A Pascal-based hardware and software description system

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:


Metadata Record: https://dspace.lboro.ac.uk/2134/33078

Publisher: © A.N. Clark

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: https://creativecommons.org/licenses/by-nc-nd/4.0/

Please cite the published version.
<table>
<thead>
<tr>
<th>AUTHOR/FILING TITLE</th>
<th>CLARK, A N</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESSION/COPY NO.</td>
<td>001033/02</td>
</tr>
<tr>
<td>VOL. NO.</td>
<td>CLASS MARK</td>
</tr>
<tr>
<td></td>
<td>LOAN COPY</td>
</tr>
<tr>
<td>- 5 JUL 1983</td>
<td></td>
</tr>
<tr>
<td>- 5 JUL 1987</td>
<td></td>
</tr>
<tr>
<td>- 1 JUL 1988</td>
<td></td>
</tr>
<tr>
<td>30 JUN 1989</td>
<td></td>
</tr>
</tbody>
</table>
A PASCAL BASED HARDWARE AND SOFTWARE DESCRIPTION SYSTEM

by

ANDREW N. CLARK

A Doctoral Thesis.

Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of the Loughborough University of Technology.

March 1982.

Supervisor: Dr. M. E. Woodward
Department: Electrical and Electronic Engineering.

© by A. N. Clark. 1982.
ACKNOWLEDGEMENTS

The writer would like to take this opportunity to extend his thanks to the people who have given assistance and support during the time it has taken to complete this Thesis.

Dr. H. E. Woodward, who has been my supervisor. My mother, S. J. Clark, for her encouragement, financial support and loan of her computer. My wife, for her assistance with shorthand and typing, economy and understanding.

This Thesis is to the memory of my father,

Mr. L. H. Clark

died 10 January 1982.
The first part of the Thesis introduces the reader to a class of computer languages known as hardware description languages. A portion of the writing is devoted to the clarification of various terms and concepts which are used throughout the rest of the work.

The relationship between software and hardware is examined closely and compatible ways of representing these are discussed and analysed. Once the representations have been established it is shown that various functions can be performed on the data so that programs, processor structure and instruction-sets can all be compared and cross-related. This is shown to provide means by which design and analysis of total systems can be accomplished all in a language which has two basic forms, a procedural form to correspond with software, and a non-procedural form allowing accurate structural description of hardware.

A language is proposed for this function and is fully described with partial syntax description to aid anyone pursuing further work in this line. A fairly complex example has been analysed using the technique proposed and the effects of the analysis can readily be seen, and the advantages appreciated.

In addition to the specification of the above-mentioned language, proposals are also laid out for a total integrated system which
is designed to handle structures, programs and instruction-sets, as single entities, and by this process reduce the need for complex interaction. The concept of the system would enable designers to solve any particular problem from whichever direction they would deem most appropriate, and the results in terms of hardware and software would become readily apparent. The package would include such facilities as simulation and assembly, structure generation and instruction-set generation.

Lastly, it is shown that once a structure is known it is then possible to take a higher level instruction that is not immediately executable and reduce it to a number of instructions which are permissible on the target machine. Techniques for performing this are explained for the lower levels, that is, at the micro-code level, but the possibility of extending the process to higher levels is thus opened up.
# CONTENTS

## CHAPTER 1 - INTRODUCTION

1.1. Introduction to structured computer systems. 1
1.2. Hierarchical structures in computing 4
1.3. Aims. 6

## CHAPTER 2 - BASIC CONSIDERATIONS AND CONCEPTS

2.1. Computers 7
2.2. Processors 9
2.3. Micro-programmability 11
2.4. The Bit Slice 14
2.5. Computer Languages 17
2.6. Structuring 19
2.7. Micro-programming 21
2.8. Hardware Description 23
2.9. Hardware and Software interaction. 24

## CHAPTER 3 - INTERNAL REPRESENTATIONS

3.1. Representation 29
3.2. Graphs and Matrices 32
3.3. Use of Graphs and Matrices 37
3.4. Petri Nets 43
3.5. State Transition Diagrams 50
<table>
<thead>
<tr>
<th>Chapter 4 - Register Transfer Languages</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1. Particular languages</td>
<td>56</td>
</tr>
<tr>
<td>4.2. Procedural and non-procedural languages</td>
<td>57</td>
</tr>
<tr>
<td>4.3. CDL</td>
<td>58</td>
</tr>
<tr>
<td>4.4. APLL</td>
<td>59</td>
</tr>
<tr>
<td>4.5. APL</td>
<td>60</td>
</tr>
<tr>
<td>4.6. ISP</td>
<td>62</td>
</tr>
<tr>
<td>4.7. Summary</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5 - Pascal</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1. The High Level Language Pascal</td>
<td>64</td>
</tr>
<tr>
<td>5.2. Structure</td>
<td>71</td>
</tr>
<tr>
<td>5.3. Scope Rules of Pascal</td>
<td>76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6 - A Pascal-Based Hardware Description Language</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1. Form of PHDL</td>
<td>79</td>
</tr>
<tr>
<td>6.1.1. The Black Box</td>
<td>79</td>
</tr>
<tr>
<td>6.1.2. Blocks</td>
<td>80</td>
</tr>
<tr>
<td>6.1.3. Source Form</td>
<td>80</td>
</tr>
<tr>
<td>6.1.4. Why Pascal ?</td>
<td>80</td>
</tr>
<tr>
<td>6.2. Descriptive form of PHDL</td>
<td>83</td>
</tr>
<tr>
<td>6.3. Constants and Meta-variables</td>
<td>85</td>
</tr>
<tr>
<td>6.4. General specifications</td>
<td>87</td>
</tr>
<tr>
<td>6.5. Substructure specification</td>
<td>91</td>
</tr>
<tr>
<td>6.6. Other substructures</td>
<td>93</td>
</tr>
</tbody>
</table>
6.7. Complex Terminal specifications 94
6.8. Function specification 96
6.9. Declaration and the Block Body 97

CHAPTER 7 - PHDL SCOPE RULES AND STATEMENTS 99
7.1. The PHDL statement 99
7.2. Overall structural description 101
7.3. The Scope Rules 103
7.4. Statement types 106
7.4.1. Simple statements 106
7.4.2. Statements 109
7.4.2.1. Triggers 109
7.4.2.2. Actions 110
7.4.2.3. Other Forms 110
7.4.2.4. Provided functions and other keywords 111
7.4.3. Compound statements 114
7.5. PHDL - Summary and Comments 115

CHAPTER 8 - EXAMPLES 118
8.1.1. Introduction 118
8.1.2. Supplied Structure Data 120
8.1.3. Supplied Program Data 126
8.2.1. PHDL Structure 131
8.2.2. PHDL Program 133
8.2.3. Instruction Set
8.2.4. Matrix Representation
8.3.1. Second example
8.3.2. Bit Sliced structure
8.3.3. Second program
8.3.4. Second instruction set
8.3.5. Expanded program
8.3.6. Expanded instruction set
8.3.7. Explanation of section 8.3.
8.4.1. Controller description
8.4.2. Controller structure
8.4.3. Analysis of controller
8.5. Structured HLL Example

CHAPTER 9 - POSSIBLE DEVELOPMENTS OF PHIL
9.1. A PHIL Package
9.2. Package functions
9.2.1. Normalisation
9.2.2. Instruction set generation
9.2.3. Structure generation
9.2.4. Simulation
9.2.5. Assembly
9.2.6. Disassembly
9.2.7. Program writers
9.2.8. High level decomposition
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.3</td>
<td>Recap of PHDL system</td>
<td>184</td>
</tr>
<tr>
<td>9.4</td>
<td>Hardware program constructs</td>
<td>186</td>
</tr>
<tr>
<td>9.5</td>
<td>Software program constructs</td>
<td>189</td>
</tr>
<tr>
<td>9.6</td>
<td>The PHDL expression</td>
<td>193</td>
</tr>
</tbody>
</table>

**CHAPTER 10 - CONCLUSIONS**

**GLOSSARY**

**APPENDIXES**

1. Pascal example  
2. PHDL examples  
3. PHDL syntax  
4. Bit Slice Device Data  

**REFERENCES**
CHAPTER 1

INTRODUCTION

1.1. INTRODUCTION TO STRUCTURED COMPUTER SYSTEMS

It has been common practice for many years to divide all aspects of computing into one of two classifications.

The term hardware is used to cover all tangible aspects of the field. All physical devices from the line printers down to logical gates are referred to as hardware and the interconnection on the physical level is the responsibility of the hardware designer.

Software is the term used to cover the remaining, intangible aspects. Under this classification will be found all computer programs and data which are necessary to make the hardware a functioning unit. This is not to say that a functioning unit cannot be built entirely in hardware, but this would be a dedicated machine, and would probably not be termed a computer.

In recent times, these terms and classifications have weakened, with the introduction of often used software in read only memories (ROM's) which cause the memory device to have certain fixed and immutable characteristics and can thus be termed hardware because the characteristics are now associated with the device. In order to overcome the problem of semantics, the new term firmware is often used.
But, this is to totally miss the main point that all main processor functions are software constructs, and that an AND gate is no more than a $4 \times 1$ bit ROM with 0 in locations 0..2 and a 1 stored in location 3.

It is not intended (nor would it be realistic) to rigidly define such terms, but nevertheless software and hardware will be referred to with the following meanings.

Hardware - a physical realised direct implementation of an algorithm

Software - an indirect implementation of an algorithm.
1.5. Software Hierarchy
1.2. **HIERARCHICAL STRUCTURES IN COMPUTING**

Order is brought in by the layering of software. This is done by the concept of a virtual machine. All high level languages (HLL's) define virtual machine. **FORTRAN, BASIC, Pascal, APL, LISP, Smalltalk, C, PL/1, etc.** are all HLL's and in their own way define a virtual machine with a particular instruction set, data types and the way they relate.

The implementation of this language may be in another language (not the language it is written in). This will usually be machine code. In its own turn, machine code is implemented either directly in hardware or in microcode, and the microcode implemented in hardware. Whatever the route, the language must ultimately be implemented in hardware. Thus it can be seen that hardware may be considered the lowest level, followed by machine code and continuing up to the HLL and the application program which itself may be considered as an English-like very high level language. See Diagram 1.a.

The operating system (OS) is a layer inserted to enable the activation of various devices. An ideal language would cause the operating system to be invisible, which is, in general, not the case. Smalltalk (Ref. 2) is an example of a complete language where no other levels or modes are involved. By no other levels it is meant that it is not necessary to cope with other languages. Levels do exist, but within the language of Smalltalk. For a full
discussion of computer levels (Ref. 1), the reader should look elsewhere. This is only to introduce layering as a concept and only pertinent details will be introduced as necessary.
1.3. INTRODUCTION

The intention in this work is to examine low level software and hardware and to demonstrate the relationship between the two; and thus to show that it is possible to bridge the apparent disciplinary gap between hardware and software and to freely interchange between the two media.

Further to this, it is intended that the possibility of designing a total hard/soft system in a high level language can be realised and to some extent automated, using such aids as simulation, assembly and instruction set generation.

In addition to the above, an HLL is proposed, and almost completely defined. But it is not suggested that this is the only possible solution nor the best possible, but nevertheless, the language will contain certain constructs that make it unique in its abilities.

Finally, ways and means of tackling remaining problems will be discussed, and possible paths for further work will be outlined.
CHAPTER 2

BASIC CONSIDERATIONS AND CONCEPTS

2.1. COMPUTERS

All electronic systems whether composed of valves, transistors, or integrated circuits (IC's) can be divided into one of two camps; these are analogue and digital. In analogue systems electronic signals are continuous variables within the bounds of an upper and lower limit. The quantity in question whether voltage or current may take on any value within the design range.

A totally different concept to this is the digital system where the chosen signal is permitted only predetermined levels or values. The number of levels chosen is purely a matter of choice, but for the greatest possible accuracy it is common only to choose two values, which in the vast majority of present-day digital equipment, e.g. Transistor Transistor Logic (TTL), are the two voltage levels of 0 and 5 volts positive. This system should be correctly termed binary digital but here we will fall in with common parlance and refer to it simply as digital.

The particular field of interest here is digital computers. The term 'computer' today covers many different circuits and systems from large main frame units down to fairly simple washing-machine control units. All these systems have one thing in common, that is, the Central Processing Unit (CPU). The differences between all the possible systems is in the peripherals attached to the CPU.
On larger systems peripherals will be devices such as printers, card readers and punches, VDU's, tele-typewriter terminals etc. On the smaller systems, in this case a washing-machine, the peripherals will be (water) valves, temperature gauges, motors and so forth. What the peripherals are on any given system is, by and large, quite irrelevant to the design and analysis of the CPU. So there is no reason why these systems cannot be grouped together and treated in a similar fashion and here they will be termed 'computers'. 
2.2. PROCESSORS

The CPU, as defined above, is composed of two parts, the processor and the main memory. The main memory may be composed of different types of memory. In older systems it was, by and large, of the core-type, but this has been replaced today by bipolar transistors and mos-transistors. Of the semi-conductor memories there are various types that function in different manners. The most common, the random access memory (RAM) possesses a number of unique locations, each of which can be written into and read from. The next most common is the read only memory (ROM) which, as its name implies does not permit the act of writing into any of its locations, but only the reading of predetermined data put in at manufacture time by the application of an appropriate mask. There are other types of read only memory which do allow reprogramming, these are generally called Erasable Programmable Read Only Memories (EPROM). The mechanisms involved are of no significance here. All that is of significance is that we have a unit with an address bus and a data bus which can be written into and read from, or just read from.

The processor is the second unit making up the so-called CPU. The processor is the unit that does the work of addition, subtraction, incrementation etc.

The processor works in the following manner (Ref. 3). It has a register which contains an address which when used enables it to select from main memory a binary word which is then transferred to
a control unit. The control unit decodes this word and
instructs the register arithmetic logic unit (RALU) to perform
certain operations which are implicit in the word which was
taken from the main memory. The original address register
is then incremented and so when this process is repeated, the
control unit will cause a different instruction to take place.
This action repeats indefinitely.

Having seen that the processor is divided into two sections, a
control unit and the RALU, it is as well to point out the basic
fact that the whole of the processor is composed of electronic
gates AND, OR and NOT. (Registers, although not actually
implemented in this fashion, can be constructed from these basic
gates). The first two are dyadic functions, that is they each
have two signals as input, whereas the NOT function is monadic,
having only one signal as input. It is assumed the reader is
quite familiar with these functions, but it is of interest to
note here that the whole system of control unit and RALU is
composed entirely of these functions. In other words, it is of
monolithic nature, and this implies that the whole system,
however complicated, may be described by a language which uses
these functions only, as its predefined building blocks.
2.3. MICRO-PROGRAMMABILITY

As has been seen from the previous subsection, the control unit is the governing 'brain' and the RALU is slave to it. This brings in the question of how the control unit functions. There are a number of techniques used in design of control units, but we are interested here in only one, which is the use of micro-programming techniques. The concept of the micro-program is quite simple if the machine level programming cursorily described above is known to the reader. The control unit itself is divided into two parts, the control memory (this is generally of ROM type) and the micro-control unit (MCU). The MCU works in exactly the same fashion as the control unit except that it obtains its instructions from the control memory and these instructions are bit-patterns which more or less require no decoding before being sent to the RALU. The one big difference between the micro-level and the machine-level is that parallelism is the rule rather than the exception. Parallelism is, as the reader might suspect, performing two or more functions simultaneously.

Micro-programming has become important in recent years because it gives versatility and simplicity with very little cost in terms of package count, etc. It also gives the possibility of changing micro-programs around thus giving a processor a different instruction set and quite different characteristics.

It is possible to go one stage further in this direction, and processors have been constructed where the functions of the MCU have
BASIC STRUCTURE OF A CPU

CPU

PROCESSOR

CONTROL UNIT

SEQUENCER

CONTROL MEMORY

ACU

MAIN MEMORY
been again, split in two; into a very small memory and a small, very simple processing unit. This is of no real significance here. Another technique which may be of interest to the reader involves not having a control memory as a separate entity, rather, the micro-program is held in main memory and accessed via a rather complicated procedure. Again this is an uncommon practice, and will not be dealt with as the advantages in terms of versatility are not great enough to warrant the loss of speed it involves.

Thus, to sum up, we have a CPU which is divided into two parts, the processor and main memory, the processor is divided again into two parts, the RALU and the control unit. The control unit, in a like manner, is split into two, the control memory and the micro-control unit. See Diagram 2.a.
2.4. THE BIT SLICE

The so-called Bit Slice [Ref. 4] is a relatively modern introduction into processor design, which has been designed to take advantage of the changes in memory fabrication.

In recent times the packing density of devices has become very great, and as a consequence, the amount of memory on each device has made it possible to have large micro-program memory, and the Bit Slice has been created to take advantage of this situation. In essence it consists of a slice through a conventional processor and contains a limited number of bits across its registers, usually two or four bits. They are constructed in such a way as to be open-ended and so they can be paralleled up to form virtually any bit-width processor desired. The design also allows for flexible access to the various units within the Bit Slice so that the units can be tailored to any particular application.

Manufacturers generally offer a selection of devices to partially handle any aspect of a target processor, for instance, a micro-program sequencer/controller. This may or may not, in fact, be Bit Slice, but it is used with an external micro-control memory to handle a RALU, which may be of any construction. The main point to note is that the micro-control memory is independent to all other units in the processor and can be changed simply by replacing the devices. It is also possible to use RAM for this memory, giving
total flexibility, but it does create the necessity for some means whereby the micro-program can be loaded into the control memory. In many applications this may be a small enough overhead to pay for the extremely fast functions that micro-programming can offer.

Micro-programming is a field that is rapidly expanding and it will not be long before it dominates the whole field of software. For this reason Bit Slice technology will become extremely important in its own right.

Typical units that are available are micro-processor slices, which consist of, for example, an ALU, a small register array, and perhaps an auxiliary register, for use as, say, an accumulator. These devices are coupled in such a way as to make the whole system expandable, offer shifting capability and give various input and output lines that permit direct access to the component parts. Also offered are micro-program sequencers, again expandable, which are used with some form of memory and perhaps some registers to form a complete control section.

The mere fact that these are self-contained functional units is another factor adding to their power and versatility. They can be used in any configuration desired and can then be micro-programmed to suit.
Registers, Bus-drivers, First in last out (FIFO) queue registers are examples of units that form a complete range which can be used to build complete control units and processors.
2.5. COMPUTER LANGUAGES

A computer is, as stated previously, an ordered set of gates, registers and so forth, and as such, if left to its own devices is just so much inanimate hardware. A computer must have a series of instructions to enable it to perform its desired function. Without any instructions the machine will not function at all. The series of instructions which enable the computer to become active are in many ways conceptual, since the form in which they are given to the computer can vary enormously. This series of instructions is called a computer program.

The most well known computer programming languages are the high level languages (HLL's), these include FORTRAN, ALGOL, APL, BASIC, etc. (Ref. 5). HLL's are designed so as to be relatively readable for humans and to provide a concise means of expressing the route to a solution. These languages then convert by various means the HLL program into a language which is readable by the machine. Any language that is readable by machine must be in the form of binary patterns, which by their very nature are illegible to humans. When the computer is given these binary patterns which are generally termed 'machine code', the computer can then perform the tasks as specified in the HLL program.

HLL's are used because they offer a number of advantages, including features which aid the programmer. The most fundamental is that by employing a HLL the programmer's task is divorced from the
intricacies of the hardware he intends to use. This is achieved by the use of variables which the ELL translates into computer main memory areas. The programmer has no idea which areas are being used, and furthermore, does not need to know. Further to this, complicated input and output procedures and peripheral handling are catered for in simplified statements where hand-shaking and timing problems are completely invisible.

When ELL's are not appropriate, or not available, then a lower language has to be employed. Next stage down are intermediate languages. These are not germane to the subject and will be omitted; which brings us further down to the assembly language level. The assembly language level is very machine-orientated and each instruction executable on the given machine has a mnemonic which corresponds to its function. There is, by and large, a one to one relationship between mnemonic and instruction. There is no automatic assignment of memory areas and the programmer as a consequence must be conversant with the functional structure of the processor.

No attempt has been made to define ELL or assembly language. All that is required is that the reader is aware of the concept of level in software and the difference between levels.
2.6. STRUCTURING

Structuring is a technique that was introduced into UMD's so as to make programming easier and to generally increase readability of programs (Ref. 6). An effect of working with a structured language is that a program has to be broken down into sub-tasks, each of which is treated independently and from the point of view of the overall program become simplified self-contained functions.

Structuring is achieved by the repeated use of a construct known as the Block. The Block is a self-contained sub-program with its own definition section and statement section just as any program would have. When variables are declared within the Block they can be referenced from within that Block only. If a variable that is not declared within that Block is found, then it is assumed that the variable has been declared in an outer Block and it is sought in the definitions of the outer Blocks. Thus, it can be seen that two Blocks, one within the other, may both contain a variable of the same name, but they will refer to different variables.

The effect of the above localisation of variables is to enable the programmer to deal with each Block virtually independently of all other Blocks. This enables him to use top-down or bottom-up analysis as he sees fit.
When a Block is executed, main memory is assigned to the variables declared and the Block is run. When the run has terminated, all the variables used by the Block, that were declared in the Block, are lost. They will of course be recreated next time the Block is activated, but the values of the variables are undefined. Since, as we have stated, Blocks may be nested one within the other, it also follows that subroutine Blocks may be nested, purely by the action of calling them. This opens up the possibility of a Block calling itself. This is known as recursion, and is permitted in many languages and can be of tremendous advantage in programming.
2.7. MICRO-PROGRAMMING

The micro-program is the lowest software level in the vast majority of digital computers, and as such, it is responsible for the direct control of all hardware units. Thus, any action that is to take place must be controlled directly by the output from the control unit, which is one of the binary patterns making up the micro-program. Every signal in the hardware which causes an action must originate from the micro-program, and obviously we cannot rely on all actions being required serially one after the other, so the micro-control unit must be capable of issuing simultaneous signals to the various elements under its control.

Micro-programming languages (Ref. 7) have been written for various machines with certain success, and attempts have been made to write general purpose micro-program assemblers, but they have not caught on particularly well. There are a number of difficulties involved in designing a micro-assembler, particularly of the general purpose type. These will be looked at one at a time.

1. Parallelism

The language must be able to express concisely all actions that are to occur simultaneously, thus we need a syntax to allow this and an internal checking mechanism to make sure that the actions are compatible. For example, it is not possible to load one register from two different sources.
2. **Permissibility**

There must be a means to test the existence of any given instruction and produce an error if it does not exist.

3. **Structure**

It would be of great advantage to be able to overlay some form of structure on the language and remove control transfer statements so as to provide readability in the program which is always a problem at this level.

What must be emphasised at this point is that the permissibility of micro-instructions is governed entirely by the hardware configuration. All functions that exist for the micro-program must exist in hardware. There can be no micro-program statements that are not present in hardware. This means that the structure of the hardware defines entirely the micro-instruction set and vice versa, the micro-instruction set partially defines the hardware structure. This is a very important point since it means that given a hardware setup, the micro-instruction set is predefined and could be found by mechanical means.
2.8. HARDWARE DESCRIPTION

Hardware description languages (HLL's) (Ref. 3) are programming languages that are designed for the express purpose of modelling hardware systems. Their variables are registers and memory units. Their data are the interaction and interconnection of these units. These languages are not like other programming languages for two reasons. Firstly, they do not define a dynamic system, there is no flow, the hardware is static, whereas normal programs define a series of actions. The HLL effectively describes a single complex 'action'. Secondly, and slightly more subtle; hardware converts the micro-program into the actions that are required. This process is analogous to the conversion performed by a HLL compiler, that produces machine executable code from non-readable source code.

Since the HLL is a medium for describing a system that converts microcode into the associated 'actions' and a HLL is a system that converts source code to machine code, it becomes apparent that the HLL is more akin to a translator writing system (TWS) (or compiler-compiler) than it is to the HLL itself. A TWS is basically a language designed for the description and programming of HLL compilers.
2.9. HARDWARE AND SOFTWARE INTERACTION

Computers are devices for solving problems. Programs are instructions to computers, directing the method of solution for a particular problem. Computer languages are media for the specification of these instructions. Given the above, it can be seen that there are two extreme design criteria around which a language may be designed. The first is the problem(s) to be solved, and this would produce a problem-orientated language. The second would be the structure of the computer itself. This produces a low level language.

From Diagram 2.b. it can be seen how the different levels and layers of computing interact. The purpose of the exercise is to find a solution to a given problem, thus we have the box top left, 'Problem', and another box bottom right, marked 'Solution'. There are various paths marked between the two, and these represent the different paths that can be taken in an attempt to solve a given problem.

The first stage represented in the Diagram has to be performed by humans and in the first four routes it is the conversion into some form of language from which the problem is soluble. Paths five and six are trivial, and are the routes that are used when the problem can be solved without mechanical aid. Square boxes have been used to show the different stages of the conversion.
process and in each of them there is a fixed or transient form expressing the remaining stages to the solution. The majority of these boxes are self-explanatory, and need not be looked at closely. There is one box, however, that may seem a little out of place. The stage 'Actions', may be said not to exist in the same way as a source program exists, but this is because they are dispersed through time. The reader may take this box as the result of the actions, it is really a question of semantics.

The first two paths use HLL's, the former being interpreted and the latter compiled. Route three is via assembly language while the next is via a micro-assembly language.

The different stages of translation will now be examined one by one.

CVT This a conversion from the HLL source to the HLL code. It is by and large a one to one conversion, and is common in languages like BASIC or APL.

CGE This is a compilation of the HLL source to produce executable machine code. The normal technique is by the look-up of Blocks of code which are then placed into the binary file. There is sometimes some form of code optimisation. Compilation is a one to many conversion.

ASS Assembly is the simple conversion from mnemonic representations of instructions to the binary code for the instructions. This is a one to one conversion although macros are commonly employed at this level.
SOFTWARE/HARDWARE RELATIONSHIPS 2b
A macro is a one to many substitution which enables possibly variable blocks of code to be placed in the binary file on a single instruction from the source.

Micro-assembly is similar to the preceding, but has lower level instructions and increased parallelisation. Macros have less applicability here.

Soft interpretation takes the HLL code and instruction by instruction converts it into machine code. This is usually done by data driven routines, where the data is the HLL code and each code is used to pick out an executable routine which is itself the machine code. This process is normally called interpretation but here has been termed soft interpretation so as to distinguish it from the following.

Firm interpretation. As in the above, this process is to expand one form of code into another. Again it is done instruction by instruction, and in many respects is the same as the previous. The main difference is that the micro-code that is substituted is held in a control memory while the source code is held in main memory, but again each instruction from the source is used to look up a batch of micro instructions which are then executed.

Hard interpretation. This is the stage where it might be said that the work is actually done. Each micro-instruction is actually performed one by one and if the actions are charted through time a whole series of actions are seen, which correspond to the micro-code instructions.
If this is unclear, then the reader should remember that the micro-instructions that are to be executed never exist as a list in its own right. It is only when all time is looked through that the list becomes apparent. Hard interpretation is one to one with no exceptions.

Conceptual interpretation. This stage may seem at best, out of place, but is in fact quite important. It must be remembered that all a computer does is switch voltage levels, nothing more and nothing less, and it is how these changes in voltage level are interpreted that has meaning and not the voltage levels themselves. This is more than a semantic argument, should a byte of memory containing a series of low voltage levels be called FALSE or ZERO, one logical, one arithmetical, the representation is the same, but meanings are incompatible.

After the preceding discussion, it should be quite apparent that the hardware of a computer behaves in much the same way as a computer language, and by similar reasoning, it can be seen that an HDL behaves like a TWS.
CHAPTER 3
INTERNAL REPRESENTATIONS

3.1. REPRESENTATION

When an HDL is employed, it is generally done so as to perform some kind of manipulation or processing upon the supplied data. The supplied data will be a description of a hardware structure and the data that is to be processed will be some form of representation of this description. As hardware description is being dealt with, one of the most obvious and useful applications is to simulation or emulation of the target machine. This would enable testing, programming, and debugging of programs before even a prototype machine is in existence.

Programming can be made easier since a complete description of the target machine could be linked to a general purpose assembly language and assembly could be performed by this system, during which process each instruction could not only be checked for syntax, but also admissibility. The assembled program could then be run on the pseudo-target machine and any alteration to either the software or the hardware could be made simply and easily. When this process has been completed, the machine would be left with a full description of the required hardware, complete with a full listing of the software in source and binary form. The hardware could be adjusted so as to fit commercially available packages and then output could be produced which could be directly interpreted into a wiring diagram.
Since the above can be performed, it may have struck the reader that the process can be used a little more deeply than is suggested. Since the machine is described fully and from this we can deduce the admissibility of instructions, then it should also be possible to provide a total listing of admissible instructions. In other words, from the hardware, it should be possible to itemise the complete instruction set.

When designing a processor from scratch, it is quite common for the instruction set to be known and the hardware structure to be secondary to this, and since from the proceeding argument we can derive the instruction set from the structure, then it would seem reasonable that the reverse would also be the case. This process would not be as simple as it may sound. One reason for this is that for one structure there is one and only one instruction set, but for one instruction set, there is any number of possible structures, and therefore, the designer would have to supply other criteria around which the structure can be produced. One further possibility that the possession of a full hardware representation can give is the decomposition of a higher level instruction which is non-executable into several permissible instructions. In other words, this would allow the software writer to use a form of language which, although not strictly of higher level, would be of a more convenient nature, since non-permissible instructions could be converted into small groups of permissible instructions. These instructions which have been converted could then be
analysed and it may be deemed prudent to alter slightly the original hardware, which may result in shorter and faster programs. In order to perform any such manipulation of the hardware and software of the target machine, it is necessary that there is some consistent means of representing the data that is in such a form that the manipulation is possible. A number of ways will be looked at in some detail. No attempt is to be made to completely define internal organisation, the proceeding section should be treated as an examination of various options, and possible ways of achieving the above specified aims. Some of the techniques have been used before and references to these can be found from the appropriate appendix. As far as is possible, the writer has assumed no prior knowledge of the various techniques on the behalf of the reader.
3.2. GRAPHS AND MATRICES

A graph is a map-like representation of a system and is simply a drawing consisting of small circles connected to one another by lines. The small circles are referred to as nodes, while the lines are referred to as edges or, sometimes, arcs. Each node may be connected to any other node, including itself; edges may not diverge, that is, each edge may have only two ends, each of them at nodes. In other words, an edge must connect two, and only two, nodes, although the two nodes can be the same one.

The application of graphs to hardware description should be obvious, the register transfer level is easily accommodated by allowing each register to be represented by a node and all interconnecting paths to be represented by edges. A path in hardware is uni-directional; in order to represent this, direction must be impressed upon the edges of the graph. This is simply done by drawing small arrows on the edges in the appropriate direction, then exclude bi-directional edges and replace with two anti-parallel edges. The graph is now termed a directed graph.

In application to hardware representation the directed graph can be viewed in one or two ways. If we take all registers and all paths, then the graph is a complete representational model of the target system. Alternatively if only certain registers and paths
are taken then the directed graph corresponding to these
may be interpreted as a representation of the transfer between
the registers along the paths in the indicated direction or as
an indication of the existence of the said registers and paths.
This smaller graph must, by its very nature, form a part of
the larger system graph and may be termed, a sub-graph. Now
the interesting situation has been reached, whence instructions
that are permissible become sub-graphs of the system graph and,
alternatively, the system graph can be found by 'summing' all
the instructional sub-graphs. This provides a means of checking
permissibility, and of obtaining a possible structure from the
instruction set. It may not be apparent at this stage, but it
is also possible to emulate the target machine by processing one
sub-graph against another. This will become more apparent later,
with the aid of diagrams.

In order to perform functions upon the directed graph, it is
necessary that a mathematical model be used. For this purpose,
we use matrices.

A matrix is simply a rectangular array of numbers usually enclosed
in square brackets. The numbers in the matrix, which are real,
are termed 'the elements of the matrix'. A matrix of \( n \) rows by
\( m \) columns will have \( (n \times m) \) elements. Rows run horizontally and
columns run vertically. The element in row \( i \) and also in column \( j \)
Example Graph (G1)

Matrix (N) of Graph G1

\[
\begin{bmatrix}
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

3.6. Corresponding graph and matrix
will be referred to as $e(i,j)$. If the number of rows is equal to the number of columns, then the matrix is said to be square, and $n$ is equal to $m$, is equal to the order of the matrix.

A graph consists of nodes $P(1)$ up to $P(q)$ and a number of edges interconnecting these $L(a,b)$ indicating an edge exists from node $P(a)$ to node $P(b)$. In order to translate a graph into matrix representation, the order of the matrix $n$ must be equal to the total number of nodes $q$ and for each edge $L(a,b)$ element $e(a,b)$ becomes 1, else remains zero. This would give a matrix where the total numbers of 1's should equal the total number of edges on the graph. The reader will find this clearer with reference to the diagram 3.e. There is a one to one relationship between graph and matrix, and it is a simple process to move between one and the other.

In order to obtain any sensible results a consistent way of operating on matrices is required. It is first essential that the reader understands what the matrix represents. It can at this stage be considered a boolean array where the 1 indicates true, an interconnection exists, and the 0 false, no interconnection exists. If we take a matrix $N$ representing a graph and perform the boolean NOT function on each element of $N$, then the result called $\neg N$ will be a matrix indicating all interconnecting paths that are not present.
In order to obtain a graph from the two sub-graphs we need to use element for element, the boolean OR function, thus

\[ M_1 = M_2 + M_3 \]. If \( M_2 \) and \( M_3 \) are the only sub-graphs of \( M_1 \) then the equation is true. If the intercept is required, then the boolean AND function is used element for element, and

\[ M_4 = M_2 \& M_3 \].

The last basic function is taken from conventional matrix multiplication, so if it is required to find the product \( M_2 M_3 \), then the normal procedure for multiplication is followed excepting the boolean functions for AND and OR replace the conventional functions of 'times' and 'plus' respectively. The result of such a multiplication is a matrix indicating the combined edges. Reference should be made to the diagrams.
3.3. USE OF GRAPHS AND MATRICES

It was stated in the previous section that graphs can be interpreted as either a representation of structure or instruction. This concept applies equally well to matrices and can be used to great advantage. Given a set of instructional graphs, then a set of matrices can be produced, and then if these matrices are OR'd together then the resulting matrix will be a representation of the hardware that would be required to execute any one of the given instructions. In other words, this simple action provides a means of obtaining the hardware structure from the instruction set, although no account of the control unit is taken all data paths can be handled.

There are a number of other tricks that can be managed with the use of matrices, some of which may be of use at the design stage. The first that shall be considered will be determination of path lengths within the structure. Let S represent the structural matrix, then $S \times S$ will be a matrix indicating paths of length 2, where the length is equivalent to the number of nodes passed through minus 1, or could be regarded as the number of individual edges passed through on the original graph from which S was taken. Also $S \times S \times S$ can be calculated and this, as expected, will indicate all paths of length 3. Unfortunately, although the above is true, no account is taken of loops that exist within S, and so the path lengths calculated may not be the shortest. It is
Graph of Structure

\[ S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \]

Graph of Instruction (1)

\[
I_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}
\]

\[ I_1 \subseteq S \]

\[ I_1 \text{ subgraph of } S \]

\[ I_1 \text{ permissible} \]

Graph of Instruction (2)

\[
I_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}
\]

\[ I_2 \subseteq S \]

\[ I_2 \text{ subgraph of } S \]

\[ I_2 \text{ permissible} \]

Graph of Compound Statement \((1, 2)\)

\[
I_1 \times I_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}
\]

\[ I_1 \times I_2 \not\subseteq S \]

\[ I_1 \times I_2 \text{ is not subgraph of } S \]

\[ i.e. \text{ compound statement is not permissible.} \]
Using \( n^2 = M_n M_{n-1} \) & \( \sum_{i=1}^{n-1} X^i \) gives:

\[
M_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}
\]

\[
M_2 = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}
\]

\[
M_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}
\]

\[
M_4 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}
\]

3.c. Path length determination
therefore necessary to find a way to exclude paths where there are routes of shorter length. The first step in this is to exclude immediate loops, that is nodes connected to themselves. This can be done by ANDing S with 'I, where I is the identity matrix and is obtained by filling the leading diagonal with 1's and all other elements with 0, thus 
\[ e(i, j) = 1 \text{ if } i = j. \]
This will be found to reduce every element of the leading diagonal of S to 0. If this action is performed at each stage of multiplication, then all plain loops will be removed. However, interrelated loops will not be removed. One way of handling these is to remove all equivalent routes of shorter length, for instance if \( S^2 \) is calculated as \((S \times S) \& 'S\) then the ANDing with 'S will exclude from \( S \times S \) all paths of length 2 which are duplicates of single length paths found in S. To calculate \( S^3 \) the following would be used \((S \times S \times S) \& 'S^2 \& 'S\). This process could be continued indefinitely. There are a number of other techniques of this nature which can be of value and if interested the reader is advised to seek references relating to graph theory (Ref. 9). See diagram 3.c.

Given two instructional matrices \( I_1 \) and \( I_2 \) then the product \( I_1 I_2 \) is a matrix representing an instruction which is equivalent to the execution of \( I_1 \) immediately followed by the execution of \( I_2 \). Similarly, the product \( I_2 I_1 \) would be an instructional matrix equivalent to the execution of the instruction \( I_2 \) immediately
followed by the execution of $I_1$. This provides a means of simulating the effect of the execution of several instructions.

If $I_1$ is a permissible instruction of structure $S$, then the graph of $I_1$ must be a sub-graph of the graph of $S$, and conversely, any instructional graph that is not a sub-graph of $S$ is not immediately executable. In order to test the admissibility of any instruction then we must check that all paths used by an instruction exist, or that no paths are non-existent, thus if the matrix $I \& 'S = 0$, that is, all elements are equal to zero, then the instruction is permissible. If, on the other hand, it is non-zero, then this matrix will indicate all paths which must be included in $S$ to make the instruction permissible. The above shows some useful techniques, it enables the checking of the validity of instruction and if the instruction is found to be invalid, then the paths necessary to make the instruction valid can be found.

If two instructional matrices are ANDed together, then the result is a matrix representing paths which are used by both the instructions. This process of ANDing instructions together can be utilised in a number of ways, one of which could be to examine the usage of various paths that exist in the hardware and exclude under used paths, for instance, paths not used at all would be indicated by the following:

$$S \& ('I_1 + 'I_2 + \ldots \ldots)$$
In concluding this section which has shown some of the ways in which matrices can be handled to great effect, it will be pointed out that true simulation of arithmetical and logical operations would have to be handled outside the matrix structure. Although this detracts to some extent the ease of use of the matrices, they still provide a powerful means of manipulating instruction sets and structures and small instruction sequences.
3.4. PETRI NETS

Petri nets are an extension of graph and net theory, developed over the last decade into a suitable model for representing and studying concurrent systems. They have a very wide field of application, but are ideally suited to the modelling and representation of computer systems. Since a great deal of work has been done in this direction, this section will only be an introduction to Petri nets, and the reader is advised to consult the references for further detail. (Ref. 10).

A Petri net can be regarded as a bipartite, directed graph

\[ N = (T, P, A) \]

where \( T = (t_1, t_2, \ldots, t_n) \) the set of transitions,

\( P = (p_1, p_2, \ldots, p_m) \) the set of places.

\( T \cup P \) are the nodes of \( N \).

\( A \subseteq (T \times P) \cup (P \times T) \) the set of directed arcs.

Petri Nets can be marked. A marking \( M \) of a net is a mapping.

\[ M : P \rightarrow I \quad I = (0, 1, 2, \ldots) \]

Tokens are assigned to places in the net by \( M \), thus \( M \) can be viewed as a vector whose \( i \)th element, \( M(i) \) represents the number of tokens \( M \) assigns to \( P_i \).

A Petri net \( N \) with marking \( M \) is

\[ C = (T, P, A, M). \]
3.6. Example Petri Net
Petri nets can be represented by drawings with small circles as places, transitions as bars (cross lines) and arcs (or edges) as lines with arrows indicating direction. See diagram 3.d.

The above definition of a marked Petri net gives the basic structure of modelling scheme but not the behaviour, this is specified by defining simulation rules.

Each transition is said to have a set of input places and a set of output places.

The set of input places of a transition \( t \), \( I(t) \) is those places that have arcs leading to the said transition.

\[
I(t) = \{ p/(p,t) \in \Lambda \}
\]

Similarly, the set of output places

\[
O(t) = \{ p/(t,p) \in \Lambda \}
\]

A transition, \( t \), is said to be enabled iff all input places to \( t \) have at least 1 token, i.e.

\[
M(p) > 0 \quad \text{for all} \quad p \in I(t)
\]

A transition can be fired if it is enabled, this is done by removing a token from each input place and placing a token in each output place. This gives a new marking.
A token cannot be divided, that is, it can be removed from a place by only one transition. Except for restrictions stated, firing of transitions is asynchronous.

In figure 3.d.1, $t_1$ is the only transition that can fire. On completion, $p_1$ is empty and $p_2$ and $p_3$ each contain a token. At this stage $t_2$ and $t_3$ are both enabled and can fire concurrently since they do not share any input places.

When these two firings are completed, $p_4$ and $p_5$ are the only places containing tokens (see diagram 3.d.2). This situation represents a conflict: both $t_4$ and $t_5$ are enabled, but firing of either disables the other. In such a case, the decision as to which one fires is completely arbitrary. The ability to represent both concurrency and conflict makes Petri nets very powerful.

Transitions in a Petri net could represent events in a real system. A marked net then represents the coordination or synchronisation of these events. The movements of tokens clearly shows which conditions cause a transition to fire and which conditions come into being on the completion of firing. Moreover, the nets are not based on any concept of a central system state.
3.e. Two-stage pipeline buffer
The nets provide a natural representation of systems where control and state information is distributed. The use of finite state machines in such cases often leads to unmanageably large single states.

Additional concepts: A marking $M'$ is immediately reachable from $M$ if the firing of some $t$ in $M$ yields $M'$. $M'$ is reachable from $M$ if it is immediately reachable from $M$ or is reachable from any marking which is immediately reachable from $M$ or is $M$ itself. The reachability set $R(M)$ of a marked Petri net $(T,P,A,N)$ is the set of all markings reachable from $M$. A place in a marked Petri net $(T,P,A,N)$ is $k$-bounded iff there exists a fixed $k$ such that $M'(p) \leq k$ for all $M' \in R(M)$. A marked Petri net is $k$-bounded if for some fixed $k$ each place is $k$-bounded. A place is safe if it is 1-bounded and a marked Petri net is safe if each place in it is safe. A transition $t$ in a marked Petri net $(T,P,A,N)$ is live if for each $M' \in R(M)$ there exists a marking reachable from $M'$ in which $t$ is enabled. A marked Petri net is live if each transition is live. Boundedness and liveness are important net properties.

Diagram 3,e, represents a simple two-stage pipeline with a buffer of size two between stages. Pipelines have frequently been used in high-performance computer systems such as the Texas Instruments Advanced Scientific Computer, the IBM 360/91, and the CRAY-1. A multiple functional unit computer resembling the CDC 6600 has also been modelled. Shapiro and Saint used Petri nets to generate
efficient CDC 6600 programs: An algorithm originally expressed in a conventional high-level language is represented as a net to remove incidental sequencing constraints imposed by the language. The sequencing constraints required by the target hardware are then introduced into the net. All sequences of which this net is capable are realisable on the target hardware and perform the desired functional mapping. (Ref. 10).
3.5. **STATE TRANSITION DIAGRAMS**

In a conventional block diagram resistors and memory units are represented as square boxes along with functional units which are represented in the same way. These are interconnected by lines which represent data paths between the data holding units through the functional units. If instead the diagram is rearranged so that the data holding units are placed in a line with the functional units below and the data holding units drawn again below the functional unit, and the interconnections drawn in such a way that data would always leave the top set of units, passed through the functional units and pass into the lower set of data holding units. This diagram could then represent one stage in a state transition diagram. The first row of data would be the state of the machine at say, time $t$, and the second row would be the state of the machine at time $t + 1$.

Calling the top row $S$, and the lower row $D$, (source and destination, respectively), then the instruction represented by the functional units would take data from $S$ and transfer it to $D$ so that $D = f(S)$. See diagram 3.f.1.

If another stage is added to the diagram then it could be interpreted as a representation of the changing state of the machine over two cycles, from $t$ to $t+2$. See diagram 3.f.2.

What is more, the combined effect that the two instructions could be interpreted via the paths selected for data transfer from the source registers to the destination registers. Even though the
\[ D := f(S) \]  \hspace{2cm} (1)
\[ D' := g(f(S)) \]  \hspace{2cm} (2)

\textit{3.f. SIMD representation of instructions}
function may not be executable than one cycle it could be decomposed into two instructions (or more) with use of the diagram. This could be accomplished by selecting appropriate paths through the two sets of functions from sources to required destinations. It would be possible to select the paths on a basic trial and error basis and when two data tokens attempt to pass through the same data port (the data port would be a register at a particular time between t and t + n), this would imply that the selector paths have failed and the process would start again on different paths, the process continuing until such time as a success is met. If success is not met, then it implies another stage should be added, and the complete process performed once again.

The best way to describe this action is by example, and so we will take a trivial example and examine the procedure.

Suppose we have a hardware structure as shown in diagram 3.6. and the instruction a := b, b:=a: is found in the source code, then with reference to the diagrams the procedure is as follows:—

no path exists from b to a', i.e. diagram must be be extended. Only one path exists from b to a', since only one token can pass through each data port the third data port can be removed, also for the time being, a' & b are of no interest.
Hardware:

```
required instruction A := B, B := A;
```

desire:

```
no path exists from B to A' i.e. we must extend.
```

Thus:

```
Only one path exists from B to A' (only one token can pass through each gate, thus the third gate can be removed. Also we are no longer interested in A' or B) thus:
```

Two paths exist for A to B'

Try gate 1.

It can be seen that this will leave no path for C to C'. Thus try A to B' through gate 2. This leaves C to C' through gate 1.

3.6 Instruction decomposition
In the next stage it is found that two paths exist from a to b' so an attempt to port no. 1. If this is done it can be seen that no path exists from c to c', therefore we take a to b' through port 2. When this is done we find left a path from c to c' through gate 1. This now leaves the final result and can be interpreted as two separate constructions viz.

\[ a := c, b := a, c := b; \]
\[ a := c, b := b, c := a; \]

In this way it can be seen that the original non-executable instruction has been translated into two executable instructions suitable for the hardware. Both the instructions and the hardware could have been represented in matrix form and it would not be difficult to include either monadic or diadic instructions. This does open up the possibility of supplying a mid-level or even possibly an HLL program and a description of the hardware on which it is to run and the executable instructions deduced from the source, and then with the aid of a built-in assembler, conversion to executable code.
This leaves us with a final successful result.

Thus:

\[
\begin{array}{ccc}
A & B & C \\
\text{ } & \text{ } & \{f_1\} \\
\text{ } & \text{ } & \{f_2\} \\
A' & B' & C' \\
\end{array}
\]

i.e. \( f_1 \equiv A := C, \ B := A, \ C := B; \)

\( f_2 \equiv A := C, \ B := B, \ C := A; \)
CHAPTER 4
REGISTER TRANSFER LANGUAGES

4.1. PARTICULAR LANGUAGES

In this chapter a brief look is taken at existing HDL's (or Register Transfer Languages). Four languages have been selected and some comparison between them will be taken. This section is not an introduction to various HDL's, it is intended as an outline to general languages in the field. CDL, APDL, APL and ISP are the four languages selected. (Refs. 12, 13, 14 & 15).

CDL is one of the most successful languages, it has been in use for several years and has been implemented for simulation and design automation. CDL is non-procedural.

APDL is an Algol based language with extensions for timing and register variables. APDL is basically a procedural language with the introduction of time blocks for concurrent execution.

APL is examined because of its very rich instruction set and its superior array handling capability. It is poor in representing parallel operations.

ISP was initially designed to describe the primitive operations from a programming angle. It can handle concurrency and sequencing in a simple manner.
4.2. PROCEDURAL AND NON-PROCEDURAL LANGUAGES

Procedural languages impose an ordering in the language in much the same manner as a conventional HLL. The activation of each function is initiated by the completion of the previous activities.

Parallel activities can be described by enclosing them in time blocks and the full description in the machine by a sequential list of time blocks.

Non-procedural languages have no order implied in the source code. Each statement is associated with a 'label' which is evaluated to determine if/when the statement(s) is executed.

The non-procedural language can be put in tabular form, each entry having the two sections i.e. activation and activity.
4.3. CDL

Computer Design Language. This language was originally proposed by Chu (Ref. 12). A translator and simulator were written for a subset of the language.

CDL is aimed at describing the structural and functional parts of a digital system. Structural components such as registers, clocks, switches and memory (register arrays) are declared explicitly at the start of the description. Functional behaviour is expressed by commonly used operators and user defined operators in the textual body of the description, and in this manner valid data paths are also implicitly specified.

All CDL variables are global and as a result there can be no partitioning of the total structure, i.e. no modular description.

The language is non-procedural with labels that are evaluated to determine when the statement is executed. Sequential actions can be expressed by employing the DECODE function on a clocked counter and then using each of the decoded lines as labels to the statements to be executed sequentially. This is naturally an inefficient means of describing sequential action, but in the writer's opinion, it is not reasonable to express hardware in a sequential form as by its nature, it is static, (it is the micro-program that is dynamically designed).
4.4. **APDL**

This language, as CDL, was adapted from ALGOL, although more rigidly. The language was written in ALGOL-60 and has been implemented on the CDC-G20. This is basically a procedural language with the employment of time blocks to represent simultaneous execution. The language overall has a block structure which allows for modular design (Ref. 13).

There are no special carriers as in CDL and it suffers from no memoryless carriers which reduces the ease of programming but does make for simpler implementation. APDL has a general base for carrier declaration whereas CDL is binary. This is perhaps an unnecessary extension but may possibly have its uses. The primitive operators provided are the standard NOT, AND, OR and X-OR which have been adjusted for non-binary elements. Vector operators are not provided although ADD and SUBTRACT arithmetic operators are provided. There is also the special exchange operator for convenience.

The full range of conditional operators are provided and also the IF THEN ELSE construct, for condition statements, along with the GOTO.

Lastly, recursive description of concurrent and sequential actions is not supported.
4.5. APL

This is a conventional HLL developed from Iverson's notation which is an extensive and powerful set of primitive operations. It has very good vector handling primitives and for this reason has been used for Hardware Description as well as a basic language for other languages, in particular AhPL and the ALERT system (Ref. 14).

APL itself is an HLL that was not designed specially to handle hardware and so will not be considered further.

All APL based languages are procedural and have poor readability due to the confusing number of operators and the unique form of condition transfer, e.g.

\[ \rightarrow (g \times 3 + g \times 10) \]

which is the AHPL equivalent of the BASIC statement.

IF \( g \) then GOTO 3 ELSE GOTO 10.

which is not only unreadable, but is also nonsensical.

Consider also the conditional assignment viz.

\[ A \leftarrow ((B \land C) \neq f) \lor(D\ast g) \]

The asterisk is used for conditional transfer but if \( f \) and \( g \) are both one, then the result in \( A \) presumably becomes \( (B \land C) \lor D \) but it is doubtful if this is the intention.

N.B. Both these statements appear in F J Hill's article 'Introducing AHPL' Computer, December 1974. The article goes
on to say that APL does grant economy of expression with its distinctive notation. While this is certainly true, the sequential nature of the language does mean that its descriptions are an abstraction of any system and specify what the system does rather than what it is.

APL has a great deal to offer the world of programming in general, but the language should be merged with a more structured language for it to be usefully universal.
4.6. I.S.P.

Instruction Set Processor is another language based on ALGOL (Ref. 15). It is unique in that it lies neither wholly in the procedural group nor in the non-procedural group. Concurrency is assumed by default with the conventional statement delimiter ';'; semicolon. If this is replaced by the introduced delimiter ';NEXT' then sequential action is assumed.

As has been stated, and the name suggests, this language was not designed to describe hardware, but to emulate a processor and as such is similar to the procedural ALGOL in describing what a given system does rather than what it is in structural terms.

I.S.P. is a powerful language that can be used to describe systems at different levels e.g. Structural, functional, behavioural (with the tendency to behavioural).

Block structuring is allowed along with recursive description of sequential and concurrent instructions.

Elements are not necessarily binary and along with APDL, the boolean operations have been redefined to accommodate this.
4.7. SUMMARY

From the proceeding section it can be seen that the general run of languages at present defined fall into two broad camps. The two camps are indicative of the problem orientated nature of the HDL field.

The first camp is typified by CDL and is the rigid description of functional units and is a language designed to reflect the structure of the unit which is given, and is therefore more difficult to design with initially.

The second camp is typified by AHPL and is aimed at reflecting the behaviour of a target system. Initially in design the required behaviour is known and the structure is not. Thus, the first trial description tends to be easy; but from structure it is difficult to relate to the description and thus further work becomes more difficult.

It must, in conclusion to this chapter, be restated that in the opinion of the writer, a non-procedural language is a more useful tool for the design of a system. This though has to be moderated with certain constructs which allow for greater use at the initial stages where the hardware has not as yet been visualised.
5.1. THE HIGH LEVEL LANGUAGE - PASCAL

In order to make the description of the proposed HDL more understandable, in this section it is intended to look at the HLL Pascal, in so far as it will relate to the syntax and structure of the proposed language (Ref. 11).

The basic structure of Pascal is based upon the repeated use of a construct known as the BLOCK. Each block has its own unique header which consists of a block type specification (which may be one of the following, PROGRAM, FUNCTION or PROCEDURE). This is followed by an identifier name, which is the means by which the block is identified, and lastly, there may be a series of parameters, which if present are enclosed in brackets. The header is then terminated by a semi-colon. The body of the block consists of six sections, all of which take the same form in each block. These sections must be present in the correct order, although the first five are optional.

Section 1  Keyword LABEL

This section will not appear in all implementations, it is nevertheless included here for completeness. This section allows the user to inform the compiler of integer values that will be utilised in the present block to represent line numbers. The line numbers may be used as labels to identify the destination...
of GOTO statements. It is as well to mention at this stage that the GOTO statement is frowned upon, and is by and large only used under exceptional circumstances. Indeed, in some versions of Pascal, the GOTO statement does not exist.

Example: \texttt{LABEL 123,456;}

Section 2 \hspace{1em} \textbf{Keyword \texttt{CONST}}

\texttt{CONST} is an abbreviated term for \texttt{Constant}, and this section is used to introduce an identifier to denote a literal numeric value, or a literal character. The identifier is used just as the literal value would be used and is compiled in the same way. Thus, a constant identifier cannot be assigned a value at all, nor can its value be changed, excepting by redeclaration in a lower block (see scope rules).

Example: \texttt{CONST ZED = 'Z'; PI = 3.142; BELL = CHAR(7);}

Section 3 \hspace{1em} \textbf{Keyword \texttt{TYPE}}

\texttt{TYPE} definition determines the set of values which variables of that type may assume. In addition to the standard simple types of \texttt{BOOLEAN}, \texttt{CHARACTER}, \texttt{INTEGER} and \texttt{REAL}, Pascal allows users to define types. In some implementations the additional types \texttt{BYTE} and \texttt{ALPHA} are predefined within the compiler.

\textbf{Simple Types} : \texttt{BOOLEAN} the two values denoted by the identifiers \texttt{TRUE} and \texttt{FALSE}. 
CHAR the characters in the ASCII standard, ordered according to their 7-bit ASCII codes.

INTEGER the whole numbers from -MAXINT to +MAXINT, where MAXINT is defined to be 32767 in most implementations.

REAL the floating point numbers within the range determined by the compiler.

* BYTE the subrange of INTEGER, 0...255.

* ALFA equivalent to the following type definition which would be permitted in full Pascal. ALFA = PACKED ARRAY [1..8] OF CHAR:

* Variants not always available as simple types.

Enumerated types (scalars): An enumerated type defines an ordered set of values by enumeration of the identifiers which denote these values. The ordering of these values is determined by the sequence in which the (constant) identifiers are listed.

Examples

PERSON = (ANN, DOB, FRED);

COLOUR = (RED, YELLOW, GREEN);
Subrange types: A type may also be defined as a subrange of another type, by indicating the smallest and largest value of the subrange. Often, this facility is restricted to sub-range of integer.

Examples  

HEXUAL = 0..15;  NEGATIVE = -MAXINT..-1;

Set types: A set type defines the range of values which is the power set of its base type. This means, that if the base type consists of a set of $N$ distinct values, then it is possible to construct $2^N$ subsets from the values of the base type. In most implementations the base type may be character, innumerated scalar, or some subrange of integer.

Examples  

GROUP = SET OF PERSON;  
TRAFFICLIGHT = SET OF COLOUR;  
Letters = SET OF CHAR;  
DayOfTheMonth = SET OF INTEGER;

Array types: An array type is a structured type consisting of a fixed number of components which are all of one type, called the component type. The component type may be any declared or simple type. The elements of the array are designated by indices which may be of type integer or sub-range of integer, innumerated scalar or character type. The array bounds may be specified as subranges, either explicitly or by subrange type identifier.
Examples

LINE = ARRAY [1..80] OF CHAR;
PAGE = ARRAY [1..24] OF LINE;
HEXLIST = ARRAY [HEXUAL] OF CHAR;

Record types: A record type is a structured type consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a field, an identifier which denotes it, and its type.

Examples

DAYNDJ1 = 1..31;
YRDATA = RECORD
  YEAR : INTEGER;
  MONTH : ALFA;
  DAY  : DAYNDJ1
END;

(* A coordinate may be considered as *)
XANDY = RECORD CASE INTEGER OF (*a single structure*)
  1 : (X,Y : INTEGER);  (*or two integers*)
  2 : (LSBX,MSBX,LSBY,MSBY : BYTE) (*or four bytes*)
END;

If the case identifier is provided then this is included in the record as a field whereas, in the case above, it is omitted.

File types: File types will not be dealt with.
Section 4  Keyword VAR

The keyword is an abbreviation for the term 'variable', and it is in this section that all variables that are to be used in the block must be defined, else they must have been previously defined from an outer block, see scope rules. The declaration of each variable consists of two parts, the former will be the name of the variable, and the latter will be the variable type, which may be any type that has previously been declared, or a predefined base type. Compile time errors will be produced for any variables used but not declared.

Examples

PASS,FINISH: BOOLEAN;
LETTER: CHAR;
MATRIX: ARRAY(1..100) OF REAL;
BUFFER: LINE;
COORD: XANDY;

Section 5  Keyword PROCEDURE or FUNCTION

This section is designed for the specification of sub-programs, FUNCTION being a value returning procedure. These sections consist of a header, as previously defined, followed by a block. The only additional point to be mentioned is in the functional block, an assignment must be made to the function block name, and this assignment should only be made once.
Section 6 Main Body

This section consists of a series of instructions separated by semi-colons and enclosed within BEGIN and END keywords. There is little point in going into the details of the language's syntax, but a simple example program can be found in the appendix.
5.2. **STRUCTURE**

Pascal is a well structured language. This has a number of advantages to the user. It enforces a strict methodology on the writing of the program and insists on a task being broken down into sub-tasks, each of which stands out relatively clearly in the source program. This is done by using a number of constructs, each of which will be looked at in turn. The first construct is the statement, which may be either a simple statement, that is one single assignment, assigning an expression to a single variable name of the same type as the processed expression. The only exception to this is the assignment of an INTEGER expression to a REAL variable and vice versa, where conversion is performed automatically in some implementations. Alternatively, the statement may be of compound form consisting of a series of simple statements separated by semi-colons, and enclosed with the bracket-like keywords BEGIN and END. Thus, anywhere a simple statement can be used a group of simple statements in the form of a compounded statement can also be used. The first construct is the alternative choice statement of

\[
\text{IF condition THEN statement 1 ELSE statement 2;}
\]

or

\[
\text{IF condition THEN statement;}
\]

It can be seen that the second version is merely a shortened form of the previous and that both merely offer an alternative path through which control can be passed. See diagram 5.a.
5.2. Basic Constructs
In order to obtain repeated execution of a statement it is necessary to be able to loop around the statement. For this purpose there are two versions of the iterative construct

\[
\text{REPEAT (statement 1; statement 2) UNTIL condition;}
\]

Where the brackets imply any number of simple statements

\[
\text{WHILE condition DO statement;}
\]

In the previous form the group of simple statements will be executed before the condition expression is tested. Whereas, in the latter form the condition is tested prior to the execution of statement, and hence the former will always execute at least one iteration, and the latter may execute none of the statements.

It will be noted that in all the above examples the condition must return a boolean function, that is, either TRUE or FALSE, see diagram 5.a.

There is a third looping construct which is for use when the number of iterations is known prior to entry into the loop. This is performed with the aid of a counting variable which is either incremented or decremented by one. There are two forms of slight difference, they are

\[
\text{FOR } A:=N1 \text{ TO } N2 \text{ DO statement:}
\]

\[
\text{FOR } A:=N2 \text{ DOWNTO } N1 \text{ DO statement:}
\]

Those statements are used with either literals or variables in place of the N's. The loop counter A must be a variable and takes
5.b. For A := B TO C DO Statement

CASE STATEMENT

A := B

IF A <= C THEN
  Statement
  A := A + 1
ELSE
  no match statement
      otherwise
  variable

Statement 1
Statement 2
Statement n
on all integer values from \( N_1 \) to \( N_2 \) inclusive. Note that the counting variable is always incremented by \( \pm 1 \).

The last construct is the multiple choice statement and has the form:

\[
\text{CASE } A \text{ OF} \\
\quad \text{label 1:statement 1;} \\
\quad \text{label 2:statement 2;} \\
\quad \vdots \\
\quad \text{label } N:\text{statement } N \\
\text{END OTHERWise statement } P;
\]

All labels must be literals (these labels need not be defined in the section \textit{LABEL}) and of the same type as \( A \), the action is that \( A \) is compared to each of the labels and when found to be equal that statement is executed. If no match is found, then statement \( P \) will be executed instead. The keyword \textit{OTHERWISE} will not be in all versions of Pascal, but is nevertheless a very useful addition. All labels must be unique, thus only one path can be taken through the \textit{CASE} statement. It is possible to give a statement two labels separated by a comma. See diagram 5.b.

The last point worth mentioning is that a statement could be a procedural name and an expression a functional name, and when found the relevant sub-programs will be called. Sub-programs may call themselves, this is called recursion. It is also possible to have an indirect recursion by allowing the execution of two sub-programs (or possibly more) calling each other.
5.3. SCOPE RULES OF PASCAL

The so-called scope rules only apply to languages which are structured with the use of the block. Each block having its own declaration of variables and these variables are referred to as local to that block, and they cannot be referenced to from without. As has been said previously, a Pascal program itself constitutes a block and therefore any variables local to this outer block are deemed global, that is, they can be referenced from anywhere within the program since by definition any further blocks must be within the outer block. The same rules apply to the calling of sub-programs so that a call to a sub-program is only allowed from blocks where the declaration is still valid.

The scope of a variable (the range over which it may be accessed) is primarily the block within which it was declared. However, this scope may be restricted if a nested sub-program includes the declaration of variables, among which is an identifier which is the same one declared in an outer block. Within the inner block, all occurrences of that variable identifier are taken to mean the one that was declared most recently (at the innermost level). This effectively excludes the block containing the inner declaration of the variables from the scope of the variable declared in the outer block.

In Pascal only sub-programs (procedures and functions) may be declared as blocks. It should be remembered that variables declared with a sub-program are only in existence during the
5.6. Stacking Procedure

```
STACK

CME

A
B
C

D

STACK POINTER

(a)

B
C
D

BLOCK ONE
VAR A, B, C: BYTE;
(SP = b)

CALL TWO

BLOCK TWO
VAR B, C, D: BYTE;
(SP = a)

LEAVE TWO

(SP = b)
```

```
execution

(SP = b)
```
execution of that block and because of this values will not be retained between successive calls to the block. Every time the routines are called the stack (the area in memory where variables are stored) is increased to accommodate the variables declared within the routine and conversely every time the routine is exited, the stack is shrunk and all local variables are lost. If the routine is called recursively (that is it makes calls to itself), then the stack is expanded the appropriate number of times and then when the routines are left it is contracted, again the appropriate number of times.

In other words, everytime a block is called, the stack is increased to accommodate the variables declared within it; also everytime a block is left the stack is shrunk and all the variables declared within are lost. Variables existing on the stack at the same time as another variable with the same name are masked from access since the last declared variable of that name will always be used. See diagram 5.c.
CHAPTER 6

A PASCAL BASED HARDWARE DESCRIPTION LANGUAGE (PHDL)

6.1. FORM OF PHDL

Hardware is a very large collection of logical gates, registers and so forth, cross connected with a multitude of simple interconnections causing the whole to be complex. Despite this complexity the totality is usually represented as a polygram with the aid of the black box technique. In other words, facets of the complete structure are abstracted and conceived of as having a planner nature of interconnection.

6.1.1. The Black Box

The black box is a means of describing units. It can be used in any field and is of particular usefulness in connection with digital design. The philosophy behind the black box is to describe the unit from an external behavioural point of view and completely ignore the mechanics of the internal actions. This gives the advantage of totally ignoring all internal details and concentrating on the aspects as they relate to the external world.

This technique also works in reverse in that all outside the subject circuit can be enclosed in the black box and the interface be viewed from within.

The reader should note that when an 8-bit width bus line is considered, the black box is being employed since the bus is
considered as a single entity and not as 8 separate lines.

6.1.2. Blocks

In order to accommodate this universal technique PHDL has been designed with an adapted form of the Pascal block, which enables the user to describe any particular unit and then refer to it by name.

The blocks are not identical to Pascal blocks, but nevertheless do function in a logical and consistent manner.

6.1.3. Source form

The source form of PHDL is very similar to Pascal and reads in much the same manner. There are exceptions, as might be expected, in the form of altered functions and conventions; but the changes are purely to accommodate the different nature of the subject matter i.e. Hardware not Software. Thus procedure and function blocks have had their name and effect altered so as to describe the static nature of hardware and not the flowing form of software programs.

6.1.4. Why Pascal?

This question by this time must have crossed the mind of the reader. It has been left to this time for the uninitiated to get some feel of Pascal and for all to appreciate the difficulty of describing hardware.
When designing a language to describe something different, it is natural to aim at a completely new language. This however, the author has rejected on several counts. Introducing a new language means that new terms, conventions and syntax must be learnt by all prospective users and this is usually either consciously or subconsciously rejected out of hand as an unnecessary labour. The living proof of this suggestion is the continued use of Fortran. The next major drawback is the sheer number of languages offered from various sources at the present time. The prospective user in effect, cannot see the wood for the trees, and will hence select the first to hand, whether the most suitable or not. An example of this is BASIC - few people could adequately defend the widespread use of this language today.

This leaves us with the adaptation of an existing language. Many might claim that this is just a half-hearted step to a new language and it thus the worst of both worlds. The writer would defend this by stressing the naturalness of adapting old forms to new, as has happened with Fortran and BASIC as well as all natural languages. APL is a very powerful language which all, including Iverson, probably thought would overrun all other languages, but it has not, and there are several reasons why. One being the non-standard character set, another the poor readability, but also because it is a long way from all other
languages.

Pascal is probably the most widely used structured language, and hence is the best choice for a hardware description language.

In conclusion, the writer will only add that it should be the aim of all compiler/interpreter writers to make their programs modifiable and hence subject to updates so that language and user can develop together.
6.2. DESCRIPTIVE FORM OF PHDL

The remainder of Chapter 6 is aimed at describing a PHDL Block section by section. The next chapter will be devoted to the scope rules of the language and the action and specifications of all predefined functions.

Statement constructs will also be handled. Particular details of syntax will not be described, the reader is referred to the partial syntax diagrams in the appendix.

Before proceeding, it is as well to note that each data item has a three part makeup, viz.

1. Definition
2. Declaration
3. Interconnection.

1. Definition is the functional/internal specification of the block (black box).

2. Declaration is the naming of the block which implies that this construct is to be used in the structure.

3. Interconnection appears in the body of a block and is how the item is interfaced with the rest of this block or structure.
All three parts must exist for all items although definition will be inherent for all predefined items.
6.3. CONSTANTS AND META-VARIABLES

A meta-variable, which is sometimes termed a book-keeping variable is a numerical field for use at compile time. It is not available at run time, or in connection with HDL's is not assignable as a register is.

A constant is a fixed numerical value, again not existing on the target system, but nevertheless is assignable to a true variable and is regarded as a fixed (hardwired) bit pattern. A constant is a fixed meta-variable.

The section for defining and declaring meta-variables is started with the keyword VALUE, then each declaration is made by the identifier being equated to the value specification, this being terminated by a semi-colon. The value specification may be an array of defined limits or a single quantity. The quantity should be a numeric literal or in the case of a meta-variable, the keyword VAR.

Examples

```
VALUE
   ONE = 1;
   MAT = ARRAY [1..3] OF 52 23 0;
   CHANGE = VAR;
   CHOICE = ARRAY [4..0] OF VAR;
   NEG = -2;
```
ONE corresponds to the numerical value 1, or it may be used to represent a bit-pattern of all zero's excepting a one in the lowest bit position (right-most bit). MAF is a group of three constants addressed as a matrix but in the same way as ONE is not assignable to. NEG, similarly to ONE can be regarded as the 1's complement of ONE. CHANGE and CHOICE may be used in much the same way as the constants. However, values can be assigned to them.

In all these cases true variables may be assigned from the identifiers and it is assumed that constant bit patterns are present in hardware and in the case of VAR type the then current value is assumed to be the constant.

Note that constant arrays are equivalent to ROM's and VAR type arrays are equivalent to PROM's.

All these book-keeping variables can be used anywhere in the current block (or lower blocks).

The bit-width of constants is not defined generally and is assumed to expand leftwards to fill the available space. It is suggested that the bit-width if it is desired to be expressed precisely then it should be done by postceeding the constant value by a colon and the required specification. As an example:

\[
\begin{align*}
9:5 & \quad 01001 \quad \text{(binary)} \\
-2:5 & \quad 11110 \quad \text{(binary)}
\end{align*}
\]
6.4. GENERAL SPECIFICATIONS

This section is initiated with the keyword TYPE and is used for the declaration of various identifiers which will be associated with various types of items. The types which may be employed here are any previously defined including all predefined base types.

There are three predefined base types which are all nodes and will be described here.

The first is the TERMINAL which is a simple memoryless carrier of single bit width. It may be used for buses and common point nodes. The output always takes on the value of the input unless it is triggered and in this case, when the trigger is false, the output is undefined or can be regarded as the high-impedance state of a tri-state buffer. When the trigger is high, it is as any other plain memoryless terminal, except when connected to its own input it latches.

Example

```
TYPE

BUS = ARRAY [1..9] OF TERMINAL;
LEAD = TERMINAL;
```

The second base type is MEMORY. This item is always triggered and behaves as a single input latch. When the trigger input is
false then the device is in the latched state, and holds the state as it was set just prior to the trigger going false.
When the trigger is true, then the device behaves as a terminal (see diagram).

The last base type is REGISTER, which is a conventional single input J-K master/slave flip-flop.

While the trigger is true the input is read, but the output is left unchanged. When the trigger falls (changes from true to false) the output is altered to the value the input was during the preceding half-cycle. The input is not read while the trigger is false. This is two Memories in series, the second with the negative trigger of the first.

<table>
<thead>
<tr>
<th>Device</th>
<th>trigger</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERM</td>
<td>T</td>
<td>Input</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>invalid/undefined</td>
</tr>
<tr>
<td>MEM</td>
<td>T</td>
<td>Input -1</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>Input</td>
</tr>
<tr>
<td>REG</td>
<td>T</td>
<td>Input -2</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>Input -1</td>
</tr>
</tbody>
</table>

See diagram.

The section is primarily for the specification of array types.
The arrays may be predefined types such as substructures, thus
allowing simple description of repetitive hardware. Any previously defined item may be arrayed.
6.5. **SUBSTRUCTURE SPECIFICATION**

A SUBStructure is a structure in its own right, although it forms a part of the overall structure.

External specifications and declarations may be used provided the identifiers are not redefined within the substructure. A substructure is a block and hence occurrence of identifiers are associated with the innermost definition of the said identifiers.

There are no restrictions on the complexity of substructures and hence any circuit may be specified as a substructure. This is done in the same way as the overall hardware is specified as a structure. That is, as a block with a header followed by the various sections, viz.

```
VALUE, TYPE, SUBS, NODE, FUNC
```

and followed by a compound statement.

From an internal point of view the writing of a substructure is independent of the external environment except where applicable in the form of particular special connections and the use of other definitions which it is not required to repeat.

It is naturally simple to construct a SUBS without any reference to external systems.
From without the item will be referenced by a single name which has been declared to be of the type specified by the SUBS header.

In order to refer to a particular location, say a terminal, then this is done by proceeding the SUBS identifier immediately by a fullstop (.) and the name of the terminal. This may be done any number of times for nested blocks.
6.6. OTHER SUBSTRUCTURES

For the convenience of the user it is necessary to have other forms of the substructure which can enable the user to model any conceptual need within the syntax of the language. What is meant by this is that although the previous described technique could handle any circuit requirement, it cannot match with the various ways that the subunits are viewed.

It will be shown later that the statement is of the basic form:

\begin{verbatim}
\text{\texttt{\textbackslash trigger}} NODE 0 := f (NODE 1 \ldots NODEn);\end{verbatim}

It has been demonstrated how the substructure can accommodate the various demands on nodal specification but this leaves two other areas to be handled within the conceptual aspect of the language. The first is the interaction between the receiving node (node 0) and the trigger and is dealt with next. The second is the relationship between the function (f) and the remaining nodes and this will be explained after.
6.7. **COMPLEX TERMINAL SPECIFICATIONS**

The **KEYWORD** to indicate that this will be a specification of a terminal type of substructure is **NODE**. For convenience the **NODE** can be given width or by default left as single bit width.

This is a different form of the previously defined item and will behave in the same manner as regards scope and access to locations.

This type of specification originates from the basic form of the statement, in that it is automatically implied that there is a particular relationship between the trigger term and the **NODE** on the left hand side of the assignment. The **NODE** specification allows the user the utmost flexibility in that he/she can define the node to perform any action on being triggered. In this way devices as complex as stacks and queues may be treated as single nodal items. In fact it is also possible to define the basic predefined nodes with this construct. Then these can be utilised anywhere within the structure under various names or even after being arrayed.

The specification header contains a trigger name enclosed in backslashes. This must be a logical variable (i.e. a single bit width variable) which will be assigned the value of the trigger as it appears in the calling statement and whatever
actions are to occur must be specified in the body of the block. Also included in the body of the block must be two assignments, one of them TO the node identifier, the other FROM the identifier. These are necessary for the transfer of the received and returned values of the node, they must appear at least once and be executed no more than once.

The width specification ability is included for convenience and should be used only when crosswise connections are included in the nodes. Despite the slightly different appearance, it must be remembered that a NODE is only a variation of SUBS and that all details of SUBS apply to NODE's.

The last point applies to all nodes. The input and output terminals of nodes are separate and distinct, despite this they are always referred to by a single name, and which of the two terminals is being referred to is determined by which side of the statement the identifier is placed. (If the identifier is on the right hand side of the statement then the trigger term is always evaluated as false). This should cause no confusion as it is the normal practice in most HDL's.
6.8. FUNCTION SPECIFICATION

A FUNCTION is an arithmetic/logical unit, examples of which are adders, subtracters, incrementers or decrementers.

Again, the function is merely an adaptation of the substructure and is designed purely for increased readability and conceptual form. The function itself is not TYPE'd although bit width may be specified and as usual default is one.

Each time a function is used it is interpreted as an equivalent piece of hardware being introduced into the overall structure, and not, a single piece being accessed from different parts of the circuit.

Operands are declared in the header, they are specified with bit-widths, enclosed in brackets and following the function identifier, this is followed by a colon as the bit width specification of the function itself. The header is then terminated with a semi-colon and a conventional block follows.

Operand terminals are considered as plain terminal as is the function output.
6.9. DECLARATION AND THE BLOCK BODY

All the above substructures were specifications and not declarations. After the definitions have been made then the language can be assumed to have such items within its capability. They cannot be used, as yet in the body of the block until they have been named and, as it were, deemed to exist.

The next section is for this purpose and appears only once in each block and always after all the specifications. The keyword is ITEMS and the section is similar to the Pascal section VAR.

In this section the item types are given names, and can then be used in the remainder of the block. All items appearing in the body of the block must be declared in this section, unless they have been declared in outer blocks and it is to these locations that the user is referring. Any undeclared items will produce compile time errors as the identifier will not be recognised. Several identifiers may be tied to the same item type by separating them by commas.

It should be noted that types are not as used in Pascal, since in EIDL items of different type may be assigned together whereas this is not allowed in a proper version of Pascal.
After the declaration of items a compound statement follows. This has been termed the body of the block since it is here that all the interconnections and assignments for this block are made. This is the description of the block from an internal aspect, and as such will stipulate how the unit behaves under various input conditions. This section is opened with the keyword BEGIN and closed with END, just as in Pascal, and in between are a number of statements. This section must be present, if only as "BEGIN END;" since it marks the termination of a block.
CHAPTER 7

PHDL SCOPE RULES AND STATEMENTS

7.1. THE PHDL STATEMENT

PHDL is a non-sequential language when used for the description of hardware and for this reason, requires some means of indicating when each statement is activated. This is the purpose of the trigger expression. Since the language is primarily aimed at low level description it is also necessary to be able to specify parallel operation. Parallel operation or execution is the simultaneous activation of two or more devices or, in the software context, statements. Thirdly, it is also advantageous to have some form of conditional statement along with other constructs and statement forms to handle decision based functions and assignments which are not so easily or sensibly modelled by the use of the trigger expression. Macros and condition compilation are also of great use, and will also be discussed.

PHDL is a language designed to have structure and thus each location within the represented hardware will possess scope. Scope is the range within the representation from which it can be reached. But if any location is limited in its scope, then this will automatically cause certain hardware configurations to be impossible to represent in a black box like fashion. This
limitation in scope has been overcome by using partial names which together make up the full and unique name of the location and thus each location has unlimited scope but a different full name from different locations.

Within this Chapter, the ordering of the different sections will also be explained.
7.2. OVERALL STRUCTURAL DESCRIPTION

After the header which simply provides a means of naming the structure described, the next section must be VALUE's. This section need not be present, but if it is present then it must be the first one. After this, optionally come in any desired order either the substructures or TYPE's.

These sections can be mixed, repeated and placed in any order. The TYPE section is a little different to the other three in that it does not form a block in its own right, and is merely used to give type names to arrays. It may seem a little unnecessary to provide a whole section for this purpose but the section has been provided to allow for expansion. The most obvious and useful addition would be the employment of the Pascal record specification which could be used to describe busses and the like as single named units with multifarious data paths within, all having their own sub-names.

These sections are Specifications. They are used to give a description of a unit association with a name and nothing more.

The next section is titled ITEM and must be placed after the aforementioned parts. This section is a declaration of what units will be present and what their descriptions are.
All items used but not declared will produce errors. Errors will not be produced for items declared but not used.

The final section is the compound statement which is the interconnection of all items declared in the preceding section.
7.3. THE SCOPE RULES

The term 'scope rules' is essentially inapplicable in context of PHDL. If the scope of any location is less than the global for an HDL then this poses a limitation on the connectivity of the structure as a whole. Despite this it is very desirable to decompose the structure into sub-units which can be dealt with on an individual basis, with little or no regard given to other units. This naturally includes what names can be given to all the locations within the subject unit.

PHDL deals with these conflicting aims by one simple expedient. This is to allow the referencing of an item from outside the block. This is done by first naming the block which is postceeded by a fullstop and the name of the item. Thus the item now has a two part name giving firstly which sub-unit the item may be found in and secondly, the actual name of the item.

This process may be extended indefinitely to allow for sub-structure nesting to any depth.

The process by which the subject is sought is as follows:

1. The first part of the identifier is compared to all items in the current block.

2. If the item is matched, then the extensions to the identifier are compared with extensions to the item
7.a. Item Access
name, if a match is not found, then the item corresponding to the identifier is not present.

3. If the item is not matched then the next block out is searched just as the 'current' block. If there are no more blocks then the search is over and the identifier is not matched.

The effect of the scope rules can be seen from the diagram 7.a.
7.4. **STATEMENT TYPES**

7.4.1. Simple statements

The simple statement is the basic building block for the language. It takes the form of a simple assignment of an expression to a 'variable'.

The expression may consist of the following functions marked with an asterisk.

<table>
<thead>
<tr>
<th>Character</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>tilde</td>
<td>NOT</td>
</tr>
<tr>
<td>-</td>
<td>minus</td>
<td>negative/subtract</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
<td>positive/add</td>
</tr>
<tr>
<td>&amp;</td>
<td>ampersand</td>
<td>AND</td>
</tr>
<tr>
<td>!</td>
<td>exclamation</td>
<td>OR</td>
</tr>
<tr>
<td>&lt;</td>
<td>left angle bracket</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>right angle bracket</td>
<td>greater than</td>
</tr>
<tr>
<td>=</td>
<td>equals</td>
<td>equal to</td>
</tr>
<tr>
<td>[ ]</td>
<td>brackets</td>
<td>enclose indices</td>
</tr>
<tr>
<td>( )</td>
<td>parenthesis</td>
<td>enclose operands</td>
</tr>
<tr>
<td>{ }</td>
<td>braces</td>
<td>group items</td>
</tr>
<tr>
<td>\ \</td>
<td>back-slashes</td>
<td>enclose trigger</td>
</tr>
<tr>
<td>:=</td>
<td>colon, equals</td>
<td>assignment</td>
</tr>
<tr>
<td>/</td>
<td>oblique</td>
<td>reduction</td>
</tr>
<tr>
<td>.</td>
<td>fullstop</td>
<td>various</td>
</tr>
<tr>
<td>:</td>
<td>colon</td>
<td>various</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
<td>delimiter</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td>separator</td>
</tr>
</tbody>
</table>

Parenthesis can also be used to indicate ordering of evaluation.
The order of processing is in effect irrelevant to the language as a whole but it is suggested that precedence be used as it is generally well established.

0) Parenthesis
1) Monadic functions
2) Dyadic functions

2.1 +,-
2.2 &
2.3 :
2.4 <,>
2.5 =

Note that the reduction function (/) in combination with a dyadic function is in effect monadic and hence should have precedence over dyadics.

Examples

\[ Z := \& / \{ ABCDEF \} \]
represents a multi-input AND gate, and
\[ Y := ! / \{ ABCDEF \} \]
represents multi-input OR gate.

Inputs to logical reduction simple statements are ignored if they are undefined due to trigger expressions. This allows the simulation of wired-OR and wired-AND.
\[ B := \{ C \, D \, E \, F \} \, [ \, A \, ] \]

This is the simple statement form of the selection function and has the effect of selectively assigning to \( B \) from \( C \) thru' \( F \) depending on the index \( A \) which has taken on values of 0, 1, 2 or 3. Any other value is in error and will be flagged as such.

\( C, D, E \) and \( F \) may be of any width but they must have the same width. The braced expression behaves just as any array type item and may be used anywhere the latter has been used.

\[ J := K \, [ \, 1, 2, 4 \, ] \]

This is equivalent to the following:


Continuous lines can also be indicated in shorthand by the use of the 'two fullstops' symbol (..), as in

\[ B := A \, [ \, 7..0 \, ] \]
\[ C := A \, [ \, 15..8 \, ] \]

which indicates that \( B \) is connected to the lower order byte of \( A \), while \( C \) is connected to the higher.

User defined functions can be used in expected simple way.

\[ J := \text{SUM} \, (I, K) \]

Up until now only simple items have been shown on the left hand side of the statement, but this need not necessarily be the case.

In fact
all forms of description, with the exception of functions
and constants may be used as receiving items, as in

\[ \{ J \[ K[1..3] \] L[7 \[ B[1] \] \} \] := Z \]

There are other forms of the simple statements involving
functions that are predefined that will be dealt with in a
later section.

7.4.2. Statements

The basic form of the statement is composed of one or more
simple statements, optionally preceded by a trigger expression.
The simple statements must be separated by commas.

7.4.2.1. Triggers

The trigger may be any logical or
arithmetic expression provided that when computed the result
is of single bit width. It must be well noted that this is not
the same as a value that computes to zero or one.

(i) \( A \[ 2..1 \] \)
(ii) \( A \[ 0 \] \)
(iii) \( A - B \)
(iv) \( A = B \)

(i) and (iii) are not acceptable as trigger expressions. (ii) and
(iv) are acceptable since they are of single bit width.

Note the equals function returns a value of 1 for equal and 0
for unequal, and it is single width.
Another example of a legal trigger is:

\& / A [ 1..4 ]

For the expression to be recognised and processed as a trigger, it must be enclosed in backslashes, as in

\ A [ 0 ] \n
7.4.2.2. Actions The action that takes place on the result of the evaluation of the trigger is determined by the characteristics of the receiving items in the simple statements that follow the trigger. If there is a change in the state of the trigger this is reported to every receiving item in the simple statements of the statement. The details of the action has been described in the case of predefined items and for others is a matter of the user definition.

7.4.2.3. Other forms In addition to the above a statement may also be of the conditional form:

IF (trigger type exp.) THEN (statement 1) ELSE (statement 2)

or without the ELSE part.

This can be used in two ways, one when the boolean expression is derived from meta-variables and constants. This constitutes the equivalent of condition assembly and the second is from declared items in which case it acts in a similar way to a triggered statement.
Another form is the iterative loop entered as:

```
FOR N1 := N2 to N3 DO statement.
```

Where N1, N2 and N3 must be meta-variables then this allows a neat and simple way to describe repetitive cross connections in an arrayed network since N1 can be used in expressions to describe various points across the network lattice.

7.4.2.4. Provided functions and other keywords. These are functions that are supplied purely for convenience and increased flexibility.

**WIDTH**

returns the width of any node.

**BITS**

returns the number of bits required to store any provided operand.

- straight binary if operand positive
- two's complement if operand negative.

**MAX**

the straight binary maximum value storable in the supplied node identifier operand.

**INC/DEC**

increment and decrement functions.

A := INC and A := DEC
equivalent to
\[ A := (A + 1) \mod (\text{MAX}(A)+1) \]
and \[ A := (A - 1) \mod (\text{MAX}(A)+1) \]
respectively.

Note the MOD function will accept negative input and return the expected value, thus
\[ (-1) \mod (8) \]
returns 7

CONTROL

This is a reserved word that identifies the control memory to the system. It is the output of the control memory and is used in such operations as assembly and micro-instruction set generation. The labelling of control gives the system a reference point from which to work, all control signals originate from this point. It must be remembered that CONTROL is not an item, it is purely and simply a label that points to the location that can be regarded as the origin of all control signals and the bit pattern found there can be considered to be the micro-code for the particular instruction being executed, or about to be executed for structures employing the pipeline technique. The reason for having this location known to the system is that during evaluation and examination of the structure under test, it is at this point that the control unit can be split to allow for full ripple through before the next instruction is tested. In other words, if the output of the terminal array CONTROL is not allowed to change then the target machine is held in
mid-instruction and by control over this small section of the whole, the target machine can be made to perform any desired function within its capability.

CLOCK
This is a self-explanatory type and provides the system supplied clock. The clock is a simple two phase device that returns values of 0 or 1. The clock is used by the system when engaged in simulation and examination of higher level instructions. Before the clock value is changed all ripple through is checked, latched states noted and race-around conditions reported as errors. Before simulation is started it must be possible to install a start value for the clock. In this way all initial conditions are specifiable.

The system having full control over the CLOCK and having the locations marked CONTROL has full control over the target machine, since any bit pattern can be pushed through location CONTROL and the effect of such clock change can be fully viewed. In the same way the process can be reversed and the instruction decoded to find what bit pattern must be present at CONTROL in order to affect the specified instruction.

DECODE
This function would have the same form as found in CDL and provide a means of decoding a binary value into single lines.
It would take the form:

\[ A := \text{DECODE}(B); \]

Where \( A \) would be a single bit array and \( B \) would be the variable to be decoded. Each element of \( A \) would be evaluated using the following rule:

\[
\text{IF } i = B \text{ THEN } A[i] := 1 \text{ ELSE } A[i] := 0; \]

This function should be provided as its use is very common in many circuits and is available as a single package from many different sources.

**ENCODE**

This is another function that some implementors may wish to include. As its name suggests it would be the reverse of the above mentioned function and would be of the form:

\[ B := \text{ENCODER}(A); \]

The encoding, would of necessity, be restricted into, say, \( B \) being assigned the ordinal position of the first bit of \( A \) set to 1. This function may seem clumsy, but nevertheless, is available as a standard package, and its use is common in such processes as interrupt encoding and keyboard encoding.

7.4.3. Compound statements

A compound statement is a number of statements separated by semi-colons and enclosed in the bracket like words

BEGIN and END.

The compound statement may be used anywhere a conventional statement can be used.
7.5. PHDL - SUMMARY AND CONTENTS

This section is a résumé of this chapter and the preceding one.

PHDL is an HDL that is based on Pascal, which is probably the most widely used of structural languages. PHDL is designed around the block concept and uses the block as a descriptive specification of the subunits that constitute the major components in the makeup of the structure.

As a language it is designed to extend no lower than the logical gate level because for any structure very little work takes place at levels lower than this.

There is no inherent upper limit to the elemental constructs and hence on upper limit.

The majority of processor designs have a central control memory from which all control signals emanate, and it is at this type of processor that PHDL is aimed. There are other techniques which are of late becoming apparent where the essence of control is not localised but is distributed throughout and behaviour of the system is determined by the system complete and not one particular locale. This second type of technique is catered for, but not aimed at.
Only a single phase clock is supplied as this is obviously the trend today, but it is always possible to immediately divide the clock and produce any number of phases.

Detailed syntactic analysis of expressions and higher level arithmetic functions have been omitted from this report. The omission is intentional since it is an arbitrary matter and does not affect the language style. The selection for inclusion for high-level arithmetic functions should be based on whether the language is aimed at modelling structures with complex arithmetic slaves or not. The choice of expression syntax is basically personal choice and there are only three main contenders.

1. Right to left precedence as in APL.
2. Reverse Polish Notation as in FORTH.
3. Conventional Functional Precedence as in Pascal.

Of these three the writer would select number three, but other designers will have various opinions.

Despite being a structured hardware description language, The scope of any item is global if the name is selected appropriately. The type of an item does not limit its name in any manner and some readers may regard this as a disadvantage. It is not uncommon for meta-variables to have a limited form of name
(e.g. first letter must be 'B').

It is strongly suggested that such a limitation is unnecessary and awkward for the user if any restriction is required by the user it can be self-imposed.

Finally the language has been arranged to handle more easily arrays and sections of arrays by freer use of indexing. Note only one dimensional indexing can be specified in brackets. If two dimensions are required then two sets of brackets should be employed. Although this may seem unorthodox, it is nevertheless legal in some versions of Pascal.
8.1.1. INTRODUCTION

This chapter is presented in order to show how the PHDL and its internal representation, using matrices, are used to analyse and adjust a given system. The result can be a hardware structure that is more tightly fitted to the program that is to be executed.

The subject matter for the whole of this chapter was abstracted from the lecture notes used at Loughborough University. The remainder of section 8.1 is a direct copy of the relevant information concerning a micro-programmed processor which was chosen as a basic model for its simplicity.

Section 8.2 contains the PHDL equivalent of the supplied processor, including the instruction set used. The section is rounded off with details of how the matrices would apply to the circuit, the instruction set and a modified version of the original structure.

In section 8.3, an alternative hardware is used to emulate the original machine. The devices selected as the alternative hardware are of the bit slice variety. This has been done to demonstrate the adaptability of the HLL to varying and relatively complex structures. In this section will be found two versions
of the program, the latter is presented merely for the reader to appreciate the way in which the language would internally represent the source form. This is not to say that the language would actually process the program in this form, but that the program is a symbolic representation of the internal form.

Section 8.4 provides a descriptive and demonstrative analysis of a very simple control unit. The control unit is designed on the most simple lines and is intended to be a suitable device to control the bit slice processor, but only in so far as the needs of the program.

The last section is merely a summary of Chapter 8.
2.1.2. SUPPLIED STRUCTURE DATA

POSSIBLE ORGANISATION OF THE EXECUTION NETWORK OF A SMALL MICROPROGRAMMED COMPUTER.

ABBREVIATIONS:

MAR : MEMORY ADDRESS REGISTER
RAM : RANDOM ACCESS MEMORY
MBR : MEMORY BUFFER REGISTER
PC  : PROGRAM COUNTER
SP  : STACK POINTER
IR  : INSTRUCTION REGISTER
R1, R2, R3, R4 : GENERAL PURPOSE REGISTERS
LC  : LOOP COUNTER
+1  : READ ONLY REGISTER CONTAINING +1
0   : READ ONLY REGISTER CONTAINING 0
-1  : READ ONLY REGISTER CONTAINING -1
SIGN : READ ONLY REGISTER CONTAINING -32768
15  : READ ONLY REGISTER CONTAINING 15
ALU : ARITHMETIC/LOGIC UNIT
A, B, C, D : BUSES

PC, SP and MAR are 13 bit registers. All other registers are 16 bit registers.
ALU FUNCTIONS

A + B (no shift)

A + B ( " )

A + B ( " )

A + B ( " )

A + B (shift right one place)

A + B ( " )

A + B ( " )

A + B ( " )

A + B (shift left one place)

A + B ( " )

A + B ( " )

A + B ( " )

+ 4 logic functions (not specified here)

RAM FUNCTIONS

READ

WRITE

NO MEMORY TRANSFER
SUPPLIED SCHEMATIC OF CPU 8a
SYMBOLS AND NOTATION

Symbol

\[
\begin{align*}
\text{Symbol} & \quad \equiv \\
\end{align*}
\]

Function: When \( A = 1 \), \( C = B \)

\( A = 0, \ C = Z \) (High Impedance)

Symbol

\[
\begin{align*}
\text{Symbol} & \quad \equiv \\
\end{align*}
\]

Function: When \( A = 1 \), \( B \) is loaded into \( C \) on a clock pulse.

When \( A = 0 \), \( C \) remains unchanged.

When \( A = \phi \), \( B \) performs the function according to the applied code \( \phi \).
## INSTRUCTION SET

### Mnemonic

<table>
<thead>
<tr>
<th>3-bit OPCODE</th>
<th>13-bit ADDRESS FIELD</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>000 M</td>
<td>Push contents of M onto stack</td>
</tr>
<tr>
<td>POP</td>
<td>001 M</td>
<td>Pop a word from the stack to M</td>
</tr>
<tr>
<td>JUMP</td>
<td>010 M</td>
<td>Jump to M</td>
</tr>
<tr>
<td>JNEG</td>
<td>011 M</td>
<td>Pop a word from stack and jump to M if it is negative</td>
</tr>
<tr>
<td>JZER</td>
<td>100 M</td>
<td>Pop a word from stack and jump to M if it is zero</td>
</tr>
<tr>
<td>JPOS</td>
<td>101 M</td>
<td>Pop a word from stack and jump to M if it is &gt;= 0</td>
</tr>
<tr>
<td>CALL</td>
<td>110 M</td>
<td>Call procedure at M and push return address onto stack</td>
</tr>
<tr>
<td>ADD</td>
<td>111 000000000000000</td>
<td>Pop T, Pop S, Push S + T</td>
</tr>
<tr>
<td>SUB</td>
<td>111 000000000000001</td>
<td>Pop T, Pop S, Push S - T</td>
</tr>
<tr>
<td>MUL</td>
<td>111 00000000000010</td>
<td>Pop T, Pop S, Push S x T</td>
</tr>
<tr>
<td>Opcode</td>
<td>OPCODE</td>
<td>13-bit ADDRESS FIELD</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>----------------------</td>
</tr>
<tr>
<td>DIV</td>
<td>111</td>
<td>0000000000011</td>
</tr>
<tr>
<td>RET</td>
<td>111</td>
<td>00000000000100</td>
</tr>
</tbody>
</table>

**NOTE:** M is a memory address, T is the word on top of the stack, and S is the word just below T on the stack.
8.1.3. SUPPLIED PROGRAM DATA

MICROPROGRAMMED INTERPRETER EXAMPLE

MAINLOOP:

MAR = PC;
MBr = RAM(MAR);
IR = MBr;
PC = PC + 1;

IF BIT (15, IR) = 1 THEN GO TO FIRST;
IF BIT (14, IR) = 1 THEN GO TO SECOND;
IF BIT (13, IR) = 1 THEN GO TO POP;

PUSH:

MAR = IR;
MBr = RAM(MAR);
MAR = SP;
RAM(MAR) = MBr;
SP = SP + 1;
GO TO MAINLOOP;

POP:

MAR = SP;
MBr = RAM(MAR);
SP = SP +(-1);
MAR = IR;
RAM(MAR) = MBr;
GO TO MAINLOOP;

FIRST:

IF BIT (14, IR) = 1 THEN GO TO THIRD;
IF BIT (13, IR) = 1 THEN GO TO JPOS;

JZER:

LC = 15;
MAR = SP;
MBr = RAM(MAR);
SP = SP +(-1);
JLOOP: IF BIT (15,MBR) = 1 THEN GO TO MAINLOOP;
MBR = LEFT SHIFT (MBR + 0);
LC = LC * (-1);
IF BIT (15,LC) = 0 THEN GO TO JLOOP;
GO TO JUMP;

JPOS: MAR = SP;
MBR = RAM(MAR);
SP = SP + (-1);
IF BIT (15,MBR) = 0 THEN GO TO JUMP;
GO TO MAINLOOP;

SECOND: IF BIT (13,IR) = 1 THEN GO TO JNEG;

JUMP: PC = IR;
GO TO MAINLOOP;

JNEG: MAR = SP;
MBR = RAM(MAR);
SP = SP + (-1);
IF BIT (15,MBR) = 1 THEN GO TO JUMP;
GO TO MAINLOOP;

THIRD: IF BIT (13,IR) = 1 THEN GO TO FOURTH;

CALL: SP = SP + 1;
MBR = PC + 0;
MAR = SP;
RAM(MAR) = MBR;
GO TO JUMP;

FOURTH: IF BIT (10,IR) = 1 THEN GO TO REP;
IF BIT (11,IR) = 1 THEN GO TO NULDIV;
ADDSUB:  
MAR = SP;
MBR = RAM(MAR);
SP = SP + (-1);
IF BIT (12, IR) = 0 THEN GO TO ADD;
MBR = MBR + 1;

ADD:  
R1 = MBR;
MAR = SP;
MBR = RAM(MAR);
R1 = MBR;
MBR = R1 + MBR;
RAM(MAR) = MBR;
GO TO MAINLOOP;

RET:  
MAR = SP;
MBR = RAM(MAR);
SP = SP + (-1);
IR = MBR;
GO TO JUMP;

MULDIV:  
IF BIT (12, IR) = 1 THEN GO TO DIV;
MUL:  
MAR = SP;
MBR = RAM(MAR);
SP = SP + (-1);
R3 = MBR;
MAR = SP;
MBR = RAM(MAR);
LC = 15;
R1 = 0 + 0; R4 = 0 + 0;
IF BIT (15, MBR) = 0 THEN GO TO NEXT;
MBR = MBR + 1;
NEXT: IF BIT (15,R3) = 0 THEN GO TO MULLOOP;
R3 = 1 + R3;

MULLOOP: IF BIT (0,MER) = 0 THEN GO TO NOADD;
R1 = R1 + R3;

NOADD: MER = RIGHT_SHIFT (MER + 0);
R4 = RIGHT_SHIFT (0 + R4);
IF BIT (0,R1) = 0 THEN GO TO NOCARRY;
R4 = SIGN + R4;

NOCARRY: R1 = RIGHT_SHIFT (R1 + 0);
IF BIT (15,R3) = 0 THEN GO TO MULEND;
IF BIT (14,R1) = 0 THEN GO TO MULEND;
R1 = R1 + SIGN;

MULEND: LC = LC + (-1);
IF BIT (15,LC) = 0 THEN GO TO MULLOOP;
MER = 0 + R4;
RAH(MAR) = MER;
GO TO MAINLOOP;

DIV: MAR = SP;
MER = RAH(MAR);
SP = SP + (-1);
R3 = MER; R2 = MER;
MAR = SP;
MER = RAH(MAR);
R1 = 0 + 0;
R4 = MER;
MER = MER + 1;
LC = 15;
IF BIT (15, R1) = 1 THEN GO TO DIVPOS;

R4 = R1;
R2 = R2 + 1;

DIVPOS:

RER = 1 + R3;
IF BIT (15, R3) = 0 THEN GO TO DIVLOOP;
R3 = RER;
RER = RER + 1;

DIVLOOP:

R1 = LEFT_SHIFT (R1 + 0);
IF BIT (15, R4) = 0 THEN GO TO NOCARRY 2;
R1 = R1 + 1;

NOCARRY2:

R4 = LEFT_SHIFT (0 + R4);
R1 = R1 + RER;
IF BIT (15, R1) = 1 THEN GO TO DIVNEG;
R4 = R4 + 1;

DIVNEG:

IF BIT (15, 1) = 0 THEN GO TO DIVPOS;
R1 = R1 + R5;

DIVPOS:

LC = LC + (-1);
IF BIT (15, LC) = 0 THEN GO TO DIVLOOP;
IF BIT (15, R2) = 0 THEN GO TO DIVEND;
R4 = 1 + R4;

DIVEND:

RER = 0 + R4;
MAX(MAR) = RER;
GO TO MAINLOOP;
STRUCTURE ORIGINAL:

(* THIS IS A PARTIAL PHDL DESCRIPTION OF THE SUPPLIED CIRCUIT. THE DESCRIPTION IS COMPLETE ENOUGH FOR AN UNDERSTANDING OF THE TECHNIQUE OF TRANSCRIBING FROM DIAGRAM TO A PHDL STRUCTURE. *)

VALUE SIGN = -32768;

TYPE T16=ARRAY[15..0] OF TERM;
R16=ARRAY[15..0] OF REG;
R13=ARRAY[13..0] OF REG;
M16=ARRAY[16..0] OF MEM;

SUBS ALU_SUB;

TYPE T4=ARRAY[3..0] OF TERM;

ITEM A,B,C:T16;
   SIGNAL:T4;
   F:ARRAY[0..11] OF TERM;

BEGIN
   F:=DECODE(SIGNAL);
   \[ F[0] \ C:=A+B; \]
   \[ F[1] \ C:="A+B; \]
   \[ F[2] \ C:=A+"B; \]
   \[ F[3] \ C:="A+"B; \]
   \[ F[4] \ C:={(A+B)[14..0] 0}; \]
   \[ F[5] \ C:={("A+B)[14..0] 0}; \]
   \[ F[6] \ C:={(A+"B)[14..0] 0}; \]
   \[ F[7] \ C:={("A+"B)[14..0] 0}; \]
   \[ F[8] \ C:=(A+B)[15,15..1]; \]
   \[ F[9] \ C:=("A+B)[15,15..1]; \]
   \[ F[10] \ C:=(A+"B)[15,15..1]; \]
   \[ F[11] \ C:=("A+"B)[15,15..1]; \]
END; (* THIS IS NOT THE SHORTEST FORM. *)

ITEM MBR, IR, R1, R2, R3, R4, LC :R16;
   MAR, PC, SP :R13;
   RAM :ARRAY[0..MAX(MAR)] OF M16;
   A, B, C, D :T16;
   R, W :TERM;
   ALU :ALU_SUB;

(* PLUS VARIOUS OTHER SIGNALS WHICH ARE UNMARKED. *)

S :ARRAY[1..15] OF TERM;
T :ARRAY[1..15] OF TERM;
U :ARRAY[1..9] OF TERM;
V :ARRAY[1..3] OF TERM;
BEGIN

ALU. B := B;
ALU. A := A;
C := ALU. C;

\SE 11 \ A := MAR;
\SE 21 \ A := MBR;
\SE 31 \ A := PC;
\SE 41 \ A := SP;
\SE 51 \ A := IR;
\SE 61 \ A := R1;
\SE 71 \ A := R2;
\SE 81 \ A := R3;
\SE 91 \ A := R4;
\SE[10] \ A := LC;
\SE[11] \ A := 1;
\SE[12] \ A := 0;
\SE[13] \ A := -1;
\SE[14] \ A := SIGN;
\SE[15] \ A := 15;

\TE 11 \ B := MAR;
\TE 21 \ B := MBR;
\TE 31 \ B := PC;
\TE 41 \ B := SP;
\TE 51 \ B := IR;
\TE 61 \ B := R1;
\TE 71 \ B := R2;
\TE 81 \ B := R3;
\TE 91 \ B := R4;
\TE[10] \ B := LC;
\TE[11] \ B := 1;
\TE[12] \ B := 0;
\TE[13] \ B := -1;
\TE[14] \ B := SIGN;
\TE[15] \ B := 15;

\UC11 \ MAR := C;
\UC21 \ PC := C;
\UC31 \ SP := C;
\UC41 \ IR := C;
\UC51 \ R1 := C;
\UC61 \ R2 := C;
\UC71 \ R3 := C;
\UC81 \ R4 := C;
\UC91 \ LC := C;

\VE11 \ D := C;
\R & \VE21 \ D := RAM[MAR]; (* THESE SIGNALS TIED TOGETHER *)
\VE31 \ MBR := D;
\W \ RAM[MAR] := MBR

END.
PROGRAM EXERCISE;

BEGIN
MAINLOOP: MAR:=PC;
  MBR:=RAM[MAR];
  IR:=MBR;
  PC:=PC+1;
  IF IR[15]=1 THEN GOTO FIRST;
  IF IR[14]=1 THEN GOTO SECOND;
  IF IR[13]=1 THEN GOTO POP;
PUSH: MAR:=IR;
  MBR:=RAM[MAR];
  MAR:=SP;
  RAM[MAR]:=MBR;
  SP:=SP+1;
  GOTO MAINLOOP;
POP: MAR:=SP;
  MBR:=RAM[MAR];
  SP:=SP-1;
  MAR:=IR;
  RAM[MAR]:=MBR;
  GOTO MAINLOOP;
FIRST: IF IR[14]=1 THEN GOTO THIRD;
  IF IR[13]=1 THEN GOTO JPOS;
JZER: LC:=15;
  MAR:=SP;
  MBR:=RAM[MAR];
  SP:=SP-1;
JLOOP: IF MBR[15]=1 THEN GOTO MAINLOOP;
  MBR:=MBR[14..0] 0);
  LC:=LC-1;
  IF LC[15]=0 THEN GOTO JLOOP;
  GOTO JUMP;
JPOS: MAR:=SP;
  MBR:=RAM[MAR];
  SP:=SP-1;
  IF MBR[15]=0 THEN GOTO JUMP;
  GOTO MAINLOOP;
SECOND: IF IR[13]=1 THEN GOTO JNEG;
JUMP: PC:=IR;
  GOTO MAINLOOP;
JNEG: MAR:=SP;
  MBR:=RAM[MAR];
  SP:=SP-1;
  IF MBR[15]=0 THEN GOTO JUMP;
  GOTO MAINLOOP;
THIRD: IF IR[13]=1 THEN GOTO FOURTH;
CALL: SP:=SP+1;
  MBR:=PC;
  MAR:=SP;
  RAM[MAR]:=MBR;
  GOTO JUMP;
FOURTH: IF IR[10]=1 THEN GOTO RET;
ADDSUB: MAR:=SP;
    MBR:=RAM[MAR];
    SP:=SP-1;
    IF IR[12]=0 THEN GOTO ADD;
    MBR:=MBR+1;
ADD: R1:=MBR;
    MAR:=SP;
    MBR:=RAM[MAR];
    R1:=R1+1;
    MBR:=MBR+1;
    RAM[MAR]:=MBR;
    GOTO MAINLOOP;
RET: MAR:=SP;
    MBR:=RAM[MAR];
    SP:=SP-1;
    IR:=MBR;
    GOTO JUMP;
MULDIV: IF IR[12]=1 THEN GOTO DIV;
MUL: MAR:=SP;
    MBR:=RAM[MAR];
    SP:=SP-1;
    R3:=MBR;
    MAR:=SP;
    MBR:=RAM[MAR];
    LC:=15;
    R1:=0, R4:=0;
    IF MBR[15]=0 THEN GOTO NEXT;
    MBR:=MBR+1;
NEXT: IF R3[15]=0 THEN GOTO MULOOP;
    R3:=R3+1;
MULOOP: IF MBROJ=0 THEN GOTO NOADD;
    R1:=R1+R3;
NOADD: MBR:=MBR[15,15,1];
    R4:=R4[15,15,1];
    IF R1[10]=0 THEN GOTO NOCARRY;
    R4:=R4+SIGN;
NOCARRY: R1:=R1[15,15,1];
    IF R3[15]=0 THEN GOTO MULEND;
    IF R1[14]=0 THEN GOTO MULEND;
    R1:=R1+SIGN;
MULEND: LC:=LC-1;
    IF LC[15]=0 THEN GOTO MULOOP;
    MBR:=R4;
    RAM[MAR]:=MBR;
    GOTO MAINLOOP;
DIV: MAR:=SP;
    MBR:=RAM[MAR];
    SP:=SP-1;
    R3:=MBR, R2:=MBR;
    MAR:=SP;
    MBR:=RAM[MAR];
    R1:=0;
R4:=MBR;
MBR:="MBR+1;
LC:=15;
IF MBR[15]=1 THEN GOTO DIVPOS;
R4:=MBR;
R2:="R2+1;
DIVPOS: MBR:="R3+1;
IF R3[15]=0 THEN GOTO DIVLOOP;
R3:=MBR;
MBR:="MBR+1;
DIVLOOP: R1:={ R1[14..0] 0 };
IF R4[15]=0 THEN GOTO NOCARRY2;
R1:=R1+1;
NOCARRY2: R4:={ R4[14..0] 0 };
R1:=MBR+R1;
IF R1[15]=1 THEN GOTO DIVNEG;
R4:=R4+1;
DIVNEG: IF R1[15]=0 THEN GOTO DIVPOS;
R1:=R1+R3;
DIVPOS: LC:=LC-1;
IF LC[15]=0 THEN GOTO DIVLOOP;
IF R2[15]=0 THEN GOTO DIVEND;
R4:="R4+1;
DIVEND: MBR:=R4;
RAM[MAR]:=MBR;
GOTO MAINLOOP
END.
INSTRUCTION SET GENERATED FROM PROGRAM.

THERE ARE 39 INSTRUCTIONS.

IR:=MBR;
LC:=15;
LC:=LC-1;
MAR:=IR;
MAR:=PC;
MAR:=SP;
MBR:=MBR+1;
MBR:=MBR+R1;
MBR:=MBR[15,15..1];
MBR:=PC;
MBR:=R4;
MBR:=RAM[MAR];
MBR:={MBR[14..0] 0};
MBR:="MBR+1;
MBR:="R3+1;
PC:=IR;
PC:=PC+1;
R1:=0, R4:=0;
R1:=0;
R1:=MBR+R1;
R1:=MBR;
R1:=R1+1;
R1:=R1+R3;
R1:=R1+SIGN;
R1:=R1[15,15..1];
R1:=[ R1[14..0] 0 ];
R2:="R2+1;
R3:=MBR, R2:=MBR;
R3:=MBR;
R3:=R3+1;
R4:=MBR;
R4:=R4+1;
R4:=R4+SIGN;
R4:={ R4[15,15..1];
R4:={ R4[14..0] 0 };
R4:="R4+1;
RAM[MAR]:=MBR;
SP:=SP+1;
SP:=SP-1;
8.2.4. MATRIX REPRESENTATION

Section 8.1. shows the original program and descriptive material on the supplied CPU. This material was used to generate the PHDL structure shown in 8.2.1, and the PHDL program shown in 8.2.2. It will be noted that the program is a direct copy, the only alterations being ones of syntax and language. The structure is a direct representation of the supplied diagram, it is as a result, incomplete, being without any description of the control unit, and is thus purely a representation of the RALU and main memory. It is from this aspect that the structure and program will be analysed.

Section 8.2.3. shows the instruction set that was used in the program. This was generated automatically by a simple BASIC program which merely stripped off all labels and rejected all non-assignment statements and stored the remainder, provided there was not an equivalent statement already saved. These were then sorted and printed. The whole program took only 40 lines, which by any measure, is a short program, yet the resulting output is informative and useful.

In this section will be found three large matrices which have been constructed in different ways. All the matrices are organised to omit any details of the ALU and purely show the
connectivity between the ALU and the remaining units making up the CPU.

The first matrix was constructed directly from the supplied hardware and examination of this clearly shows the A, B and C buses, and by counting the total number of "1's", we find that there are a total of 43 interconnections.

If attention is now turned to the second matrix, which was constructed from the instruction set which was generated automatically from the program, examination of the lists still shows the need for the same buses, plus an extra bus, which is the output from the memory buffer register, but more revealing than this is the fact that the total number of 1's, that is the total number of connections required, has fallen to 39. This is in fact a saving of 9.3% in paths which have been shown to be redundant, since the supplied program could still run on the structure represented by this matrix.

Not shown here in detail, but only as a result, further work was performed on the instruction set, but this time from the ALU's point of reference, and again it was found that there was redundancy. It was also shown that if the ALU functions were altered to become the following instructions

- increment A
- decrement A
- Shift_Left A
Shift_Hight A
Negate A
A plus D
Zero

Then the external circuit could be reduced still further, and this is shown by the third matrix which has a weight (total number of 1's) of only 32, which represents a saving of 25.6% over the original. It should also be noted that the total number of ALU functions is reduced, albeit at the expense of slightly more complex instructions.

This technique will naturally reduce the generality of the target system, but it does show that a fairly substantial saving over more orthodox techniques can be made. The same analysis of hardware can of course be applied to the control unit with similar analysis and reduction techniques being applied and the effect of all this is to tailor the hardware to the particular program that is to be run on it. The tighter the hardware is tailored to the program then the less general-purpose it becomes, but savings are made in terms of construction costs and design time.

This example has purposely been made to only show the essence of the technique so as to not cloud the issue with extraneous
<table>
<thead>
<tr>
<th>MAR</th>
<th>LDR</th>
<th>IR</th>
<th>PC</th>
<th>SP</th>
<th>LC</th>
<th>R4</th>
<th>R3</th>
<th>R2</th>
<th>R1</th>
<th>ALU.A</th>
<th>ALU.B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Matrix of Original Hardware**
<table>
<thead>
<tr>
<th>MAR</th>
<th>MBR</th>
<th>IR</th>
<th>PC</th>
<th>SP</th>
<th>LC</th>
<th>RAM</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>ALU.A</th>
<th>ALU.B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**MATRIX GENERATED FROM INSTRUCTION SET**
<table>
<thead>
<tr>
<th>MAR</th>
<th>MBR</th>
<th>IN</th>
<th>PC</th>
<th>SP</th>
<th>LC</th>
<th>RAM</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>ALU.A</th>
<th>ALU.B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Matrix with Modified ALU**
Lastly it is suggested that any reader intending to implement such a scheme should work through such an example since this is the best way to fully understand the intricacies, which it would be pointless for the writer to spell out.
8.3.1. SECOND EXAMPLE

This second example has been included in order to demonstrate the applicability of VHDL to the description of various hardwares running functionally identical software. In addition, bit slice devices were selected, to show how the language can be used to represent such structures with relative ease.

Sub-section 8.3.2 is a VHDL description of the bit slice processor that the remainder of 8.3. refers to. 8.3.3. and 8.3.4. show the program and the instruction set. The similarity between this version of the program and the VHDL version of the original is intentional. The program was only adjusted where applicable so as to perform the desired functions. Rewriting the whole example would have been a tiresome and fruitless exercise since comparison would have been made impossible.

The following equivalences have been used:

<table>
<thead>
<tr>
<th>MAR</th>
<th>ADDR.D</th>
</tr>
</thead>
<tbody>
<tr>
<td>INBUF</td>
<td>DATA.Q</td>
</tr>
<tr>
<td>OUTBUF</td>
<td>DATA.D</td>
</tr>
<tr>
<td>PC</td>
<td>RALU.RA[0]</td>
</tr>
<tr>
<td>R1</td>
<td>RALU.RA[1]</td>
</tr>
<tr>
<td>R2</td>
<td>RALU.RA[2]</td>
</tr>
<tr>
<td>R3</td>
<td>RALU.RA[3]</td>
</tr>
<tr>
<td>R4</td>
<td>RALU.RA[4]</td>
</tr>
</tbody>
</table>
Due to the limitations of the typewriter the double equals sign should be read as 'equates to' and the parentheses interpreted as brackets. In addition all items on 'the left-hand side' should be considered to be indexed as if they were arrayed 15..0.

Some means would normally be provided to declare these equivalences at the beginning of the program, in the declaration section.

Sections 8.3.5. and 8.3.6. show the expanded versions of the program and instruction set respectively. In other words, the items have been substituted for by their declared equivalents. This process, which can be automated, produces a program which resembles more closely the instructions that can be found in the structure.

The devices selected for the structure of the second example come from the Advanced Micro Devices 2900 range. Details of the particular units described may be found in Appendix 4.
8.6. E Block Diagram of Bit Slice Processor
VALUE

I=VAR;
SIGN=32768;
FIFTEEN=15;

TYPE

T4=ARRAY[3..O] OF TERMINAL;
M4=ARRAY[3..O] OF MEMORY;
R4=ARRAY[3..O] OF REGISTER;
INST_PATH=ARRAY[8..O] OF TERMINAL;

SUBS Am2901;

ITEM

Cn,Cn4,CP,QO,Q3,RAM0, RAM3, ZERO,F3:TERMINAL;
AA,AB,D,Y,F,S,R:T4;
A,B:M4;
RAM:ARRAY[0..15] OF M4;
Q:R4;
I:INST_PATH;

BEGIN
R:=A A O O O D D D I; I[2..O] I;
S:=( Q B B B A A Q Q ) I[2..O] I;
CASE I[5..3] OF
  0: {Cn4 F} := R+S+Cn;
  1: {Cn4 F} := S+R+Cn;
  2: {Cn4 F} := R+R+S+Cn;
  3: F := R!S;
  4: F := R&S;
  5: F := ~R&S;
  6: F := (~R&S)! (R~S);
  7: F := (~R~S)! (R&S)
END;

\"CP & (I[7]&I[8]) \ RAM[AB]:=D;
\"I[7]&I[8] \ D:=( RAM3 F[3..1] ),RAM0:=F[O];
\ I[7]&I[8] \ D:=( F[2..O] RAM0 ),RAM3:=F[3];
\CP & (I[8..6]=0) \ Q:=F;
\CP & (I[8..6]=4) \ Q:=( Q3 Q[3..1] ),Q0:=Q[0];
\CP & (I[8..6]=6) \ Q:=( Q[2..O] QO ),Q3:=Q[3];
\CP \ A:=RAM[AA], B:=RAM[AB]
\NOE \ Y:=(F A)[ I[8..6]=2 ];
ZERO:=F=O;
F3:=F[3];
END;

SUBS Am2907;

ITEM
A, R, BUS: T4;
D, Q: R4;
PAR, NOE, NRLE, NBE, DRCP: TERMINAL;

BEGIN
\"DRCP\ D:=A;
\"NBE\ BUS:=\"D;
\"NRLE\ Q:=BUS;
\"NOE\ R:=\"Q;
CASE NBE OF
  0: PAR:= +/A MOD 2;
  1: PAR:= +/Q MOD 2
END

END

TYPE
  T16=ARRAY[15..0] OF TERMINAL;
  BUFFER=ARRAY[3..0] OF Am2907;
  RALUTP=ARRAY[3..0] OF Am2901;
  M16=ARRAY[15..0] OF MEMORY;

ITEM
  YBUS, DBUS
  DATA, ADDR
  RALU
  R, W, LDR, LAR, ZERO, NEG
  CK
  AA, AB, SEL
  T16;
  BUFFER;
  RALUTP;
  TERMINAL;
  CLOCK;
  T4;
  ARRAY[0..MAX(13)] OF M16;

BEGIN
  YBUS:=RALU.Y;
  DATA.A:=YBUS;
  ADDR.A:=YBUS;
  DBUS:=DATA.R;
  RALU.D:=( 0 SIGN FIFTEEN DBUS )[SEL];

  DATA.BUS:=MAIN[ADDR.BUS[13..0]];
  DATA.NRLE:=0;
  DATA.NBE:=1;
  MAIN[ADDR.BUS[13..0]]:=DATA.BUS;
  DATA.NRLE:=1;
  DATA.NBE:=0;

  RALU.NOE:=0;
  DATA.DRCP:=LDR[0:4];
  ADDR.DRCP:=LAR[0:4];
  DATA.NOE:=0;

  RALU.CP:=CK[0:4];
  FOR I:=0 TO 2 DO
  BEGIN
    RALU[I+1].RAM0:=RALU[I].RAM3;
    RALU[I+1].CO :=RALU[I].Q3;
    RALU[I+1].Cn :=RALU[I].Cn4
  END;
  RALU[0].RAM0:=0;
  RALU[0].CO :=0;
RALU[3].RAM3 := RALU[3].F3;
NEG := RALU[3].F3;
ZERO := &/RALU.ZERO;
RALU.AA := AA!{0:4 0:4 0:4 0:4};
RALU.AB := AB!{0:4 0:4 0:4 0:4};
END.
PROGRAM EXERCISE2:

BEGIN
MAINLOOP: MAR:=PC;
   INBUF:=MAIN[MAR], PC:=PC+1;
   IR:=INBUF;
   IF IR[15] THEN GOTO FIRST;
   IF IR[14] THEN GOTO SECOND;
   IF IR[13] THEN GOTO POP;
PUSH: MAR:=IR;
   INBUF:=MAIN[MAR],
   OUTBUF:=INBUF;
   MAR:=SP;
   MAIN[MAR]:=OUTBUF, SP:=SP+1;
   GOTO MAINLOOP;
POP: MAR:=SP;
   INBUF:=MAIN[MAR], SP:=SP-1;
   OUTBUF:=INBUF;
   MAR:=IR;
   MAIN[MAR]:=OUTBUF;
   GOTO MAINLOOP;
FIRST: IF IR[14] THEN GOTO THIRD;
   IF IR[13] THEN GOTO JPOS;
JZER: MAR:=SP;
   INBUF:=MAIN[MAR], SP:=SP-1;
   DATA.D:=MAIN[MAR];
   IF INBUF=0 THEN GOTO JUMP;
   GOTO MAINLOOP;
JPOS: MAR:=SP;
   INBUF:=MAIN[MAR], SP:=SP-1;
   IF "INBUF[15] THEN GOTO JUMP;
   GOTO MAINLOOP;
SECOND: IF IR[13] THEN GOTO JNEG;
JUMP: PC:=IR;
   GOTO MAINLOOP;
JNEG: MAR:=SP;
   INBUF:=MAIN[MAR], SP:=SP-1;
   IF "INBUF[15] THEN GOTO JUMP;
   GOTO MAINLOOP;
CALL: SP:=SP+1;
   OUTBUF:=PC;
   MAR:=SP;
   MAIN[MAR]:=OUTBUF;
   GOTO JUMP;
FOURTH: IF IR[10] THEN GOTO RET;
ADDUB: MAR:=SP;
   INBUF:=MAIN[MAR], SP:=SP-1;
   R1:=INBUF;
   IF "IR[12] THEN GOTO ADD;
   R1:=-R1;
ADD: MAR:=SP;
 INBUF:=MAIN[MAR];
 OUTBUF:=INBUF+R1;
 MAIN[MAR]:=OUTBUF;
 GOTO MAINLOOP;
RET: MAR:=SP;
 INBUF:=MAIN[MAR], SP:=SP-1;
 IR:=INBUF;
 GOTO JUMP;
MULDIV: IF IR[12] THEN GOTO DIV;
MUL: MAR:=SP;
 INBUF:=MAIN[MAR], SP:=SP-1;
 R3:=INBUF;
 MAR:=SP;
 INBUF:=MAIN[MAR];
 XX:=INBUF;
 LC:=15;
 R1:=0;
 R4:=0;
 IF XX[15] THEN GOTO NEXT;
 XX:=XX+1;
 NEXT: IF R3[15] THEN GOTO MULLOOP;
 R3:=R3+1;
 MULLOOP: IF XX[0] THEN GOTO NOADD;
 R1:=R1+R3;
 NOADD: XX:=XX[15, 15..1];
 R4:=R4[15, 15..1];
 IF R1[0] THEN GOTO NOCARRY;
 R4:=R4+SIGN;
 NOCARRY: R1:=R1[15, 15..1];
 IF R3[15] THEN GOTO MULEND;
 IF R1[14] THEN GOTO MULEND;
 R1:=R1+SIGN;
 MULEND: LC:=LC-1;
 IF LC[15] THEN GOTO MULLOOP;
 OUTBUF:=R4;
 MAIN[MAR]:=OUTBUF;
 GOTO MAINLOOP;
DIV: MAR:=SP;
 INBUF:=MAIN[MAR], SP:=SP-1;
 R3:=INBUF;
 R2:=R3;
 MAR:=SP;
 INBUF:=MAIN[MAR];
 XX:=INBUF;
 R1:=0;
 R4:=XX;
 XX:=-XX;
 LC:=15;
 IF XX[15] THEN GOTO DVDPPOS;
 R4:=XX;
 R2:=-R2;
 DVDPPOS: XX:=-R3;
 IF R3[15] THEN GOTO DIVLOOP;
R3:=XX;
XX:=-XX;
DIVLOOP: R1:=R1[14..0] 0 ;
IF ~R4[15] THEN GOTO NOCARRY2;
R1:=R1+1;
NOCARRY2: R4:={ R4[14..0] 0 };
R1:=XX+R1;
IF R1[15] THEN GOTO DIVNEG;
R4:=R4+1;
DIVNEG: IF ~R1[15] THEN GOTO DIVPOS;
R1:=R1+R3;
DIVPOS: LC:=LC-1;
IF ~LC[15] THEN GOTO DIVLOOP;
IF ~R2[15] THEN GOTO DIVEND;
R4:=-R4;
DIVEND: OUTBUF:=R4;
MAIN[MAR]:=OUTBUF;
GOTO MAINLOOP
END.
INSTRUCTION SET GENERATED FROM PROGRAM.

THERE ARE 43 INSTRUCTIONS.

INBUF:=MAIN[MAR];
INBUF:=MAIN[MAR], PC:=PC+1;
INBUF:=MAIN[MAR], SP:=SP-1;
IR:=INBUF;
LC:=15;
LC:=LC-1;
MAIN[MAR]:=OUTBUF;
MAIN[MAR]:=OUTBUF, SP:=SP+1;
MAR:=IR;
MAR:=PC;
MAR:=SP;
OUTBUF:=INBUF+R1;
OUTBUF:=INBUF;
OUTBUF:=PC;
OUTBUF:=R4;
PC:=IR;
R1:=-R1;
R1:=0;
R1:=R1+1;
R1:=R1+R3;
R1:=R1+SIGN;
R1:=R1[15,15..1];
R1:=XX+R1;
R1:={ R1[14..0] 0 };
R2:=-R2;
R2:=R3;
R3:=INBUF;
R3:=R3+1;
R3:=XX;
R4:=-R4;
R4:=0;
R4:=R4+1;
R4:=R4+SIGN;
R4:=R4[15,15..1];
R4:=XX;
R4:={ R4[14..0] 0 };
SP:=SP+1;
XX:=-R3;
XX:=-XX;
XX:=INBUF;
XX:=XX+1;
XX:=XX[15,15..1];
PROGRAM EXERCISE2A;

BEGIN
MAINLOOP: ADDR.D:=RALU.RAM[0];
          DATA.Q:=MAIN[ADDR.D]; RALU.RAM[0]:=RALU.RAM[0]+1;
          RALU.RAM[7]:=DATA.Q;
          IF RALU[3],RAM[7][3] THEN GOTO FIRST;
          IF RALU[3],RAM[7][2] THEN GOTO SECOND;
          IF RALU[3],RAM[7][1] THEN GOTO POP;
          PUSH: ADDR.D:=RALU.RAM[7];
          DATA.Q:=MAIN[ADDR.D];
          DATA.D:=DATA.Q;
          ADDR.D:=RALU.RAM[6];
          MAIN[ADDR.D]:=DATA.D; RALU.RAM[6]:=RALU.RAM[6]+1;
          GOTO MAINLOOP;
          POP: ADDR.D:=RALU.RAM[6];
          DATA.Q:=MAIN[ADDR.D]; RALU.RAM[6]:=RALU.RAM[6]-1;
          DATA.D:=DATA.Q;
          ADDR.D:=RALU.RAM[7];
          MAIN[ADDR.D]:=DATA.D;
          GOTO MAINLOOP;
          FIRST: IF RALU[3],RAM[7][2] THEN GOTO THIRD;
          IF RALU[3],RAM[7][1] THEN GOTO JPOS;
          JZERO: ADDR.D:=RALU.RAM[6];
          DATA.Q:=MAIN[ADDR.D]; RALU.RAM[6]:=RALU.RAM[6]-1;
          DATA.D:=MAIN[ADDR.D];
          IF DATA.Q=0 THEN GOTO JUMP;
          GOTO MAINLOOP;
          JPOS: ADDR.D:=RALU.RAM[6];
          DATA.Q:=MAIN[ADDR.D]; RALU.RAM[6]:=RALU.RAM[6]-1;
          GOTO MAINLOOP;
          SECOND: IF RALU[3],RAM[7][1] THEN GOTO JNEG;
          JUMP: RALU.RAM[0]:=RALU.RAM[7];
          GOTO MAINLOOP;
          JNEG: ADDR.D:=RALU.RAM[6];
          DATA.Q:=MAIN[ADDR.D]; RALU.RAM[6]:=RALU.RAM[6]-1;
          GOTO MAINLOOP;
          THIRD: IF RALU[3],RAM[7][1] THEN GOTO FOURTH;
          CALL: RALU.RAM[6]:=RALU.RAM[6]+1;
          DATA.D:=RALU.RAM[0];
          ADDR.D:=RALU.RAM[6];
          MAIN[ADDR.D]:=DATA.D;
          GOTO JUMP;
          FOURTH: IF RALU[2],RAM[7][2] THEN GOTO RET;
          IF RALU[2],RAM[7][3] THEN GOTO MULDIV;
          ADDSUB: ADDR.D:=RALU.RAM[6];
          DATA.Q:=MAIN[ADDR.D]; RALU.RAM[6]:=RALU.RAM[6]-1;
          RALU.RAM[1]:=DATA.Q;
          IF "RALU[3],RAM[7][0] THEN GOTO ADD;
          RALU.RAM[1]:=-RALU.RAM[1];
ADD: ADDR.D := RALU.RAM[6];
DATA.Q := MAIN[ADDR.D];
DATA.D := DATA.Q + RALU.RAM[1];
MAIN[ADDR.D] := DATA.D;
GOTO MAINLOOP;

RET: ADDR.D := RALU.RAM[6];
RALU.RAM[7] := DATA.Q;
GOTO JUMP;

MULDIV: IF RALU[3].RAM[7][0] THEN GOTO DIV;
MUL: ADDR.D := RALU.RAM[6];
RALU.RAM[3] := DATA.Q;
ADDR.D := RALU.RAM[6];
DATA.Q := MAIN[ADDR.D];
RALU.RAM[8] := DATA.Q;
RALU.RAM[1] := 0;
RALU.RAM[4] := 0;
IF ~RALU[3].RAM[8][3] THEN GOTO NEXT;
MULOOP: IF ~RALU[0].RAM[8][0] THEN GOTO NOADD;
RALU[0].RAM[4][3, 1] };
IF ~RALU[0].RAM[1][0] THEN GOTO NOCARRY;
NOCARRY: RALU.RAM[1] := RALU.RAM[1][15, 15, 1];
IF ~RALU[3].RAM[3][3] THEN GOTO MULEND;
IF ~RALU[3].RAM[1][2] THEN GOTO MULEND;
IF ~RALU[3].RAM[5][3] THEN GOTO MULOOP;
DATA.Q := RALU.RAM[4];
MAIN[ADDR.D] := DATA.Q;
GOTO MAINLOOP;
DIV: ADDR.D := RALU.RAM[6];
RALU.RAM[3] := DATA.Q;
ADDR.D := RALU.RAM[6];
DATA.Q := MAIN[ADDR.D];
RALU.RAM[3] := DATA.Q;
RALU.RAM[1] := 0;
RALU.RAM[4] := RALU.RAM[8];
RALU.RAM[6] := RALU.RAM[8];
IF RALU[3].RAM[8][3] THEN GOTO DIVPOS;
RALU.RAM[4] := RALU.RAM[8];
DVDPOS: RALU.RAM[0]:=RALU.RAM[3];
RALU.RAM[3]:=RALU.RAM[8];
RALU.RAM[8]:=-RALU.RAM[8];
DIVLOOP: RALU.RAM[1]:={ RALU[3].RAM[1][2,0]
RALU[2,0].RAM[1] 0 };
RALU.RAM[1]:=RALU.RAM[1]+1;
IF ~RALU[3].RAM[4][3] THEN GOTO NOCARRY2;
RALU.RAM[4]:=RALU.RAM[4]+1;
NOCARRY2: RALU.RAM[4]:={ RALU[3].RAM[4][2,0]
RALU[2,0].RAM[4] 0 };
RALU.RAM[1]:=RALU.RAM[8]+RALU.RAM[1];
RALU.RAM[4]:=RALU.RAM[4]+1;
DIVNEG: IF ~RALU[3].RAM[1][3] THEN GOTO DIVPOS;
RALU.RAM[1]:=RALU.RAM[1]+RALU.RAM[3];
DIVPOS: RALU.RAM[5]:=RALU.RAM[5]-1;
IF ~RALU[3].RAM[5][3] THEN GOTO DIVLOOP;
RALU.RAM[4]:=-RALU.RAM[4];
DIVEND: DATA.D:=RALU.RAM[4];
MAIN[ADDR.D]:=DATA.D;
GOTO MAINLOOP
END.
INSTRUCTION SET GENERATED FROM PROGRAM.

THERE ARE 43 INSTRUCTIONS.

ADDR. D:=RALU. RAM[0]; 1
ADDR. D:=RALU. RAM[6]; 13
ADDR. D:=RALU. RAM[7]; 2
DATA. D:=DATA. Q+RALU. RAM[1]; 1
DATA. D:=DATA. Q; 2
DATA. D:=RALU. RAM[0]; 1
DATA. D:=RALU. RAM[4]; 2
DATA. Q:=MAINADDR. DJ; 5
DATA. Q:=MAINADDR. DJ, RALU. RAM[0]:=RALU. RAM[0]+1; 1
DATA. Q:=MAINADDR. DJ, RALU. RAM[6]:=RALU. RAM[6]-1; 8
MAINADDR. DJ:=DATA. D; 5
RALU. RAM[0]:=RALU. RAM[7]; 1
RALU. RAM[1]:=-RALU. RAM[1]; 1
RALU. RAM[1]:=DATA. Q; 2
RALU. RAM[1]:=RALU. RAM[1]+1; 1
RALU. RAM[1]:=RALU. RAM[1]+RALU. RAM[3]; 2
RALU. RAM[1]:=RALU. RAM[1]+SIGN; 1
RALU. RAM[1]:=RALU. RAM[1]+15,15..1; 1
RALU. RAM[1]:=RALU. RAM[8]+RALU. RAM[1]; 1
RALU. RAM[1]:=RALUC3J.RAM[1][2..0] RALUC2..0].RAM[1] 0 ]; 1
RALU. RAM[2]:=-RALU. RAM[2]; 1
RALU. RAM[2]:=RALU. RAM[3]; 1
RALU. RAM[3]:=DATA. Q; 2
RALU. RAM[3]:=RALU. RAM[3]+1; 1
RALU. RAM[3]:=RALU. RAM[8]; 1
RALU. RAM[4]:=-RALU. RAM[4]; 1
RALU. RAM[4]:=0; 1
RALU. RAM[4]:=RALU. RAM[4]+1; 1
RALU. RAM[4]:=RALU. RAM[4]+SIGN; 1
RALU. RAM[4]:=RALU. RAM[8]; 2
RALU. RAM[4]:=RALUC3J.RAM[4][2..0] RALUC2..0].RAM[4] 0 ]; 1
RALU. RAM[4]:=RALUC3J.RAM[4][3..0] RALUC2..1].RAM[4]
RALUC0].RAM[4][3..1] ]; 1
RALU. RAM[5]:=15; 2
RALU. RAM[5]:=RALU. RAM[5]-1; 2
RALU. RAM[6]:=RALU. RAM[6]+1; 1
RALU. RAM[7]:=DATA. Q; 2
RALU. RAM[8]:=-RALU. RAM[3]; 1
RALU. RAM[8]:=-RALU. RAM[8]; 2
RALU. RAM[8]:=DATA. Q; 2
RALU. RAM[8]:=RALU. RAM[8]+1; 1
RALU. RAM[8]:=RALUC3J.RAM[8][3..0] RALUC2..1].RAM[8]
RALUC0].RAM[8][3..1] ]; 1
3.3.7. EXPLANATION OF SECTION 3.3

The four preceding sub-sections showed two versions of the same program which will be termed the unexpanded and expanded versions.

The unexpanded program is of high readability despite being unstructured. (Structuring would have altered the program in such a way as to be totally dissimilar from the original). The similarity between the original and the unexpanded is obvious and demonstrates that the PDL programs for two systems with compatible capabilities will naturally be similar.

How the unexpanded program relates to the structure may seem a little obscure. This is a result of the use of, nothing more complicated then, an intelligent substitution process. This has the effect of raising the software level even though it is still a microprogram. This increase in level results in the use of registers as though they were ordinary variables in a conventional HLL. It should be noted that no functions have been altered.

The expanded program is quite unreadable and is included to demonstrate the internal representation scheme. From the expanded instruction set it can be seen that after substitution the instructions closely resemble the hardware, provided the
appropriate substitutions are made in the hardware statements. As the substitutions are made certain nodes are forced to take on specific values. These values are then traced through the structure and the consequences deduced. This may result in certain statements being illegal, or the full and more complete version of the instruction being substituted.
8.4.1. CONTROLLER DESCRIPTION

A simple address-based controller was selected as the most effective means of demonstrating the description and analysis of a control unit. The unit was designed as if it was part of the overall bit slice processor and hence can be examined with reference to the preceding section.

The micro-program counter (MPC) is simply a counter/register that can be either incremented or loaded. The output of the MPC is fed directly to the control memory (CMEM) which would be a ROM of 128 words of 23 bits each. As shown in diagram 8.3, the output of CMEM is shown as four lines. The first line corresponding to terminal J indicates whether the micro-word is a simple instruction or a jump, depending on its value 0 or 1 respectively. This line is ANDed with the output of a multiplexer (MPX) which selects one of the various input signals from the RALU section depending on the value fed to it by the second line from CMEM. The J signal is also used to stop the clock of the RALU section. This is to prevent a jump address being interpreted as an instruction. The third output line from CMEM is also the jump address that is loaded into the MPC when the 'test' succeeds.

The output of CMEM is the origin of all controlling signals and for this reason, the last statement in the PIDL description of the controller assigns CMEM to control. The reason for the assignment has been explained in previous Chapters, but a full description
of the implication of this statement appears in section 8.4.3.
8.2. SCHEMATIC OF CONTROLLER
STRUCTURE CONTROLLER;

TYPE
MEMWIDTH = ARRAY[0..22] OF MEMORY;
MPCTP = ARRAY[1..7] OF REGISTER;
T16 = ARRAY[15..0] OF TERMINAL;
T9 = ARRAY[8..0] OF TERMINAL;
T4 = ARRAY[3..0] OF TERMINAL;
T2 = ARRAY[1..0] OF TERMINAL;

ITEM
CMEM : ARRAY[0..127] OF MEMWIDTH;
MPC : MPCTP;
YBUS : T16;
I : T9;
AA, AB : T4;
SEL : T2;
J, Cn, R, S, T, ZERO, RALU_CK : TERMINAL;
CK : CLOCK;
BEGIN
[ J I Cn AA AB SEL R W ] := CMEM[MPC];
\CK & TV \ MPC := CMEM[MPC][1..7];
\CK & "TV \ MPC := INC;
T := J & S;
1 ZERO ][ CMEM[8..11] ];
RALU_CK := CK & "J;
CONTROL := CMEM
END.
3.4.3. ANALYSIS OF CONTROLLER

The controller section is described in exactly the same manner as any other unit. Although presented here as a structure, it can naturally be used as a substructure, enabling the user to completely define all sections of a processor.

The output of CIAM is labelled CONTROL, and this information would be used by the PHDL system. Under normal operation the MPC will issue, one after the other, a continuous series of addresses to CIAM. These addresses are converted by look-up into instructions, which may be conditional or unconditional jumps, or simple instructions to the RALU section. Since this is a self-perpetuating sequence the PHDL system requires some means of halting the flow temporarily. The PHDL system accomplishes this by control over the item TYPED 'CLOCK'. The clock is always a single phase output, and hence can only have the values 0 or 1.

When analysis is to be performed upon the structure the process requires either a starting or finishing point. For this purpose the predefined identifier CONTROL may be assigned any item that has been declared.

A control structure is, by its nature, a loop. For this reason analysis requires that the loop be broken else 'race around' conditions make the analysis impossible. The loop must be broken.
in a meaningful place. Electronically the loop is discontinuous at the IPC, since this contains a register where the output condition does not change until the clock permits it. From the analytical point of view the most meaningful location must be the origin of the control signals, which is the control memory. By indicating this location to the system various functions can be performed.

If bit patterns are set up at the location CONTROL then the resulting instructions can be either simulated or disassembled. If, on the other hand, an instruction is supplied to the system various nodes will be forced into conditions. These will include those which are incorporated as fields within CONTROL. This process will enable the system to build the bit pattern in CONTROL that is associated with a supplied instruction.

It should also be noted that the current address must be the input side to CONTROL. This information is also used when processing. Consider a simple GOTO instruction. The GOTO instruction is interpreted as a load into the micro-program counter which then passes the value onto the input of CONTROL. Thus, from the single assignment of the CONTROL identifier, the system can process information about data-flow within the control unit as well as in the RAM section.
(# *Eratosthenes Sieve Prime Number Program in VHDL software * )

PROGRAM PRIME;

VALUE
SIZE=8190;
TRUE=1;
FALSE=0;

TYPE
INTEGER=ARRAY[15,0] OF REGISTER;
BOOLEAN=MEM;

ITEM
FLAGS=ARRAY[0..SIZE] OF BOOLEAN;
I,PRIME,K,COUNT,ITER:INTEGER;
READY:TERMINAL;

BEGIN
COUNT:=0;
FOR I:=0 TO SIZE DO
  FLAGS[I]:=TRUE;
FOR I:=0 TO SIZE DO
  IF FLAGS[I] THEN
    BEGIN
      PRIME:=I+3;
      REPEAT UNTIL READY;
      OUT:=PRIME;
      K:=I+PRIME;
      WHILE K<=SIZE DO
        BEGIN
          FLAGS[K]:=FALSE;
          K:=K+PRIME
        END;
      COUNT:=COUNT+1
    END;
END.

PLAIN INSTRUCTIONS.

COUNT:=0
COUNT:=COUNT+1
FLAGS[I]:=TRUE
FLAGS[K]:=FALSE
K:=I+PRIME
K:=K+PRIME
OUT:=PRIME
PRIME:=I+3

STRUCTURED STATEMENTS.

FOR I:=0 TO SIZE DO
  IF FLAGS[I] THEN
    REPEAT UNTIL READY
    WHILE K<=SIZE DO
Conversion upon software constructs.

repeat
    !
    !
until con:
    if not(con) then goto label:

while con do
    begin
        !
        !
    end:
    goto label:

label: goto label2:

if con then
    begin
        !
        !
    end:
    goto label:

else
    begin
        !
        !
    end:
    goto label2:

if con then
    begin
        !
        !
    end:
    goto label2:

for a:=b to c do
    begin
        print a;
        !
        !
    end:
    if not(a<=c) then goto label2;

a:=a+1;
goto label2;

label2:
Expanding structured statements.

FOR I:=0 TO SIZE DO
    I:=0;
    if not(I<=SIZE) then goto x;
    I:=I+1;
    goto x;

IF FLAGS[I] THEN
    if not(FLAGS[I]) then goto x;

REPEAT UNTIL READY
    if not(READY) then goto x;

WHILE K<=SIZE DO
    if not(K<=SIZE) then goto x;
    goto x;

implied instructions.

I:=0;
I:=I+1;

control instructions.

if con then goto x;
    goto x;
Required instruction set:

COUNT := 0;
COUNT := COUNT + 1;
FLAGS[I] := TRUE;
FLAGS[K] := FALSE;
I := 0;
I := I + 1;
K := I + PRIME;
K := K + PRIME;
OUT := PRIME;
PRIME := I + I + 3;

if con then goto xi;
soto xi;

conditions:

not(I <= SIZE)
not(FLAGS[I])
not(READY)
not(K <= SIZE)

The plain instructions are extracted from the source without any processing, these may be passed to the instruction set intact. Identical instructions are only entered once.

The structured statements must be transformed into meaningful instructions. This is performed on Page 3 of the example using the rules shown on Page 2. As a result of the conversion some implied instructions are revealed, these may be transferred to the instruction set.

The remaining instructions are of the control type and may be divided into conditional and unconditional jumps. Further processing can extract the various conditions that require testing and this forms part of the definition of the control section.
8.6. SUMMARY OF CHAPTER 8

This section included four examples of HDL descriptions to cover various aspects of how the language can be utilised in describing both software and hardware. It was shown that the matrices when generated from the program can be of use in designing a processor that eliminates redundant paths, and hence reduce the cost of the more dedicated type of processor.

The language was used to describe two hardwares that were to be identical in operation. This was performed in such a way as to demonstrate the ease with which an automatic general purpose assembler could be organised around a description-based system.

A control unit was specified and an examination of the function of the CLOCK and CLOCK identifiers was made to demonstrate how the HDL system would use these identifiers.
CHAPTER 9
POSSIBLE DEVELOPMENTS OF PIDL

2.1. A PIDL PACKAGE
The PIDL package should be designed to interact with the user at a very high level. So that, from the user's point of view he is dealing with structure, programs and instruction sets; rather than particular facets of these. This has two distinct advantages in that it forces the user to view the target machine as a complete entity with major sub-divisions in software and hardware and an interaction between the two which can be viewed and tested in order to reach an optimum performance. Secondly it has the effect of reducing the complex interactive parts of the PIDL system which are not only complicated to write, but also lengthy and space-wasting from an execution point of view. This is not to say that interaction should be kept to the barest minimum, but that it should be employed sparingly and in appropriate situations.

The major function in the package should be as file handling processes and most of the files should be textual. As each file is called up it should be converted to the internal form and any errors detected during this conversion should be reported, and if possible, correction allowed. For example, when a structure or program is being read from a file, and an identifier that was not declared found, then it should be possible to tell the system to insert this identifier into the declaration table
along with its type, or rename the identifier to correct typographical errors. This kind of facility would save a lot of 'toing and froing' between the editor and package just to correct silly errors. It should also be possible to output such files in a standardised form, so that layout of programs and structures is always consistent and hence easier to read, correct and so forth.

Some of the functions that should be handled by such a package would be:

- Normalisation of input forms
- Instruction set generation
- Structure generation
- Simulation
- Assembly
- Disassembly
- Higher level decomposition

Each of these aspects will be examined in the next section.
9.2. PACKAGE FUNCTIONS

9.2.1. Normalisation

This has been touched on in the previous section, and as explained there, is the automatic re-writing of an input file into a standard form with inset lines and the like to make for greater readability. This function may seem to be more in the province of a text processor than a hard/soft design tool. This may be true, but only to a certain extent. Since, as it is necessary to convert the input textual file into an abstracted internal form, the return process, from internal form to textual form, is also of great benefit. This would be necessary for structure generation and thus it would be a small step to include an allied process for programs. For both processes of normalisation i.e. software and hardware, comment fields, would have to be maintained and this would be a simple process of direct copying from the original text.

The benefits of normalisation would be much more apparent in the use of such a system than in a description of the process. It would for instance make the use of a general purpose editor much more acceptable.

9.2.2. Instruction set generation

The process of instruction set generation will have two forms, depending on the input or source type. If the input is a
structure then the output would be all instructions that the structure is capable of handling. If the source is a program then it will be a summary of all the different instructions used.

The output in either case should be a textual file which lists, individually or by some method of grouping, all instructions making up the set. This output should be to a file in a normalised form. Input and normalisation of instruction sets should also be supported.

Instruction set generation from a program would be a simple matter of building a table and comparing the next instruction with the table, or of performing an equivalent process in the internal representation scheme. As for the process from the structure, this would entail examining the possibility that arises from varying inputs at the location CONTROL and examines the corresponding results for instruction.

The actual form of the output is beyond the province of this thesis. It will be stated however, that the layout and syntax for instruction set description must be prepared with care. Catering for concurrent and non-concurrent instructions and register grouping as well as memory grouping. It would for instance be quite ridiculous to list all loads to a register.
from main memory as individual instructions when the whole lot are more succinctly expressed by a single statement. Bearing these aspects in mind, instruction set input and output would be a powerful means of aiding the designer to examine the overall capability of his target. Many designers will start from the instruction set, and to be able to obtain a normalised form of same along with the necessary hardware to implement it directly would be a great saving in time and effort.

9.2.3. Structure Generation

Structure generation can only be performed from one source, namely, the instruction set. This would be the structure required for direct implementation of each instruction within the set. To obtain the structure from a program it would first be required to reduce the program to an instruction set and then convert that into a structure. When the instruction set is used to create the structure certain functions could be performed automatically. An example of this would be the number of instructions that each particular device is used by, this would give some idea of software/hardware trade-offs, that would be more profitable.

One technique for producing structure from instruction set has been dealt with in the chapter covering internal representation.
The technique involved of ORing instruction matrices together to form the structural matrix. If the ORing process were replaced by an element for element addition routine then each nodal connection of the matrix would have associated with it a number which would be equal to the number of instructions which use that particular data path. This is one way that the utilisation of various units could be found.

Input and normalised output of structure would also be permitted and would normally be in textual form. Although some implementors may prefer to have some dumping mechanism to allow for storage of the structure in internal form. Although this might be advantageous as regard speed and efficiency, it is better practice to always have such files in a man readable form at all times.

9.2.4. Simulation

Simulation is a process that would be performed on the internal structural representation. The process would require the emulation of the loaded software running in the environment of the loaded hardware. This process has been well covered elsewhere in the text, so that it is only necessary here to suggest some of the refinements that would be of advantage. It must be possible to set up the original state of the structure
before any simulation has been performed. Also, to be able to report the state at any given time. The execution of any single instruction or the run of a number of instructions, would naturally be supported. The ability to include break-points (break-points are markers in certain program locations that, when reached, cause the execution to be terminated and the status reported) could be included with little difficulty.

There is little to say about this process, except that the implementor may wish to consider the inclusion of other functions, such as the utilisation of various circuits so that when any particular circuit is employed, it has the effect of incrementing a counter, so that at the end of a run, the usage of various units would be known, and alteration to the circuits could be made based on this information. The process of simulation could also be employed to reduce a number of low level instructions into higher level ones. This would have the effect of making instructions more understandable. Many possibilities such as these, that are not directly related to simulation should occur to the reader, and their usefulness should be carefully weighed against the cost in terms of size and writing time before incorporation.

Finally, it will be stated that simulation is a must; and its inclusion should be taken for granted, however flexible or not the system is made.
9.2.5. Assembly

Assembly is a process that is essential for any programmer and it must be remembered that if assembly is included in a PHDL system, then the result would be a universal assembler, capable of assembling standard source (i.e., PHDL software) for any structure that has been defined. In fact the definition of the structure need not be accurate if only assembly is required.

The complexity of the assembler section of the package is really outside the scope of this work, but it is worthy of a few comments. The increasing complexity of an assembler can be handled separately from the main system. The process of basic assembly has been dealt with in earlier sections and will not be recounted. So, taking it for granted that a basic assembler has been written, it is a simple matter to write a one pass pre-assembler program to handle such devices as MACRO's and conditional assembly which would simply re-write the same in the same language with the MACRO's expanded and the selected code inserted as appropriate. Whether this one pass pre-assembler is included as part of the assembler section or left as a package utility is up to the designer, but if it is written as a utility, a separate program, then it should be called automatically as the first line of the assembly process.

Output from an assembler would naturally have to be a binary file and load location information would be encoded in the conventional way, although it might be safer to ignore the user's indexing
scheme for the control memory and number the lowest location as zero and increase in increments of one through the host systems memory words. This will, of course, make the process of outputting code for ROMing purposes more complicated.

The alternative is to provide special file structures that hold location and word with information encoded, where the information would correspond to the user's definitions. Whichever method is chosen, this problem should be easily overcome.

The assembly process should be regarded as a necessity. It is as the majority of other processes, a file to file process and requires little interaction, except as regards the possibility outlined in the beginning of this section, that is the prompting for undeclared identifiers to be included in the declarations table, but this could be included in the load software and excluded from the assembler.

9.2.6. Disassembly

Disassembly is the reverse process to assembly. The action of the disassembler is to take binary code and to convert into a man-readable assembly language. Thus, it is possible to understand any object code that comes to hand. This device is not an essential element of the PHDL package, and is generally not utilised much in research or educational establishments. It is nevertheless, of particular value to a practical programmer in
altering supplied programs to his own purpose. For this reason, some means of disassembly is to be considered seriously. See next section.

2.2.7. Program writers

As an addition, or as an alternative, to the direct processes of assembly and disassembly, the designer of a PHDL system may wish to consider the possibility of incorporating program writers.

As it is possible to assemble and disassemble, then it can be deduced from this; that, given the structural representation then all the information for these two processes is present. Thus, moving one step forward, it must be possible to construct an assembler/disassembler from this data, and since it is possible, then the process can be automated. This automated process would be performed by program writers. The ability to generate such programs purely from circuits, the given structure, would be of tremendous advantage to anyone who has designed a processor as a general purpose device and wishes to provide a basic amount of software support. This could be provided at no extra cost. Even more sensibly, this should be available in source form, allowing for adaptation.

The designing of such a program writer would be no mean feat. The writer believes that this technique should be examined not
only for assembly, but also for simulation. The choice of host machine on which the assembler should run could be any capable of handling whatever high level language be chosen. HDL software or Pascal would be a reasonable choice.

2.2.2. High level decomposition

This process has been explained in the section on state transition diagrams, and the result of such processing is to reduce an instruction that is non-executable into a series of instructions that are executable. The ability to perform such operations opens up vast possibilities as regards software portability and original writing.

If the handling of symbolic names of locations in memory is also supported, then by taking a program statement by statement and passing it through the decomposer, then the result would be the actions of the original program translated into the language of the second, and hence executable on that machine. Although this process is not practical as yet, it is practicable and the seeds of such portability lie in the understanding of the relationship between hardware and software. This understanding can be found at the hardware/micro-code level, with such processes, as simple decomposition.

This decomposition process is, however, practical at a low level, and even on a small scale such as this, it could benefit any programmer by simply giving him greater scope in the statements he can use.
Lastly, it is of significance that the process can be used at all because since it is at the very least practicable then the way has been opened up to the more automatic and general purpose compiler.
9.a. Block diagram of FIDL system
9.3. RECAP OF PHIL SYSTEM

Reference to the diagram should make this summary clearer.
There are three basic forms in which data can be held within this system. As a program (P) (software), as an instruction set (I), and as structure (S). There is also in an extended version, the higher level instruction section. But the interaction of this section with the program section should be fairly simple to visualise.

There are three translation processes that can be employed inside the internal system, these are:

\[ I := f(P) \]
\[ S := f(I) \]
\[ I := f(S) \]

As well as these functions there must be input routines for each of the three forms.

\[ S := \text{input} \]
\[ I := \text{input} \]
\[ P := \text{input} \]

In addition to the three obvious ones shown above, there is also micro-code (c), thus

\[ c := \text{input}. \]
Whatever form of internal representation is chosen, the translation should be possible from micro-assembly language or from micro-code. Thus, this must be included in the third statement of those shown above.

In exactly the same way as for input; output must be possible.

```
output := S
output := I
output := P
output := c
```

Adding to the above assembly and disassembly brings the total of major instructions to thirteen.

Although these functions could be written independently, certain actions overlap, and can be combined for simplification. An example of this is the syntax checking procedure for programs and structure which will have very similar routines. There are also other examples which should all be examined carefully and planned out before any attempt is made to write any section of the package.
2.4. HARDWARE PROGRAM CONSTRUCTS

Program constructs are, here, being dealt with in detail. By the term program construct, it is meant those statements in a program which control conditions or repetitive execution.

Each construct is characterised by having a single entry point, and a single exit route. The form and effect of these statements has been dealt with previously. So here their applicability to hardware will be examined.

The most common type of construct and the only one that has been included in this definition of PHDL, is the conditional execution construct, namely

\[ \text{IF condition THEN statement ELSE statement} \]

or

\[ \text{IF condition THEN statement} \]

The realisation of this form in hardware is obvious, and needs no explanation.

The second category is the repetition constructs of which there are two, viz.

\[ \text{REPTAT \ldots statement(s) \ldots UNTIL condition} \]
\[ \text{END WHILE \condition \DO \statement} \]

These two have been excluded from the definition of PHDL. The reason for their exclusion is that they assume an automaton nature. By this it is meant that the repetitive behaviour of
the construct is outside the control of the micro-program and as such is a free-working self-contained sub-unit, running independently of the micro-program. There are means of interpreting these constructs into hardware, and they may have value in this connection. The WHILE DO construct for instance, can be simply interpreted as an AND gate connected into the CLOCK line which is then used to effect the statements, a similar circuit can be utilised to translate the REPEAT statement although this will be slightly more complicated.

There is also the FOR loop which in PHDL is used exclusively as a meta-construct and not as a physically realisable statement. This type of iteration construct is, as the above, also possible to translate to hardware; this could be done by the automatic insertion into the structure of a counter and comparitor wired appropriately. This though, again, forms a self-active unit and this must be taken into account if such a construct is included in the definition of PHDL.

The writer feels that the forced nature of how these constructs are realised subtracts from the overall flexibility of the system, and should thus be excluded. The above has been outlined for implementors who feel that software constructs are essential for the greater ease they give to any designer.
2.b. REPEAT and UNTIL loops combined in FIDL
9.5. SOFTWARE PROGRAM CONSTRUCTS

Program constructs give software form and a consistency. This they do by providing one way in, and one way out units that force upon the user the decomposition of a task into subtasks; each which is executionally speaking, independent of others. All this lends itself to much clearer programs that are free of the messy GOTO statement. Unfortunately this clarity is gained at a price. The price that is paid is that of less efficient programs. The loss of efficiency at high levels is of no consequence, but at the level of micro-code it is all important. Since each micro-instruction may be executed thousands of times in any high level user program, the drop in speed of execution due to unessential instructions can prove to be expensive.

The inefficiency inherent in program constructs does not necessarily preclude their use, as the wastefulness is only present under certain conditions. These conditions will have to be avoided by the user. Thus, for MIDL software, a modified version of the iterative statements are included and the GOTO is permitted and encouraged. The modification to the iterative statements, is to link the REPEAT to the WHILE to produce a loop with two parts, one that is executed always once, and the other that may be executed no times. This can be better understood from the diagram 9.b.
The one path in and one path out requirement for constructs may seem insurmountable. There is, however, a possibility that may be worthy of further investigation. Consider the following.

During the execution of a program there are two basic ways information can be held. The first is the conventional means of using memory locations and directly storing the information for later retrieval. The second, which is less commonly realised, is the holding of data by virtue of program control position. This second method is typified by finite state automaton. Control position is equivalent to which particular instruction is being executed at the particular time in question. This amounts to information and can be converted to storeable information. In the reverse manner it also can be converted back to control position information.

Consider the Pascal program.

```pascal
READLN (D);
IF D=0 THEN
  BEGIN
    STATEMENTS (A);
    PATH := 0
  END
ELSE
  BEGIN
    STATEMENTS (B);
    PATH := 1
  END;
WRITELN(PATH);
```
At the read and write statements data is held in locations, while during statement groups A and B, the same data is held by virtue of position.

From the above it can be seen that the two forms of information retention are both valid and interchangeable. Therefore it follows that data placed in a suitable variable, say a route variable, could be translated as indicating which path is appropriate.

Such a route variable would require certain properties. Its value would indicate both which path in and which path out is taken for the relevant construct. It would be used as a conventional variable, but be translated as a metavariable governing paths selected. For example, if the statement

```
IF PATH = 1 THEN ........
```

was found then the statements conditional upon this would be assembled into the path indicated by the value one. Other peculiarities which would also follow would be that assignment to the route variable indicates the leaving of the construct and all tests upon route variable would be on the value as it enters the construct.

The implications, benefits and losses of incorporating a route variable has not been fully investigated but has been
laid out here for any interested reader to follow up.
9.6. THE PNL EXPRESSION

As no syntax diagrams exist, the PNL expressional form is outlined here. This should be understood to be a suggestion albeit a strong one that can be altered if the designer deems necessary for any valid reason. A good deal of time and effort has been spent on the expression syntax so any changes should be made with great care.

In this context, an expression is any combination of operands and functions that are strung together in a predetermined way. The expression may be assigned to an item or variable by the 'becomes' symbol (=).

The major aspect of an expression's syntax is the functional precedence, of which there are three main contender.

1. Right to left as in APL.
2. Reverse Polish Notation (RPN) as in FORKH.
3. Conventional infix as in Pascal.

Of these three, the last is selected because of it's almost universal adoption. It is the most natural of all schemes. Right to left precedence is a good scheme, but suffers the inherent problem of being against the natural scan of left to right in all English-like languages. RPN has the advantage of requiring no parenthesised terms but lacks readability, it is
much more suitable as an internal form. This leaves infix with
functional precedence. The precedence that is chosen agrees
with the majority of other such syntaxes and is shown below:

1. terms in parenthesis
2. monadic function and user defined functions
3. diadic functions,
   multiply divide (if included)
   add subtract
   AND
   OR

Note that diadic functions used in conjunction with the
reduction symbol (/), together constitute a monadic function.

Any expression may be followed by an indexing term enclosed
in brackets, and this always applies to the top-most dimensional
vector. To index in a plane, then two bracket terms must be
employed.

Undefined indexing is right to left on the first dimension,
but is left to right for higher levels and always begins with
zero.

Finally, the inclusion of a dimension specifier as in APL that
postcedes the function may be of assistance. If this is
included it should take the following form.

If MEM is a 32 by 8 bit memory,

and

\[ A := \&/MEM \]
\[ B := \&/[0]MEM \]
\[ C := \&/[1]MEM \]

then

\[ A \text{ is 8 bits width} \]
\[ B \text{ is 32 bits width} \]
\[ C \text{ is 6 bits width; equal to } A. \]

The MIDL expression is a complicated piece of syntax, but has a flexible yet easy-to-read nature. It should be implemented in its fullest form else limitations will become apparent and these should, of course, be avoided.
CHAPTER 10
CONCLUSIONS

The aim of this work was to show that the apparent gulf between hardware and software can be bridged, and also that the difference in levels of software that are machine dependant can be processed once a good method of describing the machine is found.

A powerful hardware description language has been well defined. It has universal appeal in its form and concise descriptive power with its flexible expressions. It has also been demonstrated that the language, with slight alterations to keywords and minor syntactic adjustment, can be used to describe micro-programs with parallelisation. Internal schemes of representation of both structure and program have been examined, and the way in which they can be manipulated as data held in one of three basic forms.

The three basic forms of data retention correspond to program, structure and instruction set. Means of converting from one form to another and of comparing different forms have been described also.

The overall nature of software and hardware has been analysed and it has been shown that hardware behaves in much the same way
as a translator. This can be realised once it can be accepted that 'execution' is a program that has been put into the dimension of time. From the dimension of time it can be translated back into a description that utilises state transition diagrams, and these can be compared clock-cycle by clock-cycle with the structure. By this technique a higher level construct not immediately executable can be decomposed into a number of instructions, each of which is permissible and together effect the original higher level construct. The importance of this is that the process is independent of the target machine in so far as any machine structure can be supplied and the analysis and synthesis made with this data.

A software package has been suggested with powerful macro-operations, each one of which has been shown to be realisable along with the technique that can bring this realisation about. The concept of the package is, as a designer's aid that can formulate a structure from a required program and can, in addition, simulate and assemble the program. All this has been discussed, and not only shown to be viable, but also the ways in which it can be achieved have been analysed and described.

Finally there is a relatively complex worked example to show how the representations should appear and how they can be manipulated. The next stage in this work must be to implement a small section of the overall package.
Glossary of Abbreviations
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic/Logic Unit</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus Naur Form</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processor Unit</td>
</tr>
<tr>
<td>EPROM</td>
<td>Erasable Programmable Read Only Memory</td>
</tr>
<tr>
<td>FILO</td>
<td>First In Last Out</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HLL</td>
<td>High Level Language</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>LIFO</td>
<td>Last In First Out</td>
</tr>
<tr>
<td>MCU</td>
<td>Micro Control Unit</td>
</tr>
<tr>
<td>MOS</td>
<td>Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>MPC</td>
<td>Micro Program Counter</td>
</tr>
<tr>
<td>MPS</td>
<td>Micro Program Sequence</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PROM</td>
<td>Programmable Read Only Memory</td>
</tr>
<tr>
<td>RALU</td>
<td>Register Arithmetic/Logic Unit</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>RPX</td>
<td>Reverse Polish Notation</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Language</td>
</tr>
<tr>
<td>PHDL</td>
<td>Pascal-based Hardware Description Language</td>
</tr>
<tr>
<td>TSL</td>
<td>Transistor Transistor Logic</td>
</tr>
<tr>
<td>TWS</td>
<td>Translator Writing System</td>
</tr>
</tbody>
</table>
PROGRAM SORTS;

(* A PROGRAM TO DEMONSTRATE SOME OF THE PASCAL CONSTRUCTS BY MEANS OF SOME SORTING EXAMPLES *)

CONST

\[
\text{MAX} = 101;
\]

VAR

\[
\begin{align*}
\text{CH} & : \text{CHAR}; \\
\text{SEED}, \text{VALUE}, \text{NUMBER} & : \text{INTEGER}; \\
\text{LIST} & : \text{ARRAY[1..MAX]} \text{OF INTEGER};
\end{align*}
\]

PROCEDURE GENERATE (HOWMANY : INTEGER);

VAR

\[
\begin{align*}
\text{J} & : \text{INTEGER}; \\
\text{FUNCTION RANDOM} & : \text{INTEGER};
\end{align*}
\]

CONST

\[
\begin{align*}
\text{MULT} & = 31; \\
\text{MODVAL} & = 1009;
\end{align*}
\]

BEGIN

\[
\begin{align*}
\text{SEED} & := \text{SEED} \times \text{MULT}; \\
\text{SEED} & := \text{SEED} \mod \text{MODVAL}; \\
\text{RANDOM} & := (\text{SEED} + 3) \div 4;
\end{align*}
\]

END; (* OF FUNCTION RANDOM *)

BEGIN

FOR \(J := 1\) TO HOWMANY DO LIST[J] := RANDOM;

Writeln(HOWMANY, ' RANDOM NUMBERS GENERATED');

END;

PROCEDURE SHELLSORT (LOW, HIGH : INTEGER); (* ITERATIVE *)

VAR

\[
\begin{align*}
\text{SORTED} & : \text{BOOLEAN}; \\
\text{SWOP}, \text{GAP}, \text{I}, \text{IPLUSGAP}, \text{IMAX}, \text{LENGTH} & : \text{INTEGER};
\end{align*}
\]

BEGIN

IF HIGH > LOW THEN

BEGIN

\[
\begin{align*}
\text{LENGTH} & := \text{HIGH} - \text{LOW} + 1; \\
\text{GAP} & := 255; \\
\text{WHILE GAP} \leq \text{LENGTH} \text{ DO GAP} := \text{GAP} \div 2 ;
\end{align*}
\]

REPEAT

\[
\begin{align*}
\text{IMAX} & := \text{HIGH} - \text{GAP};
\end{align*}
\]

REPEAT

\[
\begin{align*}
\text{SORTED} & := \text{TRUE}; \\
\text{FOR I} & := \text{LOW} \text{ TO IMAX DO}
\end{align*}
\]

BEGIN

\[
\begin{align*}
\text{IPLUSGAP} & := \text{I} + \text{GAP}; \\
\text{IF LIST}[\text{IPLUSGAP}] < \text{LIST}[\text{I}] \text{ THEN BEGIN}
\end{align*}
\]

BEGIN

\[
\begin{align*}
\text{SORTED} & := \text{FALSE}; \\
\text{SWOP} & := \text{LIST}[\text{I}]; \\
\text{LIST}[\text{I}] & := \text{LIST}[\text{IPLUSGAP}]; \\
\text{LIST}[\text{IPLUSGAP}] & := \text{SWOP};
\end{align*}
\]

END;
\]

END;

END;
END;
UNTIL SORTED;
    GAP:=GAP DIV 2 ;
    UNTIL GAP=0 ;
END;
END; (* OF SHELL SORT *)

PROCEDURE QUICKSORT ( LOW,HIGH : INTEGER ) ; (* RECURSIVE *)
VAR
    I,J,MID,SWOP : INTEGER ;
BEGIN
    I:=LOW; J:=HIGH; 
    MID:=LIST[(I+J) DIV 2 ];
    REPEAT
        WHILE LIST[I] < MID DO I:=I+1; 
        WHILE LIST[J] > MID DO J:=J-1; 
        IF I <= J THEN 
        BEGIN
            SWOP:=LIST[I];
            LIST[I]:=LIST[J];
            LIST[J]:=SWOP;
            I:=I+1;
            J:=J-1;
        END;
        UNTIL I > J ; 
        IF LOW < J THEN QUICKSORT(LOW, J); 
        IF I < HIGH THEN QUICKSORT(I, HIGH); 
    END;

PROCEDURE SEARCH ( LOW,HIGH,VAL : INTEGER ) ; (* BINARY SEARCH *)
VAR
    I,J,K : INTEGER ;
BEGIN
    I:=LOW; J:=HIGH; 
    REPEAT
        K:=(I+J) DIV 2 ; 
        IF VAL <= LIST[K] THEN J:=K-1 ;
        IF VAL >= LIST[K] THEN I:=K+1
        UNTIL I > J ; 
        IF I-1 > J THEN WRITE(' FOUND IN POSITION ',K:3) 
        ELSE WRITE(' NOT PRESENT IN LIST '); 
        WRITELN;
    END ;
BEGIN       (* THE MAIN PROGRAM STARTS HERE *)
REPEAT
    WRITELN;
    WRITE('ENTER A NUMBER BETWEEN 1 AND',MAX:5, ',');
    READ(NUMBER); WRITELN;
    UNTIL ((NUMBER>1) AND (NUMBER<MAX));
SEED:=123; GENERATE(NUMBER);
WRITELN('SORT USING QUICKSORT OR SHELLSORT ?');
REPEAT
  WRITE('ENTER Q OR S :- ');  
  READ(CH); WRITELN
  UNTIL (CH='Q') OR (CH='S');
IF CH='Q' THEN QUICKSORT(1,NUMBER)
ELSE SHELLSORT(1,NUMBER);
REPEAT
  WRITE(' VALUE TO BE FOUND = ? ');  
  READ(VALUE); WRITELN
  IF VALUE > 0 THEN SEARCH ( 1,NUMBER,VALUE )
  UNTIL VALUE <=0;
END.
(* EXAMPLES OF THE USE OF PHDL NODES *)

(* DEFINITION OF MEMORY FROM TERMINAL *)

NODE MEMORY \TRIG\;
ITEM A:TERM;
BEGIN
  \TRIG\ A:=MEMORY;
  ^TRIG\ A:=A;
  MEMORY:=A
END;

(* DEFINITION OF REGISTER FROM MEMORY *)

NODE REGISTER \TRIG\;
ITEM A,B:MEMORY;
BEGIN
  \TRIG\ A:=REGISTER;
  ^TRIG\ B:=A;
  REGISTER:=B
END;

(* SPECIFICATION ASSUMED BY INCREMENT (INC) FUNCTION *)

FUNC INC [ WIDTH(INC)..1 ];
BEGIN
  INC:= (INC+1) MOD ( MAX(INC)+1 )
END;
(* EXAMPLE OF THE USE OF THE PHDL SUBSTRUCTURE FOR
STANDARD TTL PACKAGES. *)

SUBS DECADE:

ITEM A, B, C, D: REGISTER;
   ROA, ROB, R9A, R9B, C1, C2: TERM;

BEGIN
   C1\ A:= ( A+1 ) MOD 2;
   C2\ {D C B} := ( {D C B}+1 ) MOD 5;
   R0A & R0B\ {D C B A}:=0;
   R9A & R9B\ {D C B A}:=0;
END!
APPENDIX 3

SYNTAX

As a late alteration the syntax diagrams were removed and replaced by a grammar representation system known as Backus Naum Form (BNF). BNF has been used, as it is generally more helpful to implementers of syntax recognisers and is, in general, a more concise representation.

Before the actual syntax is specified, the terms, used in the description will be explained.

Modified version of BNF

Syntactic entities are enclosed in angle brackets. Thus `<LETTER>`

"MAY BE COMPOSED OF SYMBOL" IS `::=`

Possible choices are separated by vertical lines `\|` Thus `<ONE_OF_FIRST_THREE_LETTERS> ::= A|B|C`

Optional entities are enclosed in braces and this implies repetition any number of times, including zero. Thus `{<LETTER>}`

If the opening brace is immediately preceded by a quote then the quantity in braces must appear at least once.

If the closing brace is immediately proceeded by a quote then the quantity in the brace must appear no more than once. Thus `{<LETTER>}` or `{<LETTER>}`
Parenthesis are used in the conventional distributive manner. Thus \( \text{A} \left( \text{B} \lor \text{C} \right) \Rightarrow \text{AB} \lor \text{AC} \). Literals are shown explicitly or if they clash with reserved terms then they will be shown in double quotes. Thus \( \text{A} \lor "\text{A}" \).
GRAMMAR (<STRUCTURE>)

<STRUCTURE>::=STRUCTURE <IDENTIF> ; <BLOCK>.
<IDENTIF> ::=<LETTER> [<LETTER>]<DIGIT>]
<LETTER> ::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<LETTER> ::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<LETTER> ::= "-
<DIGIT> ::=0|1|2|3|4|5|6|7|8|9
<BLOCK> ::=<{VALUE}> '{<SPECS> } {<ITEMS> }<COMPOUND>
<SPECS> ::=<{TYPES}>|<NODES> | <FUNCS> |<SUBS>
.VALUE ::=VALUE '{<VAL_ENTRY> ;
<VAL_ENTRY> ::=<IDENTIF> |<IDENTIF>
<VALUE> ::=<IDENTIF> '{<IDENTIF> }
<IDENTS> ::=<IDENTIF> '{<IDENTIF> }
<IDENTS> ::=<IDENTIF> '{<IDENTIF> }
<VCONST> ::=<NUMBER> | <ASPEC> (VAR'|{<NUMBER>})
<BOUNDS> ::=[<FIXED>] . < <FIXED>]
<FIXED> ::=<NUMBER> | <IDENTIFIER> '{<INDEX> } |<INvariants>
<ININVARIANT> ::=<WIDTH|MAX> "(" <FIELD> ")"
<ININVARIANT> ::=<BITS> "(" <NUMBER> ")"
<INDEX> ::=[{ <INDEXTOK> [{<INDEXTOK> ]
<INDEXTOK> ::=<FIELD> '{<FIXED> '<BOUNDS> 
<TYPES> ::= <{IDENTS} = <TSPECS> ;
<ASPEC> ::=ARRAY <BOUNDS> OF
<TSPECS> ::={<ASPEC> ' }{<BTYPE> '<IDENTIF> } |<RECTYPE>
<BTYPE> ::= REGISTERT | MEMORY | TERMINAL'
<RECTYPE> ::= RECORD <RECENT> {, <RECENT> } END
<RECENT> ::=<IDENTS> '{<BTYPE> '<IDENTIF> }
<NODES> ::=<IDENTIF> '{<BOUNDS>
<NHEAD> ::= NODE <IDENTIF> '{<IDENTIF} \{<BOUNDS>}
<FUNCS> ::=<IDENTIF> '{<BLOCK>
<FHEAD> ::= FUNCTION "(" <OPER> [{<OPER>} ")" '{<BOUNDS} '
<OPER> ::=<IDENTIF> '<BOUNDS>
<ASPEC> ::={<ASPEC> ' }{<BTYPE> '<IDENTIF> }<SUBS>
<ITEMS> ::=ITEM '{<IDENTS} <ITSPEC> ;
<ITSPEC> ::={<ASPEC> ' }{<BTYPE> '<IDENTIF> }
<COMPOUND> ::=BEGIN '{<STATMNT>}
<STATMNT> ::=<COMPOUND> |<IFSTAT> |<FORSTAT> |<SIMPLE>
<IFSTAT> ::=IF <BEXP> THEN <STATMNT> {ELSE <STATMNT>}'
<FORSTAT> ::=FOR <IDENTIF> := <EXP> (TODOWNTO) <EXP>

<BASESTAT> ::= '{<BEXP> ', <SIMPLE> '{, <SIMPLE>}
<SIMPLE> ::=<FIELD> := <EXP>
<FIELD> ::=<VARIABLE> '{" "<FIELD> ")" '{<INDEX>}
<VARIABLE> ::=<IDENTIF> '<INDEX> '{<IDENTIF> '<INDEX> 
<UNIT> ::=<FIELD> '{<FIXED>

<EXP> AND <BEXP>
ARE EXPRESSION AND BIT EXPRESSION AND ARE BUILT UP FROM
<UNIT> AND VARIOUS FUNCTIONS ALLOWED IN PHDL. THE SYNTAX
FOR THE EXPRESSION IS NOT SET OUT IN BNF BUT THE PRECEDENCE
ORDER IS GIVEN OVERLEAF. SPACES HAVE NOT BEEN DEALT WITH
IN THE SYNTAX BUT THE FOLLOWING RULES SHOULD APPLY.
MULTIPLY SPACES HAVE THE SAME MEANING AS SINGLE SPACES AND
SINGLE SPACES HAVE NO FUNCTION BUT DO ACT AS DELIMITERS.
CARRIAGE RETURN AND LINE-FEED ARE TREATED AS SPACES.
RESERVED DOUBLE SYMBOLS '(* ' AND '*)' MAY BE USED ANYWHERE
TO ENCLOSE COMMENTS AND THE ANALYSER SHOULD SKIP THESE.
CONTROL AND CLOCK SHOULD BE GLOBALLY DEFINED IDENTIFIERS THAT
EXIST IN THE IDENTIFIER TABLES BEFORE ANALYSIS BEGINS.
THE RECOMMENDED FUNCTIONAL PRECEDENCE FOR EXPRESSION EVALUATION
IS AS FOLLOWS:-

PARENTHESIS

INDICES

NOT | POSITIVE | NEGATIVE | /<DYADIC> | <OTHER FUNCTIONS>

MULTIPLY | DIVIDE
ADD | SUBTRACT
MOD
< | <= | >=
<> | =

AND

OR | XOR

INDICES ARE TREATED JUST AS ANY OTHER DYADIC FUNCTION THUS
EXPRESSIONS MAY BE INDEXED.
AND, OR AND XOR FUNCTIONS MUST BE PERFORMED ON FUNCTIONS OF
EQUAL BIT-WIDTH ELSE ONE OF THE OPERANDS MUST BE OF UNIT
WIDTH IN WHICH CASE THE FUNCTION IS PERFORMED BETWEEN IT AND
ALL ELEMENTS OF THE OTHER OPERAND.
INDEXING ON CONSTANT EXPRESSIONS AND META-VARIABLES IN THE
FIRST DIMENSION SHOULD NOT BE SUPPORTED.
**Am2901**

Four-Bit Bipolar Microprocessor Slice

**PRELIMINARY DATA**

**Distinctive Characteristics**
- 16-word x 4-bit two-port RAM.
- High speed ALU.
- 9-bit microinstruction word.
- Advanced low-power Schottky processing.
- Four-bit slice cascadable to any number of bits with full carry look-ahead.
- Three-state outputs.
- Shift left, no shift, or shift right entry into RAM from ALU.
- Output multiplexer for direct RAM A-port access or ALU output.
- Status flags include carry-out, sign-bit (negative), overflow and zero detect.
- Four-bit Q-register for scratch pad or accumulator extension.
- Direct ALU entry to Q-register.
- Shift Q-register left or right.
- RAM-shift and Q-shift are easily cascadable.

**GENERAL DESCRIPTION**

The four-bit bipolar microprocessor slice is designed as a high-speed cascadable element intended for use in CPU's, peripheral controllers, programmable microprocessors and numerous other applications. The microinstruction flexibility of the Am2901 will allow efficient emulation of almost any digital computing machine.

The device, as shown in the block diagram below, consists of a 16-word by 4-bit two-port RAM, a high-speed ALU, and the associated shifting, decoding and multiplexing circuitry. The nine-bit microinstruction word is organized into three groups of three bits each and selects the ALU source operands, the ALU function, and the ALU destination register. The microprocessor is cascadable with full look-ahead or with ripple carry, has three-state outputs, and provides various status flag outputs from the ALU. Advanced low-power Schottky processing is used to fabricate this 40-lead LSI chip.

**MICROPROCESSOR SLICE BLOCK DIAGRAM**
Distinctive Characteristics
- Quad high-speed LSI bus-transceiver
- Open-collector bus driver
- D-type register on driver
- Bus driver output can sink 100 mA at 0.8 V max.
- Internal odd 4-bit parity checker/generator

PRELIMINARY DATA
- Receiver has output latch for pipeline operation
- Three-state receiver outputs sink 12 mA
- Advanced Low-Power Schottky processing
- 100% reliability assurance testing in compliance with MIL-STD-883

LOGIC DIAGRAM

LOGIC SYMBOL

CONNECTION DIAGRAM
Top View

NOTA: Pin 1 is marked for orientation.
### TRUTH TABLE

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>INTERNAL TO DEVICE</th>
<th>BUS</th>
<th>OUTPUT</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>DRCP</td>
<td>BE</td>
<td>RLE</td>
<td>OE</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>H</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>H</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>↑</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>H</td>
<td>↑</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>L</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>H</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>L</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>L</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

H = HIGH  
Z = High Impedance  
X = Don’t Care  
↑ = LOW-to-HIGH Transition  
I = 0, 1, 2, 3

### PARITY OUTPUT FUNCTION TABLE

<table>
<thead>
<tr>
<th>BE</th>
<th>ODD PARITY OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>ODD = A0 ⊕ A1 ⊕ A2 ⊕ A3</td>
</tr>
<tr>
<td>H</td>
<td>ODD = Q0 ⊕ Q1 ⊕ Q2 ⊕ Q3</td>
</tr>
</tbody>
</table>

### DEFINITION OF FUNCTIONAL TERMS

**DRCP**  
Driver Clock Pulse. Clock pulse for the driver register.

**BE**  
Bus Enable. When the Bus Enable is LOW, the four drivers are in the high impedance state.

**BUS0, BUS1, BUS2, BUS3**  
The four driver outputs and receiver inputs (data is inverted).

**R0, R1, R2, R3**  
The four receiver outputs. Data from the bus is inverted while data from the A or B inputs is non-inverted.

**RLE**  
Receiver Latch Enable. When RLE is LOW, data on the BUS inputs is passed through the receiver latches. When RLE is HIGH, the receiver latches are closed and will retain the data independent of all other inputs.

**ODD**  
Odd parity output. Generates parity with the driver enabled, checks parity with the driver in the high-impedance state.

**OE**  
Output Enable. When the OE input is HIGH, the four three-state receiver outputs are in the high-impedance state.

### LOADING RULES IN UNIT LOADS

<table>
<thead>
<tr>
<th>Input/Output</th>
<th>Pin No's</th>
<th>Output Unit Load</th>
<th>Fan-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLE</td>
<td>1, 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A0</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BUS0</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>GND1</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BUS1</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A1</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BE</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ODD</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BUS2</td>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A2</td>
<td>13</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BUS3</td>
<td>14</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>GND2</td>
<td>15</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A3</td>
<td>17</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>DRCP</td>
<td>18</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VCC</td>
<td>20</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

A Low Power Schottky TTL Unit Load is defined as 20μA measured at 2.7V HIGH and -0.36mA measured at 0.4V LOW.
REFERENCES
REFERENCES

1. TANENBAUM, R.

2. Smalltalk

3.1. D'ANGELO, Henry

3.2. Understanding Microprocessors. (Motorola Semiconductors).

3.3. HARVEY, P.

3.4. Microprocessors (Electronics book series)

3.5. LEVIN, D.W.

4.1. NICK, John and BRICK, Jim.

4.2. Am 2900 Bipolar Microprocessor Family.
(Advanced Micro Devices Inc.)
5. TUCKER, Allen B.

6. LINZER, R.C., MILLS, H.D. and WITT, B.I.
Structured Programming in theory and practice.

7.1. BOULAY, G.G.

7.2. WOODWARD, M.E.
A Short Course on "Stored Program Controlled Systems".

8.1. SHIVA, Sajjan G.
Computer Hardware Description Languages - A Tutorial.

8.2. JORDAN, Henry F. and SMITH, Burton J.
The Assignment Statement in Hardware Description Languages.
(Computer, June 1977).

8.3. BARBACCI, Mario B.
A comparison on Register Transfer Languages for describing computers and digital systems.
February 1975.)
9. MINIMLA, Edward.

Optimisation Algorithms for Networks and Graphs.

(Karzel Dekker, AG, 1970).

10.1. AGRAWALA, Tilak.

Putting Petri Nets to Work.

(Computer, December 1979.)

10.2. PANTLI, C.A.

Interpretations of Set Theory.

(Internrer Bericht 75-05, Gesellschaft fur Mathematik und Datenverarbeitung, Bonn, W. Germany. December 1976).

11.1. LIEFFICK, B.W. ed.


11.2. TIBERGHIEN, Jacques.


11.3. TENENBAUM, Aaron M and AUGENSTEIN, Moshe J.


12. CHU, Y.

Computer Organisation and Microprogramming.

(Prentice-Hall, 1972).
13. DARRINGTON, J.A.
A Language for the description of digital computer processors.

14.1. HILL, F.J. and PETERSON, G.R.
Digital Systems: Hardware Organisation and Design.

14.2. FRIEDMAN, T.D. and YANG, S.C.
Methods used in an automatic logic design generator (ALERT)
July 1968).

14.3. THURBER, K.J.
Large Scale Computer Architecture. (Hayden, 1976).

15. BELL, C.G., GIBSON, J. and M attent, A.

Others:-

CRIES, David.

DABROUNDER, B.C.
Syntax of programming languages: theory and practice.
CASSANDRE: A language to describe digital systems, applications to logic design.
(3rd. Int. Symp. Computer and Information Science (COINS-69),
Miami Fla.) December 1969.

BARAY, N.B. & SU, S.Y.H.
A digital system modelling philosophy and design language.
(Proc. 8th Annu. Design Automation Workshop. Atlantic City,

BARBACCI, M.R., BELL, C.C. & NEWELL, A.
ISP: A language to describe instruction sets and other register
transfer systems.

BELL, C.C. & NEWELL, A.
The PMS and ISP descriptive systems for computer structures.

A common language for hardware, software and applications.

DULCY, J.R. & DIETMEYER, D.L.
A digital system design language (DDL).
Translation of a DDL digital system specification to Boolean equations.


ARKLE, R.L. & DIEHRMEYER, D.L.

DDLSIM - A digital design language simulator.


LIPOVSKI, G.J.

On gray box descriptions of microprocessors.


MUELLER, R.A. & JOHNSON, G.R.

A generator for microprocessor assemblers and simulators.


JORDAN, H.F. & SMITH, E.J.

Structure of Digital System Description Languages.

PYSHER, A.
