command    = channel "!" expr
                  | channel "!" "()"
                  | channel "!!" expr
                  | channel "!!" "()".

wait_command      = "WAIT" expr.

variable          = id [subscript] [bitslice].

channel           = id [subscript] [bitslice].

subscript         = "[" expr "]".

bitslice          = "{" expr "}".

subrange          = expr ".." expr.

constant          = id | number | hex_number | character.

expr              = element {operator element}.

element           = variable
                  | channel
                  | constant
                  | "(" expr ")".

operator          = "+" | "-" | "*" | "/" | "\" | ","
                  | "<" | ">" | "=" | "<>" | "><" | ">=" | "<="
                  | "&" | "|" | "&&" | "||".

id                = letter {alphanumeric}.

alphanumeric      = letter | digit.

number            = digit {digit}.

hex_number        = "#" hex_digit {hex_digit}.

character         = "'" any_character_except_eof_and_eoln "'".

comment           = "--" {any_character_except_eof_and_eoln} eoln
                  | "--" {any_character_except_eof_and_eoln} eof.
```

REFERENCES

[Back78] J. Backus: “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs,” Commun. of the ACM, Vol. 21, Number 8, August 1978

[Blair85] G. S. Blair, J. R. Malone, J. A. Mariani: “A Critique of UNIX,” Software-Practice and Experience, John Wiley and Sons, Vol. 15, Number 12, December 1985

[Buhr85] P. Buhr: “A Case for Teaching Multi-exit Loops to Beginning Programmers,” Sigplan Notices, ACM, Vol. 20, Number 11, November 1985

[Dijk75] E. W. Dijkstra: “Guarded Commands, Nondeterminacy, and Formal Derivation of Programs,” Commun. of the ACM, Vol. 18, Number 8, August 1975

[Deny85] P. Denyer, D. Renshaw: VLSI Signal Processing: A Bit-Serial Approach, Addison-Wesley, 1985

[ElAy85] K. El-Ayat, R. Agarwal: “The Intel 80386 - Architecture and Implementation,” IEEE Micro, Vol. 5, Number 6, December 1985

[Fay84] D. Fay: “Experiences using Inmos Proto-Occam,” Sigplan Notices, ACM, Vol. 19, Number 9, September 1984

[Gajski85]D. Gajski: “Silicon Compilation,” VLSI Systems Design, Vol. 6, Number 11, November 1985

[Harl85] D. Harland: “Towards a Language for Concurrent Processes,” Software-Practice and Experience, John Wiley and Sons, Vol. 15, Number 9, September 1985

[Hoare78] C. A. R. Hoare: Communicating Sequential Processes, Commun. of the ACM, Vol. 21, Number 8, August 1978

[Hayes83] A. Hayes: “Self-Timed IC Design with PPL's,” Third Caltech Conference on Very Large Scale Integration, Computer Science Press, 1983

[Inmos84] Inmos Ltd.: Occam Programming Manual, Prentice-Hall International, 1984

[Inmos85] Inmos Ltd.: IMS T424 Transputer: Preliminary Data, Inmos, February 1985

[Johan79] D. Johannsen: "Bristle Blocks: A Silicon Compiler," 16th Design Automation Conference, IEEE Computer Society Press, 1979

[Kato83] S. Kato, T. Sasaki: "FDL: A Structural Behaviour Description Language," Computer Hardware Description Languages and Their Applications, North-Holland, 1983

[Laws77] H. Lawson, Jr: "Computer Architecture and Microprogramming," Software Portability, Cambridge University Press, 1977

[Lesk75] M. Lesk: "Lex - A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, A T & T Bell Laboratories

[Lewin81] D. Lewin: "Computer Aided Design for Microcomputer Systems," Vol. 126: Microcomputer System Design, Springer Verlag, 1982

[Mead80] C. Mead, L. Conway: Introduction to VLSI Systems, Addison-Wesley, 1980

[Moto82] Motorola: MC68000 16-bit Microprocessor User's Manual, Prentice-Hall, 1982

[Patt82] D. Patterson, C. Sequin: "A VLSI RISC," Computer, IEEE, September 1982

[Poun86] D. Pountain: "Personal Supercomputers," Byte, Vol. 11, Number 7, July 1986

[Radin82] G. Radin: "The 801 Minicomputer," Proc. Symp. Architectural Support for Programming Languages and Operating Systems, March 1-3, 1982

[Rowen86] C. Rowen, L. Crudele, D. Freitas, C. Hansen, E. Hudson, J. Kinsel, J. Moussouris, S. Przybylski, T. Riordan: "RISC VLSI Design for System-Level Performance," VLSI Systems Design, Vol. 7, Number 3, March 1986

[Solo86] J. Solomon: IEEE International Solid-State Circuits Conference: Keynote Address, Micronews: IEEE Micro, Vol. 6, Number 2, April 1986

[South83] J. Southard: "MacPitts: An Approach to Silicon Compilation," Computer, IEEE, December 1983

[Suss81] G. Sussman, J. Holloway, G. Steel Jr., A. Bell: "Scheme-79: Lisp on a Chip," Computer, IEEE, July 1981

[Ungar84] D. Ungar, R. Blau, P. Foley, D. Samples, D. Patterson: "Architecture of SOAR: Smalltalk on a RISC," Proc. Eleventh International Symposium on Computer Architecture, 1984

[Wirth74] N. Wirth: "On the Composition of Well-Structured Programs," Computing Surveys, ACM, Vol. 6, Number 4, December 1974

[Wirth77] N. Wirth: "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?", Commun. of the ACM, Vol. 20, Number 11, November 1977