Synthesis of Hardware Systems from Very High Level Behavioural Specifications

Richard Millar Marshall

Ph D

University of Edinburgh

1986

Abstract

Electronics are now in wide use replacing mechanical controllers in products such as consumer goods and cars. Typically these electronic controllers consist of a microprocessor and some interfaces, such as digital-to-analogue converters, connected to the system being controlled. The design of the hardware and software is disjoint, with both being specified at a low level. Control software can be very complex, involving the handling of a number of concurrent tasks; consequently it is very difficult to write, and many revisions must be made before it is correct. As these controllers become single chips, mistakes in software become extremely expensive to rectify. This thesis presents a novel system which simulates and synthesises complete processor-based controllers from the software they are to run.

The programming language OCCAM is used to describe both the structure and behaviour of the controller system at an extremely high level. The structure is expressed by declaring interface circuits as imported procedures, which are executed in parallel with the behaviour of the controller, specified as OCCAM code.

The interface circuits are described in libraries. Associated with each is a number of OCCAM subprogram bodies. One of these is always a behavioural model of the circuit, for use in simulation, while the others are for different implementation technologies, communicating with the hardware and presenting a consistent interface to the control program. In this way a single design document can be used *without change* for simulation and synthesis.

An expert system, the *artificial engineer*, uses the structural information extracted from the control program to design the required controller hardware. It consults the designer over issues such as performance and cost, and designs within these constraints.

To demonstrate the practicality of the approach, a prototype has been built which can simulate designs and synthesise board-level controllers based on Z80 processors. Several complete examples are given. An architecture based on customised processors is proposed for VLSI implementation.

Declaration

I declare that this thesis has been composed by myself, and that the work it describes is entirely my own.

Table of Contents

Chapter 1

Introduction

... The hardware designer, through microprocessor and memory architectures, has merely swept the complexity of the system under the rug of software.

[Lucky 85]

Electronic control systems are becoming more and more common. Everywhere switches are being replaced with touch panels, dials with digital displays. Everything from washing machines to aircraft are now "computer controlled". Microelectronics fabrication technology has moved in step with, and in part driven, these developments. However, the methods for designing controllers have changed very little, remaining largely manual. As a result, the cost of design can become a dominant factor in the overall price of a system. This makes it an ideal area for the application of computer aided design. Before attempting to design or build design tools for controllers, we must examine what electronic controllers are, and why traditional semi-mechanical systems have been abandoned in their favour.

The introduction of electronic control can combine a reduction in size and weight with an enormous increase in functionality. An excellent example of this is the modern compact camera [Gaitonde 82], providing totally automatic control in a smaller package than the older mechanical, manual models. The flexibility of digital systems allows expensive variable analogue controls to be replaced with cheap switches, as with digital frequency-synthesis in radio and television [Groh 80]. For example, it is now possible for radios to have continuous

sweep tuning from ultra low to ultra high frequency without large range-switches selecting between banks of variable capacitors.

Microprocessor based systems permit a greater degree of control than was previously available [Clifford 78, Azuma 80]. The ability to store relatively large amounts of information and to sense conditions allows much better decisions to be taken. In many circumstances precomputed look-up tables can be used to provide complex analyses of circumstances, replacing intricate mechanical or analogue systems. Function and appearance can be changed simply and quickly by modifying the control program, allowing easier upgrading and adaptation to international markets [Penn 77].

Digital interfaces between circuits allow the easy integration of remote control [Karstand 80, Inoue 82, Platte 85], as they remove the need for bulky multi-core connections and transmission of analogue signals. Digital control can also assist in manufacture and maintenance [Wijen 86]: the provision of an external interface to the controller permits production-line computers to configure and check appliances, and service engineers can use the connection to obtain diagnostic information in the event of a breakdown.

The automotive industry is now turning to electronics: in 1984 an estimated $7.5 billion [Brandt 85] was spent in the area. Developments have taken place in all sectors [Jurgen 83, Ford 84, Weber 85], some of which are covered below. The worldwide tightening of exhaust emission regulations and the requirement for fuel economy has forced manufacturers away from traditional mechanical carburattion. The only way to produce marketable performance and stay within the law was to employ microprocessor control [Watanabe 84], using the sensing and information storage capability to optimise fuel delivery and timing. Other areas are active suspension, skid-free braking, collision detection and replacement of the expensive and error-prone car wiring harness [Preston 82],

Medicine is another field where microcomputers have made a major impact. Here all the attributes of electronic control are beneficial. Many applications are in the area of monitoring, where the ability to read several interfaces, record data *and* interpret it, makes a significant contribution to post-operative patient

care [Naghdy 84]. Miniaturisation is also important, allowing patients to carry equipment that was previously not portable [Barker 86], or even have devices implanted in their bodies.

Examining these *control computers* reveals that they are generally made up of a microprocessor with a ROM, a small amount of RAM and a number of peripheral interface chips. Most of these interfaces are off-the-shelf parts such as analogue-to-digital converters, display drivers, timers and parallel boolean input and outputs. Some applications benefit from using specially designed interfaces, but these still connect to the microprocessor like standard parts [Caironi 82].

The use of bus structures and memory mapped devices controlled centrally by microprocessor has made hardware design relatively easy. To a large extent it is now simply a case of putting together building-blocks with a small number of "glue" chips; gone are the large boards of random logic implementing the control and interfacing functions. This is yet another reason for the growth in use of electronic controllers: they are easy to build. What is not easy to build, however, is the software required to animate the hardware. All the control functions are left to the programmer to implement, usually after the hardware design has been finished. There is little integration between the two tasks.

The main difference between hardware and software is that the former is fully parallel, and the programmer emulating control logic must simulate this concurrency. The program is likely to be written in a very low-level language and it is unlikely to provide constructs to assist in this task, so a large number of revisions of the software are needed before being fully functional. As a result the software is more complex than the hardware, and it must be regarded as the major system component; control computers can be regarded as heterogeneous objects, or *heterosystems* as Organick has called them [Organick 84].

Standard-part integrated-circuit technology has provided the building blocks for control systems: originally small scale integration logic, moving on to large scale integrated computer components. Semi-custom integrated circuits and the design automation tools for them are now following the same paths for users of application-specific integrated circuits (ASICs). Initially these could integrate

random logic functions on a single chip, for example Lattice Logic's Chipsmith gate array and standard-cell compiler [Lattice 84a].

Design-tool vendors are now offering microprocessors as sub-chip components, allowing the integration of complete control computers on a single chip [Evanczuk 85]. For example, Sony use a special-purpose 4-bit microcomputer as a controller in their products [Numata 85]. VTI offer an exact replica, complete with original bugs, of the 6502 microprocessor as a "mega-cell" in their design system [Trimberger 85]; General Electric provide AMD 2900 series compatible bit-slice processors as cells [Parisoe 85], and INTEL supply "microcontroller" processors [Amini 85].

These design tools are aimed at direct replacement of printed circuit-boards, allowing engineers to work as they always have done, except that the end result is a chip. This is partly due to commercial necessity and partly because it was an easy route to follow, although only through this basic work do more advanced uses come to light. This approach is unfortunate because these tools do not realise the full potential of the technology and they preserve problems. For example the problem of separate hardware and software development is aggravated: when a complete controller system is cast in silicon it simultaneously becomes very expensive to correct mistakes in the software and impossible to test the code *in situ.*

It is obvious from the foregoing discussion that there is a need for a new design style for fully integrated control-computers. It is my hypothesis that by identifying a suitable programming abstraction the complete controller can be described at a very high level by the software.

Systems can be described in three ways: *physical*, *structural* and *behavioural.* A physical description defines how the system is built in enough detail to actually make it. A structural description states what components the system has, and how they are connected. A behavioural description says what the system *does*; this is a rather fuzzy term, but as will be seen later there is no better definition.

The most natural method of description for controllers is behavioural. For

example, a very high level definition of how a thermostat behaves might be as follows:

> *Check the temperature and if it is less than we want turn on the heater; otherwise turn off the heater; then do it all again.*

Unfortunately this description is not amenable to computer processing, but it does show something important about behavioural descriptions: in order to describe the behaviour, some information about the structure of the controller must be given. In this case we learn that there must be some device for checking temperature and something for switching the heater on and off. In essence, a structural description is *declarative* and a behavioural is *procedural.* Programming languages are, of course, procedural, and have declarative sections; so our aim of describing the controller in software can still be met.

An important feature of controllers, as stated above, is concurrency, the ability to respond to several stimuli at once for example, and this is what makes programming them very difficult. Several recent programming languages feature well defined and theoretically sound concurrency models, so are ideal for our task.

This thesis describes a system which uses software to describe controllers completely, the Unified Software and Hardware EngineeR, USHER. This uses the programming language OCCAM to define both the structure and the behaviour of the controller and can simulate and synthesise physical designs from it via technology-dependent back-ends. Each back-end comprises a code generator for the appropriate processor, a hardware allocator which extracts the structural information from the software and a knowledge-base on that technology's implementation. An expert system, the *Artificial Engineer*, creates a physical realisation from the structure derived from the software and the knowledge-base. The *Artificial Engineer* interviews the designer over issues such as performance, cost, power consumption and connections to the system under control. By the use of user-definable libraries, all parts of this system are *open*: that is, designers can extend them as required to meet new applications.

To demonstrate the validity of these ideas a prototype implementation of USHER has been developed. This includes the simulator and a back-end that designs board-level controllers using Z80 microprocessors. A Z80 microcomputer fitted with a number of common interfaces has been built as a test-bed. The interface circuits from this test-bed machine are used as the foundation for the *Artificial Engineer* knowledge-base. Three examples have been developed to test the prototype.

A structural description of this thesis might be as follows:

Chapter 2 examines other work in the field of compiling high-level languages to hardware in particular, and behavioural description and synthesis systems in general.

Chapter 3 describes the concepts behind the USHER system in detail. It gives a rationale for choosing OCCAM, describes the minor extensions to the syntax of the language, outlines the overall structure of the software, and overviews the steps involved in the making of a design.

Chapter 4 describes the USHER simulator and its environment. Various techniques for simulating interfaces are discussed.

Chapter 5 describes the implementation of the Z80 compiler and test-bed machine.

Chapter 6 introduces the *Artificial Engineer*. A limited defence of expert systems is presented, and the operation of this one described. Details of the prototype interface-designers are given.

Chapter 7 describes the three test programs. It shows the various stages in their development and demonstrates both simulation and synthesis. The simplest example is a stopwatch, another is a digital voltmeter and the largest a cash register.

Chapter 8 charts the successes and identifies the weaknesses of the USHER system, concept and implementation. It then discusses alternative back-ends for USHER, including an appropriate VLSI architecture.

Chapter 2

Solidifying Software

Reading maketh a full man; conference a ready man; writing an exact man.

Francis Bacon

Proposals to use normal software programming languages (SPLs) as hardware description languages (HDLs) are not new. In 1962 APL was proposed as a common language for hardware, systems- and applications-programming [Iverson 62]. Despite a recent suggestion of using APL for architectural development [Sinderen 86] it has never caught on, but the idea of using languages for hardware description has been widely adopted. Despite two major international standardisation efforts, still more languages appear.

Most researchers using SPLs for hardware description are forced to alter and extend the language, both in syntax and semantics, to suit their needs. In this way they are really using a different language. A number of projects, including the one described in this thesis, use the SPL unaltered except for minor changes to the syntax to make it more convenient for the application. This survey covers several systems using unaltered SPLs and a number of special purpose or adapted HDLs. These latter are either included to show how different approaches to behavioural description can be taken, or because they have made a major impact in the field. This chapter does not try to be a complete survey of HDLs as this would take far to long. Nash has provided an extensive bibliography [Nash 84] and a number of survey papers have been published [Pawlak 85].

2.1 CONLAN

CONLAN is a project to develop a CONsensus LANguage. It was first proposed in 1973, and a multi-national working party was set up in 1975, finally reporting in 1983 [Piloty 83]. This was the first major effort at standardisation in the area of HDLs and was motivated by the proliferation of languages, mainly from universities, and industry's low level of acceptance of them. The working party decided that no single HDL would be suitable for all applications, and defined a basis for the construction of a *family* of languages. An extensible syntax was developed allowing tool makers to derive new members of the the family for new applications, within the overall CONLAN framework, with a common base of syntax and semantics.

A basic *Primitive Set CONLAN* (PSCL) was defined as a parent or *reference* language from which all others are derived. PSCL is intended to provide a minimum of fundamental concepts with which the working party believe it is possible to describe all useful hardware. To demonstrate the extensibility facilities and provide a more usable language the working party also defined the first member of the family: *Base CONLAN*, (BCL). This builds all the basic facilities normally found in a language (for example, arrays) from PSCL primitives. Future family members are expected to use BCL as their reference language, not PSCL, to avoid recreation of these facilities.

CONLAN can describe both structure and behaviour of hardware at a variety of levels, however, as it stands it is too general and complex to be a useful tool. BCL must be regarded as a powerful base of constructs, not an HDL itself. By selecting the appropriate facilities a language that closely models a particular application can be derived, for example WISLAN [Engh 83] a CONLAN member for describing gate arrays.

2.2 VHDL

The United States Department of Defence is the world's largest purchaser of technology-based products, so it exerts enormous influence over industry. As part of its Very High Speed Integrated Circuits (VHSIC) initiative, the VHSIC Hardware Description Language (VHDL) project was launched in 1981. The language was jointly developed by Intermetrics, IBM and Texas Instruments, and was first published in 1984, with the current revision, 7.2, being published in June 1986 [Intermetrics 86].

VHDL is to HDLs as Ada [US DoD 80] is to programming languages. This is for three reasons: firstly its use is likely to be a requirement for defence contractors, secondly its syntax is derived from that of Ada, and thirdly it provides a vast wealth of constructs. Like the CONLAN designers, the VHDL team realised that there are many styles of design, but they decided to incorporate as many as possible into a single coherent linguistic framework.

Fortunately, the chosen design styles are well matched and form a continuum from pure structural to pure behavioural design. Circuits from complete multi-board systems to low-level cells, are described by `architecture` bodies. For each body there is a `block` which describes it. The statements within a `block` are all labelled and are executed concurrently. Any statement can be a `process` block, in which the statements that make it up are executed sequentially. Statements can be behavioural, structural or a mixture of the two. Figure 2–1[1] shows (a) a behavioural description of an adder, using the conventional assignment operator and expressions; (b) a mixed level description using signals and abstract operators; and (c) a purely structural description using named components.

[1]This example is drawn from the VHDL User's Manual, Volume 1 - Tutorial

```
architecture TOP of ADDER is
   block
   begin
      process (A, B, SUM)
         variable A, B : INTEGER range 0 to 3;
         variable SUM  : INTEGER range 0 to 6;
      begin
         SUM := A + B;
      end process
   end block
end TOP
```

(a) Behavioural Description

```
architecture MIXED of ADD is
   block
      signal S1, S2, S3 : BIT;
   begin
      S1   <= X xor Y;
      SUM  <= S1 xor CIN;
      S2   <= X and Y;
      S3   <= S1 and CIN;
      COUT <= S2 or S3;
   end block
end MIXED
```

(b) Mixed Behavioural and Structural Description

```
architecture STRUCT of HALF_ADDER is
   block
      component NAND_GATE port (A, B : in  BIT;
                                C    : out BIT);
      component XOR_GATE port (A, B : in  BIT;
                               C    : out BIT);
      component INV port (A : in BIT;
                          B : out BIT);
      signal T1;
   begin
      Z1: XOR_GATE port (X, Y, SUM);
      Z2: NAND_GATE port (X, Y, T1);
      Z3: INV port (T1, COUT);
   end block
end STRUCT
```

(c) Structural Description

Figure 2–1: An example of an adder expressed in VHDL

An unconditional behaviour is expressed as a series of assignment statements. A state-machine is used to describe conditional actions; the syntax for this is based on a case statement, which is available in two forms, firstly the long hand:

```
case OPCODE_REGISTER of
   when ADD     => ....
   when COMPARE => ....
   when JUMP    => ....
     :    :
end case;
```

Or as a short hand form in signal assignments:

```
FLAGS <= "001" when RESULT < 0 else
         "010" when RESULT = 0 else
         "100";
```

Structural information is represented at two levels, using signal assignment and abstract operators at the higher level, and component instances with port connection at the lower, purely structural level. These are both shown in Figure 2–1.

VHDL borrows a variety of features from Ada: functions and procedures, the concept of the package for hiding implementation details, and an extremely powerful type-structure. Typing is used to check interfaces between components, not allowing signals of different types to be connected. The attribute mechanism for attaching extra information to objects (accessed by *object'attribute*) can be used to support physical design and testing, down to packaging and mask level [Lowenstein 86].

VHDL is the most powerful HDL to date, combining many functions in an elegant manner. The ability to describe systems in many styles and at different levels of abstraction is extremely powerful. The verbose syntax makes the meaning and structure of a design quite clear, the use of a case statement for state machines is particularly suitable. A controversial issue with the language designer's is the degree with which Ada can be integrated into VHDL [Nash 86]. Unfortunately Ada is currently not included as part of VHDL, although adding

it would solve some problems with the current language. Chapter 3 discusses the application of Ada code integrated into VHDL with respect to the current work.

2.3 Zeus

The Zeus hardware description language has been developed jointly at ETH Zurich, Princeton University, MIT and GTE Laboratories [German 85]. It was the designers' intention that the language should closely model hardware and provide structuring facilities to support systematic design. It was also intended to be used not only as a design-documentation aid, but as an input language to a variety of design tools, including simulators and silicon compilers. Zeus syntax is based on that of Modula-2 [Wirth 82], which provides structuring via MODULES and strong typing. Modula-2 itself can be used as an extension to Zeus when greater descriptive power is required.

The basic unit of hardware description is the COMPONENT, for which the designer can specify both connections and parameters. Within a COMPONENT constants, signals, and instances of COMPONENTs, including recursive instances can be declared. The Modula-2 type mechanism is fully employed in checking signals and their connections. For example, four possible values for a tri-state signal can be declared as follows:

```
TYPE
   tri_state = (zero, one, undef, high_imp);
```

From this a range type can be derived:

```
logical = [zero..undef];
```

which is the the type of most signals. Using this basic type composite signals, such as busses, can be created using arrays and records.

The main body of a COMPONENT is specified as a CONNECT block. The WHEN construct can be used to provide a number of conditional bodies which permits

recursive instancing to terminate, and special cases such as the sides of arrays to be defined. Within each body all statements operate concurrently. Connections are specified using assignment, and provide structural information. A `FOR` construct is provided for replicated structures. Behaviour is described as a state machine and is specified using `IF` statements, which act like a case statement, except that more than one branch may be active at once since they all act concurrently.

One intended use of Zeus is as an input language to silicon compilers. In this circumstance the designer may be working within area-constraints, so she is allowed to give the tools a helping hand with the `ORDER` statement to assist with floorplanning. This uses a concept like that in ALI [Lipton 83] of gluing blocks together by their edges. It is also possible to specify the ordering of ports on each side of a component.

Zeus has been well integrated into a rich language, providing a good environment for design. The use of typing, particularly enumerated types, creates a readable document, and together with the use of built in functions such as `WIDTH`, allowing easy definition of generic components. The use of assignment to indicate connection, and allowing multiple concurrent assignments from within different `IF` sections is confusing and potentially dangerous. The use of multiple `IF` blocks to specify behaviour is more flexible than a case-statement, but it is an untidy notation, looking too much like the Modula-2 original: a sequence of actions.

2.4 MIMOLA

The MIMOLA (Machine Independent MicrOprogramming LAnguage) system has been under development at the University of Kiel, West Germany, since 1976. It was first presented in 1979 [Zimmermann 79], and again with a revised syntax in 1984 [Marwedel 84]. Two levels of description are provided: *hardware*, which is at the register-transfer level, and *action*, which is state-machine or

```
MODULE B74381 (IN  left, right : .BIT(3:0);
               IN  select      : .BIT(2:0);
               OUT result      : .BIT(3:0));
BEGIN
   result := CASE select OF
             0 : 0;
             1 : right "-" left;
             2 : left "-" right;
             3 : left "+" right;
             4 : left "XOR" right;
             5 : left "OR" right;
             6 : left "AND" right;
             7 : -1
             END
END;
```

Figure 2–2: A MIMOLA module representing a 74381 4-bit ALU

micro-program oriented. The initial language had a minimal syntax, relying on the first letter of a statement or declaration to indicate its type. The later version uses a syntax derived from PASCAL and is considerably more readable.

Hardware entities are described as MODULEs, which have connections that can be read and assigned values. It is also possible to declare variables which correspond to registers, and arrays which correspond to memories. Even in the new system, memory-element identifiers must begin with an 'S' for store. The body of a module contains *actions*, which can either take place sequentially or in parallel. This choice can be made by the designer by using either a comma as statement-separator for concurrent statements or a semi-colon for sequential statements. An optimiser attempts to maximise the parallelism within each module. A case-statement is used to implement state-machines, as shown in Figure 2–2 which demonstrates a definition of the TTL standard part 74381, a 4-bit ALU. The usual language constructs of IF, FOR etc are also provided.

Two synthesis algorithms have been used with the MIMOLA system. The most recent [Marwedel 86] produces hardware that is half the size of that produced by its predecessor [Marwedel 79]. The first step in the new approach is to compile MIMOLA algorithmic descriptions to register-transfer level, which is another level within the language. From this level various modified versions of

microcode optimisations are applied to minimise the width and number of microinstructions. This process produces another, near optimal, register-transfer description, and this is used to drive a standard-cell system. Control is provided by horizontal microcode.

The MIMOLA suite is very impressive: it can compile complete minicomputers from their microprograms. The language, although slightly stilted and lacking typing, is well suited to the task it was designed for: describing digital computers. Unfortunately the micro-code orientation of the language means that it is not suited to describing generalised hardware structures.

2.5 CAP/DSDL

The CAP/DSDL system has been developed at the University of Dortmund, West Germany. It comprises the language CAP/DSDL (Concurrent Algorithmic Programming/Digital System Description Language) [Rammig 84][2], its compiler, a stimuli compiler and a simulator [Dachauer 81]. The language is described as "broadband," providing a variety of description levels from high-level algorithmic, through register-transfer and gate to switch-level. The usual constructs are available, and the language is typed. Blocks of operations can be specified as sequential, or synchronous or asynchronous concurrent. The designer can specify *assertions* about internal states of the design, and these are continuously checked during simulation, ensuring design correctness.

The different levels of design are unified by a common basic concept, the timed Petri-Net. Petri-Nets are a simple extension of finite state machines, and can model the three classes of control described above, sequential, synchronous and asynchronous. A Petri-Net interpretation exists for each language construct. Petri-Nets are also the basis of the system description facility of CAMAD [Peng 86], developed at Linköping University, Sweden.

---

[2]Since updated and renamed DACAPO [Brueck 86]

The CAP/DSDL team have recognised that concurrent algorithmic descriptions are well suited to the design of controllers [Brück 86]. In this synthesis system a controller is made up of a number of concurrent low-level modules. Each module is implemented as a clocked, finite state-machine, but communication between modules is asynchronous, so the whole system can be regarded as self timed. This avoids problems of clock skew on large chips, making automated design easier. The CAP/DSDL language does not enforce this design style, only this synthesiser. The design is examined and any additional concurrency is automatically detected. Each module is described in a way similar to that of VHDL, using case statements for conditional behaviour and assignment for connection. Synthesis proceeds by separating the data- and control-paths, each part then being implemented by CMOS standard-cells. In the case of the control-path, each Petri-Net operator has a corresponding cell, and the control-machine is built by placing the cells in a fixed floor plan, with interconnection in a central routing channel.

An interesting feature of the CAP/DSDL system is the inclusion of a *stimuli compiler*. This accepts a language that specifies signal changes according to time, which are used to drive the inputs of the design. With this it is possible to describe an external environment, but it cannot provide feedback on the the design's outputs. An analyser is provided to interpret the output from the simulator, showing only relevant information.

The simulation and synthesis tools provided by this system are powerful, and have found acceptance in other German universities. It is also one of the very few university-developed design automation systems to find favour in industry, in this case with Siemens AG in Munich [Frantz 83]. It is also different from efforts such as MIMOLA in that it is not intended solely for designing computers, it can be turned to more general tasks. The use of Petri-Nets provides a sound basis, but it is marred by the fact that Petri-Nets cannot be composed hierarchically. This requires designs to be decomposed as a group of concurrent modules rather than the conventional hierarchy of components.

$$NAND00 \Leftarrow \alpha 1 NAND10 + \beta 1 NAND01 + (\alpha 1 \beta 1)\gamma 0 NAND11$$

$$NAND01 \Leftarrow \alpha 1 \gamma 0 NAND11 + \beta 0 NAND00 + (\alpha 1 \beta 0) NAND10$$

$$NAND10 \Leftarrow \alpha 0 NAND00 + \beta 1 \gamma 0 NAND11 + (\alpha 0 \beta 1) NAND01$$

$$NAND11 \Leftarrow \alpha 0 \gamma 1 NAND01 + \beta 0 \gamma 1 NAND10 + (\alpha 0 \beta 0)\gamma 1 NAND00$$

Figure 2–3: A CIRCAL description of a NAND gate

2.6 CIRCAL

The CIRcuit CALculus (CIRCAL) has been developed by George Milne at the University of Edinburgh since 1982 [Milne 83a]. The motivation for the language is the desire to reason about hardware at all levels, eventually leading to formal verification. CIRCAL is based on Milner's Calculus of Communicating Systems (CCS) [Milner 80], and uses a very abstract level of description. It is, for example, possible to describe the behaviour of a directional wire, as shown in this picture:

$$\alpha \rightarrow \boxed{W} \rightarrow \beta$$

with the following CIRCAL process:

$$W \Leftarrow \alpha\beta W$$

This is a recursive state-machine description meaning: an event α occurs, then an event β occurs and the whole process repeats. The event α is a signal arriving at the start of the wire, and its departure along the wire is event β.

This method of description can be used to model larger components, for example a NAND gate as shown in Figure 2–3. In this example α and β are the inputs and γ the output of the gate. There are four possible states, one

corresponding to the each combination of the two inputs. Every possible signal change must have an event associated with it. The second state ($NANDO1$) has three possible events, any one of which can occur non-deterministically, indicated by the + operator. The first event, $\alpha 1 \gamma 0 NANDO11$ means that if the input α changes to 1, then the output γ changes to 0 and the next state is $NANDO11$. The second event shows the input β falling to 0, which does not change the output, but moves to a different state. In the last event $(\alpha 1 \beta 0) NANDO10$ the brackets indicate that α must rise and β fall simultaneously, resulting in a move to state $NANDO10$.

It is easy to see from this example how descriptions at this level explode in size. The need to include the case of a signal change at each port individually and combinations of ports simultaneously enormously increases the size of a description, as well as difficulty for the designer defining each possible case. Descriptions can be *composed*, to form new descriptions with a behaviour equivalent to the primitives, permitting hierarchical design.

It is possible to use CIRCAL as an intermediate code for a silicon compiler to assist in the verification of its functionality [Milne 83b]. CIRCAL allows behaviours to be specified without any structural information. As shown above, CIRCAL descriptions are normally written in a mathematical notation which is not viable for a working HDL, so a lisp-embedding has been developed to allow machine analysis of CIRCAL expressions [Traub 83].

CIRCAL is a powerful tool for reasoning about hardware but is limited in practical application by the rapid expansion in size of designs. Unfortunately the abstract nature of CIRCAL, which makes it amenable to mathematical manipulation, also makes it unsuitable for use in real hardware design. CIRCAL is, however, not without its practical applications. It would be extremely useful for proving equivalence between different designs, possibly expressed in different HDLs. Since it is often the case that a company will market a family of products which offer identical behaviour with different performance, it is obviously very important that the members of a family actually *are* identical; so the ability to verify this automatically would be very helpful.

2.7 CMU-DA

The Carnegie-Mellon University Design Automation (CMU-DA) project started in 1973, with the objective of making exploration of the architectural design-space easier. The input language used is ISPS [Barbacci 81], which was originally devised for describing computer architectures for comparative purposes. ISPS is, however, sufficiently flexible to allow the description of other hardware. The initial implementation of the CMU-DA system [Parker 79] compiled ISPS directly into TTL or CMOS standard-cells. The demonstration example for this system was a PDP-8/E and the results of automatic synthesis from the ISPS description were claimed to compare favourably with the original manual DEC design.

Subsequent tools do not operate straight from ISPS, but use a representation call the *Value Trace* (VT). This is a directed, acyclic data-flow graph similar to that used in optimising compilers. Some simple optimisations are performed on the VT [Walker 83] which can then be used by a variety of tools. Thomas presents a good overview of the whole system base on VT manipulation [Thomas 83].

One tool built to use VTs is EMUCS, a data-path synthesiser [Hitchcock 83]. This is a conventional compiler, allocating hardware as it is needed according to a cost table. Unfortunately it has a number of major drawbacks including requiring the designer to insert busses manually. An alternative approach, using expert systems, produced the Design Automation Assistant (DAA) [Kowalski 83]. This has been used to design a 6502 microprocessor of "acceptable" quality.

Other work based on VTs includes the examination of automatic extraction of busses by employing clique theory [Tseng 83]. This has been embodied in a special tool, *Emerald*, [Tseng 84] and is now integrated into the overall CMU-DA system [Tseng 86].

The CMU-DA project seems to be successful, although, as the papers never show actual evidence to support their claims it is hard to tell. ISPS is a good language for describing processors, but is rather more detailed than required for most applications, for example requiring all registers to be sized at declaration. There is no apparent method for introducing special hardware, for example analogue-to-digital converters, into the ISPS description or the synthesis stages, which limits the kind of systems that can be defined. As long as the design in hand can be implemented as a conventional data-path, control section and memory system, ISPS is an appropriate description language.

2.8 Automatic Evaluation of Design Choices

Doug Baldwin, a postgraduate student at Yale University, has identified making *design choices* as a major element in the circuit synthesis process [Baldwin 84]. By this he means choosing the best implementation for a circuit from a number of alternative solutions. He cites the example of using an adder for performing an addition at one point, whereas a more general ALU could have been used there and also later used to perform a subtraction. The application will determine which is the most suitable: a highly-parallel solution would use an adder and a subtractor, while a low-cost solution would choose the ALU.

To investigate the automation of this decision-making he has developed a system which compiles a dialect of lisp into the control-parts of circuits built from TTL. He does not address the design of the data-part of the circuits as this field has been thoroughly explored. An example of the lisp specification, a car cruise control, is given in Figure 2–4. In the heading the input signals are declared and given a width (1 bit), followed by the output signals, with width and an initial output value (`nil`, the lisp equivalent of false). The main code outputs a true value (`T`) to reset the speed-pulse counter, then sends a true to enable the counter, and finally conditionally selects from the answer one of three options: too slow, too fast and the correct speed.

```
(define-controller Cruise-Control
        (input (Too-Fast 1)
               (Too-Slow 1))
        (output (Count       1 nil)
                (Reset-Count 1 nil)
                (Speed-Up    1 nil)
                (Slow-Down   1 nil))
        (loop
         (do (state (Reset-Count T))
             (state (Count T))
             (cond
              ((Too-Fast)  (state (Slow-Down T)))
              ((Too-Slow)  (state (Speed-Up  T)))
              (else        (state (Speed-Up  nil)
                                  (Slow-Down nil)))))))
```

Figure 2–4: A Car Cruise Control described in Lisp

The system uses state machines to describe behaviour, and can compile them into TTL circuits in a number of different styles, using a table-driven expert system. Baldwin conducted an experiment to compare manual designs with that of his synthesiser: the automatically produced designs were only slightly worse than those of humans. With this encouraging result it is a pity that only control parts were designed and that a language and synthesiser for complete systems was not produced.

2.9 MacPitts

MacPitts [Siskind 82, Southard 83] is a register transfer language based on lisp, intended for the implementation of very high performance systems, such as signal processing. The language is aimed at a specific architecture, a microprogram controlled data-path. Structural information is provided explicitly by the designer through declarations, and behaviour is specified as lisp code. The code is normally sequential, but a construct is provided to execute sections in parallel.

The MacPitts compiler can generate complete chips for signal processing applications and is one of very few systems to accomplish this high-level to working part transformation. It is not, however, without its problems [Fox 83].

```
const
   BusWidth = 23; StoreHeight = 400;
type
   Wire = (Low, High, Undefined);
   Bus  = array [0..BusWidth] of Wire;
   DMSIndex = 1..StoreHeight;

   procedure DataReg (    D : Bus;      Enable : Wire;
                      Start : DMSIndex; var Q : Bus);
   var I : 0..BusWidth;
   begin
      for I := 0 to BusWidth do
      begin
         RDMS (Start + I, Q [I], nil);
         WDMS (Start + I, D [I], Enable)
      end
   end;
```

Figure 2–5: A Register Component

Some of these are due to the simple architecture, for example limited memory capacity. Others are due to implementation expedience, such as overflow of the Weinberger array control structure. MacPitts is an example of identifying an application and tailoring a tool to be an exact match. In this way it is possible to build a design aid quickly and successfully.

2.10 Pascal to Gate-Arrays

The UK5000 is a CMOS gate-array with 5000 gates and 400 D-type master-slave (DMS) flip-flops [Grierson 83]. The standard support software provides two forms of design entry, one is workstation-based schematic capture, the other is a primitive macro-language. To provide a user interface that is more suitable for student use, a compiler which uses a Pascal program to describe the circuit has been developed at the University of Southampton [Jesshope 85]. A language subset is employed and built-in functions are provided to support activities that are not directly available in Pascal. In the example in Figure 2–5 the procedures RDMS and WDMS read and write to the flip-flops respectively.

A significant advantage of using Pascal as an input language is that the program can be compiled and run normally to provide simulation. This is only true in this case because the UK5000 circuits must be defined synchronously, if this simple clocking scheme is not employed a more complex, event-driven simulator is required. In the UK5000 values are latched into the flip-flops on clock pulses which are wide enough to allow the logic to stabilise. In the hardware all the logic operations that prepare the values are run fully parallel, but in the software simulation they are run fully serially. The synchronous updating of the latches is modelled in the program by performing all the calculations, then writing the results into the variables that represent the latches at one time. The paper reports that the programmer must impose a discipline of always performing all flip-flop reads before any writes. The run-time environment generates timing-diagrams at test-points, showing the internal state of the design. This system is very simple, the compiler does not support conditional code, but it does provide cheap access to design and simulation.

2.11 Ada as an HDL (1)

Mario Barbacci has proposed unmodified Ada as an HDL [Barbacci 85]. To utilise existing software tools he suggests that rather than having a "smart" compiler which understands hardware, the description program and libraries should provide the intelligence. The argument used to support this is that design-tools always have some design-style or technology dependence built into them. By allowing the designer direct access to the technology, he is freed from the tool-maker's preconceptions. This is in total contrast to all previous design tools, the objective of which was to liberate designers from the repetitive lower-level tasks by automating them and making designs less technology dependent.

The mainstay of the proposal is the use of Ada's advanced programming facilities. Each component type, for example a NAND gate, is represented by a package, with the public portion providing procedures for component creation,

construction (from its constituent parts) and simulation. Private types are used to store information about components that is for use only by the package. Great emphasis is placed on the use of long and meaningful identifiers, using overloading to reduce the number that the designer must remember. The main program can be modified to call either simulation or synthesis bodies, which produce a design at mask-level.

This paper advances a reworking of an old idea: embedded languages. The use of modern language facilities, such as packages and abstract data-types, permits an elegant implementation, but still leaves the designer with an enormous workload. The argument put forward to support this is counter to all previous design systems, maximising designer effort and the technology dependence of designs.

2.12 Ada Program Units to Silicon

The Ada to Silicon project at the University of Utah examined the interesting questions of where the boundary between hardware and software must be drawn [Organick 84]. If it were possible to compile programs to either hardware or machine-code with equal ease, certain high-performance sections of program (frequently used sections of an operating system, for example) could be implemented in hardware. This is what they have called a *heterosystem*: a complete entity that is made from a variety of types of components, but is described by a single document.

A useful example of a heterosystem is a computer with a network interface. Assuming that the line driver is always present, the connection can be controlled either by software on the main processor, or by a separate network controller. If the special controller is present it interprets the network protocol, presenting the operating system running in the main processor with an idealised interface. When the machine-code version of the controller is in use the operating system still uses the idealised interface, but the packaging is done on the main proces-

sor, so it has to respond to interrupts and spend time servicing them, thereby degrading overall performance.

The ability to interchange hard- and software implementations of modules makes a significant impact on system design. It becomes, at least in theory, possible to have a single document describing the whole system. Implementations with different performance criteria can be built by selecting implementation technology of critical modules. Building some interfaces in hardware introduces *redundancy*; if the hardware fails it can be replaced with its code equivalent. It is also possible to test modules before committing them to hardware by running them as code first. This can only be done satisfactorily if an appropriate timing scheme is adopted. Concurrent software tasks which communicate only when they are both ready, *rendezvous* in Ada terminology, can be compared with self-timed hardware: each section completing in its own time. So, a hardware implementation of a task should be self-timed to fit in with the software tasks it is to communicate with if it is not to restrict the programmer's style. This is in sharp contrast to the software modelling of hardware by software in Section 2.10.

The Utah researchers chose to implement an Internet control task in hardware. This had a number of benign features: small memory requirement, no recursion, simple arithmetic, and communication with only one host-processor task. This was manually *transformed* into silicon as an experiment to test the viability of Ada hardware description. Each transformation had a theoretically sound basis [Subrahmanyam 83] so, barring human error, the final chip did correspond to the Ada code. The control- and data-structures of the Ada code were separated and converted to state-machine and "path programmed logic array" descriptions respectively. These were then mechanically converted to silicon. The chip was designed to fit into an Intel 432 based system, and it was tested in this context using Ada programs. One bug was found, which was a fault in the state-machine compiler rather than the design.

While an automated system has not yet been developed, despite nine people working on the project, this is still significant work. They do not describe any

method of introducing special or user defined hardware into the design process, either for modelling in Ada or inclusion at synthesis, restricting the types of system which can be defined. Identifying the concept of "heteroware" and the attendant importance of specifying good interfaces between modules is a major contribution to hardware synthesis.

2.13 Ada as an HDL (2)

A group of researchers at Carleton University, Ottawa, are developing an Ada to standard-cell compiler [Girczyc 85]. The first stage in the compilation is converting the program into a *control-data flow graph* (CDFG), similar to the value trace used at CMU. Following various operations on the CDFG, performed by parsing it with a graph-grammar, it is transformed into a standard-cell net-list. The familiar microcode controlled data-path architecture is used, with the control graph represented as a state-machine [Girczyc 84]. The example used by the researchers, an oven-control thermostat, is shown in Figure 2–6.

The researchers aim has been to use Ada unmodified, and they have achieved this through use of the **pragma** construct and procedure calls. In the example, the procedure `DISPLAY` represents a predefined hardware entity, and is identified as such by the **pragma**. The code body is for simulation, outputting the value to be displayed to a trace file. One of the objectives of the research is to allow the designer control over the timing and performance of the circuits. To this end constraints can be given using the procedures `REFERENCE` and `CONSTRAINT` from the package `TIMING`. These calls do not generate any hardware but provide information for the hardware allocator.

Simulation is again achieved by running the program conventionally. Timing constraints are checked by a task `TIMING`, with entries corresponding to the `REFERENCE` and `CONSTRAINT` procedures. The granularity of time is important here: the clock rate of the hardware and the time used by Ada `delay` statements are not the same, so scaling is required.

```
type TEMP is INTEGER range -128..127;
type STATUS_CODE is INTEGER range 0..127;

HEAT_OFF_CODE : constant STATUS_CODE := 103;
HEAT_ON_CODE  : constant STATUS_CODE := 57;

HEAT_STATUS   : STATUS_CODE := HEAT_OFF_CODE;

task body OVEN_CONTROL is

   T1, T2, T3, T4, T_SET, T_AVG, T_DIFF, T_ERROR : TEMP;
   OVEN_READY : BOOLEAN;

   procedure DISPLAY (T : in TEMP) is
      pragma HARDWARE_CELL;
      begin
         put (TRACE_FILE, T);
      end DISPLAY;

   begin
      loop
         TIMING.REFERENCE (REF_LABEL => BEGIN_LOOP);
         GET (TEMP_PORT, T1);
         GET (TEMP_PORT, T2);
         GET (TEMP_PORT, T3);
         GET (TEMP_PORT, T4);
         GET (TEMP_DIAL, T_SET);
         T_AVG := (T1 + T2 + T3 +T4)/4;
         T_DIFF := (T2 + T3) - (T1 + T4);
         T_ERROR := (T_SET - T_AVG);
         OVEN_READY := (T_ERROR < 1) AND (T_DIFF < 2);
         TIMING.CONSTRAINT (REF_POINT => BEGIN_LOOP,
                            MAX_TIME  => 25E-6,
                            REF_LABEL => END_CALC);
         DISPLAY (T_AVG);
         PUT (STATUS_PORT, OVEN_READY);
         if T_SET > T_AVG then
            HEAT_STATUS := HEAT_OFF_CODE;
         else
            HEAT_STATUS := HEAT_ON_CODE;
         endif;
         TIMING.CONSTRAINT (REF_POINT => END_CALC,
                            MAX_TIME  => 20E-6,
                            REF_LABEL => END_DISPLAY);
         TIMING.CONSTRAINT (REF_POINT => BEGIN_LOOP,
                            MIN_TIME  => 10E-3,
                            MAX_TIME  => 11E-3,
                            REF_LABEL => END_CALC);
      end loop;
   end OVEN_CONTROL;
```

Figure 2–6: An Oven Controller

The type structure of Ada is used for typing of signals and memories. In this system it is extended, at least in theory, to include analogue signals represented by real types. If two real values are added an op-amp would be generated, and a real variable is implemented as a sample-and-hold. Single wire digital signals are represented as booleans, and ranges represent busses.

Various problem areas in the compilation process are highlighted. An obvious problem is that of dynamic creation of tasks: this cannot be done in hardware. Other problems are generally related to sharing. Ada allows sharing of global variables between tasks, so each shared variable must be created as a register and arbiter. Another, less obvious, problem is that of sharing procedures between tasks. In a machine-code implementation the code is sharable and it creates its own variables in task-local memory and so can be invoked an arbitrary number of times. This is not the case with a hardware implementation, where the procedure's locals are registers. To meet timing requirements it may not be possible to suspend one task until another task has finished with the hardware procedure: it must be replicated.

This is very significant work, having successfully compiled what is admitted a subset of Ada into hardware. Exception handling has been addressed, and hardware-error detection generated automatically. Using the type structure it is possible to specify both analogue and digital circuits in the same document. Areas of difficulty are identified, and where possible a solution has been found. The only drawback is that the primitive hardware elements are generated automatically by the compiler, and must be drawn from a standard library. If the user requires to add his own cells these can be specified but they must comply with the standard interfacing. It is not explained how the layout of user defined cells is captured by the design system.

2.14 Occam to CMOS

A team of researchers at Fujitsu have developed a compilation system that generates CMOS circuits from specifications in OCCAM [INMOS 84c], a concurrent programming language [Mano 84, Mano 85]. The actual input is a list based form of OCCAM called OCCAM-S, which can be read directly as Prolog [Clocksin 81]. For example, the following normal OCCAM code:

```
CHAN a, b:
VAR x:
PAR
  a ? x
  b ! 10
```

would be expressed in OCCAM-S as:

```
[[chan, a, b],
 [var, x],
 [par,
  [input, a, x],
  [output, b, 10]]].
```

The program is compiled into a state-machine description expressed in DDL. This is part behavioural and part structural, and the next stage in the operation is to separate out the structure as a data-path specification, and the behaviour as a control-automaton. The data-path specification is then compiled to logical-expressions and standard-cell instances, by a partially interactive process. For example, the synthesiser tries to estimate the word length needed for variables, and then confirms with the designer if it is correct. A heuristic optimiser compacts the control-automaton and generates logical-expressions for the control part as well. The next step in the design process converts the logical expressions to CMOS cells in a grid layout. These synthesis steps are all performed by part algorithmic, part expert-system Prolog programs [Maruyama 84].

For simulation, conventional compilation and execution is proposed. For this a simulated environment would have to be added to feed the inputs and display

outputs. There is no mention of how accurate a model of the timing of the final system this is, except a comment that they do not yet perform timing verification. In the OCCAM model of interprocess communication an input or an output is also a synchronisation point. If the OCCAM rule of no variable sharing is enforced the hardware can operate freely in each process and synchronise when communication occurs.

This system uses the SPL as a combined structural and behavioural specification: variables map directly onto registers, addition maps onto adders etc. The designer can specify whether sections of code are to be implemented in parallel or sequentially. An optimiser examines the sequential blocks for data dependencies, and tries to put as much in parallel as possible. The OCCAM IF statement is used to specify conditional behaviour; the multi-branch syntax for IF is more like a case-statement in other languages.

There is no mention of any problems with the design method. OCCAM does not allow the sharing of variables, but the procedure-sharing problem described in Section 2.13 is relevant and not mentioned. This project has been successful in compiling a simple OCCAM program, a pattern matcher, into random logic. There is no method of introducing user-defined special hardware: if an operation cannot be specified by the logic and arithmetic operations it cannot be done. For example, it would not be possible to define the high-current drivers of a display-driver with this system. Despite this and its simplicity, it does not handle busses correctly, this is an impressive piece of work, and would be well suited to providing a central controller on a chip, rather than a complete chip.

2.15 Summary

Having examined a number of notable HDLs and attempts to use SPLs for hardware description, it is now possible to contrast the approaches. The fundamental difference between hardware and software is that the hardware is fully concurrent, and this is reflected in the respective languages. Two sorts of concur-

rency can be identified: high level concurrency is where several large internally-sequential processes run in parallel; low level concurrency is where individual statements operate simultaneously.

In the HDLs the normal idiom is for statements to operate concurrently and sections where data dependencies require sequentiality have to be specified specially. In the SPLs statements are normally specified sequentially and special constructs are needed to indicate concurrency. Ada and Modula-2 have the concept of high level tasks or processes, whereas OCCAM allows blocks to be parallel or sequential with equal ease.

Timing is another area of difference between languages. This is particularly important for simulation and again highlights the significance of concurrency. Most of the HDLs have some concept of time built into them, for example VHDL's micro- and macro-time, or allow the designer to specify his own timing scheme easily, as in CIRCAL. A simulator then has to implement these schemes in a realistic way, and many such systems exist.

One of the mainstays of the SPL approach is that special simulators are not required: the programs need only to be compiled and run. This, however, must be viewed with caution. The University of Southampton system can only provide simulation using a sequential program by enforcing strict synchronous clocking and read-before-write use of memory elements. The other languages all provide processes, and inter-process communications can be used as synchronisation points. This is similar to self-timed hardware, where each component calculates its outputs and then waits for them to be read, allowing parts of the system to operate at different speeds.

It must be remembered that the chip produced by the compiler is not going to operate in a vacuum. When simulating a design it is also necessary to simulate its environment. When simulating with a concurrent SPL it is necessary to build a *harness* to enclose the design, supplying stimuli and recording outputs. Using the University of Utah system it is possible to run the package in situ, and they have built a test-bed in which it can be verified that the chip produced corresponds to the software. The Carleton University system uses input-output libraries to

represent file IO for simulation and external communication for synthesis. This allows a simulation to take place using data files for inputs, producing trace files as output. The CAP/DSDL system provides a language and compiler for defining simulation inputs, and an analyser for interpreting the outputs. The other systems do not make provision for these facilities.

In hardware description designers will often want to include specially designed components. This can be a problem for an SPL description, where the compiler operates with a known set of primitives. For example, it is not possible to use an RS232 terminal driver with the OCCAM-to-CMOS system, as there is no way to describe it. The Carleton University Ada compiler allows the user to include his own cells by defining procedures that simulate them.

The last comment to be made is that all these systems operate at a very low level. They are all concerned with near gate-level description, requiring the designer to work down from higher-level descriptions manually. Surely it is desirable to use a higher level of description, freeing the engineer from even more detail.

Chapter 3

The USHER System

They have been at a great feast of languages and stolen the scraps.

Love's Labours Lost

We have now examined a class of hardware systems, control computers, and a number of different language-based design automation systems. There is a clear mismatch: controller design typically operates at quite a high level, using substantial LSI building blocks and only resorting to gate-level design to glue these blocks together, while design tools are still concerned with low-level design, rising only as high as register-transfer-level. One reason why design tools are still operating at these levels is that the abstractions are well understood and have a sound basis. To proceed to higher levels we need to identify new abstractions.

To help with this task we will continue to use the thermostat example as it demonstrates the principles involved, in a small design. We will expand on the version introduced in Chapter 1 to provide a more concrete description:

> *Check the temperature, display it, and if it is less than we want turn on the heater, otherwise if the heater is on, turn it off; check if the user is typing in a new desired temperature, if yes read it and display it; then do it all again.*

From this we can derive the structure of the system as shown in Figure 3–1. Each box operates concurrently, and has a software and a hardware component. For the displays, for example, there might be some seven-segment LEDs and a display-driver, plus some software which converts integers to drive-patterns

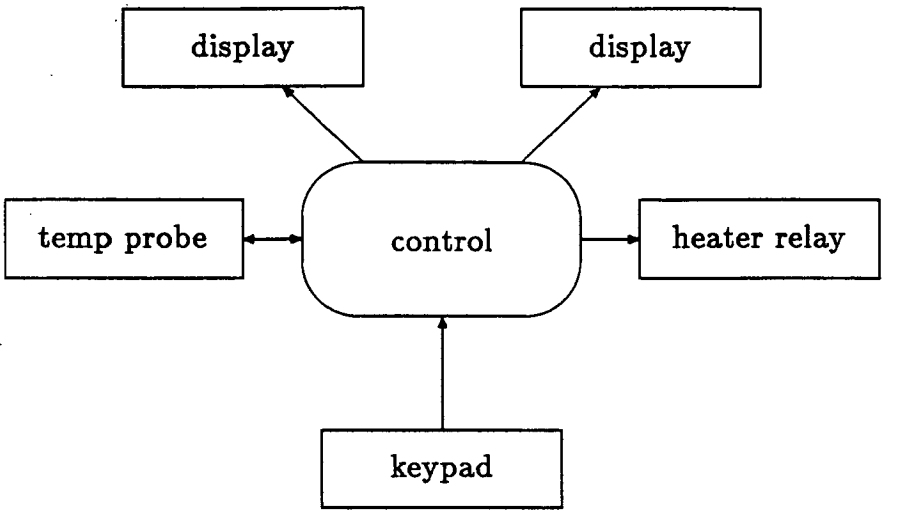


Figure 3–1: Structure of an electronic thermostat.

for the display and perhaps multiplexs the digits. Conventionally that software would be regarded as part of the control code, but it is logically part of the interface as it has to change when the hardware does.

An obvious model for the software is *concurrent processes*. The central process, which may itself contain a number of processes, communicates with the other processes when it needs them to operate. Encapsulating the hardware-driving code in processes builds firewalls round it, helping to remove technology dependency from the main code. This places considerable emphasis on the communications between processes and the importance of specifying good protocols.

Using the process model, we could describe the thermostat control program in some imaginary concurrent language as follows:

```
parbegin
   control (Probe, Heater, Keypad, Disp1, Disp2);
   temp.probe (Probe);
   heater.relay (Heater);
   display (Disp1);
   display (Disp2);
parend
```

where the parameters are some form of communication medium. This is a structural description of the system; it lists the objects which make it up and how

they are connected. The behaviour of the system is captured by the definition of each process. A circuit can be associated with each interface process, so that the activation of that process indicates that the circuit is required. In this way a concurrent program can also be a structural description of the hardware.

Concurrency has been a feature of two notable theoretical languages: CCS [Milner 80] and CSP [Hoare 78]. These were created to facilitate reasoning about concurrency rather than for implementation and real programming. However, recent practical programming languages make provision for process definitions, so it is to these that we turn when wishing to write control software.

3.1 Choosing a language

Traditionally the languages used by design automation systems were created new for each application. This do-it-yourself approach can result in languages which are appropriate to the task at hand, but rarely conform to the principles of good language design [Hilfinger 82]. It is also very difficult to define a language completely and accurately. There is already a vast number of languages, and many traditional designers find the prospect of learning yet another language daunting.

Having decided not to create a new language, a choice must be made from amongst the existing ones. In the following sections the three major languages providing concurrency facilities are analysed. The languages are examined on three points: process definition and creation, interprocess communication, and ease of implementation. This last criterion is included as I did not wish to expend a substantial amount of effort and time simply on language implementation.

3.1.1 Ada

Ada [US DoD 80] programs can be broken down into three types of subprogram unit: procedures, packages and tasks. Tasks are intended to be run concurrently,

```
task PROTECTED_ARRAY is
   entry READ  (N : in INDEX; V : out ELEM);
   entry WRITE (N : in INDEX; V : out ELEM);
end;
```

(a) The task specification

```
task body PROTECTED_ARRAY is
   TABLE : array (INDEX) of ELEM := (INDEX => 0);
begin
   loop
      select
         accept READ (N : in INDEX; V : out ELEM) do
            V := TABLE (N);
         end READ;
      or
         accept WRITE (N : IN INDEX; V : in ELEM) do
            TABLE (N) := V;
         end WRITE;
      end select;
   end loop;
end PROTECTED_ARRAY;
```

(b) The task body

Figure 3–2: An example Ada task

either time-sliced on a single processor or on multi-processors. Tasks provide *entries* which can be *called* by other tasks and are then *accepted.* This is called a *rendezvous*, and provides synchronisation as well as communication between tasks.

Like all Ada declarations there is a specification declaration, as in Figure 3–2(a), and the body declaration Figure 3–2(b). Entries are parameterised and are called like procedures. By including the word type in the specification declaration a task type is created and many processes can be declared to be of this type, including arrays. Tasks are regarded as constant values as they cannot be assigned, but it is possible to have pointers to tasks with access variables. These can be used to manipulate process identities, and with the operator new can create processes dynamically. For hardware description this dynamic creation of processes would have to be restrained, as does general recursion. Entry accep-

tance can be made conditional or timed with the `select` statement, as shown in Figure 3–3.

As described in Chapter 2, Ada is the main inspiration behind VHDL. A very interesting idea for combining the programming style of Ada and the hardware-description facilities of VHDL is the matching of task entries with ports in architecture bodies. By adding a handshake line, visible only to the VHDL end, and adopting a self-timed discipline, a software task could communicate with the hardware in a well-defined manner. An Ada task body is the obvious upper limit of a family of definitions of architecture bodies, and could provide a high level simulation facility if used appropriately.

There will always be some level of hardware that cannot be described at a high-level. It is desirable to provide a framework in which this can be described at an appropriate level, but still form part of the whole design. A language providing this framework would be a true heterosystems description language, and would provide an ideal environment for *vertical migration.* This is the process of selectively lowering the implementation level of critical sections of a design in order to improve overall performance.

Unfortunately Ada presents a major problem to a single researcher: its size. It is an enormous language, with many complex features. Subsets are regarded in a poor light, and are still difficult to build since the construction of the standard Ada environment requires the more esoteric features. For example the standard input and output libraries are defined using generic packages and overloading. Avoiding these would require programs to deviate substantially from the standard.

Ada is probably the ideal language for use in describing hardware-integrated code, and has been used widely with hardware design systems. The type and value representation structures are very extensive and allow the programmer to match any hardware data format. The rendezvous mechanism provides an appropriate model of self-timed communication between devices, and the language already provides facilities for mapping interrupts onto entries. However

```
task body TIMED_RESOURCE is
   BUSY : BOOLEAN := FALSE;
begin
   loop
      select
         when not BUSY =>
            accept SEIZE do
               BUSY := TRUE;
            end;
      or
         accept RELEASE do
            BUSY := FALSE;
         end;
      or
         delay 100.0;        -- time out requests
         BUSY := FALSE;
      end select;
   end loop;
end TIMED_RESOURCE;
```

Figure 3–3: A timed, conditional select statement

the huge amount of work required to implement an Ada system prevented it being a practical choice for this work.

3.1.2 Modula-2

Modula [Wirth 82] adopts the approach of implementing concurrency via system defined procedures, rather than language extension. A process is an independently activated procedure, created by a call to the following:

```
PROCEDURE StartProcess (P : PROC; N : CARDINAL);
```

The first parameter is the procedure, and the second the size of workspace it requires. The presence of this parameter betrays the lack of success of this style of concurrency integration: storage allocation is definitely a compiler task, and ought not to be left to the programmer.

Interprocess communication is via shared-variables, with synchronisation by variables called SIGNALs. These are semaphores and the two operations defined on them, SEND and WAIT, are directly equivalent to the P and V operations

```
CHAN To.Probe, From.Probe, Heater, Keypad, Disp1, Disp2:
PAR
  control (To.Probe, From.Probe, Heater, Keypad, Disp1, Disp2)
  temp.probe (To.Probe, From.Probe)
  heater.relay (Heater)
  display (Disp1)
  display (Disp2)
```

Figure 3–4: The thermostat main code in OCCAM

on semaphores. The Modula manual describes the use of these operations to implement monitors to control access to global variables, and this is obviously the normal method for interprocess communication.

The use of what are in effect operating-system calls to implement concurrency, rather than language features does not place sufficient emphasis on the design of protocols. It can become difficult for the compiler to identify what *is* communication code from the rest of the program. For this reason Modula-2 was discarded.

3.1.3 Occam

OCCAM [INMOS 84c] is derived from two schools of language design: BCPL [Richards 79] and CSP [Hoare 78]. It draws a minimalist approach from BCPL, with the only data-types available being machine words and bytes, and the only structuring facility the array. From CSP it takes the guarded *channel* for communication between processes, which in turn is based on Dijkstra's guarded commands [Dijkstra 75]. Every language statement is called a process, divided into two kinds: *primitives*, including assignment, procedure calls and channel input and output; and *constructors*.

Constructor processes group primitives or other constructs. Figure 3–4 shows the PAR construct and procedure calls used to implement the thermostat example. Two operations are defined on the CHAN type variables: input (?) and output (!). These provide communication and synchronisation between processes; as

```
CHAN seize, release:

  :

VAR busy, T:
SEQ
  busy := FALSE
  WHILE TRUE
    SEQ
      TIME ? T
      ALT
        NOT busy & seize ? ANY
          busy := TRUE
        release ? ANY
          busy := FALSE
        TIME ? AFTER T + 100
          busy := FALSE    -- time out
```

Figure 3–5: An example of the ALT construct

channels are not buffered, both the input and output operations must be ready for the transfer to take place. This is a very close model to the one we have developed to describe the control systems. The ALT constructor allows conditional and timed communications; Figure 3–5 shows an OCCAM version of the Ada task given in Figure 3–3. The ALT construct provides more flexibility than the Ada `select` statement, as it can be replicated and contain replicated portions.

Recursion is not allowed in OCCAM, nor are any dynamic structures such as run-time sized arrays of processes, so that the storage requirements of a program can be established at compile time. This is very helpful for describing hardware, where resources cannot be claimed dynamically, and has recommended it as the input language for at least one design automation system. Further evidence that OCCAM is suitable for description is that it has been used as a concrete syntax for the Jackson System Development Method [Featner 85] and also for general systems description [King-Smith 86]. The simplicity of the language means that it is suitable for theoretical analysis, and a set of laws describing OCCAM constructs have been developed at Oxford University [Roscoe 86], which is the first step towards program verification. Once this is possible, the same method can be used to prove hardware designs specified in OCCAM correct.

Since OCCAM is such as small language, without complex features, it is relatively easy to implement. In fact a self-compiling portable compiler is available from INMOS [INMOS 84b], but this proved too slow to be practical.

For all the foregoing reasons I decided to use OCCAM as the input language for the USHER system. Language design is frequently more religious than scientific, and the designers of OCCAM probably shaved off rather more than was prudent, as shown by the development of OCCAM 2 [May 86], so the language implemented for the USHER system is a superset of the OCCAM Programming System[1] [INMOS 85]. It provides all the vector operations, and as will be seen in the next sections, various extensions directly required for the USHER system. A complete grammar for the extended language appears in Appendix A.

3.2 Using Occam

Having decided that OCCAM is an apt language for describing control systems, it is necessary to examine in detail how it can be used for this task. Figure 3–4 shows an idealised description of the thermostat example, in it there are five parallel activities: the control software, the temperature probe, the heater switch and two displays. The control software is all that we want to describe in detail for the application, as it is the behaviour of the system. This can easily be done by using a procedure defined within the program. The information about the interfaces is an adequate but terse structural description of the system. How can we come close to this very high-level description in practice?

Ideally the designer should be able to browse through a data-book of interfaces and choose the ones most appropriate to his needs. In order to do so he

---

[1]This is an INMOS product, and is a common environment for the development of OCCAM programs on different machines. It provides a primitive text editor with integrated syntax checker, and a machine dependent code generator. The language it accepts can be regarded as the *de facto* standard.

should not need to know how the devices are implemented. This corresponds closely to the conventional programming model of subroutine libraries. So interfaces can also be modelled as imported procedures. This requires the designer to know and include in the applications program only the minimum information: the name of the interface and the source library. This allows us to approach the ideal description, and in practice the thermostat program main code is almost identical to that given above.

Most interfaces will need some *configuration* information. For example, how many digits there are in a display, or how many switches are needed. These data can be passed as parameters to the interface procedures, and the compiler can identify them as affecting the hardware.

One of the objectives of the USHER system is to be able to simulate and synthesise designs in various technologies, without changing the source code. This requires that the interface procedures can adapt to different circumstances. The effect to be achieved is that there is a version of each interface for simulation and then one for each technology. Obviously each version must present an identical interface to the application.

OCCAM cannot be used directly as described here since it lacks separate compilation. How this and other features have been added is described in Section 3.4. For readers unfamiliar with OCCAM the next section provides a brief introduction to aid the understanding of later examples. Readers who already understand the language should skip directly to page 46.

3.3 Instant Occam

The best introductory text book on OCCAM is "Programming in 'OCCAM': a tourist guide to parallel programming" [Jones 85]. To avoid the need for finding a copy of this book, however, this section outlines the features of the language in sufficient detail to understand the examples used in the rest of this thesis.

One of the novel features of OCCAM is that it is *indentation sensitive.* This means that a new scope is entered by indenting code by an additional two spaces after a constructor:

```
VAR a, b:
SEQ
  -- This is nesting level 1
  VAR c:
  SEQ
    -- This is nesting level 2
    a := c
  b := a
  -- Thus is level 1 again, c is out of scope
```

Statements are terminated at the end of the line, although a line break after a comma is ignored, and only one statement per line is possible. Comments are introduced by a double-hyphen "--", and terminate at the end of the line. The case of letters is significant, keywords must appear in UPPER CASE and identifiers must always be used exactly as they were declared.

This example also introduces the SEQ construct, in which all the statements are executed sequentially, like a normal *begin...end* block. It can be converted into a *for*-loop by adding a *replicator*:

```
SEQ i = [0 FOR 10]
  :
```

The loop counter i is created, it need not be declared and any other variable of the same name is out of scope. The loop *executes* ten times, ten is *not* the last value of the counter; in fact i will go through the range zero to nine. The only other cyclic constructor is WHILE. Sequential conditions, both of the *if* and *case* variety, are implemented by the IF construct:

```
IF
  bool
    Screen ! 'T'
  n > (3*q)
    Count := Count + 1
  TRUE
    SKIP
```

Else clauses are introduced by a TRUE condition. At least one condition must be valid, or the construct is equivalent to the STOP statement, which suspends the current process indefinitely[2]. For this reason most IF statements end with TRUE...SKIP.

Declarations precede blocks and are followed by a colon:

```
VAR s, a [10], line [BYTE 80]:
DEF ESC = 27, Limit = 25:
DEF Prompt = "(Y or N) ? ":
CHAN Transfer, Lights[Limit]:
```

The first declaration introduces a host-word-size variable, an array of ten words in which the index range is zero to nine, and an array of eighty bytes, index zero to seventy-nine. The next two declarations define two constants and a constant array. Strings are arrays of bytes, with the first byte the length of the string. The last declaration is of a single channel and an array of channels.

Arrays can be indexed as either words or bytes, determined by the presence of the keyword BYTE before the index, but are only one dimensional. Sections of arrays, or *slices* can be operated on at once, for example:

```
VAR v [10], a [20]:
SEQ
  :
  v [BYTE 0 FOR n] := a [BYTE 26 FOR n]
  :
  a [1 FOR 5] := TABLE [13, 27, 19, 126, 99]
  :
```

Slices can be used in assignment as shown here, or in channel operations or passed as array type parameters to procedures.

Procedure declarations look like this:

```
PROC id (VALUE p1, VAR p2, CHAN c []) =
  SEQ
    c [0] ! p1
    c [1] ? p2:
```

---

[2]This feature was introduced with the OCCAM Programming System and does not appear in the original manual.

Note that the last statement of the procedure ends with a colon, which makes the declaration conform to the

Declaration → Type *Identifier* "=" Value ":";

syntax of the other declarations. VALUE parameters appear like constants within the procedure; they cannot be assigned new values. Array parameters do not have bounds, and indexing is unchecked.

This brief description covers everything except the concurrent facilities of the language. The PAR construct is used to execute its component processes in parallel. There is no concept of a process-block: any statement can be executed as a concurrent process. Figure 3–4 on page 39 demonstrates this construct. Arrays of processes can be created with a replicated PAR, but the number must be a compile-time constant.

Processes communicate via *channels*, which are single-direction and can link only two processes. Communication only occurs when both parties are ready. Any value can be sent via channels, they are not typed or checked, so the programmer must ensure that processes use the same protocol. The special value ANY can be used for both input and output when only the synchronisation effect is important.

The ALT construct provides a choice between inputs, whichever happens first activates its associated process; Figure 3–5 on page 40 shows an ALT. Each process *guard* can either be an input, the always-ready SKIP, or a boolean expression plus either an input or a SKIP. If the condition is false, that guard is not open. It is one of the deficiencies of the language that output guards are not permitted. The special channel TIME can be used as a guard to specify a time-out on a communication. It can also be used outside of ALT guards to enquire the current time, and to specify a delay with the AFTER operator.

Channels are used to implement file input and output on conventional computers. The OCCAM Programming System provides channels Screen, Keyboard, and Infile*n* and Outfile*n*. Files are opened and closed by outputting special

values to these channels. These are the only facilities provided by INMOS, even a procedure for outputting numbers must be written by the user. The USHER OCCAM system corrects this, providing a number of output and utility procedures, and these are used in some of the examples. Also included with USHER is the special channel `Terminal` which is a terminal-independent screen driving package. This provides facilities for moving the cursor and clearing sections of the display, which is useful for simulating display devices.

3.4 Extending Occam

Modifying programming languages is not sensible unless it is absolutely vital. It has the effect of decreasing portability or requires the programmer to work with a subset, so what starts as an extension actually reduces the available facilities. The first rule of expanding a language is "don't". The second rule is, if you must do it, try and match the original language as closely as possible.

3.4.1 Defining procedure libraries

Bearing in mind these maxims, we approach OCCAM requiring to define libraries of procedures and refer to them from other programs. At present the language makes no provision for this facility. The OCCAM Programming System supports what is called separate-compilation, but this is really partial pre-compilation. Within the Programming System procedures can be held in *folds* and it is possible to compile individual folds, which are then bound into one object when the outermost fold is compiled. This presents a number of source-code-control problems, such as what happens if prior declarations change, and does not provide any way of sharing procedures between programs.

The Ada package is a very elegant method of defining a library of subroutines. It allows the programmer to hide or make available as much or as little as is wished. Unfortunately it is at odds with the OCCAM syntax and philosophy.

Modula-2 uses *export* and *import* lists to display or aquire routines from its modules. This is still a clean method and less syntactically demanding. Most Pascal compilers now allow the inclusion of externally defined procedures by replacing the code body of a procedure declaration with the keyword EXTERN. This greatly reduces the violence done to the syntax but is not very readable, and has already been used with OCCAM [Stallard 85].

An infusion of these schemes, kept within OCCAM's declarative syntax has resulted in the following organisation. The keyword PROC is replaced with IMPORT or EXPORT as required. For an import, the code body is replaced with a FROM statement identifying the source library. An example of an imported routine is

```
IMPORT Write (CHAN To, VALUE Number, Places) =
  FROM STANDARD:
```

which shows one of the output routines provided in my OCCAM system in the library STANDARD. Library files are headed by a LIBRARY declaration, so the export definition of the previous example would be

```
LIBRARY STANDARD:

  :

EXPORT Write (CHAN On, VALUE Number, places) =
  VAR s [BYTE Buffer.Size]:
  SEQ
    IToS (Number, Places, s)
    PrintString (On, s):
```

Library files may declare global variables to compensate for the lack of *own* data. Since declarations cannot include initialisation, libraries can contain main-code bodies. These are defined to be executed before the main-code of the final program, but if more than one library is in use the order of execution is not. It is dangerous for one library initialisation code to call procedures from another library as it may not have been initialised. Exported procedures can call other exports or procedures imported from other libraries.

One possible reason why original OCCAM does not include external linkage is to allow the compiler to enforce the no-recursion rule. It is not practical to

check for mutually cross-calling procedures in different libraries at compile time. The programmer is required to keep a strict discipline and check that this does not occur; however this is hardly an arduous task and is far outweighed by the benefits of having libraries.

3.4.2 Defining interface libraries

We can now define libraries of procedures, but what is really required is the definition of libraries of interfaces. Section 3.2 identified three requirements for interfaces: external definition, configuration parameters and technology adaptation. It would be possible to provide all these functions using the basic facilities described above. The identifiers of EXPORT procedures that are interfaces could all start with "IF.", marking them as such, and the configuration parameters could all start "CONFIG.", and there could be a different library for each technology. These are all inelegant solutions, and since what they are adapting is not part of the standard language it is preferable to extend the extension instead.

For both the export and import cases the keyword PROC is replaced with INTERFACE. This does not allow interfaces to invoke other interfaces, but as will be seen later this is not possible anyway. Configuration parameters are defined with the keyword CONFIG, and appear as value parameters, except that the value passed must be a compile time constant. An import of an interface appears like:

```
INTERFACE seven.segment (CONFIG digits, initial.value,
                         CHAN data) =
  FROM display.drivers:
```

The syntax for interface library definitions is the same as ordinary libraries, in fact they can contain ordinary export procedures. The difference arises in the definition of the code body. Each interface must have a simulation body, and a body for each synthesis technology. The closing colon of the procedure comes at the end of the last body; the end of the other bodies is detected by the indentation level. A definition of the above interface would be of the following form:

```
LIBRARY display.drivers:

  :

INTERFACE seven.segment (CONFIG digits, initial.value,
                         CHAN data) =
  SIMUL
    :
  SYNTH Z80, six.seven
    :
  SYNTH CMOS.1, big.drive.SS
    :
```

The keyword SIMUL introduces the simulation body, then each synthesis body is headed with SYNTH. After this keyword are the *technology-identifier* and the *hardware-identifier*. The technology-identifier is used during linking the applications program to select the appropriate code body. The hardware-identifier associates a circuit with the procedure and is used by the hardware allocator. Several interface procedures can use the same hardware, presenting different appearances to the control program. For example a seven-segment display driver could be used to display time, decimal numbers without leading zeroes, or hexadecimal numbers. The hardware- and technology-identifiers exist in separate name spaces, and do not clash with program identifiers.

3.4.3 Within a SIMUL body

A simulation body runs in the normal OCCAM environment, so is free to use the special input-output channels. These can be used for reading drive-files of precomputed data or writing trace-files to allow later analysis. The channel Screen has the special property that it can be used by more that one process at once. The simulator must manage this, providing the applications program and each interface with a screen "window". The Keyboard channel cannot be similarly shared.

3.4.4 Within a SYNTH body

It is intended that in a highly integrated synthesis system the hardware would communicate directly with the control program via the channel parameters. For example the IMS T424 transputer [INMOS 84a] has two interrupt signals which map onto OCCAM channels. In such a technology the synthesis body can simply be the OCCAM no-operation statement SKIP. In a conventional technology, however, the synthesis body must drive the hardware of the interface. Since the hardware is allocated automatically, the interface can make no assumptions about device addresses, so this must be packaged by the compiler. Two special channels are provided to support this: HARD and EVENT.

HARD is a dynamically sized array of channels that can be used for both input and output. It is used for reading and writing to device registers. Figure 3–6 shows the actual code for driving a Z80 parallel input-output circuit (PIO) as a bargraph driver. The indexing of HARD varies according to the implementation. Technologies based on 8-bit microprocessors, for example, use byte indexing irrespective of the presence or absence of the BYTE keyword. In these systems the unit of data transfer will also be a byte.

The special channel EVENT is used for trapping interrupts. The term "event" is derived from the transputer implementation of OCCAM. It can only be used for input, and some circuits will present the data that caused them to interrupt at that input. Others will only perform the equivalent of an ANY-output to the channel. An example of the former kind appears in Figure 3–7; this is the real code for an analogue-to-digital converter. Channel communication is normally synchronised, but conventional peripheral devices do not operate in this manner, and if a response is not forthcoming data can be lost. To avoid this happening the EVENT channel is always buffered by at least one item. If an interrupt occurs and the relevant process is not waiting on EVENT, the interrupt is answered and the data stored until the next input on EVENT which does not involve any delay. Normally the buffer is only one unit long, so interfaces should also have a process

```
DEF A.Data    = 0,          -- device register addresses
    A.Control = 1,
    B.Data    = 2,
    B.Control = 3:

DEF Control.Mode   = #CF,  -- control codes for the PIO
    Interrupts.Off = #07:

INTERFACE Bar.Graph (CONFIG Bits, CHAN Show) =
  SYNTH Z80, Bar.Graph
    -- set up PIO as desired
    HARD [A.Control] ! Interrupts.Off
    HARD [A.Control] ! Control.Mode
    HARD [A.Control] ! 0                -- All outputs
    HARD [B.Control] ! Interrupts.Off
    HARD [B.Control] ! Control.Mode
    HARD [B.Control] ! 0                -- All outputs
    -- now send it some initial data
    HARD [A.Data] ! 0
    HARD [B.Data] ! 0
    WHILE TRUE
      VAR Data:
      SEQ
        Show ? Data
        HARD [A.Data] ! Data/\#00FF
        HARD [B.Data] ! Data>>8:
```

Figure 3–6: The channel HARD in use

```
INTERFACE Int.ADC (CHAN Request, Data) =
  SYNTH Z80, ADC
    WHILE TRUE
      VAR D:
      SEQ
        Request ? ANY       -- wait to be asked to do something
        HARD [0] ! ANY      -- set up conversion
        EVENT ? D           -- read result on interrupt
        Data ! D:
```

Figure 3–7: The channel EVENT in use

waiting for an event. It is possible for interface procedures to contain nested PAR constructs, with one component always waiting for data from the interface.

Neither HARD nor EVENT can be passed as parameters. As synthesis bodies will be running on minimum-configuration machines, none of the normal input-output channels are available. This is also true of applications programs when used for synthesis, but for development purposes with the simulator, diagnostic information can be generated through these channels.

3.4.5 Instantiating Interfaces

The action of creating an interface in an application program is called *instantiation*, and is distinct from the declaration of the interface-procedure. Two things happen at instantiation, firstly the procedure is activated as a process, and secondly the relevant hardware, as defined by the hardware-identifier, is marked for inclusion during synthesis. Syntactically interface instantiation is identical to a procedure call, but it can only occur as an element of a PAR or replicated PAR construct with a unique path to the top level.

As each interface is instantiated it is allocated an address, which is stored as part of the information in the enclosing PAR construct. Obviously there must be a one-to-one mapping between addresses and interfaces, so instantiation must be in a PAR that is activated only once. This is an aspect of the code-sharing problem, introduced in Chapter 2. An interface, which is a unique, unshareable object, cannot exist in a control sequence that is common to several processes. For example, the following code for a chart-recorder is illegal:

```
CHAN data [channels]:
PAR i = [0 For channels]
  PAR
    data.probe (data [i])
    pen.driver (data [i])
```

This is not valid because the simple PAR is used to store the addresses of the interfaces, and is required here to hold them for all the devices of the replication. By inverting the nesting, however, the above example can be described:

```
CHAN data [channels]:
PAR
  PAR i = [0 For channels]
    data.probe (data [i])
  PAR i = [0 For channels]
    pen.driver (data [i])
```

Interfaces can only be instantiated within a procedure if that procedure is only called once. This is because the address information is stored in the PAR within the procedure, not at the higher level. The same reasons prevent interfaces invoking other interfaces. The compiler checks that these rules are not violated.

3.5 Software Structure

Having established the aims of the system and defined the input language, an implementation stragtegy must be devised. The end product will be a simulator and a framework for building synthesisers, with a prototype to demonstrate the validity of the approach.

Implementation is constrained by available computing resources. The possibilities were a heavily overloaded VAX 11/780 running VMS or the Departmental Advanced Personal Machine (APM) with very little software but providing adequate performance. In light of the response times the APM was chosen as the most suitable computing engine, and this influences other decisions. All the conventional programs are written in Imp77 [Robertson 80], which is available on all the principle available computing services, so does not restrict the system to the APM.

The simulator and each synthesiser will both accept the same applications program and interface libraries, so could use a common parser. Each tool will require a different code body from the libraries, but a single parameterised linker-selector could be used by all of them. This suggests the development of a common front-end and intermediate-code [Robertson 81].

To assist customers in developing OCCAM systems INMOS distribute a portable compilation system called the "portakit". This compiles full OCCAM into a byte-stream code called the Portakit Instruction Set (PIS *sic*). This compiler is itself written in OCCAM and is distributed in source and PIS form, with interpreters in several languages. This was evaluated as a possible basis for the front-end and intermediate-code for USHER. The first step was to mount a PIS interpreter on the APM[3] and this demonstrated that the compiler was much too slow to be practical. For example a three line test program to write the printable ASCII characters to the terminal took 4 minutes 25 seconds to compile. In addition the compiler source was 15,000 lines long and was not amenable to modification, and no run-time checking or diagnostic information was generated.

This left the field open for developing a completely new system, the overall structure of which is shown in Figure 3–8. An OCCAM compiler, OC, generates an OCCAM Intermediate-Code (OIC) module, which can then be linked with other OIC modules and then either executed directly or used as input by the synthesis tools. Each synthesiser consists of three parts, an *allocator*, a code-generator, and an *Artificial Engineer*. The allocator identifies interface instantiations and allocates them addresses and interrupt vectors, outputting a *hardware-requirements list*. The code-generator produces native code to run on the appropriate processor, and possibly an architecture specification for customised processors. The *Artificial Engineer* assembles the required hardware from the hardware-requirements list.

This is better explained by outlining the steps involved in the creation of a design. From the specification of the product, the engineer identifies which interfaces are required, and matches them with ones available from the USHER libraries. Assuming that they are all available, the engineer writes the control program, and runs it on the simulator until it works as desired. He then compiles it with the appropriate back-end, which produces a code file and a hardware-

---

[3]Thanks are due to Dr David May of INMOS Ltd. for supplying the portakit, and to Iain Baird for mounting it as part of his final year project.

requirements list. The *Artificial Engineer* is then applied to the hardware-requirements list and produces the fabrication details. The human engineer is then responsible for the manufacture of the hardware.

If the design under development requires interfaces which are not already available, then these must be created ad installed in the USHER library. The interface designer first defines the specification of the interface, and its OCCAM procedure-header. He must then design, construct and test the hardware, SYNTH and SIMUL code-bodies for the interface. The OCCAM definitions of the interface are installed in the USHER library using standard tools from the OCCAM system. The hardware-design knowledge must then be captured for the *Artificial Engineer.*

OC is documented separately [Marshall 86], but is described briefly in the next section. Chapter 4 describes the simulator, Chapter 5 covers the prototype Z80 allocator and code-generator, while Chapter 6 documents the *Artificial Engineer.* Table 3–1 shows the size in lines of each of the parts of the USHER system.

3.6 An Occam front-end

The most common form of intermediate code is the *byte-stream.* This is like the order code of a real computer, but is defined on an imaginary architecture suited to the execution of the language. The best know example of this is P-Code, used in the UCSD Pascal System, which has been successfully mounted on many machines. The portakit instruction set is another example.

These codes present two problems, the first is that the compiler has to package up information which is then unpacked by the interpreter or code generator. In this process a considerable amount of information can be lost, leading to the second problem: diagnostics. For the USHER system the OCCAM is to be executed for simulation and diagnostic purposes, so code and variable tracing is very important. When an error occurs within a PIS program, for example, the

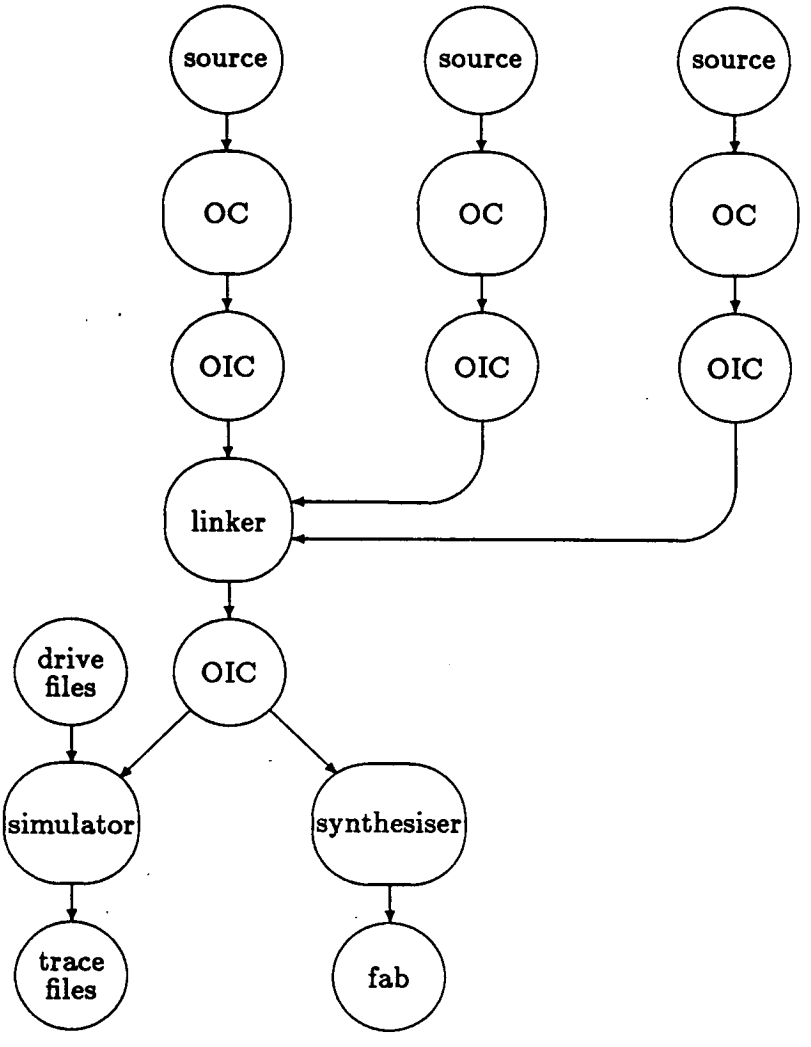


Figure 3–8: The structure of the USHER system

Program	Language	Modules	Size
Grammar	APG	2	530
Front End	Imp77	21	7272
Simulator	Imp77	13	4527
Run-Time Library	OCCAM	1	250
Z80 Code Generation	Imp77	7	4828
Z80 Allocator	Imp77	3	488
Z80 Run-Time Support	Z80 Assembler	1	719
Z80 Executive	Z80 Assembler	1	780
AE Shell	Prolog	4	99
AE Designers	Prolog	5	740
Z80 Processor Base	ESDL	1	74
		Total	20307

Table 3–1: The size of USHER

only information presented to the programmer is the PIS code address and the address of the workspace of the failing process, or even worse simply the message "deadlock". This is totally unhelpful, and is typical of the form of report that byte-stream based systems can produce.

When presented with the same problem, the definition of an intermediate code, for the VLSI design-language SCALE [Marshall 83] I developed an alternative type of code, a high-level homogeneous data-structure, and this was used as the basis for OIC. Essentially OIC is an executable, abstract syntax-tree, which combines the understandability of the source language with the run-time efficiency of the byte-stream code. To understand this, consider the representation of a variable in each form.

The programmer associates an identifier with the variable, and uses that to refer to it in the source program. It is in these terms that diagnostic information should be presented. Within the byte-stream code variables are identified by an address, probably an offset within a stack-frame. To present the variables in

diagnostics a separate table of source code tie-ins is needed and then the stack must be unwound. In the data-structure based scheme a record is associated with each program object, and this contains both the identifier and the stack offset. Instructions refer to the variable by using a pointer to the record, providing the efficiency of byte-stream, and allowing diagnostic information to be generated on-the-fly and for stack dumps.

There are, of course, disadvantages to the data-structure code. The most obvious is that the size of the structure is very large, at times larger than the original source file. This is only a severe drawback if the compiler is to be shoe-horned into a small computer. Another is that it is slightly slower than byte-stream as an additional dereference is required to access the stack offset. There is also the well known problem of saving a heap-based data-structure in a file and then having to restore it at a different address. This, however, is simply an implementation problem and has been solved for OC.

So far only the advantages for interpretation have been identified. OIC is also well suited to code generation. Typically intermediate codes have some fixed architecture in mind: P-Code and PIS are stack based, Z-Code for Algol68 assumes the presence of eight registers. In OIC, expressions are stored in tree structures, and can easily be mapped onto either style of architecture. OIC is also word length independent, and can work with both 32-bit and 16-bit words, although modules of different word length cannot be mixed.

Two kinds of parsing technology are currently in vogue: programmed recursive-descent and grammar generated. In the recursive-descent style the grammar is built into the compiler and modifying the accepted language can be a substantial task. Parsers that are built automatically from the grammar they are to accept allow changes to syntax to be made very quickly. Since this project involves language development the grammar-generated scheme was adopted.

The local utility APG [McCaskill 85] was used to generate the parser in Imp77. This tool reads a lexical definition file, containing definitions of the keywords and tokens of the language, and a grammar. Actions can be associated with each production, so that when a phrase is recognised a procedure is called.

The OIC file is generated by this mechanism. As well as the obvious checks for syntactic errors, extensive semantic checking is performed.

OIC structures can be stored in files, by converting the internal pointers to block number and offset tuples. This is always the case when library files are compiled, but programs are normally parsed and executed or compiled to code immediately from the in-store OIC. OC includes a linker and dictionary mechanism. When a library is compiled an entry is made in a dictionary, which is then used to check that import declarations are correct. The dictionary also contains the name of the OIC file for the library, so programs are linked automatically. Utilities are provided for accessing other user's dictionaries. After linkage a parameterised routine selects the code bodies from the libraries according to the appropriate technology-identifier.

A version of OC and the USHER simulator without the interface extensions has been successfully used to teach OCCAM in an undergraduate class. It can also run the test programs distributed with the portakit, providing evidence that OC is a sound base on which to build.

Chapter 4

The Simulator

Simulate:
1. *to give a false appearance of; feign*
2. *to look or act like*

Webster's New World Dictionary

In the traditional hardware-first design style for control-computers, the software is developed using an *in-circuit emulator*. This is a device which plugs into the board where the microprocessor would normally be mounted, performing exactly as the microprocessor would as far as the hardware is concerned. It runs the software and the hardware is exercised: data is captured, displays display. However, via the emulator, the programmer can see what is happening inside the system. He can set traps, examine the contents of registers and memory, and even specify assertions which are continuously checked.

If the controller is implemented in silicon this technique is physically not possible, and if the software is being developed first it is contrary to the design method. In both cases it is not the control processor that must be emulated, but the surrounding interface circuitry and its environment. The programmer, however, must still be able to examine the internal state of the software. This chapter describes the design, implementation and use of such a tool.

Chapter 3 defines an extension of OCCAM that allows the interface designer to specify a simulation body with each interface. This code emulates both the interface and its environment. In this way the control program can be linked with the simulation bodies of the interfaces and executed conventionally, with normal program debugging tools used for internal examination.

With a programming language two kinds of execution are possible: compiled and interpreted. Compilation is undoubtedly faster for execution, but has a longer set up time as the intermediate code must be translated and then the new machine-code loaded, linked and run. Debugging tools must work at a very low level, and the implementation of watch-points can slow execution considerably. Interpretation's main advantage is that it is machine independent, and all the source-code information is available for generating diagnostics. During program development the slower execution is offset against the compilation time.

An interpreter was chosen for this research for several reasons. The first was that it does not tie the simulator to the APM. The next was that the APM did not provide any existing code-generation or machine-level debugging tools that could have been adapted. OIC is equally suited to compilation or execution, but diagnostic information can be accessed immediately from an interpreter, whereas it would have had to be packaged for use with a native-code module.

The first sections of this chapter describe how the interpreter part of the system was implemented: Section 4.1 outlines the store allocation strategy, Section 4.2 describes how the concurrency operations are implemented. Section 4.3 shows how the basic OCCAM interpreter was extended to support simulation, and examines some example SIMUL bodies, while Section 4.4 looks at the diagnostic tools which are provided. Finally, Section 4.5 analyses the simulator to see how reliable it is: whether it corresponds to definition 1 or 2 at the start of the chapter.

4.1 Storage Allocation

OCCAM has been defined such that it is possible to work out the store requirements of a program at compile time. Since this calculation is back-end specific, OC provides a generic procedure for performing it. This executes a depth-first traversal of the OIC tree, filling in two fields of each OIC instruction record. One field is the store requirement for any variables local to that instruction,

and the other is the total store needed for the locals and any nested code. For PAR constructs the total requirement is the sum of all the requirements of its components, but for the other constructs it is the maximum requirement that is used. Procedure calls have a requirement equal to that of the code of the called procedure.

Any process, including primitives, can have local variables, so that the following code is valid, although pointless:

```
VAR temp:
temp := 5*q
```

with the variable temp existing only for the duration of the assignment. This form of declaration does have a use for declaring variables to be passed as dummy parameters to a procedure. To avoid having to allocate store for primitives OC moves these declarations out to the enclosing construct, into an area at the end of the normal variables. This is shared by any other temporary declarations and is the size of the largest object.

An OCCAM program is a tree of processes, with the outermost level also a process, and sub-processes created with the PAR construct. The store required by a sub-process is part of the store area of the enclosing process. The store area used by a process is called its *workspace*. Since the interpreter does not rely on the host architecture to execute the program it is free to implement workspaces in a manner appropriate to the language; in fact this interpreter uses multiple stacks.

All program data exists in a single data-stack, which is the workspace of the global process. This stack does not contain any control-flow information, which is kept in a stack per process. Each process is represented by a *process control block* (PCB). These are created off the heap and are linked in a tree from the global process. A *workspace stack* is attached to each PCB, implemented as a linked list. When a new sequential construct is started, or a procedure is called a new workspace record is pushed onto this list. Each record contains the addresses of the ends of the newly extended workspace and a pointer to the return instruction.

The workspace records also contain a pointer to a *display.* Displays are devices to allow code to access variables at an outer level of static nesting. Since variables are addressed by an offset within the workspace, the address of the workspace which contains the variable must be known. Each level of nesting is allocated a number, its static level, and this is used to index into an array of workspace pointers, providing very fast access to the variable. With a conventional sequential language this is easy to implement, using a global array, with portions of it cached in registers to accelerate access. Unfortunately this does not work for a highly concurrent language like OCCAM, where each process will have different procedures active at different static levels. To overcome this, each process has its own display derived from the parent process.

Displays are only one way of accessing non-local variables. The use of *static links* is now rather more common, and they are used for both the portakit and transputer instruction sets [Whitby-Strevens 85]. In this scheme each stack frame includes a back pointer to the static level above the current one. This is not always the one immediately enclosing the current workspace since it is possible to call procedures declared at further out levels. There is a considerable penalty in accessing variables that are several frames higher, so most architectures include a pointer to the outermost block. In this way both local and global variables can be accessed quickly, but intermediate accesses are still slow. This is justified by analyses of the usage patterns of typical programs, where most variable references are either local or global. With displays all accesses are of uniform high speed, but more space is required to implement them. I chose to use displays for the interpreter as it is already slow, and implementing the look-back loops in a high-level language would be inefficient.

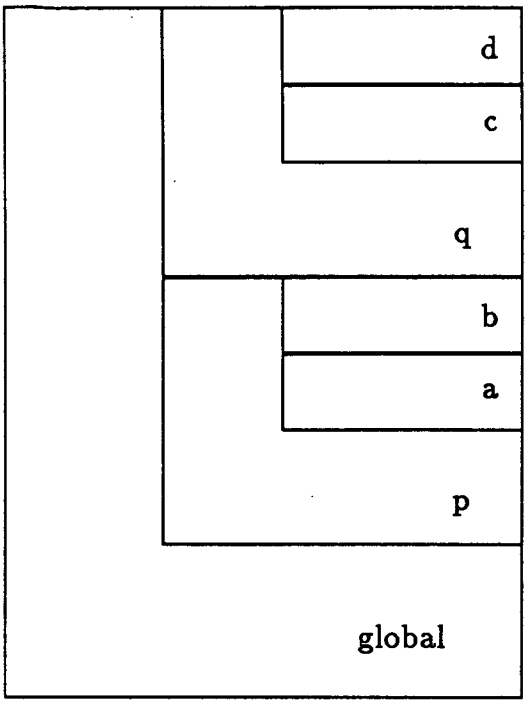
4.2 Implementing Concurrency

The implementation of the sequential constructs of OCCAM is unremarkable, so is not described here. The simulated concurrency, however, is more unusual so the following sections describe how the PAR, ALT, and communication constructs are implemented. In this section the words "process", "task", and "job" are used interchangeably to improve readability.

4.2.1 PAR

The PAR construct launches its component statements as parallel processes and suspends the parent until all the subprocesses terminate. Two things must be done before starting a process: a workspace and a *process control block* (PCB) must be created for it. The subworkspaces are created within the parent's workspace, as shown in Figure 4–1. Note that this is like a procedure call, except that several subspaces are created simultaneously. The PCB is created off the heap, and must be added to a run-queue. The PAR construct can be preceded by the modifier PRI, converting it to a prioritised construct, where the priority of each created process is determined by the order they appear: first is highest and the same as the parent, last is the lowest. The INMOS products, including the transputer, only have two levels, but since many is as easy to provide as two, this interpreter provides six but can be constrained to use as few as one for simulating different architectures. In addition to the run-queues, all PCBs are kept on a global list to allow the debugger to access all the processes.

In order to simulate multiprocessing, all processes which are ready to run must be given some processor time. Several schemes are possible, the simplest being self-scheduling, which relies on processes performing some function, such as channel communication, which cause them to suspend. This is used by the portakit, but does not prevent a high priority process that is looping from locking out other processes. So some control is needed in addition to the self-scheduling.




```
PROC p =
  PAR
    a
    b:

PROC q =
  PAR
    c
    d:

PAR
  p
  q
```

Figure 4–1: Nested process workspaces

One possible method is to execute one statement from each process that is ready in a round-robin fashion [Ainscough 85], but this results in a considerable overhead in context changing. The best option is *time-slicing.* The transputer microcode implements this, dropping the current process in favour of the next one in the high priority queue after 4096 processor cycles (a time of $819 \cdot 2\mu$S). If a high priority job becomes ready while a low priority one is executing, the low job is suspended. This interpreter has adopted a similar approach, since it does not have a clock interrupt it deschedules a process after executing 100 instructions.

When a process is time-sliced or deschedules itself two operations are performed. First the timer queue is checked; tasks can wait until after a specified time with the following statement[1]:

```
TIME ? AFTER last.time + alarm.period
```

[1]This is another substantial change introduced with the Programming System.

Whether a process is ready to execute is determined by the value of the "wake-after" time in the PCB. This is compared with the current time using arithmetic modulo the word length so that time is monotonically increasing, even when the representation wraps round. If any of the wake-after times are before the current time, those processes are added to the appropriate run-queue according to their priorities. Note that this definitely implements the AFTER operation and is not equivalent to being woken at an exact time. Details of the clock are given in the next section.

The second task performed at rescheduling is checking for keyboard input. The special channel Keyboard suspends the inputting process until a character is typed, unless some have been typed ahead. The waiting task is placed in the keyboard queue, which must only ever contain one process to enforce the one process per channel rule. When a character is detected the waiting process is placed on the run-queue appropriate to its priority. This implements asynchronous terminal input, but unlike other channels it is buffered. Once these two checks have been performed, the next process is chosen from the highest priority run-queue that is not empty. A low priority process is activated only when all higher priority tasks are suspended.

Processes can die in two ways: they can *terminate* or *stop*. Termination is the normal way, happening when all the components of a process have been executed. When a job terminates it must reactivate its parent if it is the last subprocess running. To implement this the PCB contains a "child count" field which is the number of subprocesses still running and is filled in at the start of the PAR construct. When a subprocess terminates it decrements the child count of its parent and when that reaches zero the process deletion procedure reschedules the parent. If the global process terminates then execution ends.

The stopped state is described as being for "error containment". The effect is that the process remains in existence but is permanently descheduled. Any other process which is dependent on that one for communication will hang and eventually the whole system deadlocks. While this does have the effect of "gracefully" stopping the program when an error occurs it is somewhat hard to

trace what started the collapse. My interpreter extends the stopping facility: normally when a process stops the whole program halts and the debugger is entered, clearly showing which task stopped. Alternatively a warning message can be generated and execution proceeds, or it can operate compatibly with the INMOS products.

4.2.2 Channels

Channels provide communication between processes and between the program and its environment. These two uses are described separately.

Interprocess Channels

A channel is implemented as a simulation-host size word, which is either the special value `Not.Process`, or a pointer to a PCB. If it is `Not.Process`, then the channel is not in use and a process that wishes to make a communication fills in the channel transfer record in its PCB, places a pointer to the PCB in the channel and deschedules itself. When the other process is ready to complete the transfer it takes the pointer to the first process's PCB, performs the transfer as described in the record, resets the channel to `Not.Process`, puts the other process back on the appropriate run-queue and then performs a reschedule.

This pattern is followed whichever process starts the communication, although it is the inputting task that always performs the copy. The form of the transfer record varies depending upon which kind of communication it is: word, `ANY`, or slice. Neither the OCCAM Programming Manual or System give any guidance on what happens when the type of the input is different from that of the output. For example, can an `ANY`-input be used to ignore the argument of a word output? Is a word length slice transfer the same as a word communication? This interpreter implements all these cases, but faults slice transfers which are of different lengths. It also checks for processes sharing channels.

Special Channels

The only channel that is built into OCCAM is TIME. This is used for reading the current time and is equivalent to assigning the clock value to the variable and does not actually communicate or cause a reschedule. It does, however, suggest that the clock is in some way a separate device. The TIME channel can also be used to suspend a process, as described on page 65.

The resolution of the clock is an important issue, and can vary between implementations. For example in the VAX version of the Programming System the clock is in 100nS units, updated every 10mS and transputer time is measured in 1μS units. This is due to implementation expedience, for example the odd VAX format is due to the hardware. In this interpreter the basic clock unit is 1mS as this is the time provided by the APM. The word length of the machine also affects practical clock units, for example, a 1μS clock on a 16 bit machine would make the longest possible delay 0·065 seconds, which is much too short to be useful. Even on the transputer the maximum delay time is approximately 72 minutes, which again may not be long enough, requiring the programmer to build longer delays herself.

To allow simulation of different architectures an integer *clock factor* can be set to adjust the basic clock rate. If this is a positive number it is divided into the time, or if negative the absolute value is used to multiply the time. It is possible to change value as the program runs as an aid to debugging: events that happen very fast can be slowed down, or very slow processes speeded up.

The other special channels are declared with the special syntax:

```
CHAN Screen AT 1:
```

which was introduced to allow hardware links on the transputer to be declared, but has now been used for integrating OCCAM into programming environments. It is, in effect, a form of operating system call, where the compiler or interpreter maps the "address" onto a procedure [Hazari 86]. The complete set of special channels is shown in Figure 4–2 and with the exception of Keyboard they all

```
CHAN Parameters   AT 0:    -- Command line parameter
CHAN Screen       AT 1:    -- Output on the terminal
CHAN Keyboard     AT 2:    -- Input from the terminal
CHAN FileIn0      AT 3:    -- Communication with the
CHAN FileIn1      AT 4:    -- file system, channels
CHAN FileIn2      AT 5:    -- used in in/out pairs
CHAN FileOut0     AT 11:
CHAN FileOut1     AT 12:
CHAN FileOut2     AT 13:
CHAN ErrorMessage AT 20:   -- Reports from the file system
CHAN Date         AT 21:   -- The current date
CHAN Time         AT 22:   -- The time of day
CHAN Terminal     AT 23:   -- Cursor moves etc.
```

Figure 4–2: The special channel declarations

operate immediately. It would be possible to implement the file communication channels asynchronously, suspending the process until data arrives from the disk, but in the interests of portability and simplicity they are mapped onto Imp77 input-output routines.

One of the limitations of using channels for implementing what are in effect procedure calls is that parameter passing is difficult. To cure this, each channel that requires more than one parameter per operation is implemented as a state-machine. For example the channel `Terminal` can be used to move the cursor on the screen to point (3, 10) with the following statement:

```
Terminal ! Term.Cursor; 3; 10
```

When the channel receives the value `Term.Cursor` it moves to a state requiring an X-coordinate, then a Y-coordinate, then performs the cursor jump and returns to the waiting state. Another approach would be to use reverse-polish, where parameters are pushed onto a stack to be popped off by the operators, but this is a less elegant solution. It is obvious that the one process per channel rule must be adhered to with these channels with state.

This complicates matters as a channel transfer record must be created for each element of the loops and the replication values must be set up before making the transfer. The facilities that make ALTs difficult to implement also make it powerful, but it is essentially an inefficient construct with very high overheads.

4.3 Assisting Simulation

Once more we have reached a point where a conventional OCCAM system has been described, and it can either be extended to support interfaces or the onus can be placed on the programmer. Assuming that the one process per channel rule can be bent slightly to allow multiple processes to use the channels `Screen` and `Terminal`, each SIMUL body could always use a particular pair of `File` channels and a fixed area of the screen. A simulation program for the thermostat was built to test this method, and uncovered a number of problems.

The first difficulty emphasised the need for strict enforcement of the channel rules. One process would output a cursor jump only to be descheduled and another one put out its message at the wrong point on the screen. A solution that the designer can employ is to use a screen *mixer* process which drives the screen from an array of channels, with each outputting process claiming the driver until it has finished. This approach works for a static program, but is not suitable for a dynamically constructed simulation.

The second complication is that of replicated interfaces. If a simulation body has screen coordinates built-in to it, then only one interface can be instantiated. The solution to these two problems is for the interpreter to provide a window manager, giving each interface a portion of screen of its own. A modified version of the hardware allocator code is used for this, allocating one line at the top of the screen for each interface. This was developed to use a character terminal rather than a graphics device for portability and simplicity. The allocator draws a frame around the lines or, *micro-windows*, used for the interfaces, and prints the name of each interface at the left. The bottom of the screen can be used

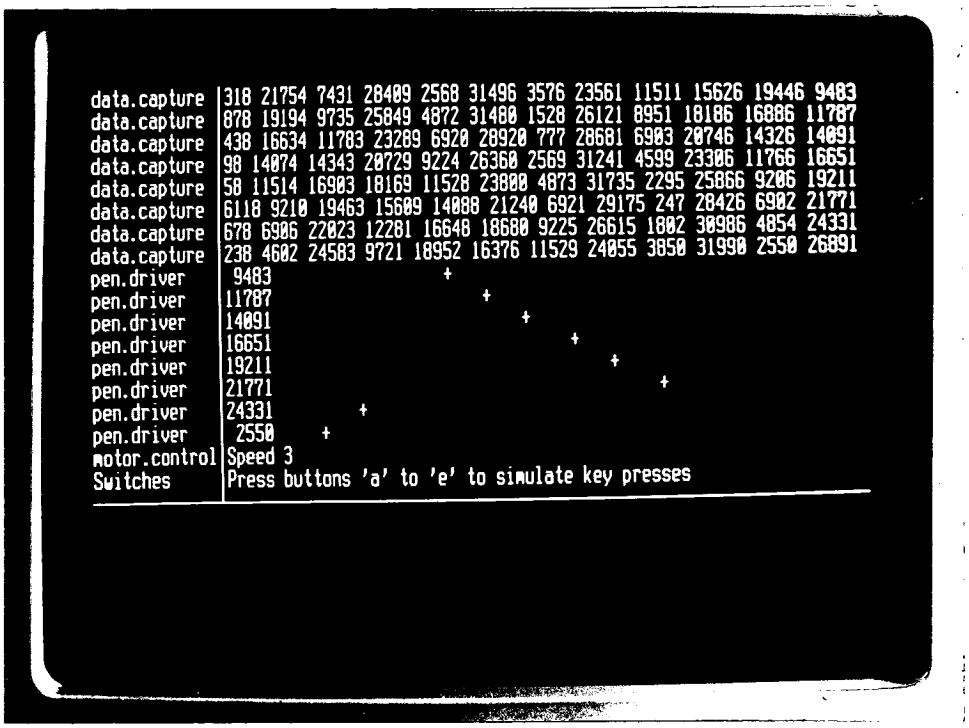


Plate 4–1: Simulating an eight-channel chart-recorder

for diagnostic messages from the control program, or the debugger. Within each micro-window text is scrolled horizontally, newlines appear as "|" and the channel `Terminal` can be used to clear areas and move the cursor, although the Y-coordinate is ignored. An example demonstrating these facilities appears at the end of the section.

Multiply instantiated interfaces currently cannot read from different data files. One of the major limitations of the APM is that it only allows three files to be open in each direction, which would limit the usefulness of this facility. At present each interface choses one of the file-channel pairs, and opens a file of a fixed name. The way this could be implemented, given an arbitrary number of input-output streams, is to provide each interface with channels `DRIVE` and `TRACE`, which would be connected to files during the allocation phase.

The use of different word lengths, priority ranges and change the clock rate all contribute to simulation. By default the simulator is set up to mimic the Z80 system described in the next chapter, with the word length set to 16 bits, only one level of priority and the clock period set to 20mS, giving a maximum

```
INTERFACE data.capture (CHAN data) =
  FROM chart.recorder:
INTERFACE pen.driver (CHAN with) =
  FROM chart.recorder:
INTERFACE motor.control (CONFIG start, CHAN speed) =
  FROM chart.recorder:
INTERFACE switches (CONFIG number, CHAN return[]) =
  FROM switches:

DEF channels = 8:     -- number of data channels
CHAN transfer [channels]:

DEF buttons = 5:      -- number of speeds + stop
CHAN speed [buttons], motor:

PAR -- Chart Recorder
  PAR i = [0 FOR channels]
    data.capture (transfer [i])
  PAR i = [0 FOR channels]
    pen.driver (transfer [i])
  motor.control (0, motor)
  switches (buttons, speed)
  WHILE TRUE          -- listen for the buttons
    ALT i = [0 FOR buttons]
      speed [i] ? ANY
        motor ! i     -- set the motor speed
```

Figure 4–3: Chart-Recorder program

```
INTERFACE data.capture (CHAN data) =
  SIMUL
    VAR seed:
    SEQ
      Random.Seed (seed)
      WHILE TRUE
        SEQ
          Random (seed)          -- get new random number
          IF                     -- get its absolute value
            seed < 0
              seed := -seed
            TRUE
              SKIP
          Write (Screen, seed, 0)
          Screen ! ' '
          data ! seed:
```

Figure 4–4: The Data-Capture Simulator

of about 22 minutes delay. Any PRI PAR constructs are implemented as normal PARs with a warning.

A simulation example

To demonstrate the use of the simulator a simple example is presented. This is an eight-channel chart-recorder with five speed motor. The very short source code is shown in Figure 4–3. Assuming that all the interfaces were already designed, this is all that the applications engineer would need to specify. By changing the value of `channels` the number of data-capture/pen-driver pairs can be adjusted allowing controllers for a family of chart-recorders to be created instantly. A simulation of the eight-channel device is shown in Plate 4–1.

Figure 4–4 shows the SIMUL body for the data-capture interface. Since the device can potentially be connected to anything, the random number package provided with the interpreter environment (written in OCCAM) is used to generate the incoming data. To check that the data at the pen is the same as the probe, the value is written to the interface's micro-window. The pen-driver simulation is shown in Figure 4–5. This code writes the actual value at the left of the micro-window, and then uses the cursor to simulate pen movement. To minimise screen updates, which are costly in time and look jerky, it does not redraw the window when the value has not changed.

The switches simulator is shown in Figure 4–6. This uses the terminal keyboard to act as the buttons, allocating a letter per button. Each press is returned via an array of channels, allowing more than one key press at once. The motor driving procedure, shown in Figure 4–7, simply prints out the speed number. Note that these interfaces do not use messages like "Press button 'b' for 2 cm/s" as they are intended to be general, for example the switches interface is used in the stopwatch example in Chapter 7.

```
INTERFACE pen.driver (CHAN with) =
  SIMUL

    PROC display (VALUE clear, what) =
      DEF screen.compress = 32767/60:
      SEQ
        Screen ! '*c'                  -- carriage return
        Write (Screen, what, 5)
        Terminal ! Term.Cursor; (clear/screen.compress) + 6; 0
        Screen ! ' '                   -- clear last 'pen'
        Terminal ! Term.Cursor; (what/screen.compress) + 6; 0
        Screen ! '+':                  -- draw new 'pen'

    VAR last, data:
    SEQ
      last := 0
      display (0, 0)
      WHILE TRUE
        SEQ
          with ? data                  -- get the data
          IF
            data <> last
              SEQ
                display (last, data)
                last := data
            TRUE                       -- ignore if the same
              SKIP:
```

Figure 4–5: The Pen-Driver Simulator

```
INTERFACE switches (CONFIG number, CHAN return[]) =
  -- provides number push button switches
  SIMUL
    VAR ch:
    SEQ
      PrintString (Screen, "Press buttons 'a' to '")
      Screen ! number + ('a' - 1)
      PrintString (Screen, "' to simulate key presses")
      WHILE TRUE
        SEQ
          Keyboard ? ch
          IF
            (ch >= 'a') AND (ch <= ((number + 'a') - 1))
              return [Ch - 'a'] ! ANY
            TRUE                       -- ignore out of range
              SKIP:
```

Figure 4–6: The Switches Simulator

```
INTERFACE motor.control (CONFIG start, CHAN speed) =
  SIMUL
    VAR S:
    SEQ
      S := start                -- initial speed
      WHILE TRUE
        SEQ
          Terminal ! Term.Clear.Line
          IF
            S = 0
              PrintString (Screen, "Stopped")
            TRUE
              SEQ
                PrintString (Screen, "Speed ")
                Write (Screen, S, 0)
          speed ? S:
```

Figure 4–7: The Motor-Control Simulator

4.4 Diagnostic Facilities

Great emphasis has been placed on the provision of diagnostics by this interpreter. These are provided by way of an *interactive debugger*, which permits examination of the interior of process workspaces in terms of source identifiers and line numbers. There are three routes into the debugger: program failure, user interrupt and watch-points.

The program is continuously checked for potentially hazardous behaviour. If any of the following conditions arise, the program fails. When this happens the current line number and a pertinent message is printed, followed by a *trace-back*, as shown in Plate 4–2.

Array accesses are checked for being within bounds. If this is not the case then the requested index and the actual bound for the array are given. Array parameters are currently not checked.

Channels are checked for being shared. Other possible errors are related to

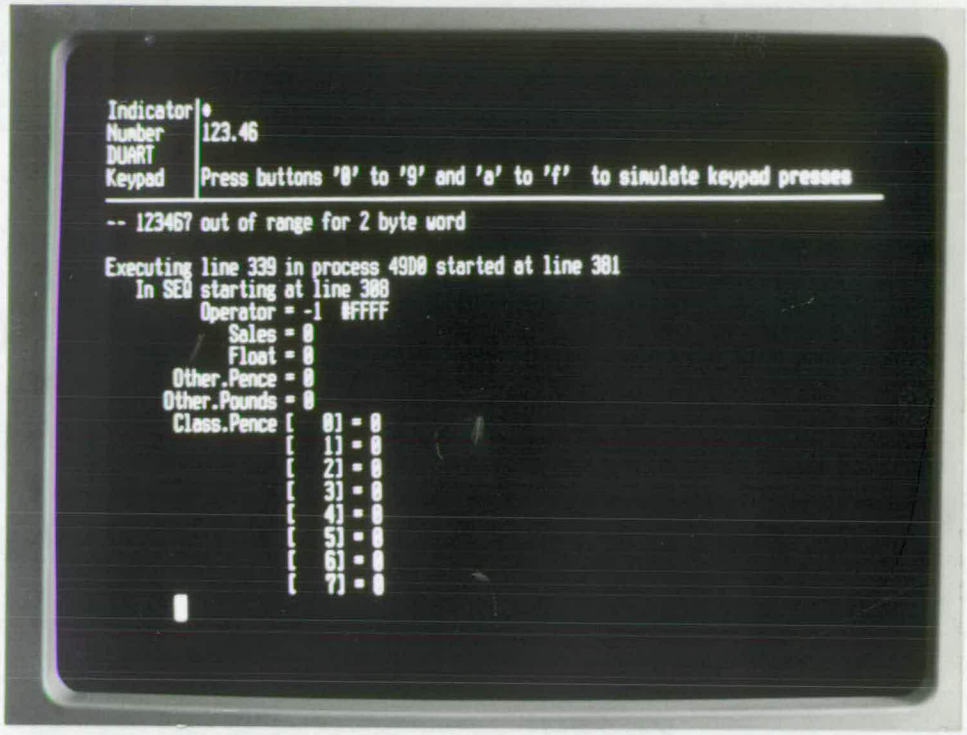


Plate 4–2: A program failure

inappropriate use of the special channels. Note that file errors, such as file not found, are not included as they are reported to the program.

Constructs with replication counts are checked that the replication limit is a natural number.

Deadlock arises when no processes are available for execution and none are delaying or waiting for external events. No process is obviously at fault, so rather than the trace-back being printed, the table of processes is given.

Expressions are checked for the usual errors like division by zero, and negative shift counts. When operating in 16-bit mode, results are checked to be within that range.

Slice lengths are checked for match during assignment and communication. Indices are also checked for being within bounds.

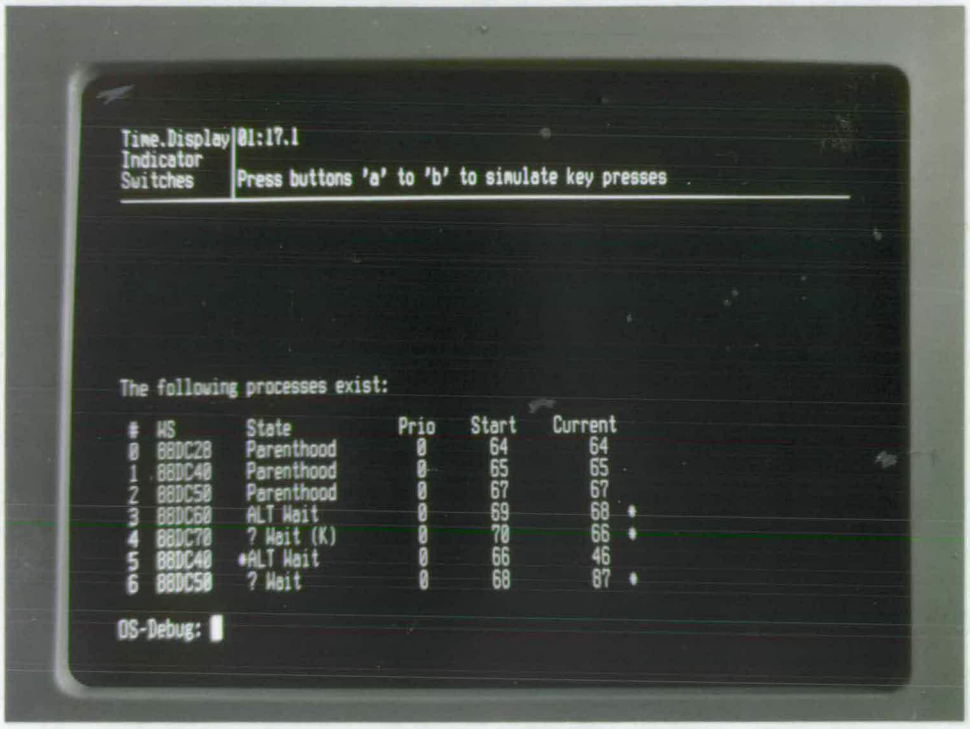


Plate 4–3: A process table

Variables are checked for being unassigned if running in 32-bit mode. The range of 16-bit numbers is too small to be restricted still further by the use of an unassigned pattern.

Typing control-C during program execution enters the debugger. One of the available commands sets *watch-points* on variables. By giving an identifier, possibly with an index, all accesses to objects of that name are logged, and optionally the debugger entered. Similar watch-points can be placed on line numbers. Both these watch-points are checked lexically, so it is not possible to restrict matching to a particular scope. If a line number is executable in several modules, each occurrence will set off the trap.

Once in the debugger it is possible to obtain a table of active processes as shown in Plate 4–3. The first column is a reference number for use inside the debugger, the second is the workspace address and is useful in debugging the interpreter only. The next column is the process *state*. If this is `Parenthood` it means that the process is executing a `PAR`. The meanings of the other states are all obvious: `Runnable`, `Stopped`, `! Wait`, `? Wait`, `ALT Wait`, and `Delaying`. A

star before the state indicates that it is being executed, and a (K) after a wait indicates that it is on the keyboard queue. The fourth column is the process priority, with 0 as the highest and 5 the lowest. The last two columns are line numbers, the line that starts the process and the line currently being executed. If this is followed by a star it means that it is an interface process.

Using the process number it is possible to obtain a stack trace back for any process, similar to that shown in Plate 4–2. The values of all the variables are displayed, including arrays which are printed as text strings if they are of the correct form. There is currently no way to investigate the state of channels, which would be a useful addition.

Commands are provided for putting these tracing tables in files. It is also possible to change the clock rate interactively, although this has undesirable side effects. If a process enters a delay state while the clock rate is set high, it can take a very long time to reach the desired time when the clock rate is reduced. Execution can be resumed if entry was from an interrupt or a watch-point, or it can be stopped.

4.5 Timing: Accurate or not?

One of the problems with using simulators is knowing whether to trust the results or not. All simulators, including this one, can be made to produce false or misleading results. First of all we shall identify some problem areas. The most obvious is speed of execution: as the program is being executed by an interpreter written in a high-level language it is very slow, especially since all operations are rigorously checked. Execution on an 8MHz M68000 with physical store is about the same as native code on a 4MHz Z80. This is slow, but is not actually a significant disadvantage when using the prototype back-end.

Execution speed is primarily an implementation problem, and can be cured by the insertion of machine code in critical paths or moving to a more powerful machine. One factor that does slow simulation unavoidably is terminal and file

communication: it is much slower to write six characters to a terminal than write them to registers for display by hardware; an analogue-to-digital converter presents its data much faster than a file-server.

Different implementations will use varying scheduling algorithms, resulting in processes being executed in a different order. SYNTH code bodies are, by definition, different from their respective SIMUL bodies. These two points are inevitable, and are a common problem in writing portable programs. The solution is *defensive programming.* An example of this appears in the stopwatch program in Chapter 7. Here the watch suspends itself until 0·1S later, but it does not assume that this is actually the time when it is woken. By doing this ten stopwatches can be run simultaneously, and each one shows the correct time when it is updated, although it is not updated every 0·1S.

To a large extent the accuracy of an USHER simulation is dependent on how well the SIMUL bodies emulate the actual interfaces. This code is created by interface designer, and when employing defensive programming techniques, which most programmers do anyway, this simulator provides the facilities to allow the effective simulation of any design. All the examples in this thesis have been simulated, and those that can run on the Z80 test kit have done so without modification.

Milne has pointed out that a major problem with conventional simulators is understanding their output [Milne 86]. By using the very high-level simulation model of USHER this problem is greatly reduced as the volume of data is lower. The clear visual presentation of high-level output makes mistakes more obvious, and the debugger provides access to as much low-level data as is required to fix the problem.

Chapter 5

The z80 Back-End

The Z80 Family handles most microprocessor applications with little additional logic. Z80 designs are efficient and cost effective microcomputer systems.

Zilog Publicity Material

A prototype back-end for USHER was required to demonstrate that the ideas presented in this thesis work in practice. The ideal technology to have developed would have been single-chip, custom-silicon controllers, and some possible approaches are outlined in Chapter 8. Unfortunately no appropriate building blocks were available, and much of the circuitry would have had to have been built from scratch. This would have required a considerable amount of work with little new content. Together with the infrequent fabrication runs, this made a VLSI-based solution impractical. The alternative was to develop a board-level back-end using a standard-part microprocessor.

The obvious processor to use for this system would be the INMOS transputer since it was designed specifically for running OCCAM programs. This device was just becoming available on evaluation boards at the time when it would have been needed, but at a very high price and no funds were available to purchase one. It is also likely to be several years before electronic controllers will use 32-bit processors. This led to the decision to use an 8-bit microprocessor since they were available and controllers are currently built with them.

A simple hardware prototyping system had been developed some years ago for undergraduate teaching in Edinburgh University's Computer Science Department, and was no longer in use. These provide a four-rail power supply, an area of

solderless bread-board, and slots for the insertion of "personality-cards". Cards were available with M6809 and Z80 processors on them, with a small quantity of RAM and a zero-insertion-force socket for an EPROM. These were ideal for USHER experimentation, as the bread-board could be used for testing various interfaces.

This left the choice between M6809 and Z80 microprocessors. While the M6809 has a very orthogonal instruction set, the Z80 offered a number of advantages: simple interfacing without handshake, a separate input-output address space reducing the amount of decoding needed, a larger number of registers, 16-bit arithmetic and a shadow register-set for interrupt handlers. With these advantages the Z80 was the best option.

This chapter describes the Z80 environment that was developed, and then concentrates on code generation. Section 5.1 provides a short introduction to the internal architecture of the Z80. Section 5.2 describes the development of a test microcomputer. Next Section 5.3 describes the code generator, showing examples from both the sequential and concurrent language facilities. The last section describes how the hardware and software parts of the interfaces are integrated.

5.1 The z80 Architecture

The Z80 offers a rich variety of instructions and features, most of them useless. It is plagued by instructions which only operate on particular registers, and special optimisations which are designed for the assembly-language programmer, but not the compiler writer. Despite this it is quite possible to generate acceptable Z80 code from a high-level language.

There are three register sets: main, alternate, and special purpose. The alternate set are identical to the main set, and can be used by interrupt handlers to avoid having to save and restore contexts, the `EX` and `EXX` instructions switch between sets. Each set provides an 8-bit accumulator `A`, a flags register `F`, and

six other 8-bit registers, B, C, D, E, H, and L. H and L are used together as the 16-bit accumulator, and the others can be used as the 16-bit registers BC and DE under certain circumstances, particularly for holding addresses. The useful special registers include the 8-bit interrupt register I, which is described later, and three 16-bit registers which are used for addressing. These are the index registers IX and IY, and the stack pointer SP.

The most powerful instruction is LD (LoaD), which moves 8-bit and 16-bit data around. The destination is the first operand, the source the second, and indirection is indicated by brackets. The most glaring omission is a direct 16-bit register-to-register move, and these must be done either with two byte-sized moves or with the PUSH and POP instructions, which automatically decrement and increment SP. For the compiler writer, however, the most restrictive omission is the absence of a 16-bit load from an address held in a register. This must be done with two byte-sized loads, and is usually done via the index registers, as explained in Section 5.3. The local assembler allows these two-byte moves to be abbreviated to single pseudo-instructions.

Three styles of interrupt response are available, but only mode two is useful. In this scheme the interrupting device places the low byte of an address on the data-bus. This is concatenated with the contents of register I to form an address which contains the address, or *vector*, of the interrupt handler. These vectors are normally grouped in a table. Interrupt handlers are terminated with the RETI instruction, which is recognised by some peripheral chips. Interrupts can be enabled or disabled with the EI and DI instructions respectively.

5.2 A Simple Microcomputer

Facilities were required in which to test both the code generated by the compiler and to develop the interface hardware. The solution to both of these was to build a microcomputer, which could be connected between the APM and the terminal. This machine is called the *exercise machine* because it is used to exercise the

USHER system and its output. Normally the machine is in "transparent" mode and is invisible to the user, but on receipt of a particular control character it down-loads a program and then executes it, with the special channels `Screen` and `Terminal` connected to the terminal. Such a system was constructed and the next two subsections describe the hardware and software provided by it.

5.2.1 Exercise Machine Hardware

The Z80 personality-card provides the basic services of a clock, 128 bytes of RAM, a socket for a 1k byte EPROM, run/halt logic, and address and data bus buffering. In addition to this, the exercise machine required two RS232 lines, one for the APM and one for the terminal, a clock, an address decoder for the interfaces and a large amount of memory. The first version used eight 64k-bit dynamic RAMs, however the bread-board was too electrically noisy for these sensitive devices to operate satisfactorily. The final solution was to use four 8k byte static RAMs, which proved to be enough memory. A schematic for the memory system is shown in Figure B–1 in Appendix B on page 158.

The two RS232 connections are provided by the Signetics SCN2681 Dual Asynchronous Receiver/Transmitter (DUART). This contains two independent, fully software-configurable RS232 input-output devices in one package. Since this is not a member of the Z80 family it does not provide the various interrupt modes used by the Z80. In particular it cannot support mode two, where the interrupting device must place the low byte of the vector address on the data-bus in response to the $\overline{\text{M1}}$ and $\overline{\text{IORQ}}$ signals being asserted. This function is provided by a 74LS244 octal buffer connecting an eight-way DIL switch to the data-bus, allowing the interrupt vector to be moved during software development. This circuit is shown in Figure B–2 on page 159.

A clock chip is required to assist with time-slicing processes and to provide a basis for the OCCAM channel `TIME`. The chosen device is the Z80 family Counter Timer Controller (CTC). This device connects to the Z80 with no additional logic as it has interrupt mode two hardware on chip. It does need a chip-select signal,

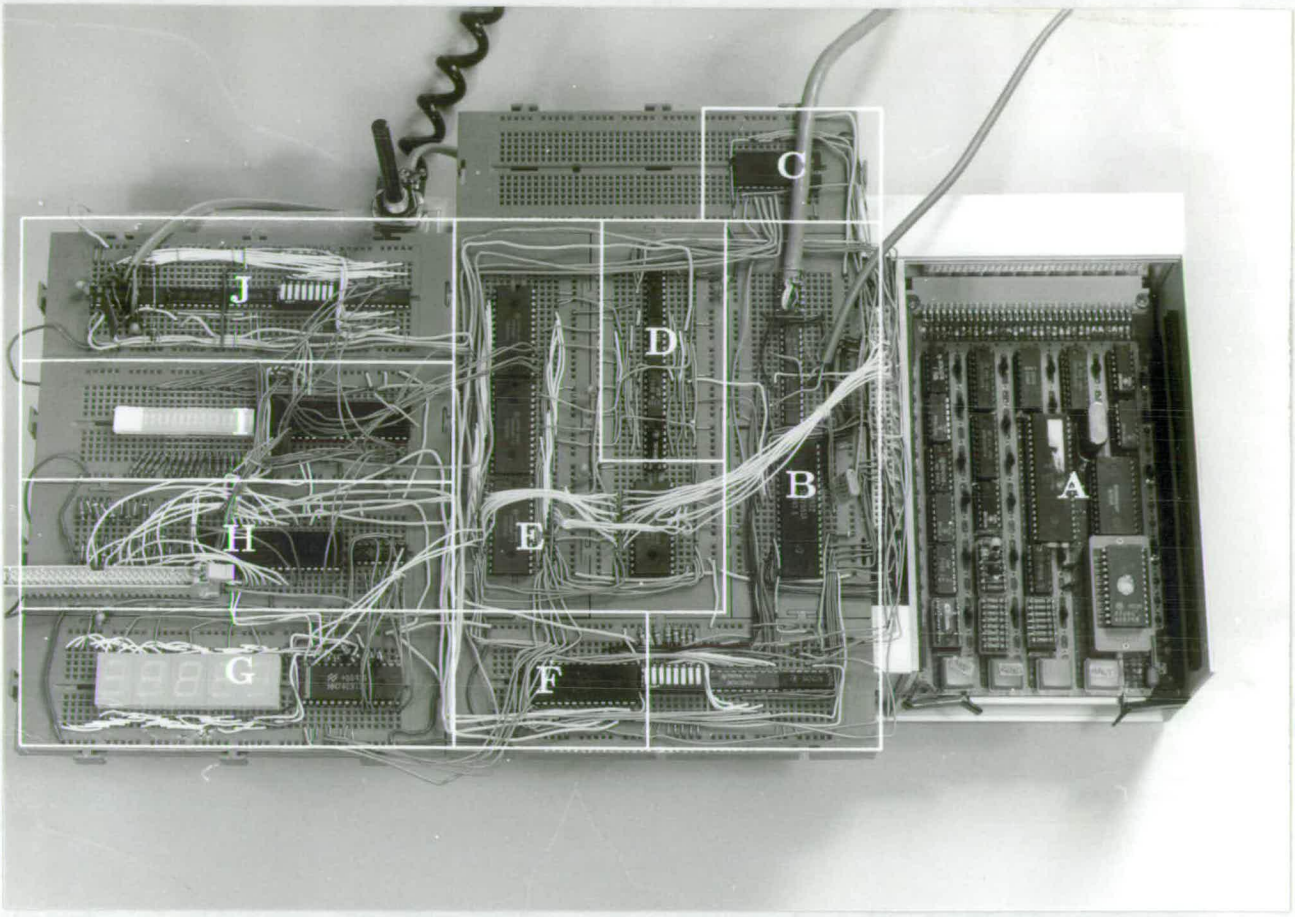
and this is drawn from a four-to-sixteen decoder (74LS154) which is used to select all the interfaces. This is connected to address lines A_4 to A_7, which leaves the four lowest addresses lines for selecting the internal registers of interface chips. The CTC and decoder are shown in Figure B–3.

This completes the description of the basic hardware facilities provided on the exercise machine. Plate 5–1 shows the different parts of the machine, including other interfaces which are described in the next chapter.

5.2.2 ZERO

The Z80 Executive Running OCCAM (ZERO) provides two classes of services: communication with the APM and a multiprocess kernel to support OCCAM execution. Three control-characters interrupt the tight polling-loop that is used to implement transparent mode. One starts a load of up to 256 bytes from the APM into any part of memory, another prints a screen sized hexadecimal memory-dump on the terminal starting at a given address, and the last one starts a process running from a given address. Normally the program is loaded at the low end of the 32k bytes of RAM, and the workspace extends downwards from the top. Most of the 1k byte of ZERO is devoted to the support of multitasking, because of the limited size of the operating system only one level of process priority exists, but otherwise it is a full implementation of parallel processes. Some of the features are described below.

ZERO contains procedures for the creation, scheduling and destruction of processes. Like the simulator, process control blocks are used when manipulating processes, but unlike the simulator these are held at the start of the process workspace, not in a separate area. The format of the PCB is shown in Figure 5–1. Note that the stack pointer for the process is saved in the PCB when it is not active, however the other registers are saved at the end of the workspace, and the stack pointer contains the address of the end of this save area. Processes are referred to by a *process identifier* (PID), which is the address of the next-in-queue field in the PCB. The uses of the different fields will become clear through



A The Z80 personality card.

B The DUART and interrupt circuit.

C The interface address decoder.

D The memory control logic.

E The static RAM chips.

F The Counter Timer Controller.

G The seven-segment displays and driver.

H The switch interface.

I The bar-graph display interface.

J The analogue-to-digital converter.

Plate 5–1: Exercise Machine Anatomy

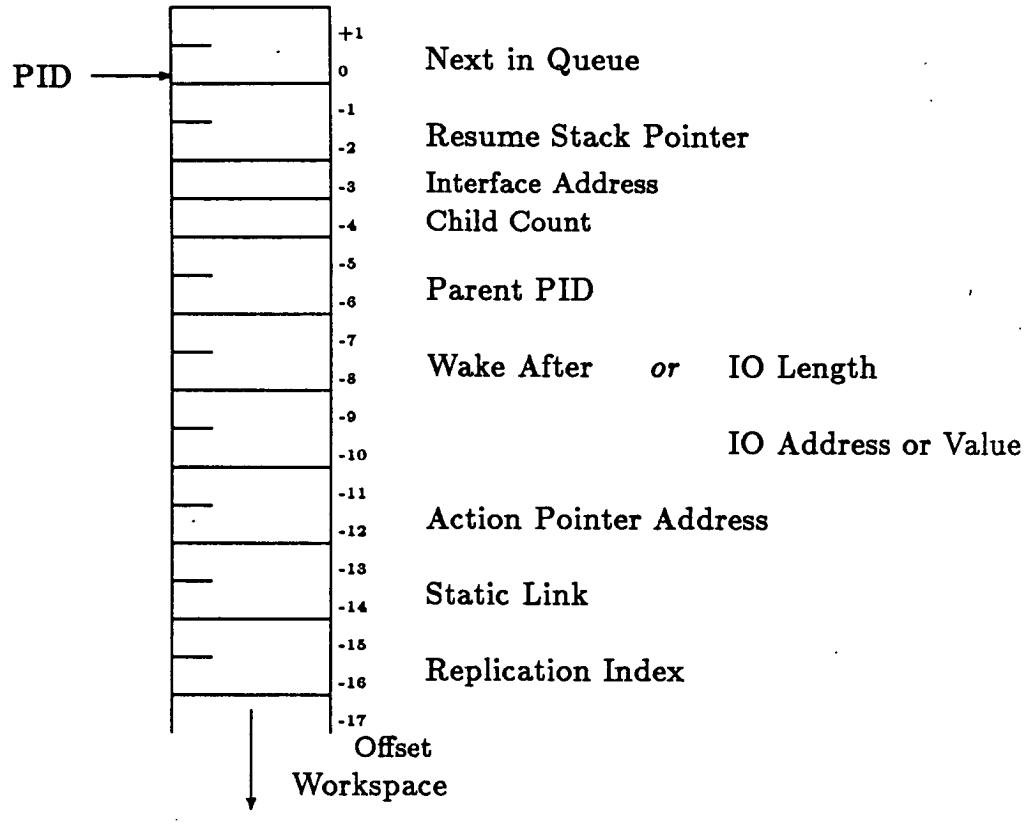


Figure 5–1: ZERO Process Control Block Structure

the rest of this chapter. Suspended parent processes are reactivated by use of the child-count and parent-pointer fields by the process destruction procedure.

ZERO provides a single run-queue and a timer-queue. The CTC chip is programmed to generate an interrupt every 20mS, and on this the current process is descheduled in favour of the next in the run-queue. At the same time the timer-queue is examined, using the wake-after field to determine if any of the processes are ready. The time is kept in the personality-card RAM, and is incremented at each interrupt. The scheduler performs deadlock detection, printing out an error code and returning to transparent mode if it happens. This is the strategy taken for all errors, it is assumed that the simulator is used to debug the program before attempting to run it on the exercise machine. Error trapping is the only occasion where the `RST` instruction is used, all other system calls are invoked with the normal `CALL` instruction.

An interrupt handler for the DUART is provided to support asynchronous input for the channel Keyboard. Up to sixteen characters can be typed ahead, after that a beep is generated and the characters discarded. The keyboard "queue" is checked so that it can only contain one process, verifying that the channel is not being shared. If a process is waiting on the keyboard-queue when a DUART interrupt occurs, it is added to the run-queue. Various output routines are provided to help in diagnosing code-generation faults.

5.3 Code Generation

OCCAM was first released in the form of an evaluation kit, which compiled to UCSD P-Code. Since then at least one portakit interpreter has been mounted on a Z80, but I am aware of no attempts to build a native code generator for an 8-bit microprocessor. This caused some trepidation, but the compiler proved quite easy to write, and with the 4MHz clock rate chip, execution speed is impressive. To test relative performance, the Newton-Raphson square-root approximation program from the OCCAM Programming Manual was used to calculate six roots with a pipeline of ten processes. The average time over several runs using the INMOS native-code compiler on a VAX 11/780 was 160mS, and the time on the exercise-machine was 450mS. This section describes how standard OCCAM is compiled, first the sequential parts then the concurrent constructs.

5.3.1 Sequential Occam

The storage allocation scheme is different to that of the simulator, as it is influenced by the Z80 instruction set and the requirement to keep the size of the code as small as possible. Since nested variables cannot be allocated fixed addresses they must be indexed from a base for each context, and this is done by always having the address of the current stack-frame in the index register IX. To save space, the static-link method of non-local access is used rather than displays. Since the global process is only created once, all global variables can be allo-

cated fixed addresses, which means that both local and global variables can be loaded or stored with a single instruction. Variables declared at the level above the current one are accessed by linking back through the static link with register IY. Further out levels are accessed by a *primitive* procedure. These are routines written in Z80 assembler which the compiler knows about and uses to extend the instruction set. There are 32 such primitives used by this compiler. Figure 5–2 shows examples of each level of variable access.

The indexed load instructions restrict the range of displacements to ±127 bytes, and this further constrains the form of the stack-frame. To use this range to its full, yet keeping IX pointing to the static link, parameters appear above the link, with local variables below. This is in keeping with procedure calls where the parameters are pushed and the procedure is called, pushing the return address onto the stack. Figure 5–3 shows the layout of the stack-frame. The dynamic-link contains the frame pointer of the calling context, and the static-link points to the frame that is lexically above the current one. If the procedure has parameters, to save popping them off individually, the value of the SP from before pushing them is saved. The code generated for a procedure call is shown in Figure 5–4, demonstrating the large amount of code which can be needed. If arrays are included in the local space they are always placed at the end of the frame. This means that if the array size exceeds the 127 byte limit, simple variables do not suffer from more complex indexing. An example of indexing into a local array with a value parameter is shown in Figure 5–5. If the array bounds are within the 127 byte range, the indexed load instruction is used.

The Z80 provides sufficient registers that most expressions do not need to use temporary variables, but too few to make remembering register contents between statements important. This allows considerable freedom in the compilation of expressions. Most common optimisations are implemented, for example using Inc to add one, and some reordering of expressions is performed to avoid temporaries. As OCCAM does not have functions there is no problem with side-effects. Since variables are indexed off IX, rather than SP, the stack is used for any temporaries that are needed. The more complex operations (multiplication,

```
VAR global:

PROC outer =
  VAR L1:

  PROC wrap =
    VAR L2:

    PROC inner =
      VAR local:
      SEQ

        global := 3
          |   LD        HL,3                value to be assigned
          |   LD        (UserRAM+0),HL      at a fixed address
        L1 := 2
          |   LD        HL,2                value to be assigned
          |   LD        BC,HL               save it
          |   LD        E,2                 the number of levels
          |   Call      P8                  call primitive
          |   LD        HL,BC               recover value
          |   LD        (IY+-4),HL          assign it

        L2 := 1
          |   LD        HL,1                value to be assigned
          |   LD        BC,(IX+0)           get static link
          |   Push      BC                  copy it to IY
          |   Pop       IY
          |   LD        (IY+-4),HL          assign
        local := 0:
          |   LD        HL,0                value to assign
          |   LD        (IX+-4),HL          in local frame

    inner:
  wrap:
outer
```

Figure 5–2: Different levels of variable access

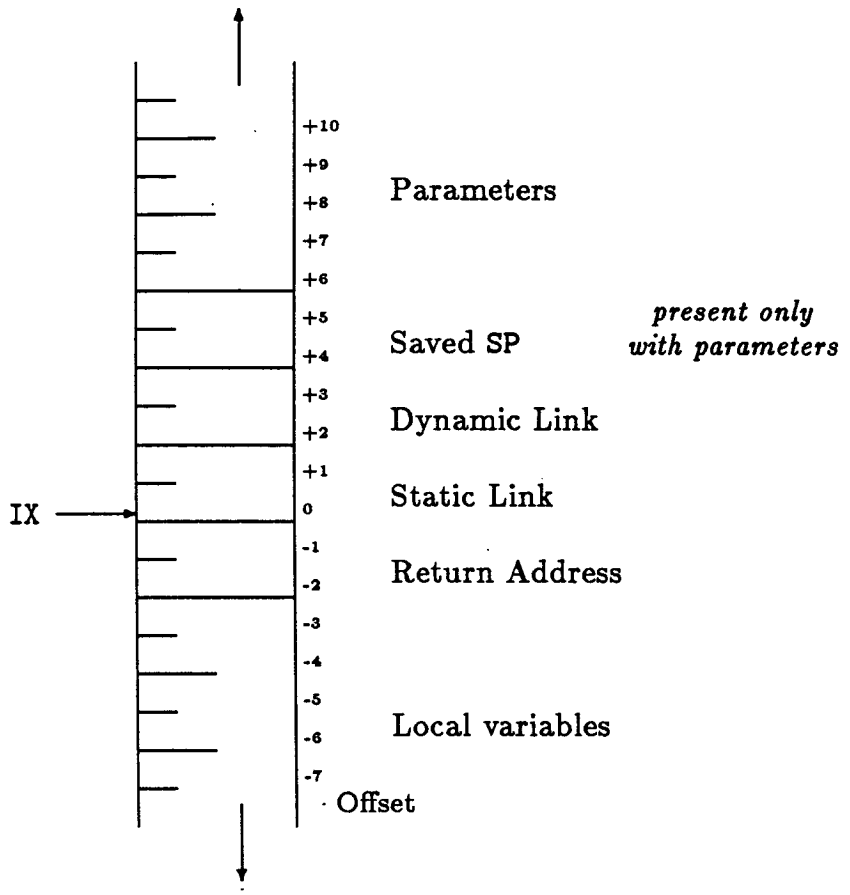


Figure 5–3: Stack-Frame Layout

division, and variable sized shifts) are implemented using primitive routines. Some example expressions are shown in Figure 5–6. Many other optimisations are possible, some of which are noticeable in the examples in this chapter, but could only be implemented at the cost of compiler modularity.

When a construct is entered, stack space must be created for any local variables, and any channels that are created must be initialised to the `Not.Process` value, which for this version is zero. For this reason, and to clear down any previous values in memory, the value zero is pushed onto the stack enough times to provide space for all the locals. How this is done depends on the amount of space to be claimed, either with several pushes, a loop, or a primitive procedure

```
PROC p (VALUE e, VAR s, v[]) =
    :

p (10, simple, array)
    |   LD      (StackTmp),SP      save current value of SP
    |   Push    IX                 copy frame pointer
    |   Pop     HL                 to HL
    |   LD      BC,-24             add frame offset
    |   Add     HL,BC              for array
    |   Push    HL                 push it as parameter
    |   Push    IX                 copy frame pointer
    |   Pop     HL                 to HL
    |   LD      BC,-4              add frame offset
    |   Add     HL,BC              for simple
    |   Push    HL                 push it as parameter
    |   LD      HL,10              push 10
    |   Push    HL                 as parameter
    |
    |   LD      HL,(StackTmp)      load old stack value
    |   Push    HL                 and push it
    |   Push    IX                 push dynamic-link
    |   Push    IX                 push static-link
    |   LD      (StackTmp),SP      set up new frame pointer
    |   LD      IX,(StackTmp)
    |   Call    R16                call procedure
    |   Pop     HL                 discard static-link
    |   Pop     IX                 restore frame pointer
    |   Pop     HL                 restore old SP
    |   LD      SP,HL              to skip parameters
```

Figure 5–4: An example procedure call

```
array[s] := 56
    |   LD      HL,56          the value
    |   LD      BC,(IX+6)      get index
    |   SLA     C              convert to words
    |   RL      B
    |   Push    IX             get frame pointer
    |   Pop     IY             into IY
    |   Add     IY,BC          add index
    |   LD      BC,-402        add frame offset
    |   Add     IY,BC          of array base
    |   LD      (IY+0),HL      assign it
```

Figure 5–5: A local array access

```
a := a + 1
    |   LD      HL,(IX+-14)    load a
    |   Inc     HL             add one
    |   LD      (IX+-14),HL    assign new value
a := b + 2
    |   LD      HL,(IX+-12)    load b
    |   Inc     HL             increment twice is shorter
    |   Inc     HL             than load 2 and add
    |   LD      (IX+-14),HL    assign it
a := b - c
    |   LD      BC,(IX+-10)    load c
    |   LD      HL,(IX+-12)    load b
    |   And     A              clear carry bit
    |   SBC     HL,BC          subtract with carry
    |   LD      (IX+-14),HL    assign it

 a := 89 + ((b*c)>>6)
    |   LD      HL,(IX+-12)    load b
    |   LD      DE,(IX+-10)    load c
    |   Call    PO             call multiply primitive
    |   LD      B,6            number of shifts
    |  L1:                     top of shift loop
    |   SRL     H              do the 16-bit shift
    |   RR      L
    |   DJNZ    L1             loop until B = 0
    |   LD      BC,89          for adding
    |   Add     HL,BC          do the add
    |   LD      (IX+-14),HL    assign it
```

Figure 5–6: Example Expressions and their code

call. Space is recovered at the end of the context by either the required number of pops, or by direct arithmetic on the stack pointer, which ever needs less code.

This implementation fully supports the slice operations. These are implemented using the Z80 block move instructions packaged in primitive routines, which also check that the slices are of equal length.

5.3.2 Concurrent Occam

The concurrent statements operate in a similar way to that of the interpreter. Primitive routines are used to implement most of the operations, which results in compact code. Again the `ALT` construct is the hardest to implement. Rather than have a list of channel-transfer records, one for each guard, there is a single record in the PCB which is shared by all. Two loops are used, one to setup and one to clear-down. Deciding which guard has fired is done by examining each channel, with the active one having another process's PID in it. Since the channel-transfer record shares PCB space with the wake-after field, timer guards are tested by examining the clock directly. The ZERO scheduling routines automatically remove a process from the timer-queue if it is put on the run-queue, avoiding double scheduling.

The special channels `Keyboard`, `Screen`, and `Terminal` are supported. These are very simple to implement for direct use, for example `Terminal` output statements are compiled to primitive procedure calls, but when they are passed as parameters it is more complex. The only restriction is that `Terminal` cannot be passed. The others are passed as their `AT` value. As these addresses are in the space occupied by ZERO they cannot be confused with ordinary channels, so each primitive routine supporting channel communication tests for these addresses.

A number of test programs are supplied with the portakit to verify that interpreters are fully functional. One of these programs uses an array of processes to simulate thermal conduction in a metal plate. Heat is applied at one point, and the temperature rise across the plate is displayed on the terminal via a

screen-mixer process. This was used to test the concurrency features of the compiler and ZERO: a ten-by-ten array was successfully run in the available store, using a total of 142 processes.

5.4 Interfaces

Programs can be compiled either for execution on the exercise-machine, or for inclusion in a specially built stand-alone system. In the first case the program must be adapted to fit into an existing environment, and in the second case, an environment must be created for the program. The first task to be performed in either case is matching software specified interfaces with the required hardware, the *allocation* phase.

The information required for each interface is its address, any configuration data, and if it uses interrupts, an interrupt vector, a queue for EVENT inputs, and a buffer for the incoming data. On the exercise machine these are all fixed, but for the stand-alone compilation these values must be calculated and added to a *hardware-requirements list.* For stand-alone machines, interface addresses are allocated sequentially from one (zero is always the CTC). Interrupt handlers are created semi-automatically, with the interface designer supplying a piece of code which clears the interrupt for the device. This is not done in the OCCAM SYNTH body as it requires interrupts to be disabled, and that would require further OCCAM extensions. At entry to this code, register C contains the address of the device, and the data that raised the interrupt must be loaded into register A. The rest of the handler is then compiler generated. For the analogue-to-digital converter the designer-coded handler consists of one IN instruction. For the DUART it adds three to register C to select the appropriate internal register and then executes an IN; a total of four instructions. If a device needs to be initialised with the interrupt vector, for example a Z80 PIO or CTC, another portion of interface-designer written code can be included.

```
HARD [A.Control] !  Control.Mode
     |   LD        BC,1              value of A.Control
     |   LD        IY,(CurrPRC)      get pointer to PCB
     |   LD        L,(IY-3)          load interface address
     |   LD        H,0               clear top byte
     |   Add       HL,BC             add index
     |   LD        C,L               copy it to reg for Out
     |   LD        A,207             the value of Control.Mode
     |   Out       (C),A             output the value

HARD [Receive.Reg] ?  Data
     |   LD        BC,5              value of Receive.Reg
     |   LD        IY,(CurrPRC)      get pointer to PCB
     |   LD        L,(IY-3)          load interface address
     |   LD        H,0               clear top byte
     |   Add       HL,BC             add index
     |   LD        C,L               copy to reg for Out
     |   In        A,(C)             read the interface
     |   LD        H,0               clear top word
     |   LD        L,A               to make data 16-bit
     |   LD        (IX+-24),HL       assign it to Data
```

Figure 5–7: HARD communications and generated code

```
% Build file RMM_U:DVM.BLD created on 24/07/86 at 10.30 %
?- technology(z80).
?- make(adc, 1, 8, []).
?- make(six_seven, 2, 0, [3]).
?- make(bar_graph, 3, 0, [16]).
?- ram(474).
?- rom(2340).
```

Figure 5–8: An example hardware-requirements list

The interface address and the address of the event queue (called the *action pointer*) are stored in the interface-process PCB. These are installed in the PCB by the PAR construct as the process-creation routine does not support this. Any PARs within the SYNTH body also copy the details into the subprocess PCBs. These values are needed to implement the HARD and EVENT channels. Figure 5–7 shows the code generated for HARD inputs and outputs.

EVENT inputs are more complex as they can involve descheduling the process. The action-pointer in the PCB holds the address of a three byte block, the first two bytes of which is a process pointer and can be either Not.Process, 1, or a PID. The third byte is any data waiting to be read. On executing the input, if the process pointer is Not.Process, the process puts its PID there and deschedules itself, to be woken by the interrupt handler. If it is set to 1, then an interrupt has already occurred and data is waiting. If there is already a PID there, then the channel is being shared, which constitutes an error. These actions are implemented by a primitive procedure, so an EVENT input compiles to a procedure call and a load instruction to save the incoming data.

The hardware-requirements list is in the form of a Prolog program which is used to drive the *Artificial Engineer*. It represents the abstract structural information extracted from the OCCAM program, plus the allocated addresses. An example appears in Figure 5–8, the ?- at the start of each line indicates that these are *queries*, in effect procedure calls, rather than definitions. The first statement identifies the synthesis technology. The make statements create interfaces; the first parameter is the hardware-identifier, converted to lower case and dots translated to underlines to meet prolog lexical requirements. The next parameter is the address, followed by the interrupt vector. The last parameter is a list of the configuration values, passed as CONFIG parameters. The last two statements specify how much memory is needed.

The size of ROM required includes a cut-down version of the operating system. This does not include transparent mode, nor does it provide any input-output routines or a keyboard interrupt-handler. If a design requires terminal input-output, a DUART must be included as a normal interface. The primitive

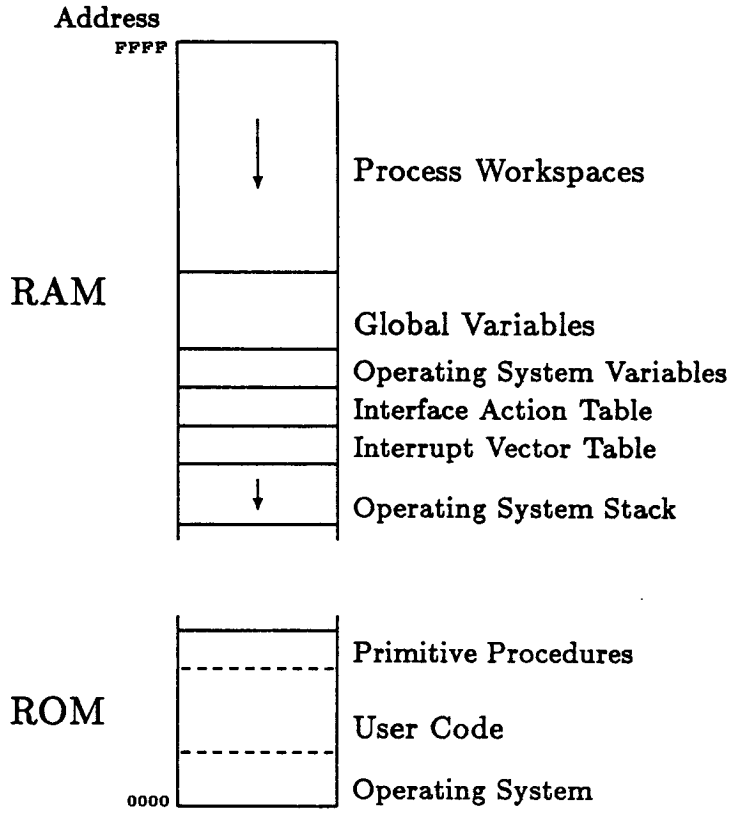


Figure 5–9: The memory map of stand-alone systems

routines are also simplified, as there is no longer any need to provide code to recognise `Screen` and `Keyboard`.

Stand-alone systems use the same store arrangement as the exercise machine: ROM at the bottom end and RAM at the top of the address range. The ROM has to start at address 0000 as that is the power-on reset address. The RAM could start immediately after the ROM, but placing it at the top means that in a minimum system the top address line, A_{15}, is sufficient to select between them. A memory map of a stand-alone system is shown in Figure 5–9.

Any error condition, such as channel sharing, causes the program to perform a complete restart. Control programs should be robust against this, and should not make assumptions about device states when initialising.

Chapter 6

The Artificial Engineer

An expert is someone who has made all the mistakes, which can be made, in a very narrow field.

Niels Bohr

The second part of a back-end is an *Artificial Engineer*. This takes the hardware-requirements list and fleshes it out to a description which can be used to drive fabrication tools. This is a process of hardware-assembly, and has two aspects. The first is technology independent, analysing the hardware-requirements list and communicating with the user in a consistent fashion. The second part deals with the implementation technology, making a processor-block and its memory, and then creating the required interfaces. The prototype back-end provides a simple technology-independent part, or *shell*, and a Z80-based part which generates Elementary Structural Design Language (ESDL) [Smith 80] files, which can be used to drive a solder-wrap machine that wires prototype boards automatically.

The fleshing-out process can be viewed as a straight-forward database look-up task, with a one-to-one mapping between interface specifications and implementations. This would result in a proliferation of interfaces, as each minor variation in implementation would be distinct. For example, an interface which reads a set of switches could include the switches on the circuit-board, or provide a connector to allow the switches to be mounted remotely. With the database approach each of these would require a different hardware-identifier, forcing the interface-designer to replicate the interface definitions for each variation in implementation. The system-level designer would have to change the source-program

to influence these implementation-level details, which are not relevant at that level of description.

An alternative is what is now called an *expert system.* These are programs which attempt to emulate the actions of human experts by capturing their "knowledge" and drawing "inferences" from it. Traditionally the structural information provided in the hardware-requirements list would be handed to a skilled design-engineer, who would then ask questions about *constraints* on the design, for example cost, power consumption or performance. From this information he would produce the detailed design. This is the essence of engineering: applying knowledge and experience to convert a high-level specification into a working device. We want to emulate this process and create an *Artificial Engineer.*

Most expert systems are written in special *production system* languages, such as OPS5 [Forgy 81]. The logic language Prolog [Clocksin 81] is also in wide use for knowledge-based systems [Yazdani 84], but they can also be built using conventional languages [Forsyth 84]. Prolog and Imp77 are available on the APM, so the choice was between them. I decided to use Prolog because it is *extensible*, allowing the shell to consult only the required parts of the rule base. Prolog does have a number of serious drawbacks (lack of modularity, rule-interpretation is order-sensitive, back-tracking is hard to control and costly to implement), but these are outweighed by the advantages of extensibility and interactive development.

The next section examines some existing hardware-design expert systems. Following that, Section 6.2 outlines the overall structure of the *Artificial Engineer* and describes the shell. Section 6.3 describes the base-module of the Z80 designer, while Section 6.4 covers the prototype interfaces. Finally some issues involved in maintenance of the knowledge-base are discussed in Section 6.5.

6.1 In Defence of Expert Systems

Expert systems have gained a bad reputation in some circles because of the unrealistic claims made for them. Too often they have been presented as a revolutionary concept that can solve all known problems. In reality, the knowledge-based approach is just another form of programming, and is well matched to some problems, but not to others. Computer-aided design is an area where the knowledge-based approach seems to be successful, and the literature reveals a number of interesting projects [Birmingham 86].

Expert systems for hardware synthesis is the only area which includes work at the very high level of specification used in the USHER system. MICON [Birmingham 84] starts with a hardware-requirements list and produces single-board computers. The systems MICON is intended to produce are identical to those with USHER, featuring a microprocessor, some ROM and RAM, and a few interface devices. It supports the Z80, TI-9900 and iAPX-186 processor families. MICON reinforces the hardware-first style of design by choosing the processor that it "thinks" most appropriately meets the designers specifications, although the designer must then specify how much memory is needed. Synthesis is performed using *templates*, which are predefined groups of components. Statistics are provided claiming that designs produced by MICON are three-quarters of the size of commercial boards, once the errors in its output have been manually corrected. No concrete evidence, such as circuit diagrams or photographs, is provided, but assuming that it has all been implemented, MICON is an impressive tool.

MAPLE [Bowen 83] is another tool to assist microcomputer design at a very high level. Its most significant feature is an *interviewer*, which consults the designer about what is to be produced until it has gathered sufficient information to create a system from existing, commercially-available boards. The choice of processor is determined by the designer selecting between assembly and high-level programming languages. Development cost and time can be supplied as

constraining factors, as can production cost. After completing the interview, a microcomputer configuration is proposed, along with any constraints that it breaks. If this is accepted, a block-diagram of each board is produced, along with details of board-interconnection.

Descart is described as an expert silicon-compiler [Gajski 86]. It accepts TI-HDL behaviour sections, which are compiled to a control/data-flow graph, which is further compiled to a register-transfer structural description and symbolic microcode to animate it. This is translated into a standard-cell network, from which silicon is created by a commercial package. Each of these stages of translation is performed by part algorithmic, part expert system programs. Each level of description is scrutinised by other expert systems. One is the *constraints allocator*, which accepts design constraints in a "planning" language and ensures that they are met. Each intermediate level of design representation is optimised by what is called a "critic". For example, at the register-transfer architectural level, registers which are not used simultaneously are concatenated to form register files. No examples are given, and it is not clear what stage of implementation this system is at.

The CMU-DA project, described in Chapter 2, includes an expert system, the Design Automation Assistant (DAA) [Kowalski 86], which translates Value-Trace representations of ISPS descriptions into a register-transfer level structural description. This has successfully been used to redesign two well-known computers, the PDP/8 and the IBM 370. The DAA has been modelled on the methods used by human designers, and each design subtask they perform is represented by a module in the system. The first task is to develop an approximate global structure and floor plan. This is refined by hardware allocators working at increasingly detailed levels. The current system uses 314 production rules, as well as a number of algorithmic portions written in C rather than OPS5, which has been used for the rest of the system.

Two systems compile register-transfer descriptions specified in DDL to standard-cell networks. One is the back-end of the OCCAM-to-CMOS project described in Chapter 2 [Maruyama 84]. The other system is the Design Expert (DE) devel-

oped at Nippon Telegraph and Telephone [Takagi 84], which creates datapaths, and then verifies that they match the specification given for them. The verifier provides two functions: the first is that it can check the output of the synthesis system; and the second is useful for upgrading designs. If a design has been built, and subsequently needs modification in line with a new specification, the verifier proposes "patches" that can be made to bring them into line.

Talib [Kim 86] is a very low-level system which takes a transistor, gate and interconnection list and produces NMOS layout using Mead and Conway design rules. This process is constrained by cell size, and port placement. Once more a combination of conventional algorithmic and and knowledge-based programming is used.

These systems demonstrate that the use of knowledge-based expert systems, while still at an early stage of development, is practical in computer-aided design. Recent papers all admit that some parts of their systems are more efficiently implemented algorithmically, and have used incompatible languages communicating via data files. The solution to this would be to develop either a production-rule language with algorithmic features, or provide knowledge representation facilities in a conventional language.

6.2 Structure of the Artificial Engineer

The prototype *Artificial Engineer* has been built to provide full generality; it can readily be extended to include other technologies. The *shell* provides three functions: technology selection, interface-designer invocation, and user interaction. Referring to the example build file on page 96, the first statement is a query to the *technology data-base*. This is a file of Prolog facts which is *consulted* (the Prolog term for reading a file of facts) when the *Artificial Engineer* starts. In this file there is a definition for all the technologies known to the *Artificial Engineer*. Each technology entry creates any base hardware that is always present, sets a default directory for finding interface designers, and reads

```
technology(z80) :-
        consult('rmm_ae:z80_utils'),
        consult('rmm_ae:z80_base'),
        consult('rmm_ae:ram'),
        consult('rmm_ae:rom'),
        make_default('rmm_ul:').
```

Figure 6–1: The z80 technology definition

any technology dependent utility procedures. Figure 6–1 shows the technology definition for the Z80 designer. The first statement consults some Z80 specific utilities, which are mainly concerned with generating the ESDL output. The next three consultations read in the Z80 base designer, which is described in the next section.

The last statement in Figure 6–1 sets the default directory for use with the make query. This shell function takes the name of the interface and reads a file of that name from the default directory to find a procedure to create the hardware for the interface, and then invokes that procedure. For example, the hardware-requirements list statement:

```
?- make(adc, 1, 8, []).
```

will consult the file `rmm_ul:adc.ae`, which should contain the definition of a procedure called `adc_make` which takes the interface address, interrupt vector, and configuration data as parameters.

It is in these `_make` definitions that the design knowledge is captured. Each `_make` is an expert in creating its own type of interface. A second aspect of expert-systems construction is *knowledge capture*. The prototype uses entirely manual capture, that is the designer must code-up his knowledge as Prolog rules. It is not clear how this process could be automated. The knowledge needed to create the interface is limited to adapting the circuitry to differing circumstances, but to build a board-level interface in the first place requires a very wide knowledge of available chips.

The base and interface-designer procedures can require additional information from the human engineer. These are constraints on the design, such as performance, power consumption or cost. An important factor in a control computer is how the interface chips are connected to the devices they control. To collect this information in a user-friendly style, each designer procedure asks the human engineer questions about the implementation issues. The details of the circuit are stored in an ESDL file, but a brief description of the circuit and connection information are printed on the terminal. The questions and the construction information form a *dialogue* between the human and artificial engineers. As an example, the dialogue that resulted in the board design for the cash register is shown in Appendix D on page 176.

6.3 The z80 Base Designer

A prebuilt "personality card" was used to provide the basic components of the exercise machine. This is obviously not possible for a stand-alone system, which must include the processor, clock and reset circuitry. This is shown in Figure B–4 on page 161. The base also includes the CTC and interface decoder, which are the same as those used in the exercise machine, and shown in Figure B–3. The complete controllers are built on single double-height eurocards, with 96-way edge connectors providing power and ground lines, as well as external connections.

The Z80 base designer includes the memory-subsystem designers. For simplicity, the maximum size of either RAM or ROM is restricted to 32k bytes, allowing the top address bit, A_{15}, to select between them. EPROM design is very simple as chips are available in all sizes up to 32K bytes, so only one device is ever needed. Device select for the EPROM is generated by the logical OR of A_{15} and $\overline{\text{MREQ}}$. The EPROM generator provides the option of using low-power CMOS or conventional devices; Table 6–1 shows the chips that are used.

Size	EPROM		RAM
(k bytes)	NMOS	CMOS	
1			4118
2	2516	27C16	6116
4	2732A	27C32	
8	2764	27C64	6264
16	27128		
32	27256	27C256	

Table 6–1: Memory chips used by the Z80 designer

It was decided to use static RAM as small amounts of memory are needed, and these devices are available in 8-bit widths, reducing chip count. Dynamic RAMs require a clean electrical environment, and this cannot be guaranteed in a control computer, and they are mostly one bit wide. The largest static RAM currently available is an 8k byte by 8-bit device, so up to four of these could be needed. In the simple case of only one device, a select signal can be generated by the logical OR of $\overline{\text{MREQ}}$ and the inverse of A_{15}. If more than one chip is needed, a 74LS138 three-to-eight decoder is used with address lines A_{15} to A_{13}.

6.4 The z80 Interface Designers

Chapter 1 examined some existing control computers, and from these identified some typical interfaces. Five of these have been built for the exercise machine, and are used as examples of *Artificial Engineer* designers. These are a seven-segment LED display driver, a general LED driver, a switch reading interface, an analogue-to-digital converter (ADC), and the DUART. Each is described below, with schematics appearing in Appendix B.

6.4.1 A Seven-Segment LED Display

Seven-segment displays are one of the most common interfaces in current use. They are extremely flexible, and can display any form of numerical data. The displays are constructed as eight light-emitting diodes (LED) (including the decimal point), with their cathodes or anodes connected, which implies that for n displays, $8n$ drive signals are needed, requiring a considerable number of driver devices. To avoid this, display digits are *multiplexed*, with the common cathode being used to select which digit is active, requiring only $8 + n$ drive lines. As long as the digits are refreshed regularly, the combined persistence of the LED and the eye are sufficient to give the appearance of constant illumination.

This device driving is normally performed by clock-interrupt driven software. This would, of course, be possible in a SYNTH body, but would impose a load on any computation-intensive tasks. To avoid this effect in the general case, the exercise machine uses a special-purpose display driving chip, the 74C917, which drives up to six digits of display and includes an internal oscillator for multiplexing. The device appears in the Z80 input-output address space as six registers which can be set using the Out instruction, or output to channel HARD in a SYNTH body. The only other hardware needed to support this device is an inverting buffer to sink the current on the display commons, which is performed with a 74H04. Figure B–5 on page 162 shows the circuit.

The 74C917 has two drawbacks, the first is that it can only display the "hex" digits 0 to F, with or without decimal point, and cannot blank off digits programmably. The second is that it is currently quite expensive. In an application where a software multiplexer would not result in an unacceptable degradation of performance, the Z80 parallel input-output (PIO) chip would be an ideal solution. This device provides two 8-bit ports, one of which could drive the segments, and the other could select digits, possibly via a decoder.

The Prolog code for this interface synthesiser is too long to appear in the body of the thesis, so it appears in Appendix C on page 164. It mainly consists of output statements generating the ESDL file, and uses very little of Prolog's

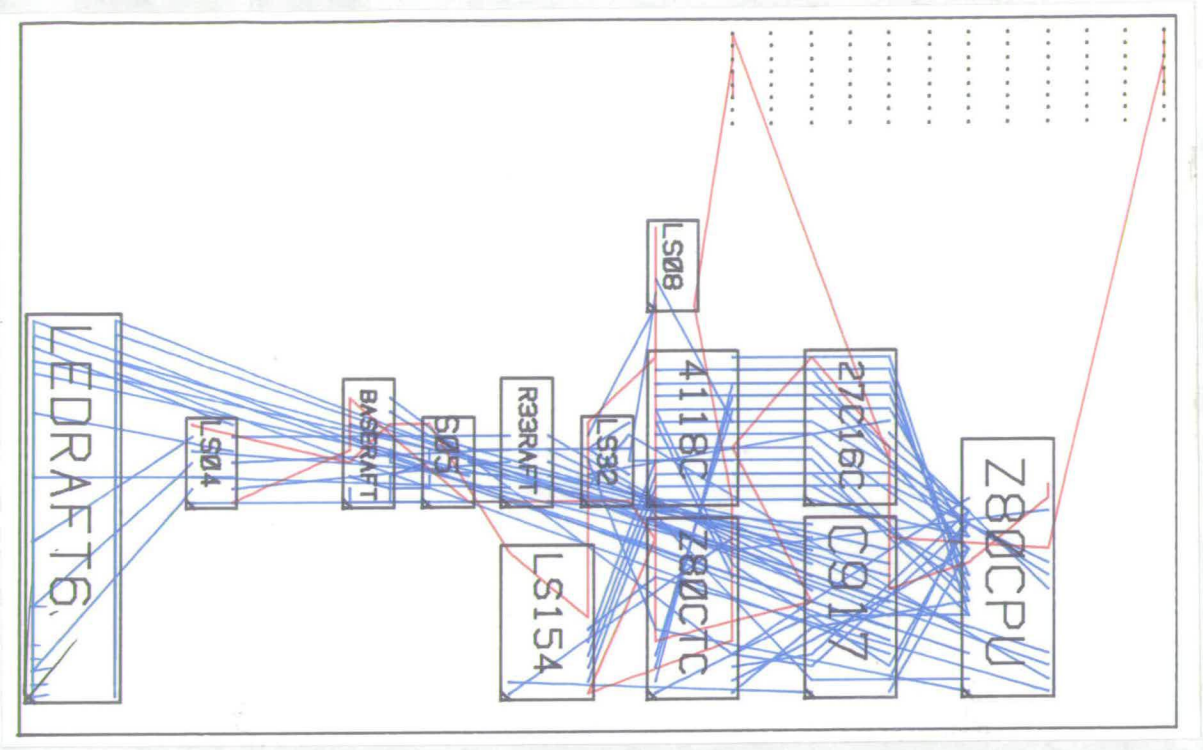


Figure 6–2: A board with six seven-segment displays

inference and back-tracking facilities. The configuration parameter list is used to determine how many digits of display are provided. The designer is presented with three options for the placement of the displays: on the board, connected to pins on the edge-connector, or via a 14-pin DIL connector. Other possible options include different heights of displays, and different current-limiting resisters to control brightness. Figure 6–2 shows the board layout produced for a system with the full six digits of display.

6.4.2 General LED Driver

This is a multi-purpose interface, and demonstrates how the same hardware can be made to appear as different interfaces by using several `INTERFACE` definitions with the same hardware-identifier. The implementation consists of a Z80 PIO, with its sixteen outputs directly connected to LEDs. The configuration parameter determines how many bits are actually used. The designer is offered a choice of using "bargraph" LED blocks on the circuit board, or placing the displays off-board via the edge connector or a 20-pin DIL connector. When using external

LEDs, the designer can select whether to use on-board current limiting resistors or not.

With the generality of this interface, it can be used for several purposes. Two possibilities are as a set of indicators or as a bargraph. In the examples given in the next chapter, the stopwatch has an indicator to show when it is in "lap mode", and the cash-register illuminates different LEDs to show what the value displayed on the seven-segment display is. The digital volt-meter, however, uses the LEDs as a bargraph, to provide an analogue scale. The code for the indicator `INTERFACE` is shown in Figure 6–3, and that for the bargraph in Figure 6–4. An example board, using bargraph mouldings, appears in Figure 6–5. Two displays are present, one of sixteen, and the other of eight elements.

6.4.3 Switch Reader Interface

This is another very simple interface, consisting of a PIO, with each of its inputs connected to a pull-up resistor and a switch. This allows the switches to be read independently of each other. The *Artificial Engineer* provides four synthesis options. The switches can be off-board, wired up either by way of the edge connector or a DIL connector. Alternatively, there can be either push-buttons or DIL switches on the board. Figure 6–6 shows a board with six push switches, while Figure 6–7 uses a 20-pin DIL connector.

6.4.4 The DUART

The stand-alone DUART circuit is the same as the one used in the exercise-machine, except that the interrupt-vector address is selected by direct connections to power and ground rather than a DIL switch. The circuit appears in Figure B–2. The discrete components used with the DUART are included in the ESDL description by means of a *raft*, which is a DIL component-carrier. This interface allows the serial signals to go off-board either via the edge connector, as shown in Figure 6–8, or spare pins on the raft.

```
INTERFACE Indicator (CONFIG Number, CHAN Switch[]) =
  SYNTH Z80, Bar.Graph
    -- set up PIO as desired
    HARD [A.Control] ! Interrupts.Off
    HARD [A.Control] ! Control.Mode
    HARD [A.Control] ! 0                    -- All outputs
    HARD [B.Control] ! Interrupts.Off
    HARD [B.Control] ! Control.Mode
    HARD [B.Control] ! 0
    -- now send it some initial data
    HARD [A.Data] ! 0
    HARD [B.Data] ! 0
    VAR State:
    SEQ
      State := 0
      WHILE TRUE
        ALT i = [0 FOR Number]
          Switch [i] ? ANY
            VAR Mask:
            SEQ
              Mask := 1<<i
              IF
                State/\Mask
                  State := State/\(NOT Mask)
                TRUE
                  State := State\/Mask
              HARD [A.Data] ! State/\#00FF
              HARD [B.Data] ! State>>8:
```

Figure 6–3: The indicator driving interface

```
INTERFACE Bar.Graph (CONFIG Bits, CHAN Show) =
  SYNTH Z80, Bar.Graph
    -- set up PIO as desired
    HARD [A.Control] ! Interrupts.Off
    HARD [A.Control] ! Control.Mode
    HARD [A.Control] ! 0                    -- All outputs
    HARD [B.Control] ! Interrupts.Off
    HARD [B.Control] ! Control.Mode
    HARD [B.Control] ! 0
    -- now send it some initial data
    HARD [A.Data] ! 0
    HARD [B.Data] ! 0
    WHILE TRUE
      VAR Data:
      SEQ
        Show ? Data
        HARD [A.Data] ! Data/\#00FF
        HARD [B.Data] ! Data>>8:
```

Figure 6–4: The bargraph driving INTERFACE

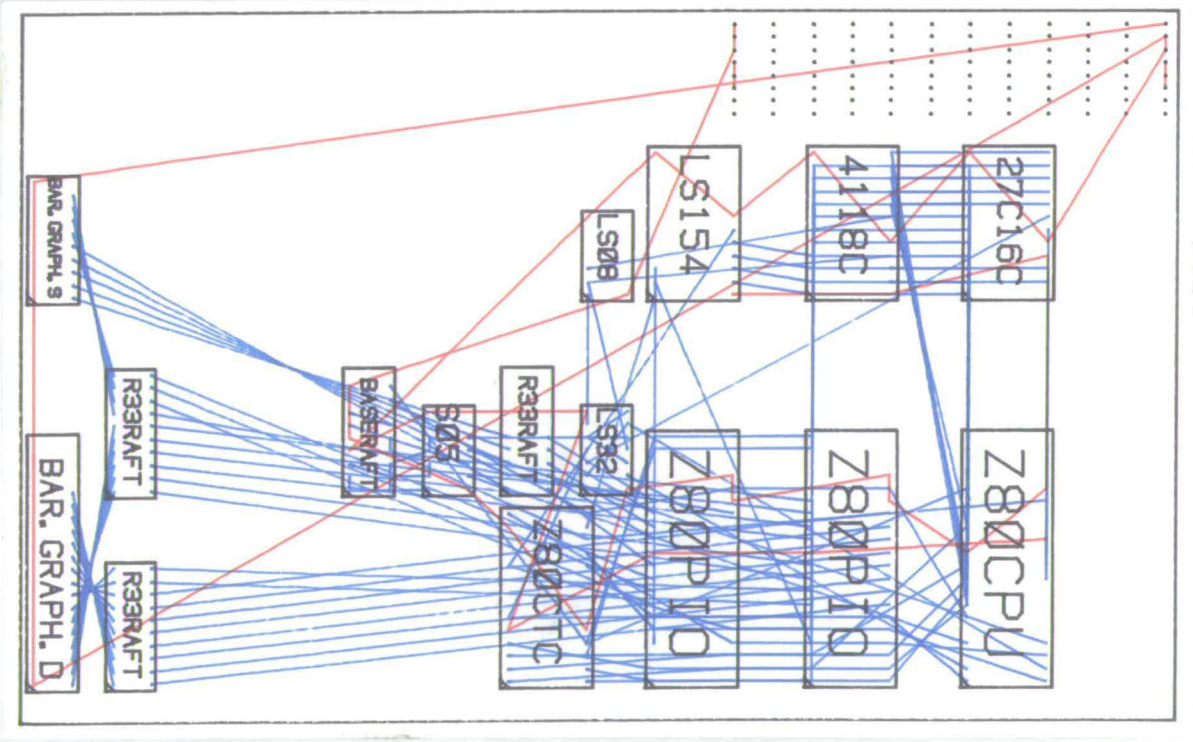


Figure 6–5: A board with two LED displays

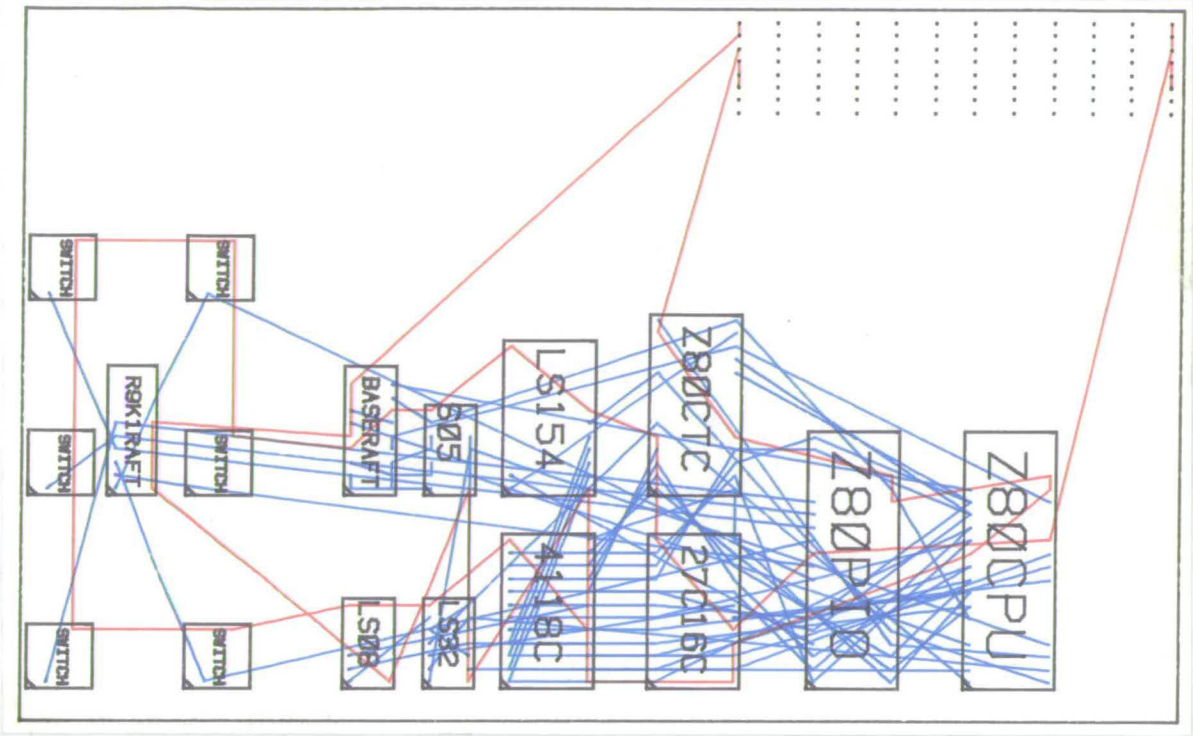


Figure 6–6: A board with push-buttons

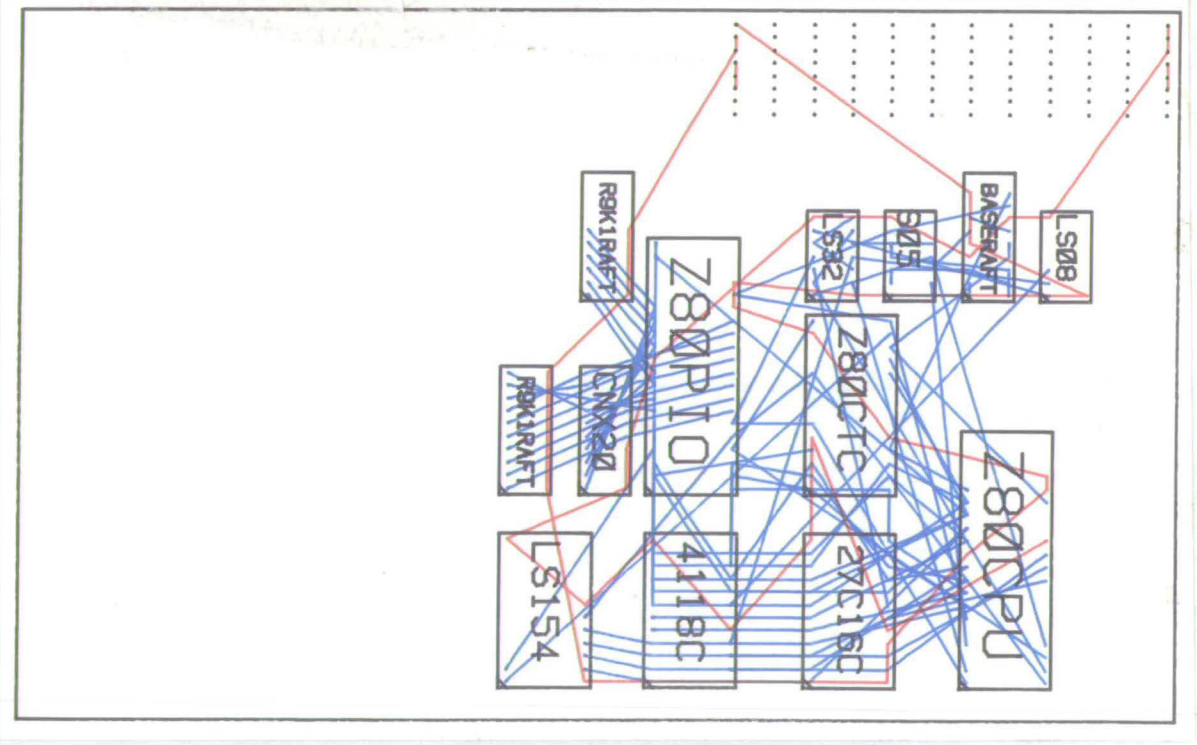


Figure 6–7: A board with switches connected via a 20-pin DIL connector

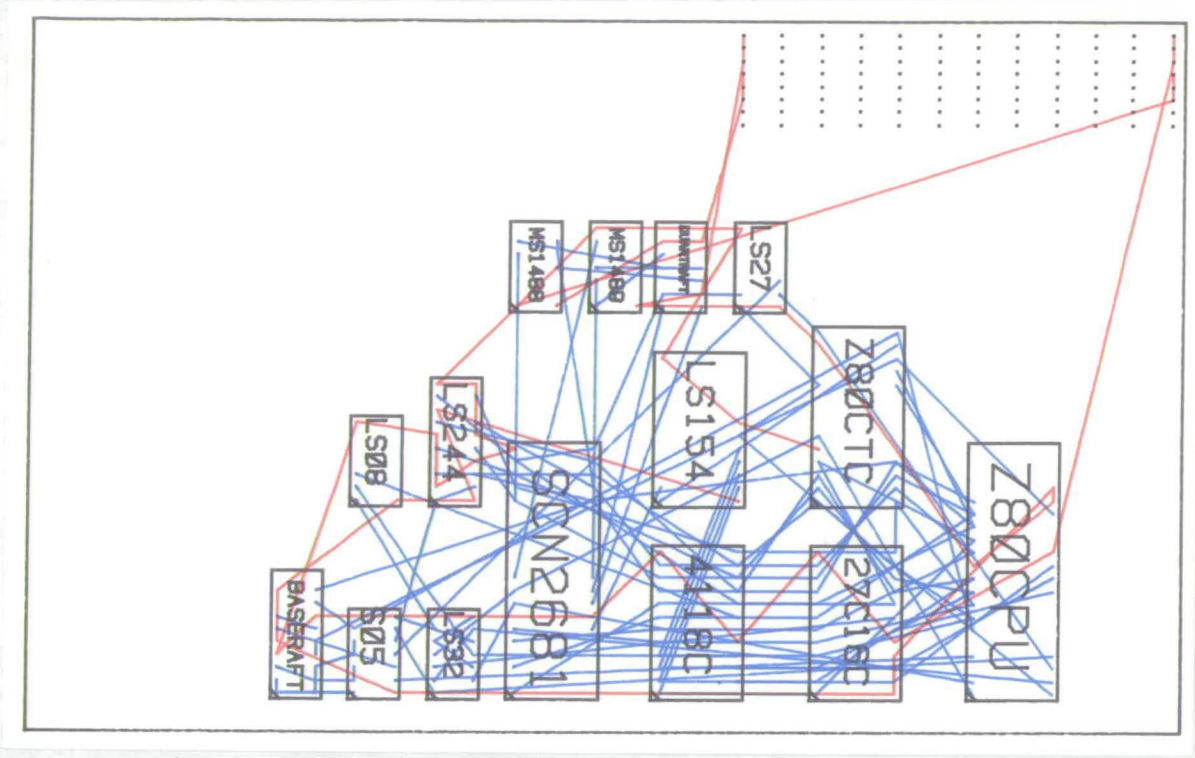


Figure 6–8: A board with a DUART

6.4.5 Analogue to Digital Converter

The analogue-to-digital converter is the most complex interface in the prototype *Artificial Engineer*. There are several families of ADCs, with differing properties and electrical interfaces. The device chosen was the ADC0804LCN, which is both compatible with the Z80 and inexpensive. A conversion is started by performing a write to the device, and it generates an interrupt when the data is ready. As this is not a Z80 -family device, it does not support mode two interrupts. This is done by the random logic shown in Figure B–6. The $\overline{\text{INT}}$ line of the ADC is connected to the Z80 $\overline{\text{INT}}$ via an open-collector buffer, allowing the device interrupt to trigger the vector circuit when the $\overline{\text{M1}}$ and $\overline{\text{IORQ}}$ signals are asserted. A board using an ADC appears in Figure 6–9. External connection is always via the discrete-components raft, and the *Artificial Engineer* reminds the designer to use screened cable to attach the transducer.

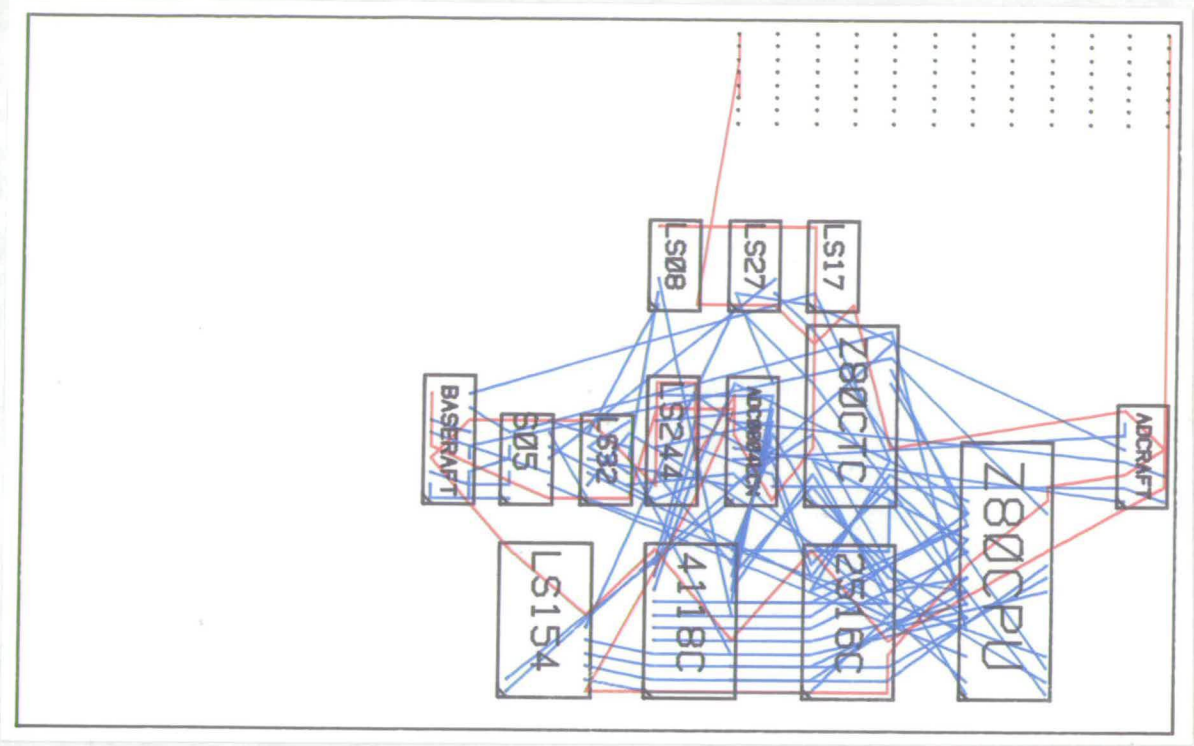


Figure 6–9: A board with an ADC

6.5 Maintenance

Any synthesis system will require updating as new devices and applications appear. In the USHER system two kinds of update can be identified: interface modifications and new interfaces. The latter kind requires the creation of a new `INTERFACE` definition, possibly in a new library, and a new *Artificial Engineer* interface designer. For devices which use interrupts it may also be necessary to create interrupt initialisation and handler code for the Z80 compiler.

Interface adaptation, for example allowing the use of multi-digit, daughterboard mounted displays with the seven-segment display interface, is performed by modifying the Prolog interface designer. The prototype designers use backtracking to form a chain of possible definitions, selected by the user. In the example in Appendix C, the clause `make_ss`, which defines the seven-segment displays or the appropriate connector, is defined three times: once each for on-

board, via the edge connector, or via a DIL connector. By making the last clause conditional with the `get_yes` clause, another option can be added at the end.

If major changes in implementation are required, for example moving to liquid-crystal displays (LCD), where the controller is different, the relevant SYNTH body must be changed. This identifies a weakness in the prototype implementation of USHER. For example, LCD and LED displays are functionally identical, and could be driven via the same INTERFACE definition, but the choice must be made in the control program as each implementation requires a different SYNTH body. This is not a problem intrinsic to the USHER approach, as allowing multiple SYNTH bodies per technology:

```
INTERFACE Seven.Segment (CONFIG Digits,
                         VALUE Initial,
                         CHAN Drive) =
  SIMUL
    :

  SYNTH Z80, LED
    :

  SYNTH Z80, LCD
    :
```

and then offering the designer the choice during linkage, would avoid the problem.

Chapter 7

Examples

Example is always more efficacious than precept.

Dr Johnson

Several test programs were developed to exercise specific parts of the USHER system, and a number of these have appeared in earlier chapters. To demonstrate the efficacy of the system with "real world" problems, however, three more complex examples were developed. The first two are a stopwatch and a digital volt-meter, two devices which are now commonly integrated onto single chips. Each uses three interfaces, has significant performance requirements, and has source code of under a page. The third example is a cash register, which is a device requiring considerable complexity of control. Each example is shown in source form, under simulation, and as a circuit board. They have all been thoroughly tested on the exercise machine, and perform as predicted by the simulator.

7.1 A Stopwatch

The stopwatch program provides facilities similar to those found on digital watches, or devices like the LCM7045 Precision Timer/Stopwatch chip. It displays time in minutes, seconds, and tenths of seconds on five seven-segment LEDs. It uses two switches, one to start and stop timing, and another which combines the reset and "lap" functions. A single LED indicator shows whether

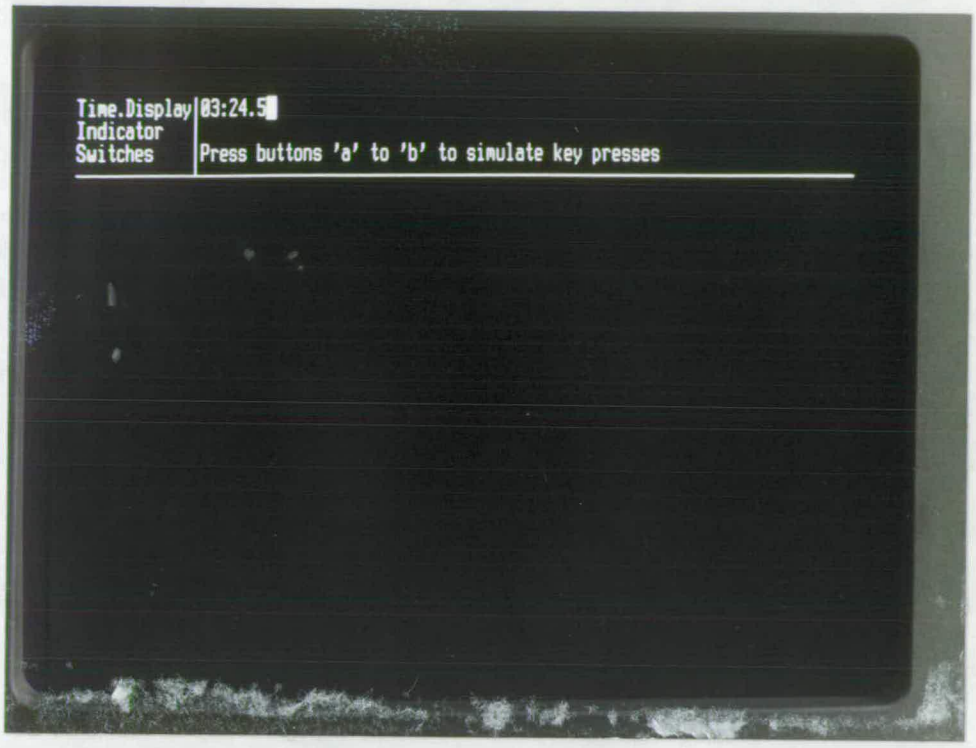


Plate 7–1: Simulating a stopwatch

lap mode is active. Figure 7–1 shows the source code for a single stopwatch, while Plate 7–1 shows it being simulated. Figure 7–2 contains the hardware-requirements list, and Figure 7–3 the board created by the *Artificial Engineer* from it. In this case all the interfaces are on the circuit board.

OCCAM's replicated PAR construct allows the definition of arrays of processes, and this is extended to the instantiation of arrays of interfaces. In certain circumstances, such as motor racing, it is often required to time several activities at once. By modifying the main code of our stopwatch to use replicated PARs, as shown in Figure 7–4, it is possible to define an arbitrary number of timers by adjusting the value of `Watches`. Plate 7–2 shows eight stopwatches being simulated. Unfortunately the *Artificial Engineer* is unable to create a board for this as there is insufficient room on a single card.

The OCCAM channel `TIME` is used to control the time displayed. The value of the current time is stored in the 20ms units used by the internal clock, so this value is divided by five to produce a count in tenths of seconds. The `Time.Display` interface procedure converts this value to minutes and seconds

```
INTERFACE Time.Display (CONFIG Digits, VALUE Initial, CHAN Data) =
  FROM Seven.Segment:
INTERFACE Switches (CONFIG Number, CHAN Return []) =
  FROM Switches:
INTERFACE Indicator (CONFIG Number, CHAN Switch []) =
  FROM Bar.Graph:

PROC Watch (CHAN Start.Stop, Lap.Button, Display, Lap.Sign) =
  VAR Counting, Time, Last, Displaying, T:
  SEQ
    Time := 0
    Counting := FALSE
    Displaying := TRUE
    WHILE TRUE
      SEQ
        WHILE NOT Counting
          ALT
            Start.Stop ? ANY
              Counting := TRUE
            Lap.Button ? ANY
              IF
                Displaying
                  SEQ -- set the time back to zero and redisplay
                    Time := 0
                    Display ! Time
                TRUE
                  SEQ  -- first stroke - unlap the display
                    Displaying := NOT Displaying
                    Display ! Time/5
                    Lap.Sign ! ANY
        TIME ? Last
        WHILE Counting
          SEQ
            T := Last PLUS 5
            ALT
              Start.Stop ? ANY
                Counting := FALSE
              Lap.Button ? ANY
                SEQ
                  Displaying := NOT Displaying
                  Lap.Sign ! ANY
              TIME ? AFTER T
                SEQ
                  TIME ? T
                  Time := Time PLUS (T MINUS Last)
                  Last := T
                  IF
                    Displaying
                      Display ! Time/5
                    TRUE
                      SKIP:

CHAN Buttons [2], Lap.Sign [1], Display:
PAR
  Watch (Buttons [0], Buttons[1], Display, Lap.Sign[0])
  Time.Display (5, 0, Display)
  Indicator (1, Lap.Sign)
  Switches (2, Buttons)
```

Figure 7–1: Source code for the stopwatch

```
% Build file RMM_U:SW.BLD created on 24/07/86 at 10.28 %
?- technology(z80).
?- make(six_seven, 1, 0, [5]).
?- make(bar_graph, 2, 0, [1]).
?- make(buttons, 3, 0, [2]).
?- ram(554).
?- rom(3305).
```

Figure 7–2: The stopwatch hardware-requirements list

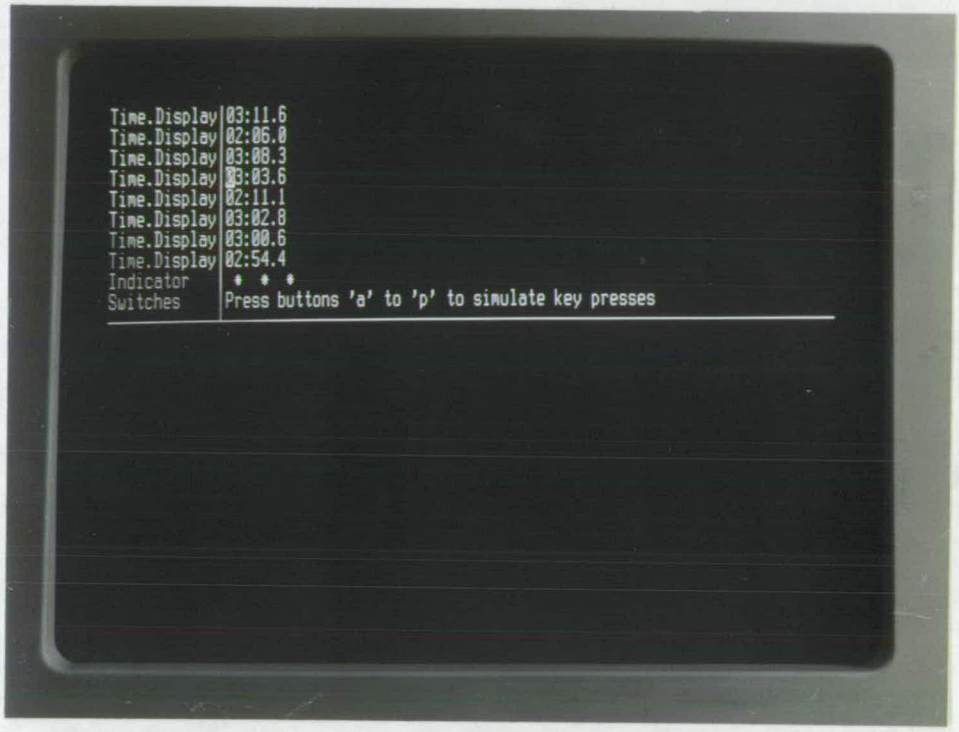


Plate 7–2: Eight stopwatchs being simulated

for display. The control procedure consists of two loops, the first is the idle state, waiting in an **ALT** for either the reset or start buttons to be pressed. When the start switch is pushed, control enters the other loop, which contains another **ALT**, with one branch for the stop button, one for the lap button, and one branch waiting 0·1s. If the timer branch is activated, this updates the time, and if not in lap mode, displays the new value. Note that it does not assume that it has been kept suspended for exactly 0·1s, but rather reads the time after reactivation and calculates the delay from that. This defensive practice ensures that the time is kept correctly when several stopwatches are running.

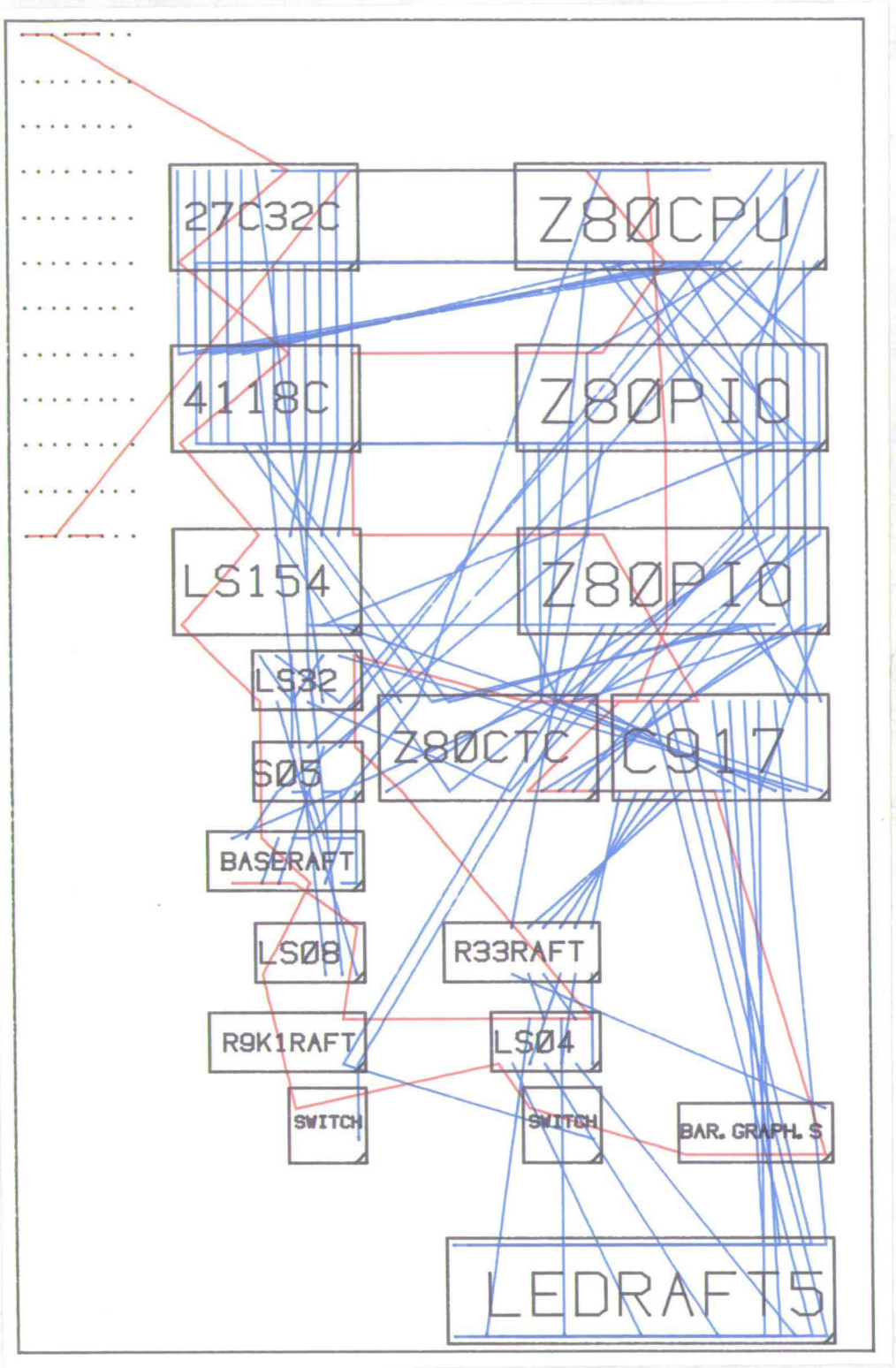


Figure 7–3: The stopwatch circuit board

```
DEF Watches = 8:
CHAN Buttons [2*Watches], Lap.Sign [Watches], Display [Watches]:

PAR
  PAR i = [0 FOR Watches]
    Watch (Buttons [2*i], Buttons [(2*i) + 1],
           Display [i], Lap.Sign [i])
  PAR i = [0 FOR Watches]
    Time.Display (5, 0, Display [i])
  Indicator (Watches, Lap.Sign)
  Switches (2*Watches, Buttons)
```

Figure 7–4: The main code for replicated stopwatches

7.2 A Digital Volt-Meter

This design uses the analogue-to-digital converter to read a voltage between 0 and 5 volts, then displays it on three seven-segment displays, and provides an "analogue" scale on a sixteen-element bar graph. Figure 7–5 shows the OCCAM code for the volt meter, and Plate 7–3 shows it running on the simulator. The simulation body for the ADC reads data from a file; the code for this appears in Figure 7–6 (the corresponding SYNTH body appears on page 51). OCCAM's asynchronous input can be used to build a program for generating such drive files interactively. Such a program is shown in Figure 7–7, and this was run on the simulator to generate the file used in Plate 7–3. Figure 5–8 on page 96 shows the hardware-requirements list for the digital volt-meter, and Figure 7–8 shows the board produced from it.

ADCs are notoriously unstable and sensitive devices, and the cheaper ones can give wildly varying readings. The volt-meter program must be proof against this, and provide a smooth but fast responding display. This is achieved by storing the last 32 samples in a circular buffer, and displaying the average value. To reduce "jitter" in the display, very small changes are ignored. The ADC data is a byte, so that the 0 to 5 volts are mapped onto 0 to 255. For simplicity the conversion is performed by multiplying by 2 and inserting the decimal point

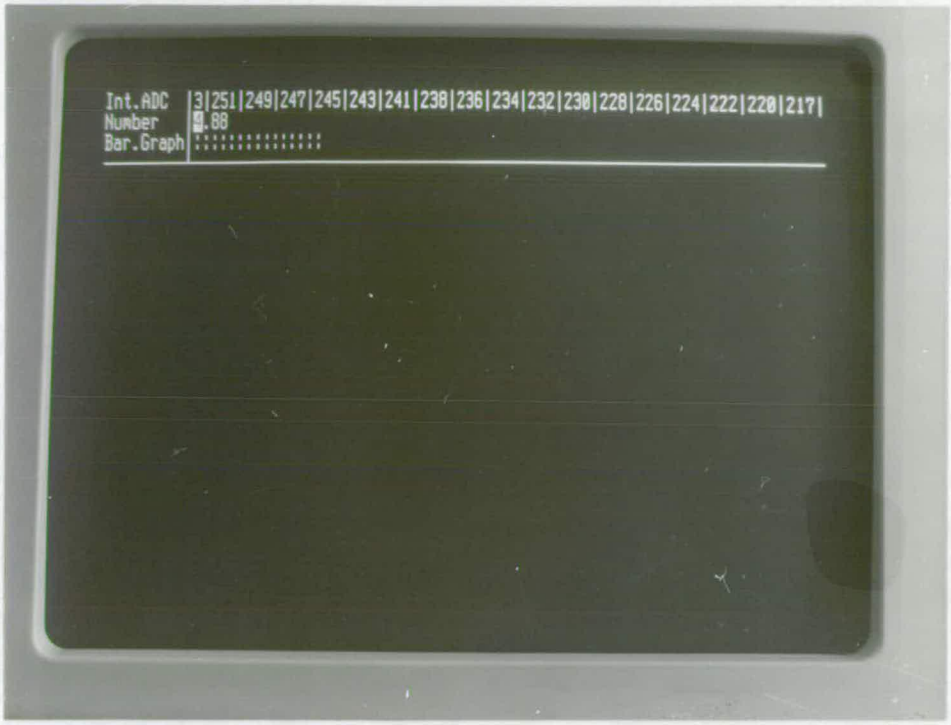


Plate 7–3: The digital volt-meter being simulated

appropriately. This results in a slightly high reading, and in a product would be corrected by altering the ADC reference voltage slightly. The bargraph pattern is created by bit manipulation.

```
INTERFACE Int.ADC (CHAN Request, Data) =
  FROM ADC:
INTERFACE Number (CONFIG Digits, VALUE DP, Initial, CHAN Data) =
  FROM Seven.Segment:
INTERFACE Bar.Graph (CONFIG Bits, CHAN Show) =
  FROM Bar.Graph:

PROC Controller (CHAN ADC.Request, ADC.Reply, Display, BG) =
  VAR Data, Samples, Queue [BYTE 32], QP, LD ,D:
  SEQ
    SEQ i = [0 FOR 32]
      Queue [BYTE i] := 0
    Data := 0
    Samples := 0
    QP := 0
    LD := 0
    WHILE TRUE
      SEQ
        ADC.Request ! ANY
        ADC.Reply ? D
        Data := (Data - Queue [BYTE QP]) + D
        Queue [BYTE QP] := D
        QP := (QP + 1)/\31
        IF
          Samples < 32
            Samples := Samples + 1
          TRUE
            SKIP
        D := Data/Samples
        IF
          ((LD - D) < 5) AND ((D - LD) < 5)
            SKIP
          TRUE
            SEQ
              Display ! D*2        -- scale it to 0 to 5V
              BG ! NOT ((-1) << (D/16))
              LD := D:

CHAN Go, Getter, Out, BG:
PAR
  Int.ADC (Go, Getter)
  Number (3, 2, 0, Out)
  Bar.Graph (16, BG)
  Controller (Go, Getter, Out, BG)
```

Figure 7–5: The digital volt-meter source code

```
INTERFACE Int.ADC (CHAN Request, Data) =
  SIMUL
    VAR Res:
    DEF FileName = "ADC.DATA":
    SEQ
      OpenRead (FileOut0, FileName)
      FileIn0 ? Res
      IF
        Res = OpenedOK
          SKIP
        TRUE
          SEQ
            PrintString (Screen, "ADC Data File open failed -> ")
            ReportError (Screen)
            STOP
      VAR D:
      WHILE TRUE
        SEQ
          Request ? ANY
          Read (FileIn0, D, TRUE)       -- echo it in micro-window
          Data ! D:
```

Figure 7–6: The analogue-to-digital converter simulation body

```
-- Program to generate a data file for the ADC to read

DEF FileName = "ADC.DATA":
VAR Res:
SEQ
  OpenWrite (FileOut0, FileName)
  FileIn0 ? Res
  IF
    Res = OpenedOK
      SKIP
    TRUE
      SEQ
        PrintString (Screen, "ADC Data File open failed -> ")
        ReportError (Screen)
        STOP
  VAR V, Going:
  SEQ
    V := 0
    Going := TRUE
    PrintString (Screen, "Type '+' to raise voltage, ")
    PrintString (Screen, "'-' to lower it, '.' to stop")
    Screen ! '*c';'*n'
    WHILE Going
      VAR Ch, T:
      SEQ
        TIME ? T
        ALT
          Keyboard ? Ch
            IF
              Ch = '.'
                Going := FALSE
              (V < 255) AND (Ch = '+')
                V := V + 1
              (V > 0) AND (Ch = '-')
                V := V - 1
              TRUE
                SKIP
          TIME ? AFTER T + 25
            SEQ
              Write (FileOut0, V, 0)
              FileOut0 ! '*n'
              Write (Screen, V, 0)
              Screen ! '*n'; '*c'
```

Figure 7–7: An OCCAM program for generating a drive file

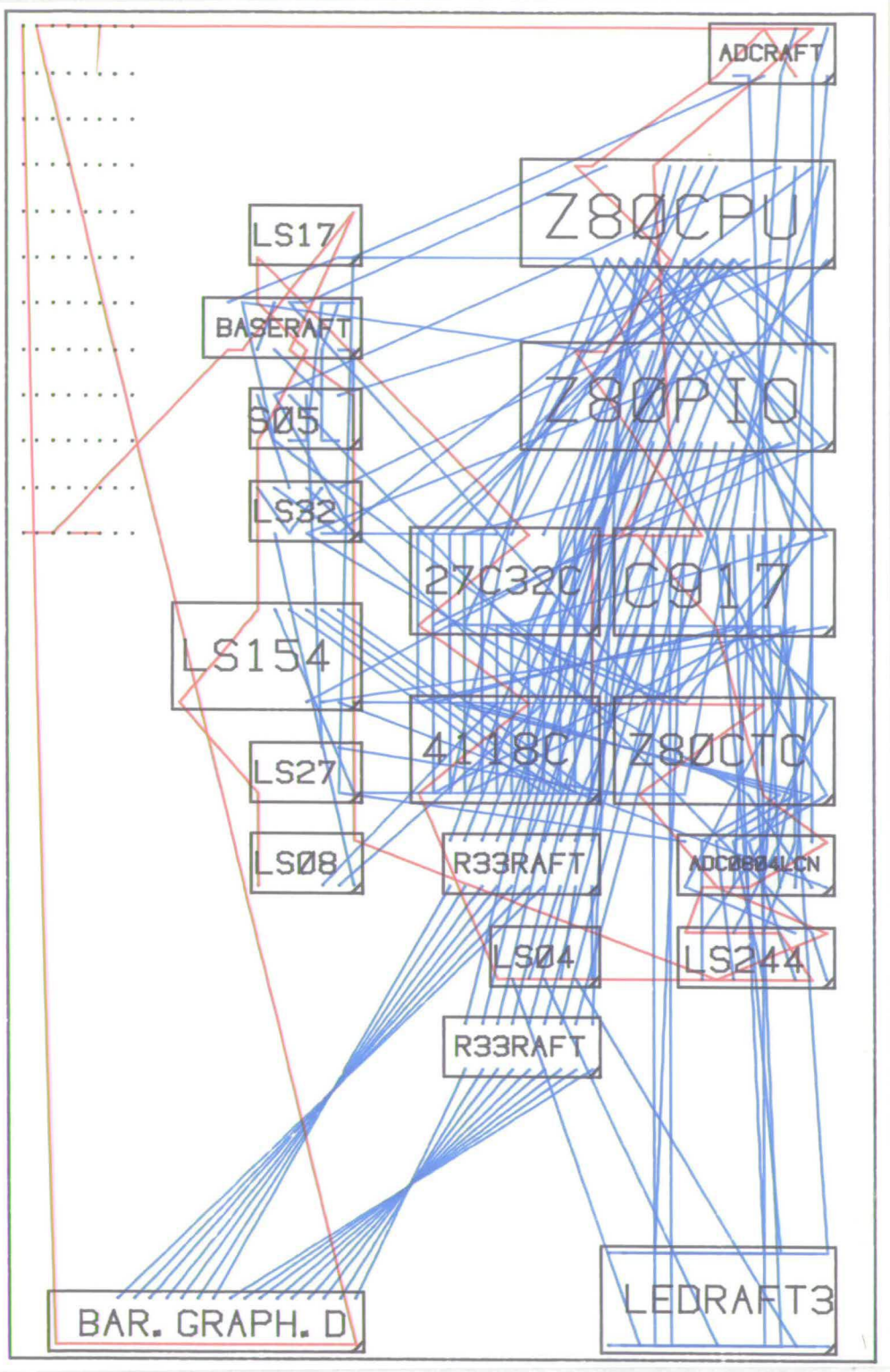


Figure 7–8: The circuit board for the digital volt-meter

```
% Build file RMM_U:TILL.BLD created on 07/09/86 at 13.53 %
?- technology(z80).
?- make(bar_graph, 1, 0, [6]).
?- make(six_seven, 2, 0, [5]).
?- make(duart, 3, 8, []).
?- make(buttons, 4, 0, [16]).
?- ram(994).
?- rom(12143).
```

Figure 7–9: The cash register hardware-requirements list

7.3 A Electronic Cash Register

The last and largest example is a typical electronic cash-register. It uses four interfaces: 5 seven-segment displays, 6 indicator LEDs, a 16-button keypad, and a printer driver. In a real system an extra interface would be needed: a solenoid actuator to release the lock on the cash drawer. The source code is too long to include in the body of the thesis, so it appears in Appendix D on page 168. Plate 7–4 shows the till being simulated, and Plate 7–5 shows it running on the exercise machine. The hardware-requirements list is shown in Figure 7–9, and the circuit board created from it in Figure 7–11.

The numeric part of the keypad is used for entering data, and the top six keys ('A' to 'F') select functions. One keys adds the current price to the bill, another allows the item to be repeated, another totals the bill and calculates the required change. Items can also be put into "classes" (groceries or stationery, for example), and the total value of goods sold in each class can be printed out at the end of the day. This is achieved via the "master" key, which acts like a shift, followed by another keystroke. Using this facility the sales assistant can identify himself with a number, and the size of the "float" of change can be set. All transactions are recorded on a printed receipt, examples of which are shown in Figure 7–10. Since the DUART interface was available, this was used to drive the printer, although it only uses one of the channels. Since the Z80 system uses

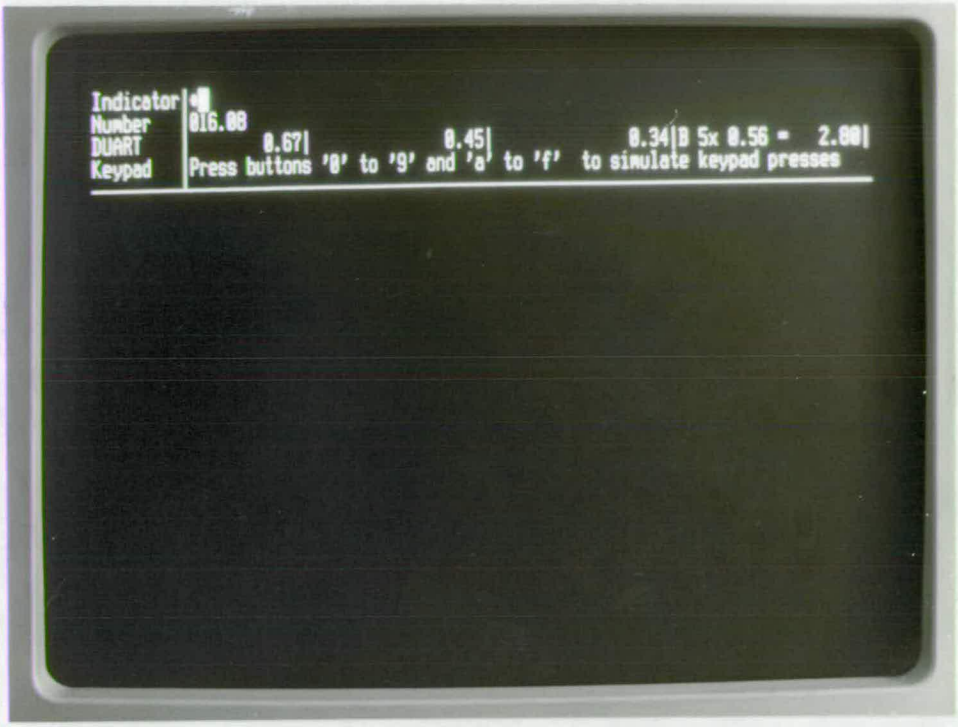


Plate 7–4: The cash register being simulated

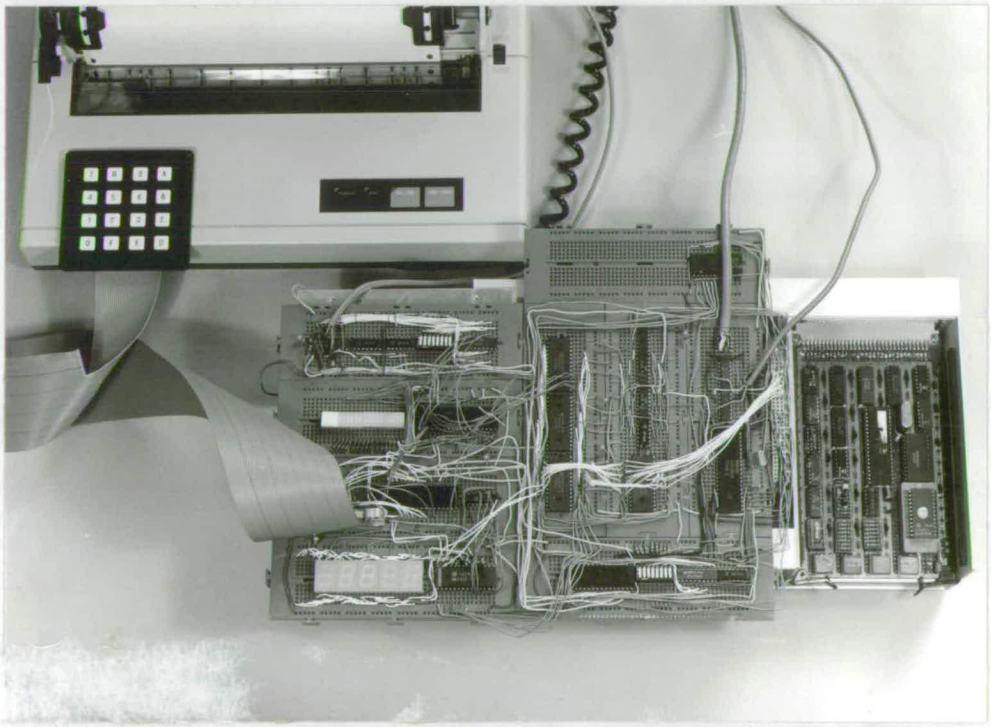


Plate 7–5: The cash register on the exercise machine

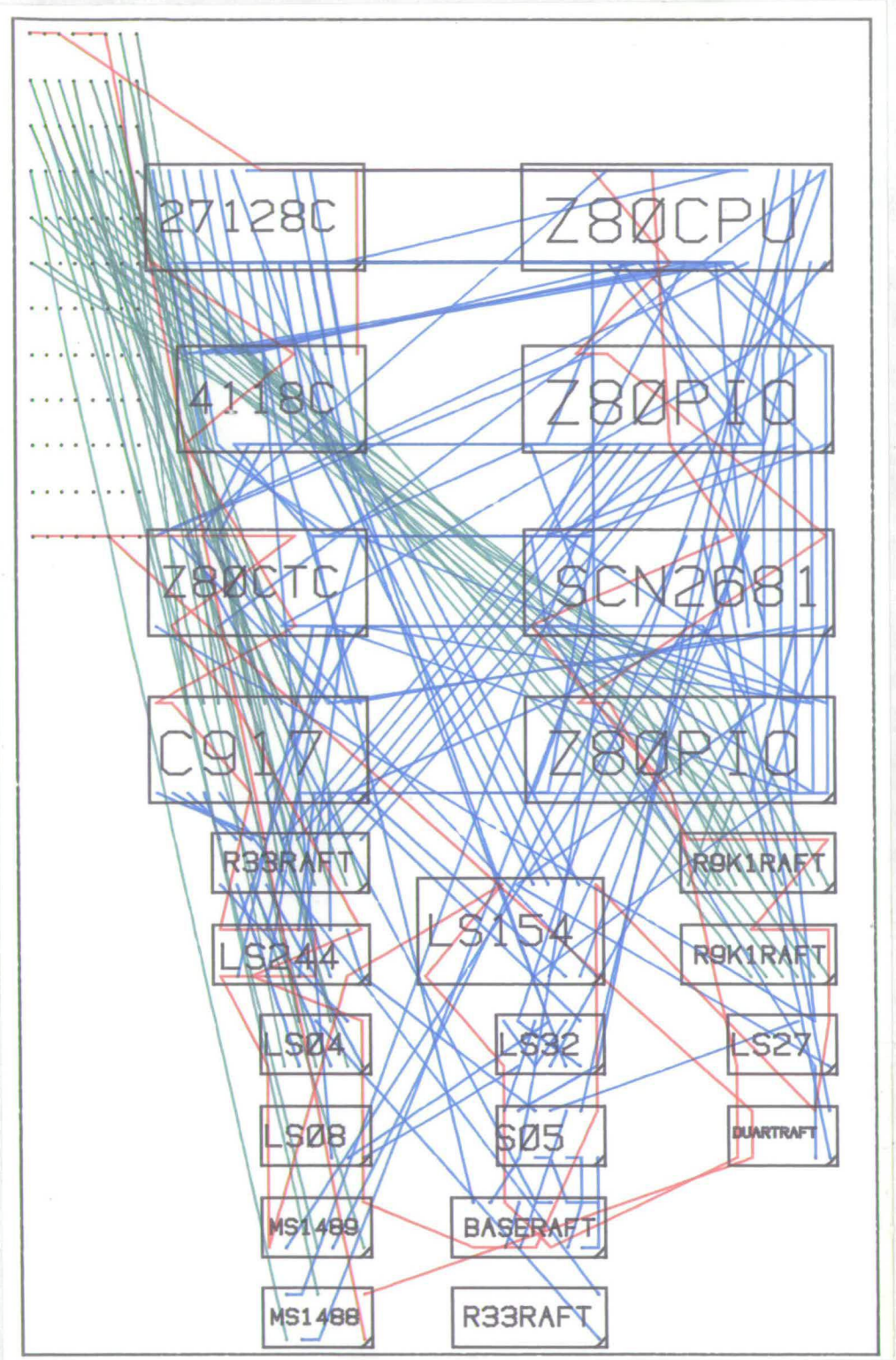


Figure 7–11: The circuit board for the cash register

Chapter 8

Conclusions

In my end is my beginning.

East Coker
by T. S. Elliot

This work is based on the hypothesis that control computers can be described completely by the software they are to run. The preceding chapters have described a suite of design tools, the USHER system, which can simulate and synthesise controllers from programs written in an extended version of OCCAM. The examples in Chapter 7 demonstrate that the USHER system works, thereby proving the hypothesis correct. The USHER system represents a significant step forward in tools for the design of systems that involve both software and hardware components. No hitherto published work can simulate and synthesise complete control-computers from a single design document.

The simulator allows designs to be tested and debugged prior to fabrication, and the high-level nature of USHER descriptions avoids some of the common problems with simulation. As the description is in the form of a programming language, simulation is by direct execution, and does not incur the speed penalties that are normal in simulation. The high level of description also reduces the quantity of output, and what is produced is readily intelligible. Not only is the description functional, but both input and output can be regarded as functional. For example seven-segment display output is shown as digits, not segment-driving waveforms.

The prototype synthesis system produces complete board-level, Z80-based microcomputers. These are fully functional, but designs would not be commer-

cially acceptable for reasons outlined in the following sections. This is due to implementation expedience, not flaws in the approach. The experimental version of USHER could still be used to produce prototype systems or one-off devices. It is particularly suitable for use in areas of scientific research where specialised monitoring and control equipment is needed. Here the function is often complex and obscure, and the task of conveying it to a design engineer difficult and error prone. USHER users need only understand programming to create a complete computer system, requiring no knowledge of hardware design.

The following sections examine each component of the USHER system in detail, identifying strengths and weaknesses. The last section outlines further work that could be done within the USHER framework, including an examination of other possible back-ends, and in particular a potential VLSI architecture.

8.1 The Language

The USHER input language can be viewed as two parts: standard OCCAM, and extensions. OCCAM was chosen for its concurrency facilities, and these have served well. Experience has shown that processes communicating via channels are an appropriate model for controllers, as proposed in Chapter 3. The small number of facilities offered by the language makes it easy to compile and amenable to theoretical manipulation, but it is not helpful for writing programs. Particularly lacking is any form of data typing or structuring facilities. OCCAM 2 has had types added, but these are very limited and are not user definable. As the control programs used with USHER are unlikely to be very large, these disadvantages are probably insignificant, and OCCAM is definitely a more appropriate language in which to implement concurrency than assembly code.

In order to use OCCAM to define control computers, we need to reference interface devices, modelled as procedures. This could be done using textual manipulation and by adopting lexical conventions, but this is inelegant. Instead, language extensions were defined that allowed the definition of imported proce-

dures from libraries, with dictionary-based, automatic linkage. These were used for the definition of an OCCAM runtime environment, and proved most effective. The extension was then itself extended to allow the definition of interfaces, both as import specifications and as library entries. Interfaces can then be instantiated using the same syntax as procedure calls. The only restriction imposed was forcing each instantiation to be in a distinct PAR branch. This is merely an inconvenience, however, as it does not limit what can be described, only the form that descriptions must take.

Interface library entries allow the definition of multiple code-bodies for each interface. One entry is a simulation body, and the others are for synthesis with different technologies. Each of these have a technology-identifier and an associated hardware-identifier. This syntax was successful, and is analogous to the multiple architecture bodies possible in a VHDL description. A problem associated with these definitions is the case where the code body changes with variations in implementation within one technology, as described in Chapter 6. This can be solved by allowing multiple SYNTH definitions for each technology, and delaying the binding of modules and code generation until after the hardware-requirements list has been drawn up.

The very high-level abstractions used in the USHER system allows the inclusion of different low-level design styles. In many cases, OCCAM could not be used directly to capture the structure of devices which can be successfully integrated as interfaces. The most obvious of these are the analogue converters, but it is also true for devices like RS-232 interfaces which have analogue line-drivers.

8.2 The Simulator

The simulator can also be divided into standard and extended parts. The conventional OCCAM interpreter has proved itself by supporting an undergraduate programming practical. Additionally, a suite of over forty test programs was developed to check the interpreter prior to release. The extensions not only include

support for the interface definitions, but the ability to vary default word-length and clock-rate. These facilites allow the simulator to model different fabrication technologies.

Simulation is definitely at a behavioural level, and does not attempt to provide accurate electrical information. The interface designer is required to use a low-level simulator to verify that interfaces work before they are used with USHER. By doing this, the USHER user can work at a high level without concern for implementation detail. As outlined at the end of Chapter 4, it is possible to abuse the simulator and make it produce false results, but this is true of all simulation tools. In normal use it provides an accurate model of the final environment, and correctly shows how control programs will operate.

In the hardware-first design style, control programs can be developed *in situ*. The purpose of the simulator is to support this development and debugging phase. To do this, it provides a variety of tracing and diagnostic facilities. These are superior to those provided by any other OCCAM system, and have proved useful for debugging both USHER programs and programming exercises. The screen driving facilities built into the simulator and the provision of a micro-window for each interface enhances the comprehensibility of simulator output.

8.3 The z80 Back-End

The Z80 microprocessor family was chosen on the basis of availability rather than desirability. Despite this inauspicious start, it proved a good research vehicle. The processor architecture is such that it is possible to generate native code of good quality from OCCAM. The code contains a number of optimisations, and these (combined with the clock frequency of 4MHz) ensure good performance. Further optimisations were possible, but were outside the field of research. On an 8-bit processor it is inevitable that hand-written machine code will always be faster and more compact than that generated by a compiler. Many USHER applications will not find this a problem as they do not require enormous speed.

But for some designs, high performance is essential for them to function at all, and the solution for these is to use a more powerful microprocessor. Section 8.5.1 discusses other microprocessors which could be used as a basis for USHER back-ends.

Unquestionably the most severe drawback in a Z80 based systems is the 16-bit address space. There will frequently be enough processing power available, but insufficient memory space. This is compounded in the prototype *Artificial Engineer* by using A_{15} to select between RAM and ROM, but this could easily be corrected leaving the 64k bytes total limit. The solution that is commonly used to work round this problem is *paging*. Here a range of addresses can be used to access several areas, or *pages*, of memory, depending on the state of a page-select register. The OCCAM compiler and operating system could be extended to implement this automatically. If the main code of a program is, for example, a PAR construct, each branch could reside in its own page of memory, with the executive selecting between them when it reschedules processes. Global variables and code can exist in unpaged memory and be available to all. The alternative and preferable solution is to use a processor with a larger address space.

The use of user-definable libraries of interfaces means that the back-end software does not "compile" the hardware-requirements list; it simply assembles it according to the designer's plan. This has the advantage that the back-end does not need to be changed when implementations change, nor does it restrict the range of devices that can be used by the designer. However, it does have one disadvantage: the back-end cannot optimise the hardware in any way. In the stopwatch example in Chapter 7, two Z80 PIOs are used: one to monitor two switches, and one to illuminate a single LED. Obviously a single PIO could perform both of these jobs and still have spare capacity. The interface designer could provide a bidirectional interface that would perform this function, but the back-end cannot automatically merge the two.

Whether to leave the "smart" part of the design process in the hands of the human engineer or to build it into the compiler is a difficult question. The technology that is being used must influence where the division is placed. In

the case of a board level system, such as this one, the range of possible interface devices is unbounded since it can include custom chips; so it is probably better to leave the interface designer free to choose which devices he or she offers the high-level designer. In a closed technology like VLSI, the range of available interfaces will be much smaller, and their use will be amenable to automated optimisation. Automation might even be essential to make designs fit into the available silicon area.

8.4 The Artificial Engineer

The prototype *Artificial Engineer* successfully produces single-board, Z80-based computers; but it does not reach its full potential as an expert system. This is due to the small number of options that are available with each interface, giving no room for automated decision making as there are none to be taken. The shortage of interface options is, in turn, due to a lack of resources and my inexpertise in hardware design. An electronic engineer with access to components with which to build prototypes would soon be able to construct an extensive library of interfaces.

The use of Prolog and a rule-based idiom has, however, been most effective. The shell structure, which loads interface-designer modules as required, palpably demonstrates the value of Prolog's dynamic syntax. The interactive development environment is also helpful for creating and extending interface definitions. My experience confirms the conclusions of the OCCAM-to-CMOS researchers at Fujitsu, that Prolog can be used to succinctly express algorithms and represent knowledge effectively, making it suitable for use in the construction of intelligent design-tools.

8.5 Further Work

This section identifies some directions in which the USHER system could be developed. The extended OCCAM input language is complete and would not benefit from further development. The extensions follow the OCCAM philosophy by providing the minimum sufficient features. Similarly the only work which could usefully be done on the simulator would be optimisation to improve performance. Different back-end technologies, however, can be developed without limit. The following two sections discuss first possible board-level technologies, and then a possible VLSI based synthesiser.

8.5.1 Other Standard-Part Processors

There are now many microprocessors on the market, with word lengths from 1 bit[1] to 32 bits. Any processor with a word length of 8 bits or greater could be used as the basis of an USHER back-end, but some are more appropriate than others. INMOS have coined the term *transputer* to mean a complete computer on a chip, comprising processor, memory and input-output connections. The term has become synonymous with the INMOS products, T414 and T212, but there are other devices which qualify for the title.

Several members of the Motorola 6800 family, for example, now have on-chip interfaces. The HD68P05W0 features an analogue-to-digital converter on chip, as well as a "piggyback" EPROM socket. From the same family, the HD63701X0C has a clock, serial input-output, 4k bytes of EPROM, 192 bytes of RAM, and 53 parallel input-output lines. The Z8 microprocessor has on-chip clock, two timers and serial communications. These and other devices are intended for use in control computers, and greatly reduce chip count in systems using them. The 6800 family offers a variety of different members with different configurations,

[1]This is the MC14500BP "Industrial Control Unit," which has 16 instructions.

more memory but less input-output for example, and the *Artificial Engineer* would be able to select the optimum device.

The only processor to offer the larger address space and improved performance of 16/32 bit words as well as these on-chip interfaces is the 68070. Preliminary data for this device was released at the end of 1985. It includes on-chip memory management, two direct-memory-access channels, a serial interface, two 16-bit timers and two 16-bit data-capture registers. At the time of writing this is the ideal chip for the USHER system. The 68000 processor is sufficiently powerful that the loss of efficiency caused by compiler-generated code is insignificant. The address space is 16M bytes, so removing the need for paging. The bus connection is compatible with the 6800 peripheral chips; so many interface devices are already available.

As stated at the start of Chapter 5, the INMOS T414 or T212 are the obvious processors to use with this system. They are fast, with a large address space, and are specifically designed for running OCCAM. They are also partially intended for use in control applications, as graphics- and disc-control transputers are available, but unfortunately they were too expensive to be used. A disadvantage of the transputer is that non-INMOS peripherals must be connected via a serial link and a link adaptor. Transputers only provide four links, so if more than four interfaces are needed, they require either two link adaptors or transputer machine code to drive them. Because OCCAM does not provide any means of explicit memory access, if the interface itself is memory-mapped in the address space, it must be driven by machine code. Alternatively the ! and ? operations can be used if a link adaptor is placed in the address space, followed by a second adaptor to connect to the interface.

8.5.2 A VLSI Architecture

Chapter 1 describes the use of core-processors; that is, the use of a standard-part microprocessor as a subcomponent of a chip. An *Artificial Engineer* could be built to create chips in this style, but it does not make good use of the medium.

Chapter 2 describes a number of design systems which take a behavioural description of a task and produce a datapath and microcode to animate it. This is suitable for small tasks, typically the definition of a microprocessor intended to run other programs. For a task of larger size, the size of both the datapath and the microcode becomes unwieldy. An alternative approach has been developed.

This new approach is embodied in the Plex system, developed at Bell Laboratories [Buric 83, Buric 84]. This is based on a pre-designed, modular microcomputer, with variable word length, number of registers, and size of ROM and RAM. The input is in the form of a C program, which is compiled to an intermediate code which is examined to see what hardware resources are needed. For example, if the program does not use the exclusive-OR function, then the ALU need not contain that logic. Having calculated the number of registers and required operations, the instruction set is created from a set of templates. This is then used to generate code for the program and to define the datapath and microcode. In this way, a manually designed processor is optimised for a particular application.

The INMOS generic transputer architecture, shown in Figure 8–1, is already modular and so ideally suited to this type of compilation. The instruction set is designed to work with any word length that is a multiple of eight bits, and the processor-block microarchitecture is also highly modular. Since the transputer uses an evaluation stack, there is no need to adjust the instruction format to allow for different sizes of register files. The hardware-requirements list could be extended to include processor configuration data, and the *Artificial Engineer* could generate a custom processor-block. By implementing the process-to-processor configuration parts of OCCAM, multiprocessor control systems could be built, using the serial links for communication. The links could also be used for maintenance and diagnostic purposes, with the service program running on a separate transputer, and plugged in only when required.

The R^2 system [Widdowson 84], on which I collaborated, provided a Prolog-based environment for the development of VLSI design tools. It provided facilities for the definition of geometric items and cells as Prolog facts. These could be

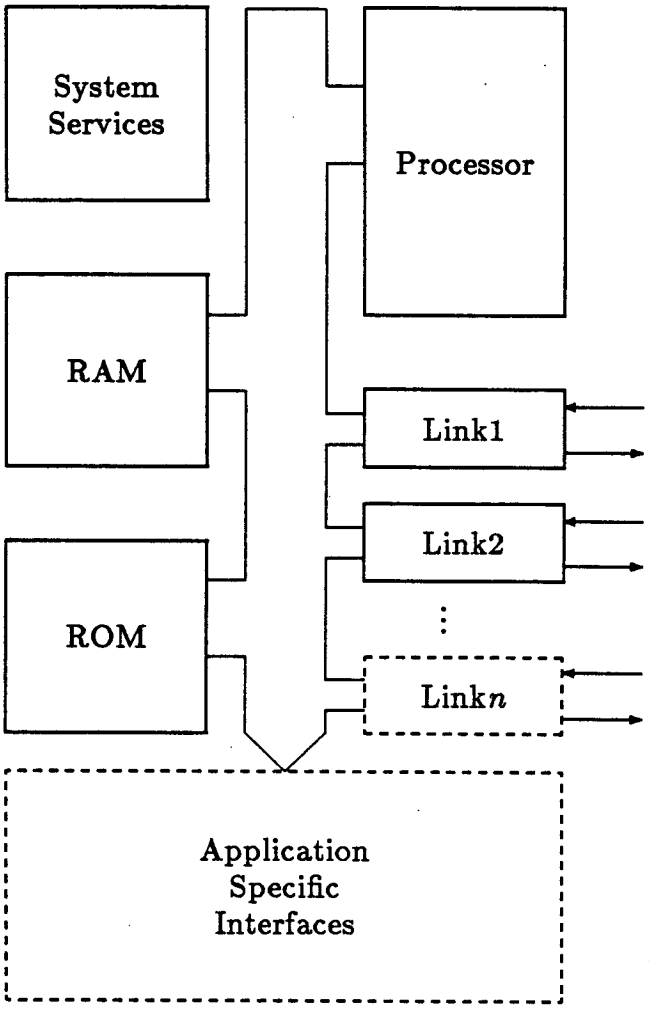


Figure 8–1: The Generic Transputer Architecture

manipulated by the applications program, or by procedures in the system. The design could be stored as a Prolog program, or dumped as CIF. By providing facilities for the construction of user interfaces, such as menus and command line interpretation, R^2 encouraged consistency in the appearance of design tools.

Electronic design at the silicon-level is much harder than at the board-level. When designing with standard parts, many functions which the silicon designer must create for herself already exist. It follows that fewer engineers will be able to develop interfaces for a VLSI based USHER system. Various methods of semi-custom design have been developed to help combat this problem, and these could be integrated with an USHER system generating custom transputers. The Lattice Logic Chipsmith system [Lattice 84a] generates gate arrays from a structural description. Using this system interfaces could be implemented as small gate arrays, and driven as channels via a standard logic-block. Interfaces defined like this would not require SYNTH bodies as they respond directly to channel communications. It would also be possible use the Switchsmith simulator [Lattice 84b] instead of SIMUL bodies. This simulator has a procedural interface, which could be connected to the channel-communication code of the USHER simulator, creating a mixed structural and behavioural simulator.

Bibliography

[Ainscough 85] John Ainscough and Alistair Brightman. An OCCAM compiler. OCCAM *User Group Newsletter*, (3):7–12, Summer 1985.

[Amini 85] Hamed Amini, Steve Hunt, Rod Rebello, and Mark Schuelein. ASIC design with microcontrollers in INTEL's iCEL™ design system. In *Custom Integrated Circuits Conference*, pages 442–446, IEEE, May 1985.

[Azuma 80] N. Azuma, K. Mohri, and E. Funaki. Microprocessor controlled variable playback for video tape recorder. *IEEE Transactions on Consumer Electronics*, CE-26(1):121–128, February 1980.

[Baldwin 84] Doug Baldwin. *Automatic Evaluation of Design Choices in Digital Controller Synthesis.* PhD thesis, Department of Computer Science, Yale University, December 1984.

[Barbacci 81] Mario R. Barbacci. Instruction set processor specifications (ISPS): the notation and its applications. *IEEE Transactions on Computers*, C-30(1):24–40, January 1981.

[Barbacci 85] Mario R. Barbacci, Steve Grout, Gary Lindstrom, Michael P. Maloney, Elliott I. Organick, and Don Rudisill. Ada as a hardware description language: an initial report. In C. J. Koomen and T. Moto-oka, editors, *Computer Hardware Description Languages and their Applications*, pages 272–302, North Holland, August 1985.

[Barker 86] J. R. Barker and E. F. J. Ring. A portable microprocessor-based temperature recorder. *Journal of Microcomputer Applications*, 9(2):143–145, April 1986.

[Birmingham 84] William P. Birmingham and Daniel P. Siewiorek. MICON: a knowledge based single board computer designer. In *21st Design Automation Conference*, pages 565–571, June 1984.

[Birmingham 86] William Birmingham, Rostam Joobani, and Jin Kim. Knowledge-based expert systems and their application. In *23rd Design Automation Conference*, pages 531–539, June 1986.

[Bowen 83] J.A. Bowen and Smith M.F. Expert systems for analysis and design of microprocessor applications. *Journal of Microcomputer Applications*, 6(2):155–161, April 1983.

[Brandt 85] Richard Brandt. Next from Detroit: the computer on wheels. *International Business Week*, 2880(210):62–63, 11th February 1985.

[Brück 86] R. Brück, B. Kleinjohann, T. Kathöfer, and Franz J. Rammig. Synthesis of concurrent modular controllers from algorithmic descriptions. In *23rd Design Automation Conference*, pages 285–292, June 1986.

[Brueck 86] Rainer Brueck, Klaus Groening, Christoph Ohsendoth, Peter Schmitz, Ulrike Schroeder, Karl-Heinz Temme, and Norbert Wandet. DACAPO II: User manual. Technical Report, Department Informatik, University of Dortmund, March 1986.

[Buric 83] Misha R. Buric, C. Christensen, and Thomas G. Matheson. Plex: automatically generated microcomputer layouts. In *International Conference on Circuits and Devices*, pages 181–184, 1983.

[Buric 84] Misha R. Buric. Microcomputers as components of custom ICs. *VLSI Design*, V(5):33–39, May 1984.

[Caironi 82] J. Caironi, G. Torelli, and D. Devecchi. A TV frequency synthesis system with a single chip microprocessor interface including non-volatile memory. *IEEE Transactions on Consumer Electronics*, CE-28(3):363–372, August 1982.

[Clifford 78] C. Clifford, R. Landuyt, and J. Schmitt. Microprocessor based software defined television controller. *IEEE Transactions on Consumer Electronics*, CE-24(3):436–441, August 1978.

[Clocksin 81] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Dachauer 81] R. J. Dachauer, K. Gröening, K.-D. Lewke, and Franz J. Rammig. The CAP/DSDL system: simulator and case study. In Melvin A. Breuer and Reiner W. Hartenstein, editors, *Computer Hardware Description Languages and their Applications*, pages 213–227, North Holland, September 1981.

[Dijkstra 75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[Engh 83] M. K. Engh, A. K. Vaidya, and D. L. Dietmeyer. WISLAN — a CONLAN member for gate array design. In Takao Uehara and Mario P. Barbacci, editors, *Computer Hardware Description Languages and their Applications*, pages 31–42, North-Holland, May 1983.

[Evanczuk 85] Stephen Evanczuk. Core-based custom processors. *VLSI Design*, VI(5):20–38, May 1985.

[Featner 85] A. H. Featner. OCCAM *as a design notation in Jackson System Development Method.* Master's thesis, Oxford University Programming Research Group, September 1985.

[Ford 84] Ford Motor Co. Automotive electronics developments. In *Ford Energy Report Volume 3*, pages 2–10, Ford Motor Co., 1984.

[Forgy 81] Charles L. Forgy. *OPS5 User's Manual.* Technical Report, Department of Computer Science, Carnegie-Mellon University, July 1981.

[Forsyth 84] Richard Forsyth. The architecture of expert systems. In Richard Forsyth, editor, *Expert Systems: Principles and Case Studies*, pages 9–17, Chapman Hall Computing, 1984.

[Fox 83] Jeffrey R. Fox. The MacPitts silicon compiler: a view from the telecommunications industry. *VLSI Design*, IV(3):30–37, May/June 1983.

[Frantz 83] Dieter Frantz and Franz J. Rammig. The impact of an advanced CHDL on VLSI design. In *International Conference on Circuits and Devices*, pages 173–176, IEEE, 1983.

[Gaitonde 82] G. Gaitonde. Microcomputer in photography. *IEEE Transactions on Consumer Electronics*, CE-28(4):625–637, November 1982.

[Gajski 86] Daniel D. Gajski, Alex Orailoglu, and Charles D. Bosco. An expert silicon compiler. In *Custom Integrated Circuits Conference*, pages 116–119, IEEE, May 1986.

[German 85] Steven M. German and Karl J. Lieberherr. Zeus: a language for expressing algorithms in hardware. *IEEE Computer*, 18(2):55–65, February 1985.

[Girczyc 84] Emil F. Girczyc and John P. Knight. An Ada to standard cell hardware compiler based on graph grammars and scheduling. In *International Conference on Circuits and Devices*, pages 726–731, IEEE, 1984.

[Girczyc 85] Emil F. Girczyc, Ray J. A. Buhr, and John P. Knight. Applicability of a subset of Ada as an algorithmic HDL for graph based hardware compilation. *IEEE Transactions on Computer Aided Design for Integrated Circuits*, CAD-4(2):134–142, April 1985.

[Grierson 83] J. R. Grierson, B. Cosgrove, R. Daniel, K. I. H. Halliwell, J. C. Knight, J. A. McLean, J. M. McGrail, and C. O. Newton. UK5000 – successful collaborative development of an integrated design system for a 5000 gate CMOS array with built in test. In *20th Design Automation Conference*, pages 629–636, June 1983.

[Groh 80] R. M. Groh. The microprocessor: the key to an advanced frequency synthesised HF SSB amateur radio transceiver. *IEEE Transactions on Consumer Electronics*, CE-26(3):234–246, August 1980.

[Hazari 86] C. Hazari and H. Zedan. System calls for OCCAM. OCCAM *User Group Newsletter*, (5):27–31, July 1986.

[Hilfinger 82] Paul N. Hilfinger. *Abstraction Mechanisms and Language Design. ACM Distinguished Dissertations*, The MIT Press, 1982.

[Hitchcock 83] Charles Y. Hitchcock III and Donald E. Thomas. A method of automatic data path synthesis. In *20th Design Automation Conference*, pages 484–489, June 1983.

[Hoare 78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[INMOS 84a] INMOS Ltd. *IMS T424 Transputer Reference Manual.* November 1984.

[INMOS 84b] INMOS Ltd. OCCAM *Portakit.* November 1984.

[INMOS 84c] INMOS Ltd. OCCAM *Programming Manual.* Prentice Hall International, 1984.

[INMOS 85] INMOS Ltd. OCCAM *Programming System.* July 1985.

[Inoue 82] M. Nonoue, M. Fujii, M. Esaki, T. Hasegaura, T. Kobayashi, M. Shinma, and T. Miki. A remote controller for room air conditioner using an optical fiber and voice synthesis LSI's (sic). *IEEE Transactions on Consumer Electronics*, CE-28(4):571–584, November 1982.

[Intermetrics 86] *VHDL Language Reference Manual, Version 7.2.* IR-MD-045-2 edition, June 1986.

[Iverson 62] Kenneth E. Iverson. A common language for hardware, software and applications. In *Fall Joint Computer Conference*, AFIP, 1962.

[Jesshope 85] C. R. Jesshope. Pascal circuit compiler for UK5000. *IEE Proceedings on Computers and Digital Techniques*, 132 Part E(2):116–120, March/April 1985.

[Jones 85] Geraint Jones. *Programming in 'occam': A tourist guide to parallel programming.* Technical Monograph PRG-43, Oxford University Programming Research Group, March 1985.

[Jurgen 83] Ronald K. Jurgen. Detroit unveils sophisticated electronics. *IEEE Spectrum*, 20(10):33–39, October 1983.

[Karstand 80] K. Karstand. Microprocessor control for color-TV receivers. *IEEE Transactions on Consumer Electronics*, CE-26(2):149–146, May 1980.

[Kim 86] Jin Kim and John McDermott. Computer aids for IC design. *IEEE Software*, 2(3):38–47, March 1986.

[King-Smith 86] Tony King-Smith. OCCAM as a systems description language. In *3rd Silicon Design Conference*, pages 59–63, July 1986.

[Kowalski 83] Thaddeus J. Kowalski and Donald E. Thomas. The VLSI design automation assistant: prototype system. In *20th Design Automation Conference*, pages 479–483, June 1983.

[Kowalski 86] Thaddeus J. Kowalski and Donald E. Thomas. The VLSI design automation assistant: what's in a knowledge base. In *23rd Design Automation Conference*, pages 252–258, June 1986.

[Lattice 84a] Lattice Logic Ltd. *ChipSmith: A Random Logic Compiler for Gate Arrays, Optimized Gate Arrays and Standard Cells.* Lattice Logic, 1984.

[Lattice 84b] Lattice Logic Ltd. *SwitchSmith: A Switch-Level Logic Simulator.* Lattice Logic, 1984.

[Lipton 83] Richard J. Lipton, Jacobo Valdes, and Gopalakrishnan Vijayan. VLSI layout as programming. *ACM Transactions on Programming Languages and Systems*, 5(3):405–421, July 1983.

[Lowenstein 86] Al Lowenstein and Greg Winter. VHDL's impact on test. *IEEE Design and Test of Computers*, 3(2):49–53, April 1986.

[Lucky 85] R. W. Lucky. Coping with complexity. *IEEE Spectrum*, 22(3):14, March 1985.

[Mano 84] Tamio Mano, Fumihiro Maruyama, Kazushi Hayashi, Taeko Kakuda, Nobuaki Kawato, and Takao Uehara. OCCAM *To CMOS.* Technical Report TR-093, Institute for New Generation Computer Technology, December 1984.

[Mano 85] Tamio Mano, Fumihiro Maruyama, Kazushi Hayashi, Taeko Kakuda, Nobuaki Kawato, and Takao Uehara. OCCAM to CMOS: experimental logic design support system. In C. J. Koomen and T. Moto-oka, editors, *Computer Hardware Description Languages and their Applications*, pages 381–390, North Holland, August 1985.

[Marshall 83] Richard M. Marshall. *A Compiler for Large Scale.* Technical Report CSR-150-83, University of Edinburgh, Department of Computer Science, October 1983.

[Marshall 86] Richard M. Marshall. *OC: A Portable* OCCAM *front-end.* Technical Report CSR-201-86, University of Edinburgh, Department of Computer Science, August 1986.

[Maruyama 84] Fumihiro Maruyama, Tamio Mano, Kazushi Hayashi, Taeko Kakuda, Nobuaki Kawato, and Takao Uehara. Prolog-based expert system for logic design. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 563–571, November 1984.

[Marwedel 79] Peter Marwedel. The MIMOLA design system: detailed description of the software system. In *16th Design Automation Conference*, pages 59–63, June 1979.

[Marwedel 84] Peter Marwedel. The MIMOLA design system: tools for the design of digital processors. In *21st Design Automation Conference*, pages 587–593, June 1984.

[Marwedel 86] Peter Marwedel. A new synthesis algorithm for the MIMOLA software system. In *23rd Design Automation Conference*, pages 271–277, June 1986.

[May 86] David May. *Occam 2.* May 1986.

[McCaskill 85] George A. McCaskill. *APG: A Parser Building Package.* May 1985.

[Milne 83a] George J. Milne. CIRCAL: a calculus for circuit description. *Integration: The VLSI Journal*, 1(2 and 3):121–160, October 1983.

[Milne 83b] George J. Milne. The correctness of a simple silicon compiler. In Takao Uehara and Mario P. Barbacci, editors, *Computer Hardware Description Languages and their Applications*, North-Holland, 1983.

[Milne 86] George J. Milne. Behavioural description and VLSI verification. *IEE Proceedings on Computers and Digital Techniques*, 133 Part E(3):127–137, May 1986.

[Milner 80] Robin Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Spinger-Verlag, 1980.

[Naghdy 84] F. Haghdy, K. D. C. Stoodley, R. M. Henry, and A. D. Crew. Development of a microcomputer based monitoring system for post-surgical cardiac patients. *Journal of Microcomputer Applications*, 7(1):41–49, January 1984.

[Nash 84] J. D. Nash. Bibliography of hardware description languages. *ACM SIGDA Newsletter*, 14(1):18–37, 1984.

[Nash 86] J. D. Nash and Larry F. Saunders. VHDL critique. *IEEE Design and Test of Computers*, 3(2):55–65, April 1986.

[Numata 85] Jun Numata and Nobuhisa Watanabe. The CPU core architecture of a 4-bit microcomputer. *VLSI Design*, VI(5):40–48, May 1985.

[Organick 84] Elliott I. Organick, Tony M. Carter, Michael P. Maloney, Alan L. Davis, Alan B. Hayes, Dan Klass, Gary Lindstrom, Brent E. Nelson, and Kent F. Smith. Transforming an Ada program unit into silicon and verifying its behavior in an Ada environment: a first experiment. *IEEE Software*, 1(1):31–50, January 1984.

[Parisoe 85] L. F. Parisoe and G. W. Knapp. 2901/2910 core microprocessor cells bring 'systems-on-a-chip' to semicustom standard cells. In *Custom Integrated Circuits Conference*, pages 264–266, IEEE, May 1985.

[Parker 79] Alice C. Parker, Donald E. Thomas, Daniel P. Siewiorek, Mario P. Barbacci, L. Hafer, G. Leive, and J. Kim. The CMU design automation system: an example of automated data path design. In *16th Design Automation Conference*, pages 73–80, June 1979.

[Pawlak 85] Adam Pawlak. A tutorial guide to modern hardware description and design languages. In Klaus Waldscmidt and Bjørn Myhrhaug,

editors, *Proceedings of the 11th Euromicro Symposium*, pages 507–516, North-Holland, September 1985.

[Peng 86] Zebo Peng. Synthesis of VLSI systems with the CAMAD design aid. In *23rd Design Automation Conference*, pages 278–284, June 1986.

[Penn 77] M.G. Penn. A low cost dedicated microprocessor forms the heart of an electronic control system for appliance applications. *IEEE Transactions on Consumer Electronics*, CE-23(2):182–187, May 1977.

[Piloty 83] Robert Piloty, Mario Barbacci, Dominique Borrione, Donald Dietmeyer, Frederick Hill, and Patrick Skelly. *CONLAN Report.* Volume 151 of *Lecture Notes in Computer Science*, Spinger-Verlag, 1983.

[Platte 85] H-J. Platte, G. Oberjatzas, and W. Voessing. A new intelligent remote control unit for consumer electronic devices. *IEEE Transactions on Consumer Electronics*, CE-31(1):59–69, February 1985.

[Preston 82] N. C. G. N. Preston and J. Lucas. Multiprocessor implementation of the logic function of a multiplexed wiring system for automotives. *IEE Proceedings on Computers and Digital Techniques*, 129 Part E(6):223–228, November/December 1982.

[Rammig 84] Franz J. Rammig. *CAP/DSDL: Hardware Description and Simulation Tools.* Technical Report, Department Informatik, University of Dortmund, April 1984.

[Richards 79] M. Richards and Colin Whitby-Strevens. *BCPL - the Language and its Compiler.* Cambridge University Press, 1979.

[Robertson 80] Peter S. Robertson. *The Imp77 Language.* Technical Report CSR-19-77, University of Edinburgh, Department of Computer Science, November 1980.

[Robertson 81] Peter S. Roberston. *The Production of Optimised Machine-Code for High-Level Languages using Machine-Independent Intermediate Codes.* PhD thesis, University of Edinburgh, Department of Computer Science, November 1981.

[Roscoe 86] A. W. Roscoe and C. A. R. Hoare. *The Laws of* OCCAM *programming.* Techical Monograph PRG-53, Oxford University Programming Research Group, February 1986.

[Sinderen 86] M. van Sinderen, C. Huijs, and G. A. Blaauw. APL: an effective design language. *IEE Proceedings on Computers and Digital Techniques*, 133 Part E(2):102–103, March/April 1986.

[Siskind 82] Jeffrey M. Siskind, Jay R. Southard, and Kenneth W. Crouch. Generating custom high performance VLSI designs from algorithmic descriptions. In Paul Penfield, Jr., editor, *Conference on Advanced Research in VLSI*, pages 28–39, MIT, 1982.

[Smith 80] Lee D. Smith. *The Elementary Structural Description Language.* Technical Report CSR-53-80, University of Edinburgh, Department of Computer Science, January 1980.

[Southard 83] Jay R. Southard. MacPitts: an approach to silicon compilation. *Computer*, 6-12:74–82, December 1983.

[Stallard 85] R. P. Stallard. OCCAM activities at Loughborough. OCCAM *User Group Newsletter*, (2):15–17, January 1985.

[Subrahmanyam 83] P. A. Subrahmanyam. Synthesising VLSI circuits from behavioral specifications: a very high level silicon compiler and its theoretical basis. In Francois Anceau and E. J. Aas, editors, *VLSI 83*, pages 195–210, North Holland, 1983.

[Takagi 84] Shigeru Takagi. Rule based synthesis, verification and compensation of data paths. In *International Conference on Circuits and Devices*, pages 133–138, IEEE, 1984.

[Thomas 83] Donald E. Thomas, Charles Y. Hitchcock III, Thaddeus J. Kowalski, Jayanth V. Rajan, and Robert A. Walker. Automatic data path synthesis. *Computer*, 6-12:59–70, December 1983.

[Traub 83] Niklas G. Traub. *A Lisp Based Circal Environment.* Technical Report CSR-152-83, University of Edinburgh, Department of Computer Science, November 1983.

[Trimberger 85] Stephen Trimberger and Jim Rowson. Automatic layout in an open design system. *VLSI Design*, VI(5):88–98, May 1985.

[Tseng 83] Chia-Jeng Tseng and Daniel P. Siewiorek. FACET: a procedure for the automated synthesis of digital systems. In *20th Design Automation Conference*, pages 490–496, June 1983.

[Tseng 84] Chia-Jeng Tseng and Daniel P. Siewiorek. Emerald: a bus style designer. In *21st Design Automation Conference*, pages 315–321, June 1984.

[Tseng 86] Chia-Jeng Tseng and Daniel P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer Aided Design for Integrated Circuits*, CAD-5(3):379–395, July 1986.

[US DoD 80] United States Department of Defence. *Ada Reference Manual.* Spinger-Verlag, July 1980.

[Walker 83] Robert A. Walker and Donald E. Thomas. Behavioral level transforms in the CMU-DA system. In *20th Design Automation Conference*, pages 788–789, June 1983.

[Watanabe 84] K. Watanabe and M. Tumer. An automotive engine calibration system using microcomputer. *IEEE Transactions on Vehicular Technology*, VT-33(2):45–50, May 1984.

[Weber 85] David M. Weber. Slow and steady is the new strategy in automotive electronics. *Electronics*, 47–52, 17th June 1985.

[Whitby-Strevens 85] Colin Whitby-Strevens. RISC and the I1 instrcution set for the transputer. In *12th International Symposium on Computer Architecture*, June 1985.

[Widdowson 84] Rod D. W. Widdowson and Richard M. Marshall. R^2: an intelligent toolmaker's assistant. In *The 1984 Edinburgh VLSI Workshop: Expert Systems and Silicon Compilation*, June 1984.

[Wijen 86] Hans Wijen. Hardware and software aspects of computer controlled TV. *IEEE Transactions on Consumer Electronics*, CE-32(1):32–36, February 1986.

[Wirth 82] Niklaus Wirth. *Programming in Modula-2. Technical monographs in Computer Science*, Springer-Verlag, 1982.

[Yazdani 84] Masoud Yazdani. Knowledge engineering in Prolog. In Richard Forsyth, editor, *Expert Systems: Principles and Case Studies*, pages 92–111, Chapman Hall Computing, 1984.

[Zimmermann 79] G. Zimmermann. The MIMOLA design system: a computer aided digital processor design method. In *16th Design Automation Conference*, pages 53–58, June 1979.

Acknowledgements

I must learn to write more genteelly.

Conversations at Mr Hume's
by Roger Savage

I would like to thank my supervisor Dr David Rees. I have also had many helpful discussions with Dr John Gray and Dr Irene Buchanan, now with European Silicon Structures. Dr David May of INMOS Ltd. spent some time discussing my ideas and telling me about the intestines of the transputer. Dr Rob Kersopp of Zonal Retail Systems Ltd. also provided useful information on the design of microprocessor-based systems in the real world.

My poor Mother has read numerous drafts of this deadly prose to correct the speling, typnig and missed out. Dr Roger Savage of the English Literature Department also showered-this-text-with-hyphens, trying to correct the malignant hyphenectomy that plagues technical prose.

This work was funded under the Cooperative Awards in Science and Engineering scheme by the Lattice Logic Ltd. and the United Kingdom Science and Engineering Research Council.

Appendix A

Extended Occam Grammar

Less is more.

Mies Van der Rohe

We are tied to a language that makes up in obscurity what it lacks in style.

Rosencrantz and Guildenstern are Dead
by Tom Stoppard

This grammar is based on that given in the Occam Programming Manual, but includes the changes made in the VAX Occam Programming System. Further extensions introduced in this implementation are marked with the ▷ character. Note that this grammar is significantly longer than that in the manual. This is because the optional and conditional clauses of productions in its grammar have been expanded to the full form for the parser generator, and the indentation and line continuation syntax is included in the grammar whereas it is only described informally in the manual.

Terminal symbols are shown in "`typewriter`" if they are punctuation or keywords, more complex ones are defined by regular expressions and are shown in *slanted* text. The symbol "⇓" means an end-of-line character, and "␣" means a space character. These whitespace characters appear in the grammar due to the way OCCAM is defined with indentation significance.

```
Occam →
  Statement_List;

Statement_List →
  Statement_List Statement |
  Statement;

Statement →
  Indents Process "⇓" |
  error;

Indents →
  "␣" Indents | ;

CommaNL →
  "," WhiteSpace;

WhiteSpace →
  WhiteSpace WhiteElem | ;

WhiteElem →
  "⇓" |
  "␣";

— Processes —

Process →
  Primitive_Processes End_Proc |
  Declaration |
  Construct |
```

```
    Compiler_Option |
    Procedure_Call End_Proc |
    Proc_At_End | ;

Procedure_Call →
    Identifier Actual_Parameters;

Actual_Parameters →
    "(" Actual_Param_List ")";

Actual_Param_List →
    Actual_Param_List CommaNL
            Actual_Param |
    Actual_Param;

Actual_Param →
    Expression |
    Slice_Value;

Proc_At_End →
    Identifier ":";

End_Proc →
    ":" | ;


— Primitive Processes —

Assignment →
    Variable ":=" Expression |
    Slice ":=" Slice_Value;

Input →
    Variable "?" Input_List;

Input_List →
    Input_Var_List |
    Slice |
    "ANY";

Input_Var_List →
    Input_Var_List ";" Variable |
    Variable;

Output →
    Variable "!" Output_List;

Output_List →
    Output_Exp_List |
    Slice_Value |
    "ANY";

Output_Exp_List →
    Output_Exp_List ";" Expression |
    Expression;

Skip →
    "SKIP";

Stop →
    "STOP";

Timer_Delay →
    "TIME" "?" "AFTER" Expression;

Timer_Input →
    "TIME" "?" Variable;

Primitive_Processes →
    Assignment |
    Input |
    Timer_Input |
    Timer_Delay |
    Output |
    Skip |
    Stop;


— Constructors —

Replicator →
        Identifier "=" "[" Expression
                "FOR" Expression "]" | ;

Guard →
        Expression "&" Guard_Primitive;

Guard_Primitive →
        Input |
        Timer_Delay |
        Skip;

PAR_Construct →
        "PAR" |
        "PRI" "PAR";

ALT_Construct →
        "ALT" |
        "PRI" "ALT";

▷ Synthesis_Body →
▷       "SYNTH" Identifier "," Identifier;

Construct →
        "SEQ" Replicator |
```

```
    Compiler_Option |
    Procedure_Call End_Proc |
    Proc_At_End | ;

Procedure_Call →
    Identifier Actual_Parameters;

Actual_Parameters →
    "(" Actual_Param_List ")";

Actual_Param_List →
    Actual_Param_List CommaNL
            Actual_Param |
    Actual_Param;

Actual_Param →
    Expression |
    Slice_Value;

Proc_At_End →
    Identifier ":";

End_Proc →
    ":" | ;


— Primitive Processes —

Assignment →
    Variable ":=" Expression |
    Slice ":=" Slice_Value;

Input →
    Variable "?" Input_List;

Input_List →
    Input_Var_List |
    Slice |
    "ANY";

Input_Var_List →
    Input_Var_List ";" Variable |
    Variable;

Output →
    Variable "!" Output_List;

Output_List →
    Output_Exp_List |
    Slice_Value |
    "ANY";

Output_Exp_List →
    Output_Exp_List ";" Expression |
    Expression;

Skip →
    "SKIP";

Stop →
    "STOP";

Timer_Delay →
    "TIME" "?" "AFTER" Expression;

Timer_Input →
    "TIME" "?" Variable;

Primitive_Processes →
    Assignment |
    Input |
    Timer_Input |
    Timer_Delay |
    Output |
    Skip |
    Stop;


— Constructors —

Replicator →
        Identifier "=" "[" Expression
                "FOR" Expression "]" | ;

Guard →
        Expression "&" Guard_Primitive;

Guard_Primitive →
        Input |
        Timer_Delay |
        Skip;

PAR_Construct →
        "PAR" |
        "PRI" "PAR";

ALT_Construct →
        "ALT" |
        "PRI" "ALT";

▷ Synthesis_Body →
▷       "SYNTH" Identifier "," Identifier;

Construct →
        "SEQ" Replicator |
```

```
        PAR_Construct Replicator |
        ALT_Construct Replicator |
        "IF" Replicator |
        "WHILE" Expression |
▷       "SIMUL" |
▷       Synthesis_Body |
        Guard |
        Expression;

  — Declarations —

  Variable_Declaration →
        "VAR" Identifier_Dec_List ":";

  Identifier_Dec_List →
        Identifier_Dec_List CommaNL
              Identifier_Dec |
        Identifier_Dec;

  Identifier_Dec →
        Identifier Subscript |
        Identifier;


  Subscript →
     "[" Byte_Option Expression "]";

  Byte_Option →
     "BYTE" | ;

  Channel_Declaration →
     "CHAN" Channel_Def_List ":";

  Channel_Def_List →
     Channel_Def_List CommaNL
          Channel_Def |
     Channel_Def;

  Channel_Def →
     Identifier Chan_Subs At_Place;

  Chan_Subs →
     "[" Expression "]" | ;

  At_Place →
     "AT" Expression | ;

  Constant_Declaration →
     "DEF" Constant_Def_List ":";

  Constant_Def_List →
     Constant_Def_List CommaNL
          Constant_Def |
     Constant_Def;

  Constant_Def →
     Identifier "=" Constant_Value;

  Constant_Value →
     Expression |
     Vector_Constant;

  Formal_Parameters →
     "(" Formal_Parameter_List ")" | ;

  Formal_Parameter_List →
     Formal_Parameter_List CommaNL
     Formal_Parameter |
     Formal_Parameter_1;

  Formal_Parameter →
     Formal_Parameter_1 |
     Param_Name;

  Formal_Parameter_1 →
     Param_Class Param_Name;


  Param_Class →
     "VAR" |
     "VALUE" |
     "CHAN" |
▷    "CONFIG";

  Param_Name →
     Identifier "[]" |
     Identifier;

  Procedure_Declaration →
     Proc_Id Formal_Parameters "=";

  Proc_Id →
     Procedure Identifier;

  Procedure →
     "PROC" |
▷    "IMPORT" |
▷    "EXPORT" |
▷    "INTERFACE";

▷ Library_Declaration →
▷    "LIBRARY" Identifier ":";

▷ Source_Declaration →
▷    "FROM" Identifier ":";
```

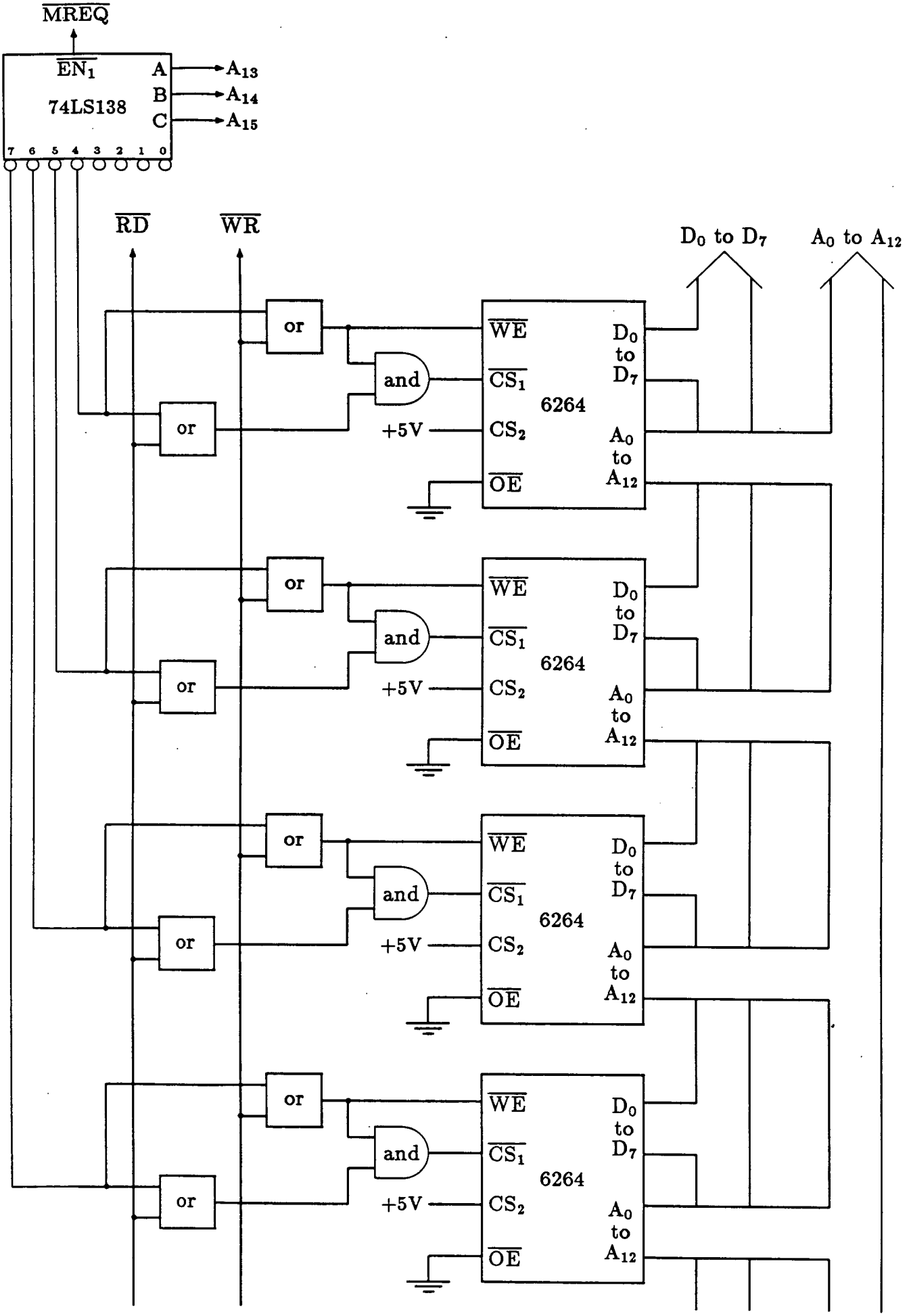
```
Declaration →
    Variable_Declaration |
    Channel_Declaration |
    Constant_Declaration |
    Procedure_Declaration |
▷   Library_Declaration |
▷   Source_Declaration;

— Expressions —

Vector_Constant →
    Table |
    StringConst;

Table →
    "TABLE" "[" Byte_Option
        Expression_List "]";

Expression_List →
    Expression_List CommaNL
        Expression |
    Expression;

Arithmetic_Op →
    "-" |
    "MINUS" |
    "/" |
    "\";


Comparison_Op →
    "=" |
    "<>" |
    "<" |
    ">" |
    "<=" |
    ">=";

Logical_Op →
    "/\" |
    "\/" |
    "><";

Boolean_Op →
    "AND" |
    "OR";

Shift_Op →
    "<<" |
    ">>";

Monadic_Op →
    "-" |
    "NOT";

Assoc_Arith_Op →
    "+" |
    "PLUS" |
    "*" |
    "TIMES";

Assoc_Op →
    Assoc_Arith_Op |
    Logical_Op |
    Boolean_Op;

Operator →
    Arithmetic_Op |
    Comparison_Op |
    Shift_Op;

Element →
    Number |
    HexNumber |
    Variable |
    Vector_Constant Subscript |
    CharConst |
    "TRUE" |
    "FALSE" |
    "(" Expression ")";

Expression →
    Element |
    Element Assoc_Element |
    Element Operator Element |
    Monadic_Op Element;

Assoc_Element →
    Assoc_Element Assoc_Op Element |
    Assoc_Op Element;

Variable →
    Identifier |
    Identifier Subscript;


— Vector Operations —

Slice →
    Identifier "[" Byte_Option
        Expression "FOR" Expression "]";

Slice_Value →
    Slice |
    Vector_Constant;
```

Appendix B

Collected Schematic Diagrams

This appendix contains all the schematic diagrams used in this thesis. They have been produced using the same formatter as the rest of the thesis, and this constrains the form some of the symbols take. For example, or and nor gates are drawn as boxes as LaTeX does not provide general arcs.

Contents


MREQ
EN1
74LS138
A
B
C
A13
A14
A15
7 6 5 4 3 2 1 0
RD
WR
D0 to D7
A0 to A12
or
and
+5V
WE
CS1
CS2
OE
6264
D0 to D7
A0 to A12

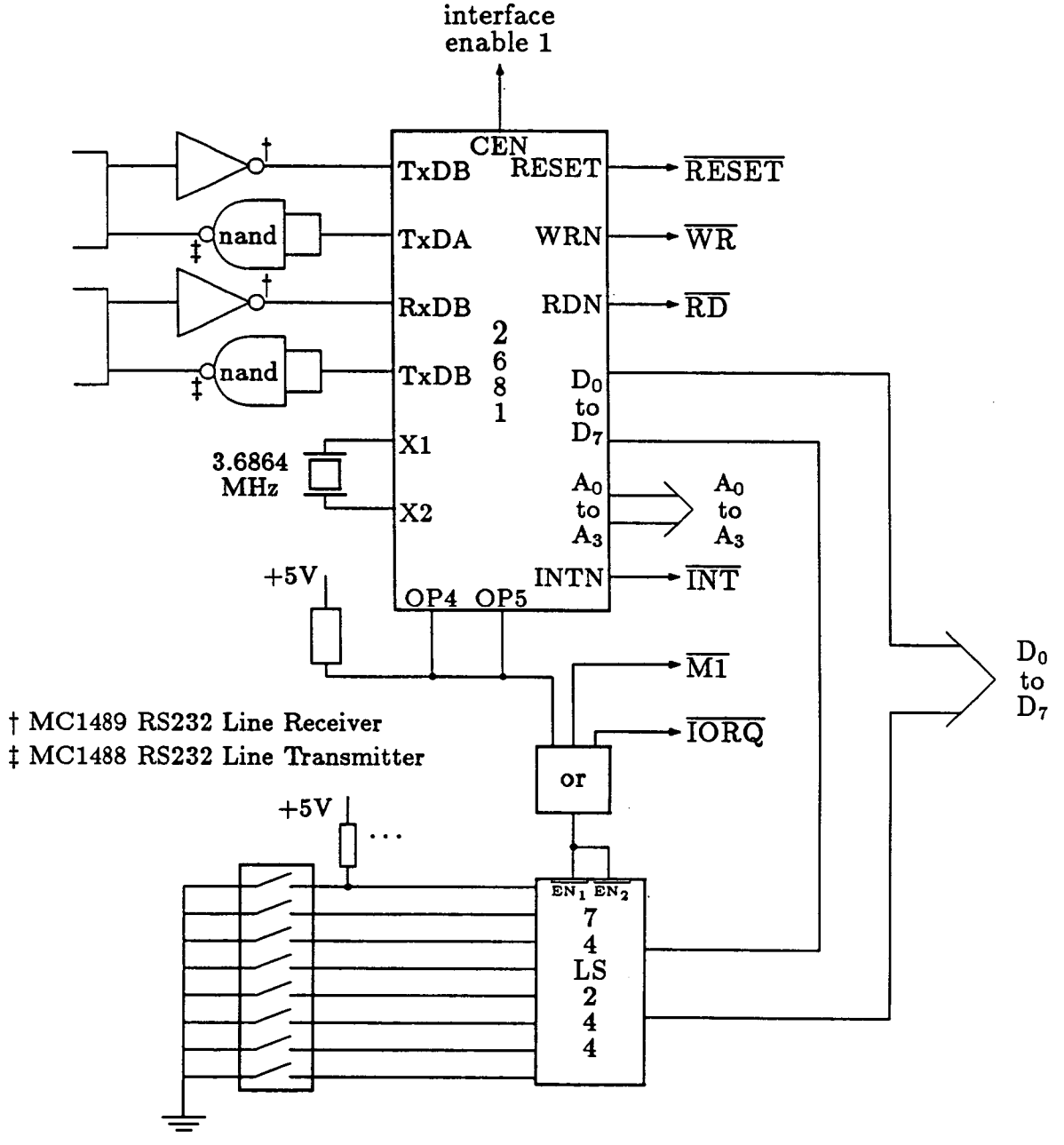


Figure B–2: The DUART Subsystem

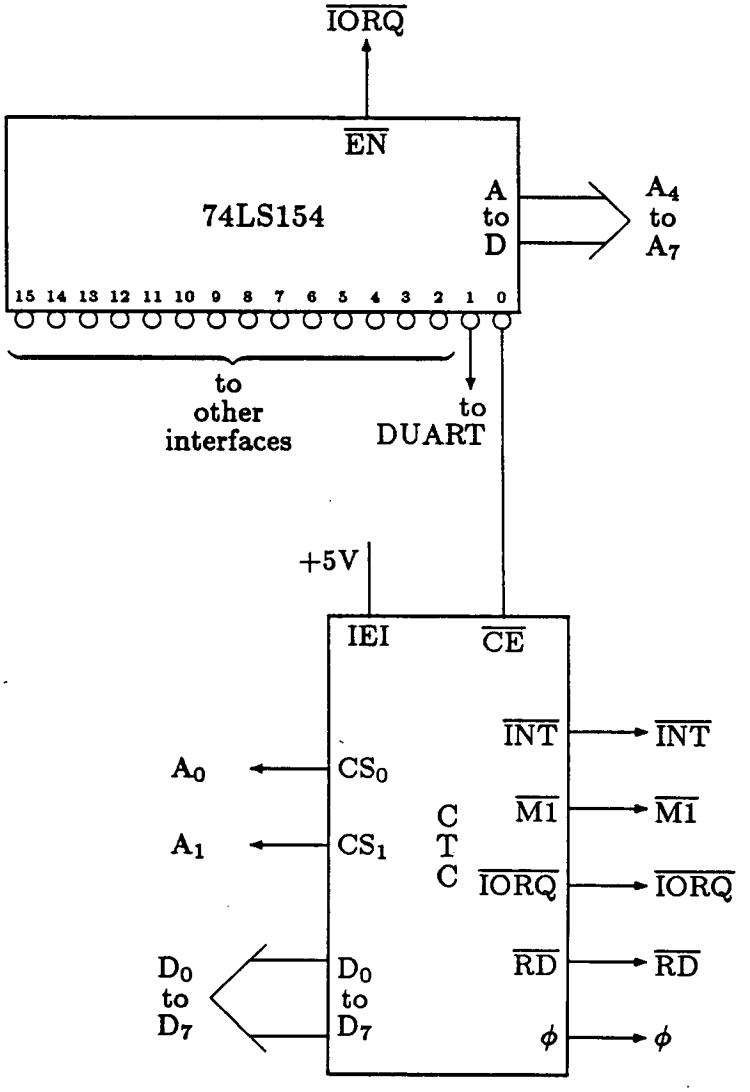


Figure B–3: The CTC and Interface Decoder

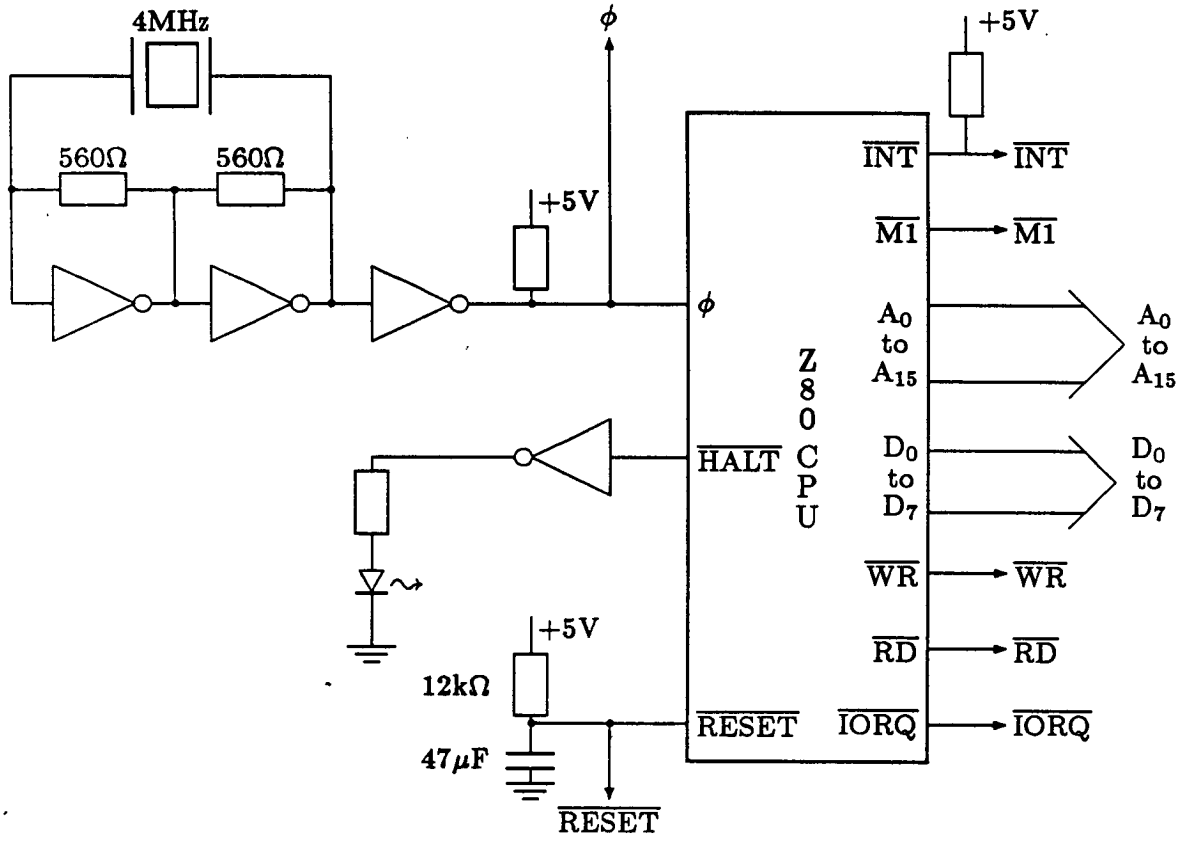


Figure B–4: The Z80 processor base circuit

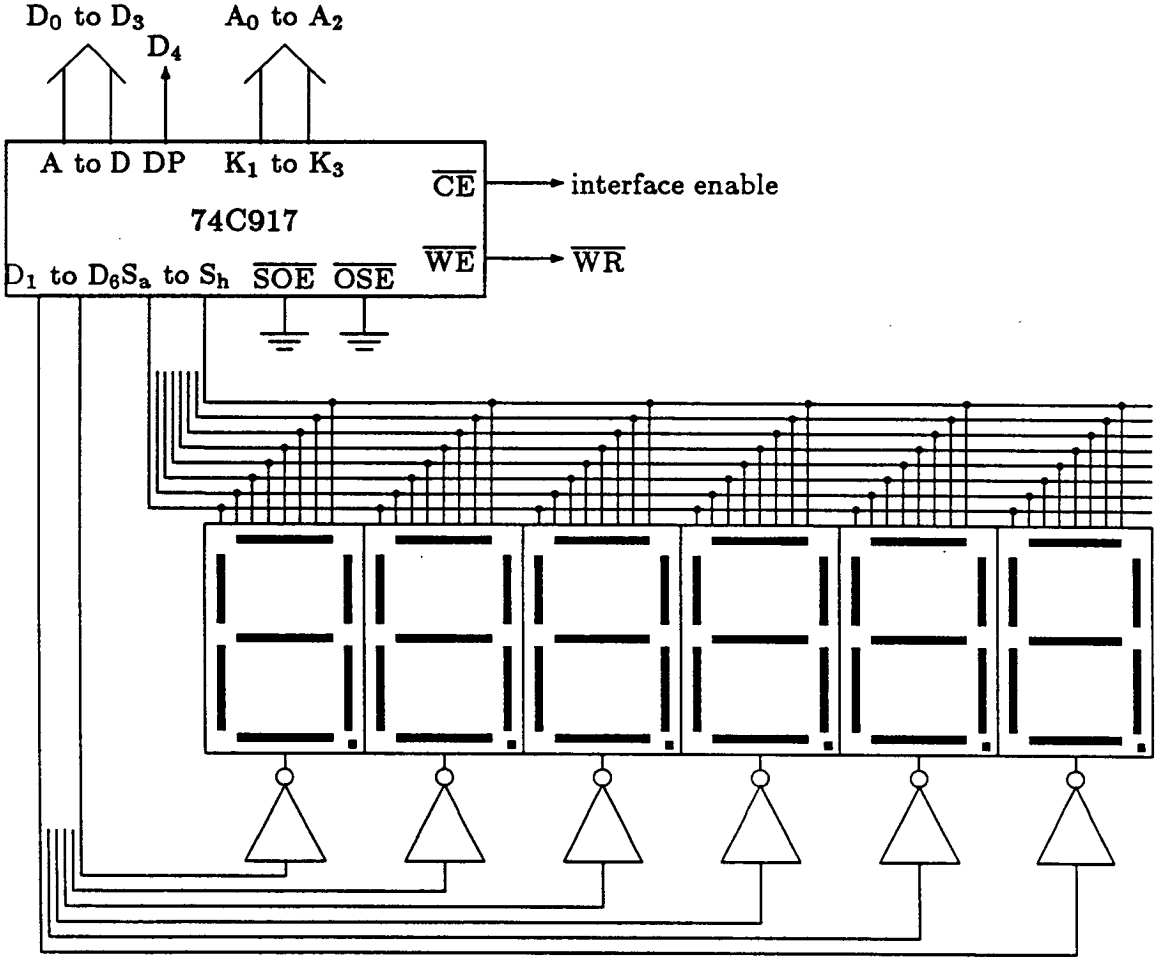


Figure B–5: The Seven-Segment Display Interface

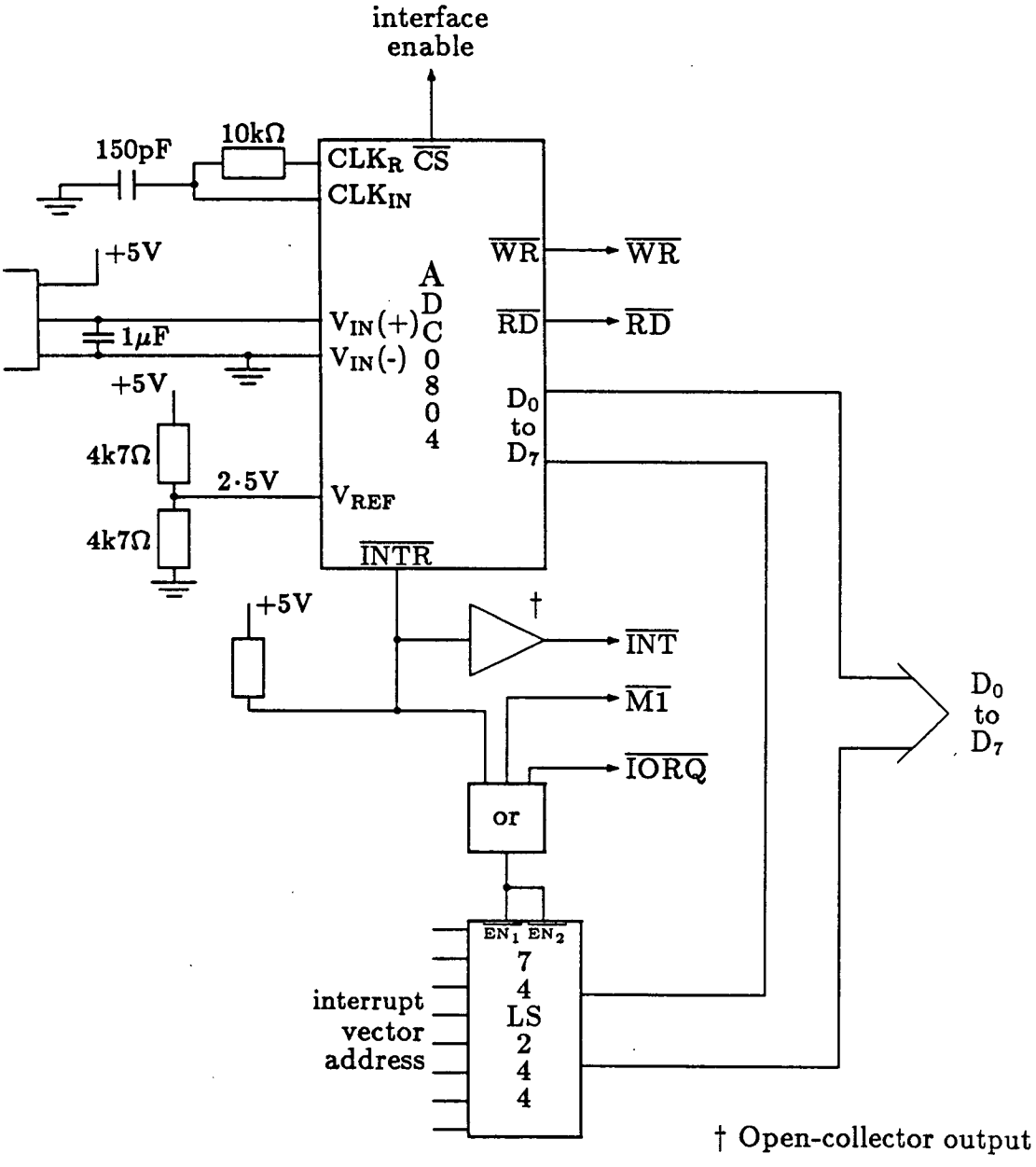


Figure B–6: The Analogue-to-Digital Converter

Appendix C

An Example Interface Designer

This appendix contains the Prolog code that synthesises the seven-segment display interface, described on page 107. This is achieved by outputting ESDL code with the utility procedures starting with `esdl_`. These are defined in the utilities library that is consulted by the base-designer.

```
six_seven_consulted.

six_seven_make(Address, _, [Digits | _]) :-
        write('Making a '),
        write(Digits),
        write(' digit seven-segment display at address '),
        write(Address),
        nl,
        plant_controller(Address, Digits),
        make_ss(Digits, Address).

plant_controller(Address, Digits) :-
        esdl_string('   hex.display (A<0:2>, D<0:4>, interface.EN''<'),
        esdl_string(Address),
        esdl_string('>, WR'', .GND, .GND) ->'),
        esdl_nl,
        esdl_string('                segments'),
        esdl_string(Address),
        esdl_string('<1:7>, DP'),
        esdl_string(Address),
        esdl_string(', digits.drive'),
        esdl_string(Address),
        esdl_string('<0:5>'),
        esdl_nl,
        digits_res(Address, Digits),
        digits_inv(Address, Digits),
```

```
        esdl_string(' -> external<'),
        esdl_string(DP_Edge),
        esdl_string('>'),
        esdl_nl,
        write(', DP on pin '),
        external_name(DP_Edge),
        nl,
        Last_Digit_Pin is DP_Edge + N,
        connect_digits(Last_Digit_Pin, N, Ad),
        write('The commons are connected on pins '),
        First_Digit_Pin is DP_Edge + 1,
        external_name(First_Digit_Pin),
        write(' to '),
        external_name(Last_Digit_Pin),
        !.

make_ss(N, Ad) :-
        nl,
        write('Using a 14 pin dil connector, pins 1 to 7 for the segments,'),
        nl,
        write('and 8 for the DP.  The commons are wired up to pins 9 to '),
        Last is 8 + N,
        write(Last),
        write('.'),
        nl,
        esdl_string('   connector14 (segments'),
        esdl_string(Ad),
        esdl_string('<1:7>, DP'),
        esdl_string(Ad),
        esdl_string(', digits'),
        esdl_string(Ad),
        esdl_string('<0:'),
        M is N - 1,
        esdl_string(M),
        esdl_string('>'),
        ss_dil_un(N),
        esdl_string(')'),
        esdl_nl.

ss_dil_un(6).

ss_dil_un(N) :-
        write(', '),
        Left is 6 - N,
        unused_signals(Left).

wire_up_ss(N, Ad) :-
        ss_tree(N, N, Ad).

ss_tree(0, _, _).

ss_tree(This, All, Ad) :-
```

```
        Next is This - 1,
        ss_tree(Next, All, Ad),
        place_ss(All, Next),
        esdl_string('   SS.LED'),
        esdl_string(All),
        esdl_string(' (segments'),
        esdl_string(Ad),
        esdl_string('<1:7>, DP'),
        esdl_string(Ad),
        esdl_string(') -> digits'),
        esdl_string(Ad),
        esdl_string('<'),
        esdl_string(Next),
        esdl_string('>'),
        esdl_nl.

place_ss(LEDs, 1) :-
        Pins is LEDs * 10,
        place_device(Pins).

place_ss(_, _).

connect_ss(_, 0, _).

connect_ss(Edge, N, Ad) :-
        Next_Edge is Edge - 1,
        M is N - 1,
        connect_ss(Next_Edge, M, Ad),
        esdl_string('   Wire segments'),
        esdl_string(Ad),
        esdl_string('<'),
        esdl_string(N),
        esdl_string('> -> external<'),
        esdl_string(Edge),
        esdl_string('>'),
        esdl_nl.

connect_digits(_, 0, _).

connect_digits(Edge, N, Ad) :-
        Next_Edge is Edge - 1,
        M is N - 1,
        connect_digits(Next_Edge, M, Ad),
        esdl_string('   Wire digits'),
        esdl_string(Ad),
        esdl_string('<'),
        esdl_string(M),
        esdl_string('> -> external<'),
        esdl_string(Edge),
        esdl_string('>'),
        esdl_nl.
```

Appendix D

The Cash Register

This appendix contains the OCCAM code for the cash-register example from Chapter 7, and the *Artificial Engineer* dialogue that was used to create the board shown in the same chapter.

D.1 Source Code

```
INTERFACE Number (CONFIG Digits, VALUE DP, Initial, CHAN Data) =
  FROM Seven.Segment:
INTERFACE Indicator (CONFIG Number, CHAN Switch []) =
  FROM Bar.Graph:
INTERFACE Keypad (CHAN Return []) =
  FROM Switches:
INTERFACE DUART (CHAN In1, Out1, In2, Out2) =
  FROM DUART:

DEF Price.Ind  = 0,        -- codes for the indicator LEDs
    Times.Ind  = 1,
    Class.Ind  = 2,
    Change.Ind = 3,
    Paid.Ind   = 4,
    Master.Ind = 5:
DEF Max.Indicators = 6:

DEF Add.Key    = 10,       -- the function keys
    Times.Key  = 11,
    Class.Key  = 12,
    Total.Key  = 13,
    Error.Key  = 14,
    Master.Key = 15:
```

```
DEF Price.State  = 0,      -- internal state of the
    Total.State  = 1,      -- control software
    Change.State = 2,
    Times.State  = 3:

-- Control --

PROC Control (CHAN Keypad [], Display, Indicators [], Printer) =
  VAR Pressed,             -- last key pressed
      Value,               -- number being typed in
      Price,               -- price of current item
      Total,               -- total for this bill
      State,               -- current software state
      Last.Show:           -- last indicator illuminated

  VAR Class.Pounds [16],   -- pounds spent, by class
      Class.Pence  [16],   -- pence  spent, by class
      Other.Pounds,        -- unclassified sales
      Other.Pence:

  VAR Float,               -- till float of change
      Sales,               -- total number of sales
      Operator:            -- assistant number

  -- Show --

  PROC Show (VALUE What) =
    SEQ
      IF
        Last.Show < 0
          SKIP
        TRUE
          Indicators [Last.Show] ! ANY
      Indicators [What] ! ANY
      Last.Show := What:

  -- Print --

  PROC Print (VALUE What, Size) =
    SEQ
      Write (Printer, What/100, Size - 3)
      Printer ! '.'
      IF
        (What\100) < 10
          SEQ
            Printer ! '0'
            Write (Printer, What\100, 1)
        TRUE
          Write (Printer, What\100, 2):

  -- Multiple.Add --
```

```
PROC Multiple.Add (VALUE Class) =
  VAR TP:
  SEQ
    TP := Price*Value
    Total := Total + (TP)
    IF
      Class < 0
        SEQ
          Printer ! ' '
          Other.Pounds := Other.Pounds + (TP/100)
          Other.Pence  := Other.Pence + (TP\100)
      TRUE
        SEQ
          Printer ! ('A') + Class
          Class.Pounds [Class] := Class.Pounds [Class] + (TP/100)
          Class.Pence  [Class] := Class.Pence  [Class] + (TP\100)
    Printer ! ' '
    Write (Printer, Value, 0)
    Printer ! 'x'
    Print (Price, 5)
    Printer ! ' '; '='
    Print (Price*Value, 7)
    Printer ! '*n'
    Price := 0
    Value := 0
    Display ! Total
    Show (Price.Ind)
    State := Price.State:


-- Single.Add --

PROC Single.Add (VALUE Class) =
  SEQ
    Total := Total + Value
    IF
      Class < 0
        SEQ
          Printer ! ' '
          Other.Pounds := Other.Pounds + (Value/100)
          Other.Pence  := Other.Pence + (Value\100)
      TRUE
        SEQ
          Printer ! 'A' + Class
          Class.Pounds [Class] := Class.Pounds [Class] + (Value/100)
          Class.Pence  [Class] := Class.Pence  [Class] + (Value\100)
    Print (Value, 17)
    Printer ! '*n'
    Price := 0
    Value := 0
    Display ! Total:

-- Add --
```

```
PROC Add (VALUE Class) =
  SEQ
    IF
      State = Times.State
        Multiple.Add (Class)
      TRUE
        Single.Add (Class):

-- Give.Total --

PROC Give.Total =
  SEQ
    Display ! Total
    Value := 0
    State := Total.State
    PrintString (Printer, "==================")
    Printer ! '*n'
    PrintString (Printer, " Total  ")
    Print (Total, 10)
    Printer ! '*n'
    Show (Change.Ind):

-- Give.Change --

PROC Give.Change =
  VAR Give:
  SEQ
    Give := Value - Total
    Display ! Give
    PrintString (Printer, "------------------")
    Printer ! '*n'
    PrintString (Printer, " Päid   ")
    Print (Value, 10)
    Printer ! '*n'
    PrintString (Printer, "------------------")
    Printer ! '*n'
    PrintString (Printer, " Change ")
    Print (Give, 10)
    Printer ! '*n'
    PrintString (Printer, "==================")
    Printer ! '*n'; '*n'
    IF
      Operator < 0
        SKIP
      TRUE
        SEQ
          PrintString (Printer, "Assistant ")
          Write (Printer, Operator, 0)
          Printer ! '*n'; '*n'
    PrintString (Printer, "*#OETHANK YOU")
    Printer ! '*n'; '*n'
```

```
    PrintString (Printer, "  Have a nice day")
    SEQ i = [0 FOR 5]
      Printer ! '*n'
    Sales := Sales + 1
    Value := 0
    Total := 0
    State := Change.State
    Show (Paid.Ind):


-- Master.Mode --

PROC Master.Mode =

  -- Print.Split --

  PROC Print.Split (VALUE Pounds, Pence) =
    SEQ
      Write (Printer, Pounds, 0)
      Printer ! '.'
      IF
        Pence < 10
          Printer ! '0'
        TRUE
          SKIP
      Write (Printer, Pence, 0):

  DEF Set.Operator = 0,
      Set.Float    = 1,
      Print.Totals = 2,
      Clear.Totals = 3:

  VAR What,
      TSP,
      TSp:
  SEQ
    Show (Master.Ind)
    ALT i = [0 FOR 16]
      Keypad [i] ? ANY
        What := i
    IF
      What = Set.Operator
        SEQ
          Operator := Value
          Value := 0
          Printer ! '*n'; '*n'
          PrintString (Printer, "Assistant ")
          Write (Printer, Operator, 0)
          Printer ! '*n'; '*n'

      What = Set.Float
        SEQ
```

```
      Float := Value
      Value := 0
      Printer ! '*n'; '*n'
      PrintString (Printer, "Float ")
      Print (Float, 0)
      Printer ! '*n'; '*n'

    What = Print.Totals
      SEQ
        Printer ! '*n'; '*n'
        PrintString (Printer, "Assistant ")
        Write (Printer, Operator, 0)
        Printer ! '*n'; '*n'
        PrintString (Printer, "Number of sales ")
        Write (Printer, Sales, 0)
        Printer ! '*n'
        IF
          Other.Pence > 100
            SEQ
              Other.Pounds := Other.Pounds + (Other.Pence/100)
              Other.Pence  := Other.Pence\100
          TRUE
            SKIP
        SEQ i = [0 FOR 16]
          IF
            Class.Pence [i] > 100
              SEQ
                Class.Pounds [i] := Class.Pounds [i] + (Class.Pence [i]/100
                Class.Pence  [i] := Class.Pence [i]\100
            TRUE
              SKIP
        TSP := Other.Pounds
        TSp := Other.Pence
        SEQ i = [0 FOR 16]
          SEQ
            TSP := TSP + Class.Pounds [i]
            TSp := TSp + Class.Pence  [i]
        TSP := TSP + (TSp/100)
        TSp := TSp\100
        PrintString (Printer, "Total Sales ")
        Print.Split (TSP, TSp)
        Printer ! '*n'; '*n'
        PrintString (Printer, "Till balance ")
        TSP := TSP + (Float/100)
        TSp := TSp + (Float\100)
        TSP := TSP + (TSp/100)
        TSp := TSp\100
        Print.Split (TSP, TSp)
        Printer ! '*n'; '*n'
        PrintString (Printer, "Unclassed sales ")
        Print.Split (Other.Pounds, Other.Pence)
        Printer ! '*n'; '*n'
```

```
          SEQ i = [0 FOR 16]
            SEQ
              PrintString (Printer, "Class ")
              Printer ! 'A' + i
              Printer ! ' '
              Print.Split (Class.Pounds [i], Class.Pence [i])
              Printer ! '*n'
          SEQ i = [0 FOR 5]
            Printer ! '*n'

      What = Clear.Totals
        SEQ
          Sales := 0
          SEQ i = [0 FOR 16]
            SEQ
              Class.Pounds [i] := 0
              Class.Pence  [i] := 0
              Other.Pounds := 0
              Other.Pence  := 0
      TRUE
        SKIP
    Printer ! '*n'
    Show (Price.Ind):


SEQ -- Main code of Control
  Value := 0
  Price := 0
  Total := 0
  State := Price.State
  Sales := 0
  Float := 0
  Operator := -1
  Last.Show := -1
  SEQ i = [0 FOR 16]
    SEQ
      Class.Pounds [i] := 0
      Class.Pence  [i] := 0
  Other.Pounds := 0
  Other.Pence  := 0
  Show (Price.Ind)
  WHILE TRUE
    SEQ
      ALT i = [0 FOR 16]
        Keypad [i] ? ANY
          Pressed := i
      IF
        State = Change.State
          SEQ
            Show (Price.Ind)
            State := Price.State
        TRUE
          SKIP
```

```
        IF
          (0 <= Pressed) AND (Pressed <= 9)
            SEQ
              Value := (Value*10) + Pressed
              Display ! Value
          (Value > 0) AND (Pressed = Add.Key)
            Add (-1)
          (Value > 0) AND (Pressed = Class.Key)
            VAR Class:
            SEQ
              Show (Class.Ind)
              ALT i = [0 FOR 16]
                Keypad [i] ? ANY
                  Class := i
              Add (Class)
              Show (Price.Ind)
          Pressed = Total.Key
            IF
              State = Price.State
                Give.Total
              State = Total.State
                Give.Change
          Pressed = Times.Key
            SEQ
              Price := Value
              Value := 0
              Show (Times.Ind)
              State := Times.State
              Display ! 0
          Pressed = Error.Key
            SEQ
              Value := 0
              Display ! 0
          (State = Price.State) AND (Pressed = Master.Key)
            Master.Mode
          TRUE
            SKIP:

CHAN Display,
     Indicators [Max.Indicators],
     A, C, D,
     Printer,
     Keys [16]:

-- MAIN CODE OF CASH REGISTER --

PAR
  Control (Keys, Display, Indicators, Printer)
  Indicator (Max.Indicators, Indicators)
  Number (5, 2, 0, Display)
  DUART (A, Printer, C, D)
  Keypad (Keys)
```

nd

D.2 Artificial Engineer Dialogue

```
C Prolog version 1.4e.edai
% Restoring file CPLOG:START.UP
| ?- [ae].
-A-r-t-i-f-i-c-i-a-l- -E-n-g-i-n-e-e-r-
Version 1.0
yes
rmm_ae:utils consulted 452 bytes .46078 sec.
rmm_ae:tech consulted 224 bytes .30125 sec.
rmm_ae:make consulted 828 bytes .50187 sec.
yes
ae consulted 1764 bytes 1.9005 sec.
yes
| ?- ae till.
rmm_ae:z80_utils consulted 4300 bytes 2.8804 sec.
generating Z80 processor and services: please wait....
yes
rmm_ae:z80_base consulted 744 bytes 8.2405 sec.
rmm_ae:ram consulted 1956 bytes 1.4416 sec.
rmm_ae:rom consulted 1160 bytes 1.1801 sec.
yes
RMM_UL:bar_graph.AE consulted 2716 bytes 1.6416 sec.
Making a bargraph at address 1 with 6 elements.
Do you want the bar graphs on the board? (y/n): n
Do you want to use the edge connector? (y/n): y
Edge connector pins 03a to 04c are the bar graph anodes, in increasing order.
yes
RMM_UL:six_seven.AE consulted 4036 bytes 2.2809 sec.
Making a 5 digit seven-segment display at address 2
Do you want the displays on the board? (y/n): n
Do you want to use the edge connector? (y/n): y
Segments wired up on pins 05a to 07a, DP on pin 07b
The commons are connected on pins 07c to 09a
yes
RMM_UL:duart.AE consulted 3480 bytes 2.0602 sec.
Making a Dual Asynchronous Receiver Transmitter (DUART) at address 3
Remember that the design will now require a +/- 12V power supply
Do you want to connect the RS232 lines via the Raft?(y/n): n
Connecting to the edge connector
Receive A is on pin 09b
Transmit A is on pin 09c
Receive B is on pin 10a
Transmit B is on pin 10b
yes
RMM_UL:buttons.AE consulted 3232 bytes 1.9413 sec.
Making a set of 16 buttons at address 4
Do you want the switches on the board?(y/n): n
Do you want the switches to be connected to the edge connector? (y/n): y
Do you want pull-ups on the switches? (y/n): y
```

```
Edge connector pins 10c to 15c are the switch lines, the switches must ground
them.
yes
RAM design:
        1K - 4118 @ 7
        invert A15 and OR with MREQ'
yes
ROM design:
Do want to use the more expensive but lower power CMOS EPROMs? (y/n): y
The appropriate EPROM is a 27128, but that is not available in CMOS
Do you want me to use an NMOS 27128? (y/n): y
16K - 27128
yes
till.BLD consulted 21624 bytes 28.641 sec.
yes
| ?- ^Z
% Prolog execution halted
```