Accelerating the parsing process with an Application Specific VLSI RISC processor

by

John Derek McMullin

Thesis submitted to the University of Central Lancashire in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

October 1997

The work presented in this thesis was carried out in the Department of Electrical and Electronic Engineering at the University of Central Lancashire.

Closed Access
b/c003 975 320

006.35 MAC
m0001433ul

Declaration

I declare that while registered with the University of Central Lancashire for the degree of Doctor of Philosophy I have not been a registered candidate or enrolled student for another award of the University of Central Lancashire or any other academic or professional institution during the research programme. No portion of the work has been submitted in support of any application for another degree or qualification of any other University or Institution of learning.

Signed ....

John Derek McMullin

Table of Contents

Table of Figures

Table of Tables

Acknowledgements

I would like to express my sincere thanks to my Supervisor, Greg Stevenson for his help, guidance, support and patience during the various stages of this research project.

I wish also to acknowledge the support of the University of Central Lancashire to this research. I wish to thank Mr. Simon Hill for the technical support during the project.

Finally, I wish to give special thanks to my wife, Jacqueline and daughter, Caroline for their love and support throughout this research.

Abstract

This thesis investigates the topic of the design, implementation and potential use of specialised hardware used to accelerate the recognition and translation of computer programs expressed in a range of computer languages. This investigation focuses specifically on the twin processes of parsing and lexical analysis.

The research described was carried out in two areas namely, the feasibility of designing a specialised instruction set for a RISC like processor able to accelerate the parsing and lexical analysis process, and the physical implementation of a RISC processor in CMOS VLSI technology able to execute the designed instruction set.

The feasibility of mapping the process of language recognition onto the instruction set of a RISC processor is investigated. This involves an assessment of the suitability of the LL(1) and LALR(1) algorithms, both of which are used for parsing, and other associated algorithms, used for lexical analysis, as a basis for an appropriate instruction set architecture. The feasibility of an instruction set design which uses fixed size instructions with variable size data fields to ensure scaleable operation is also investigated. The appropriate software mechanisms used to validate the instruction set architecture are outlined.

The practical implementation using CMOS technology of a RISC processor able to execute the new instruction set is investigated. In particular the feasibility of using bit-slice technology to implement the processor having fixed size instructions with variable size data-paths and address ranges is investigated.

The combination of novel instruction set with variable data-widths and the fabricated devices able to activate semantic actions directly from hardware together form an original contribution to the field of parsing and lexical analysis.

1. Introduction

The use of high level languages such as PASCAL, C++, or Java to specify computer behaviour depends on the ability of computers to analyse the source text and translate the implied meaning (or semantic specification) into executable machine instructions.

This research investigated the feasibility of designing a specialised processor having an instruction set aimed specifically at the problem of recognising computer programs written in a wide range of computer languages. The architecture of the RISC processor also had to provide interfaces to allow external hardware to implement the semantics implied by the computer program. This research, therefore provides a practical application of the theory of parsing and lexical analysis, two fundamental concepts of language theory extensively used within computer science. These theories of lexical analysis and parsing are well known.

In the following sections of Chapter 1, the main concepts involved in parsing and lexical analysis are briefly described. This is needed to provide an informed background before discussing both the research objectives and the overview of the thesis.

For more information refer to the literature on the theory of parsing and lexical analysis. The following is a partial list of useful references :-

[Aho 1977] "Principles of Compiler Design"

[Brown 1981] "Writing Interactive Compilers and Interpreters"

[Denning 1978] "Machines, Language and Computation"

[Fischer 1991] "Crafting a Compiler with C"

[Hunter 1981] "The Design and Construction of Compilers"

[McGettrick 1980] "The Definition of Programming Languages"

[Minsky 1972] "Computation: Finite and Infinite Machines".

1.1 Parsing and Lexical Analysis - An Overview

Just as the process of reading a book uses the twin processes of lexical analysis and parsing, so does the example of reading and translating a computer program. Firstly the sequences of characters are analysed into words (possibly requiring a dictionary). Next the combinations of words are analysed to see if they form correct sentences. The final stage is to recognise the meaning of the individual sentences and perform any actions implied by the meaning.

Lexical Analysis is defined as being the process of recognising "words" from sequences of characters in the computer program source text. The dictionary (or lexicon) of words depends on the language being used.

Parsing is defined as being the associated process of checking that the sequence of recognised "words" forms a valid sentence in the language. Different languages will have different structures for a legal sentence.

Together these two processes recognise source code as belonging to a specific language and also provide hooks to allow the generation of executable code based on the semantics implied by the source text. The "definitions" provided describe the behaviour of a parser and lexical analyser as if they were "black boxes". Some awareness of the internal operations of a parser and lexical analyser is needed for an understanding of both the processor architecture (and its instruction set) and its practical implementation in hardware. The actual internals and concepts involved in both parsing and lexical analysis are introduced informally. To highlight the important concepts in parsing and lexical analysis, the example (mentioned above) of reading a sentence in a book will be used.

1.2 Parsing - The Basic Concepts

As indicated in the previous section, the parsing process checks that the tokens (or words) provided by the lexical analyser form a legal structure or sentence in the given language. This section outlines some of the terminology used in parsing theory (as it relates to the research).

1.2.1 Sentences and Language

Informally, a **sentence** consists of a number of words with some constraints on the sequences of words allowed. There will be a fixed collection of words (known as a lexicon or dictionary).

Each **word** (or **token**) in the lexicon will consist of a sequence of letters where each letter is taken from a fixed alphabet.

A **language** will be defined by the combination of constraints on the word sequences and the lexicon. A language could therefore consist of many possible sentences.

For example, given a dictionary D of tokens where

D = {“a” , “on”, “cat”, “ball”, “sat”, “the”, “threw” }

then we could form the sequences R, S, T where

R = “a” “ball” “threw” “a” “cat”.

S = “a” “cat” “threw” “a” “ball”.

T = “the” “cat” “sat” “on” “the” “ball”.

(Note that the full-stop is only used to indicate the end of the token sequence)

We could define all three sequences to be a sentence and state that the language L only contained these three sentences. Thus we could formally define the language L to be a set of sequences. That is,

L = { R, S, T } where R, S, T are defined above.

For a language containing many sequences (possibly an infinite number) it could be difficult to verify if a particular sequence of tokens is a sentence due to the large number of sentences.

1.2.2 Languages and Grammars

Instead of defining a language by listing all its sentences, an alternative mechanism is to generate the sentences from a simple set of rules known variously as **re-write** or **production** rules.

The following are all examples of productions.

L = R “cat”

L = X “ball”

X = “a” S

X = “the” T

R = “a” “ball” Y

S = “cat” Y

T = “cat” “sat” “on” “the”

Y = “threw” “a”

A sequence of symbols can be used more than once as the left hand side of a production. Thus L and X each have two productions.

Productions can also have more than one symbol on the left hand side. The following are also examples of productions.

A B C = X “sat”

A C B = X “saw”

Grammars which have productions with rules having more than one symbol as part of the left-hand side are called **context-sensitive**. Grammars where all productions have only one left-hand symbol are called **context-free**. Natural languages such as English or French can only be described by context-sensitive grammars. Computer languages such as Java or PASCAL can be described using context-free grammars. The research only investigated the acceleration of recognition of languages generated from context-free grammars. Accordingly from this point, any grammar described will be context-free.

Also a production can have an empty right-hand side. That is the right hand side contains the null symbol, called **epsilon**. The production is also known as an **epsilon** production.

X =

X = '0'

This defines X as being null or '0'.

A symbol which appears on the left-hand side of a production is known as a **non-terminal** symbol. A non-terminal symbol can appear on the right-hand side of a production. Thus L, X, R, S, T and Y are all non-terminal symbols. A symbol which only appears on the right-hand side of a production is known as a **terminal** symbol. Thus "cat", "ball", "a", "the", "sat", "on" and "threw" are all terminal symbols. The **epsilon** symbol is an example of a terminal symbol. A symbol can be either a terminal or non-terminal symbol. It cannot be both.

Starting with the symbol L and using the productions

L = R "cat"

R = "a" "ball" Y

Y = "threw" "a"

generates the chain

L

=> R “cat”

=> “a” “ball” Y “cat”

=> “a” “ball” “threw” “a” “cat”

Similarly starting from the symbol L and using the productions,

L = X “ball” etc.

gives the two chains

L

=> X “ball”

=> “a” S “ball”

=> “a” “cat” Y “ball”

=> “a” “cat” “threw” “a” “ball”

L

=> X “ball”

=> “the” T “ball”

=> “the” “cat” “sat” “on” “the” “ball”

The generation of all possible chains from a starting expression or symbol (such as L) by means of the productions is known as a **derivation**. The derivation set is the set of sequences obtained when no further production can be used.

Thus we can see that the three sequences

“a” “ball” threw” “a” “cat”,

“a” “cat” “threw” “a” “ball”,

"the" "cat" "sat" "on" "the" "ball"

are all derivations from the symbol L. Notice also that the set of derivations from the symbol L is identical to the set of sentences which represent the language L as defined earlier.

We define a **goal** symbol to be a unique non-terminal symbol whose set of derivations forms the language or the set of sentences.

A **grammar** is defined to be the combination of productions, goal symbol, non-terminal and terminal symbols. The set of all derivations from the goal symbol (using the productions) gives the **language** defined by the grammar.

Finally, grammars can be described using notation other than re-write rules. One such notation uses ? + and * to indicate repetition of symbols.

Define a* = {} u { a } u {aa} u {aaa} u ... = {epsilon, a, aa, aaa, ... }. That is, a* represents the repetition of "a" from 0 to many times. (Sometimes the notation [a] is used instead of a*).

Define a+ = { a } u { aa } u { aaa } u ... = { a, aa, aaa, ... }. That is, a+ represents the repetition of "a" from 1 to many times.

Define a? = { } u { a } = { epsilon, a }. That is, a? represents the repetition of "a" from 0 to 1 times.

Note also that a* is the same as a+?.

Define "," to represent the choice between two sequences.

Thus a , b = { a , b }

A grammar expressed using the "?" "*" "+" and "," symbols can be easily converted to re-write rules. For example, A = C* can be converted to

A = X

X =

X = X C

or

A = X

X =

X = C X

and A = C+ can be converted to

A = X

X = C

X = X C

or

A = X

X = C

X = C X

The "*", "+", "?" and "," notation is used extensively in the following chapters.

1.2.3 Parsing and Grammars

The use of grammars to define languages aids greatly in the process of recognising whether a sequence of tokens is a sentence. Grammars do this by converting the parsing process into a game of "**syntactic dominoes**". Each production becomes a domino where the tokens represent the domino spots. The game starts with the goal symbol on one side and the proposed sentence (or sequence of terminal symbols) on the other side. The two sides are then joined together by using the legal dominoes (or productions). If it is possible to join the goal symbol to the token sequence using only legal dominoes then the token sequence is a sentence of the language. The game is identical to finding a derivation of the goal symbol except that the end result is known from the start.

There are a number of strategies (or parsing methods) possible which can be used to play this game. Two of these are important for use as potential parsing mechanisms and are outlined below. The first strategy is where dominoes are always added from the goal symbol side down towards the token sequence side. This is known as a **top-down** parse. The other strategy of interest is where dominoes are always added from the token sequence side up towards the goal symbol side. This is known as a **bottom-up** parse. The pattern made by the dominoes is called the **parse-tree**. An example parse-tree is shown below.

This example parse-tree is generated when recognising that the token sequence "10 + 11" belongs to the language defined by the grammar with goal symbol E and productions,

E = E + T

E = T

T = D

T = T D

D = 0

D = 1

This parse-tree could be created either bottom-up (from the token sequence 10+11) or top-down (from the goal symbol E).

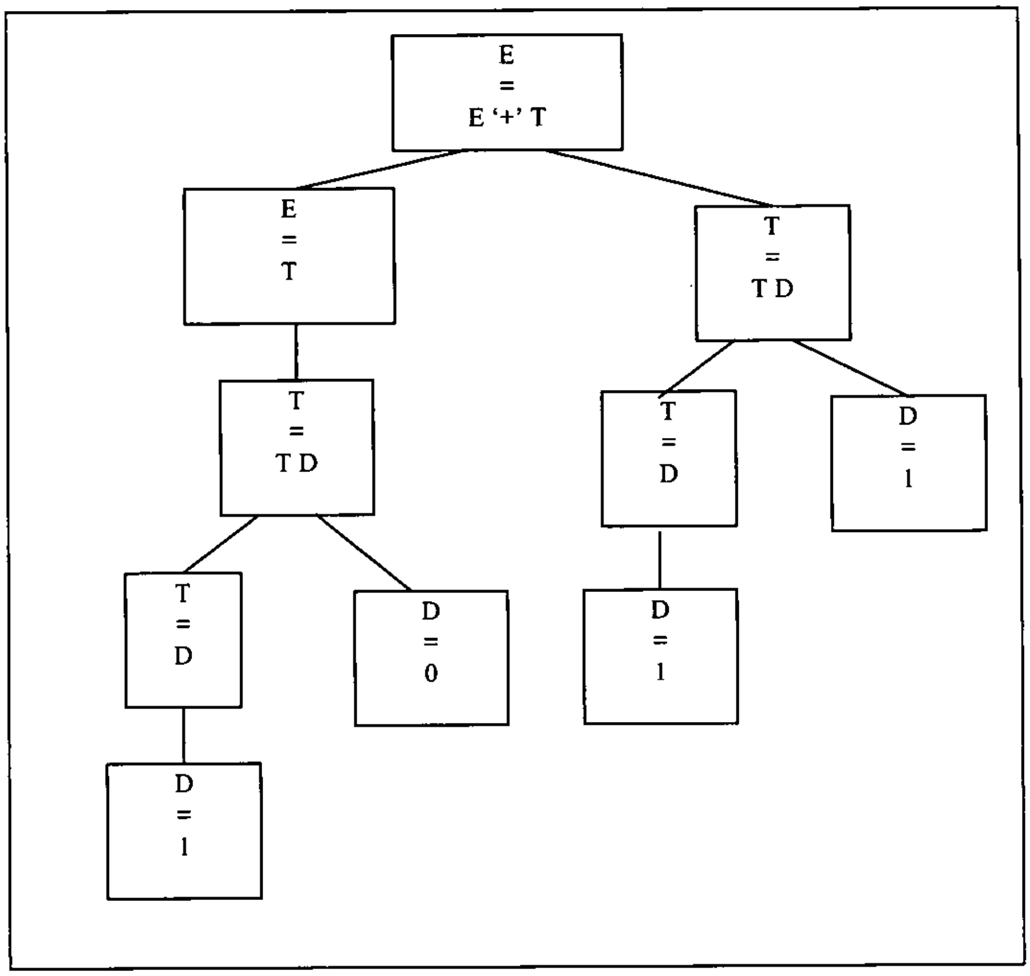

Figure 1 - A Parse-Tree

The following diagram shows a left to right traversal of the parse-tree starting from (and returning to) the top (or goal symbol)

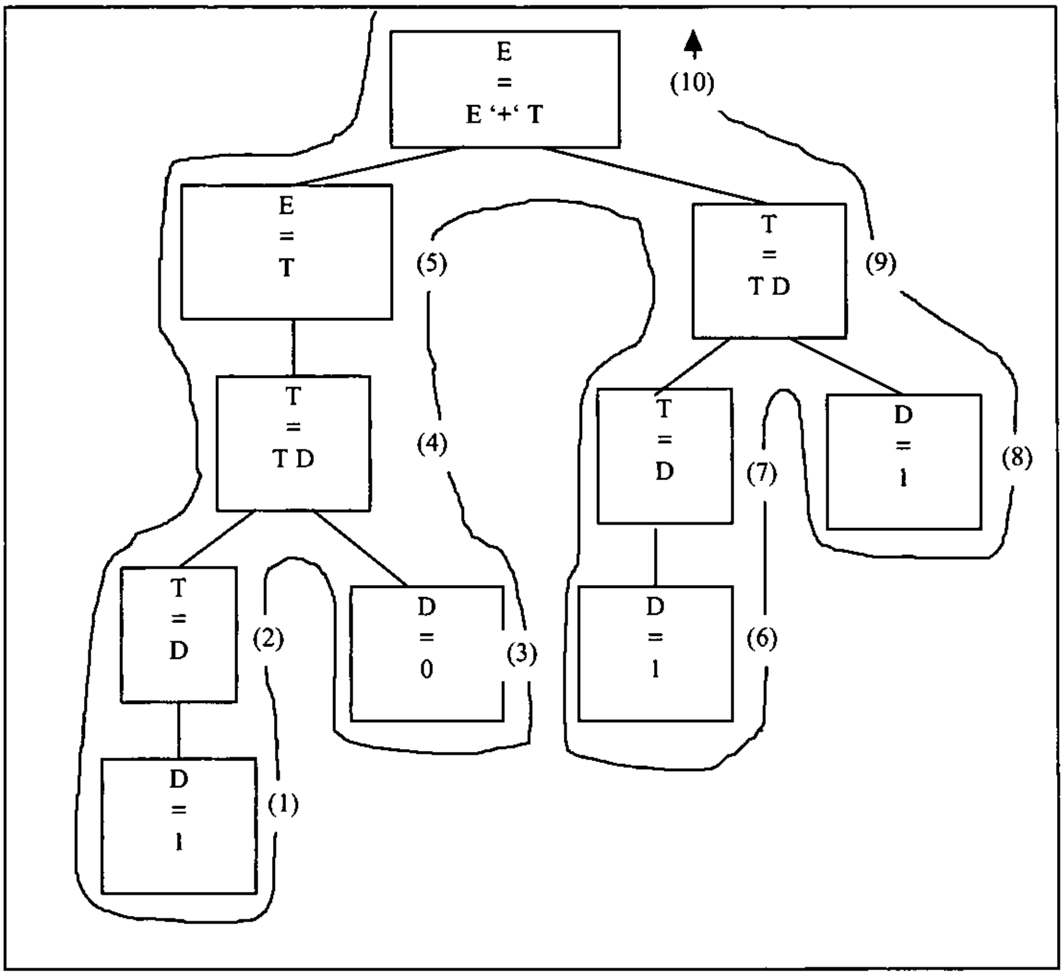

Figure 2 - Parse-Tree Traversal

A **pre-order traversal** of a parse-tree is a visit of the node N then a recursive visit of the sub-trees rooted at children N 1..k of a node N. A **post-order traversal** of a parse-tree is a recursive visit of the sub-trees rooted at children N 1..k of a node N then a visit to N.

For the above parse-tree the sequence of rules given by both post-order and pre-order traversals are shown.

Parse Tree Label	Pre-Order Traversal Rule sequence	Parse Tree Label	Post-order Traversal Rule sequence
10	E = E + T	1	D = 1
5	E = T	2	T = D
4	T = T D	3	D = 0
2	T = D	4	T = T D
1	D = 1	5	E = T
3	D = 0	6	D = 1
9	T = T D	7	T = D
7	T = D	8	D = 1
6	D = 1	9	T = T D
8	D = 1	10	E = E + T

Table 1 - Pre and Post-order Traversal of Parse Trees

It is important to note that a pre-order traversal predicts the rule to be recognised, and that a post-order traversal is able to refer to data already recognised.

1.2.4 Grammars and Semantics

The purpose of a language is to communicate meaning or semantics. A grammar with its productions can only define the structure of the sentence. It cannot normally indicate the semantics or meaning of a particular sentence. However, it is possible to attach semantic actions to the individual productions. That is each production can have an action to perform when the rule is recognised.

The following example shows how binary numbers can be recognised from text and then added. This example uses the grammar and parse-tree shown in the previous section, where the following table shows the productions and associated actions to recognise binary numbers. The goal symbol of the grammar is E.

Production	Action
E = T	E := T
E = E '+' T	E := E + T
D = '0'	D := 0
D = '1'	D := 1
T = D	T := D
T = T D	T := 2*T + D

Table 2 - Productions and semantic actions

The semantic actions shown assume that there are three registers E, T and D. For the token sequence 10 + 11, we use the post-order traversal of the previously mentioned parse-tree. Writing the rules with associated semantic action in reverse order (starting from the token sequence to the goal symbol i.e. bottom up) (evaluating E,T,D as we go)

Parse Tree Label	Rule	Action	Value of D	Value of T	Value of E
1	D = 1	D := 1	1	?	?
2	T = D	T := D	1	1	?
3	D = 0	D := 0	0	1	?
4	T = T D	T := 2*T + D	0	2	?
5	E = T	E := T	0	2	2
6	D = 1	D := 1	1	2	2
7	T = D	T := D	1	1	2
8	D = 1	D := 1	1	1	2
9	T = T D	T := 2*T + D	1	3	2
10	E = E + T	E := E + T	1	3	5

Table 3 - Semantic actions and Parse-Trees

Note that E contains the answer 5 (in decimal) which is the value of 10 + 11 in binary.

Thus we can see that semantic actions can be attached to the syntax productions to obtain the desired meaning. In pre-order traversal the semantic action is triggered at the start of a predicted production. For post-order traversal the semantic action is triggered after the production has been recognised. The left-right post-order traversal emits the productions in a sequence where all information is acquired before using it. The left-right pre-order traversal of a parse-tree predicts but does not acquire the information required at the time of triggering of a semantic action.

Therefore, the design of semantic actions to attach to productions works best when the rules are recognised in the order given by the post-order traversal of the parse-tree. It must be noted that the semantic actions are specific to the productions. Changes to the productions will require different semantic actions.

1.2.5 Grammar Hierarchies

For a given language there may be more than one grammar which can "generate" all the sentences in that language. However the converse of a grammar generating multiple languages is not possible. This is due to the uniqueness of the derivation set (or set of sentences in the language) obtained from the goal symbol.

1.2.5.1 Ambiguous Grammars

It is possible to have a grammar which allows a sentence to be derived in more than one way from the goal symbol. This is equivalent to stating that there is more than one parse-tree for at least one sentence in the language. An **ambiguous** grammar is defined to be a grammar with this property. The following is an example of an ambiguous grammar.

Using E as the goal symbol with the following productions,

E = E '-' E

E = number

where number and '-' are both terminal symbols. '-' represents the arithmetic subtraction operator.

The sentence "n '-' n '-' n" can be derived from E in two possible ways. Note that the two routes generate the same sequence or derivation.

Parse Tree 1.

E

=> E '-' E

=> number '-' E

=> number '-' E '-' E

=> number '-' number '-' number

This is equivalent to working out the value of number - (number - number)

Parse Tree 2.

E

=> E '-' E

=> E '-' number

=> E '-' E '-' number

=> number '-' number '-' number

This is equivalent to working out the value of (number - number) - number.

These values are not normally the same. They are only equivalent if the last number is zero or just one of the first two numbers is infinite.

A parser for an ambiguous grammar could generate an incorrect sequence of actions. Therefore ambiguous grammars must not be used.

1.3 Parsing Methods

The practicality of using productions to both define a language and also to assist as place-holders for semantic actions should have been demonstrated. What has not been shown are any algorithms which can recognise a token sequence as being a sentence, or how productions can be used in the process. The format of productions should indicate that state machines could be used to implement the parsing and lexical analysis processes. The following sections indicate the use of state machines to implement the parsing process which are based on productions.

1.3.1 State Machines

A state machine can be represented by a graph where each node corresponds to a state and the arcs linking the states represent the state transitions. Each transition is a directed arc (or one-way street) from one state to another state and is tagged with the transition trigger. For the parsing process, state transitions are triggered by the currently visible terminal or non-terminal symbol. Each production could be mapped onto a number of states (depending on the number of symbols in the rules' right-hand side) where the transitions are triggered in sequence by the symbols of the production.

For example,

The production R = A '+' B could become the state machine S with states S1, S2, S3, S4 where S1 is the initial state and S4 the final state.

$$\begin{tikzcd}
\text{S1} \arrow[r, "A"] & \text{S2} \arrow[r, "'+'"] & \text{S3} \arrow[r, "B"] & \text{S4}
\end{tikzcd}$$

Figure 3 - State Machine

State machines can be classified depending on the transitions and number of states. Generally there can be many start states and many end states for a given machine.

A **deterministic** state machine is one with many states (possibly infinite) where each input symbol has at most one transition from each state. Thus for each state and for each possible input symbol (terminal or non-terminal) it is possible to "determine" the next state. This machine ensures that there is no ambiguity of transitions when reading a stream of input symbols.

A **finite-state** state machine is one with a countable (finite) number of states. When used to implement a parser there will be a single start state with a single end state. The end state will indicate that the goal symbol has been recognised. This type of state machine just stores the current state as a single variable. State transitions are decided by noting the current state and current symbol in the input queue, using the state transition table to determine the next state, which becomes the new value of current state. The current symbol is consumed and the following symbol becomes the current symbol. This machine has two types of instructions, a **shift** to state instruction and an **accept** instruction attached to the end state.

A **finite-state machine with stack** is similar to a finite-state machine except that a stack is used to store the current states. State transitions are decided by noting the state on the top of the state stack and next symbol from the input queue and using the transition table to determine the next state. This next state is then pushed onto the state stack. These transitions are **shift** instructions. There are some states where a production is recognised (possibly determined by the next input symbol). The production will be **reduced** at this point. The state stack is popped by the number of symbols in the right hand side of the rule and the top of the stack becomes the current state. The symbol on the left of the rule is inserted into the input queue to become the new next symbol. This has an additional **reduce** instruction as well as the shift and accept instructions. The top-down and bottom-up strategies can both be emulated using this type of state machine.

In both types of finite state machines it is possible to have the state transition or reduce conditional on knowing more that just the current next symbol but the subsequent next n symbols. In practice n is either 0 or 1. The next n symbols are known as the **look-ahead** symbols.

There are many algorithms to construct the transition (or **goto**) tables and the reduce tables are discussed next.

1.3.2 Algorithms

There are a large number of parsing algorithms, however only the three most common will be outlined. Each algorithm involves creating a mechanism to decide which production to recognise. A grammar may have a non-terminal symbol which defines a number of productions. Deciding which rule to start to recognise requires knowing the set of symbols which can possibly start a rule.

1.3.2.1 First and Follow Sets

A parser needs to decide which rule to start recognising for those productions defined by a common non-terminal. Knowledge of which terminal symbols can be seen first when starting a production is then required. Also knowledge of which terminal symbols can be seen following after a rule is useful to decide whether to recognise a rule or continue shifting to other states.

A **first** set for a non-terminal symbol N contains those terminal symbols which can be present at the start of the productions for the symbol. It is the set of those terminal symbols which appear first in the set of derived sequences from N. The epsilon (or null) symbol can also be in this first set. Note that the first set for a terminal symbol is the set containing itself.

Example.

Given the productions for a grammar as being

```
S = A '*'
S = A Z '$'
A =
A = B
A = C A
B = '1'
C = '0'
Z =
Z = '.'
```

then the derivatives D of A are
D = { epsilon, '1', '0'... '1', '0'... '0'}

The start set is therefore { epsilon, '0', '1' }

Figure 4 - Start Sets

Elimination of the epsilon symbol from the first set of a non-terminal symbol N depends on where N is referenced in the other productions. One method is to augment the first set with the first set of each symbol immediately following N when N appears on the right-hand side of a production. Should there be no following symbol in the rule where R is the left-hand symbol then use the first sets of the symbols following references to R. Repeat this until epsilon is eliminated.

Using this method for the above example gives the start set S for A where epsilon has been removed as S = { '*', '.', '$', '0', '1' }.

A **follow** set for a non-terminal symbol contains those terminal symbols which can be legally expected to occur after the symbol. It can be generated by forming the set of non-terminal and terminal symbols which can follow the specified non-terminal symbol in all productions which reference it. The non-terminal symbols are repetitively replaced until no more terminal symbols can be added. Any non-terminal symbols are replaced by the non-terminal symbol starter set q.v. A non-terminal symbol which is defined by an epsilon rule is replaced by its own follower set. A non-terminal symbol which is the last symbol in a production and is defined by an epsilon rule is replaced by the follower set of the non-terminal on the left of the production.

Example.

Given the productions for a grammar as being

```
S = A '*'
A =
A = '1'
A = '0' A
```

then the follower set for A is { '*' } , since only '*' occurs after A

Note.
Non-terminal A has an epsilon rule,

A =

Figure 5 - First and Follow Sets

First and Follow sets are used by the following three parsing algorithms.

1.3.2.2 Recursive Descent

This algorithm uses the grammar rules and involves writing a sub-routine for each production. The routine corresponding to the rule defining the goal symbol is the first to be called. The sub-routines are potentially recursive and use the top-down approach. For a non-terminal symbol which has many productions, deciding on the appropriate rule will require examining the current next symbol and comparison with the start set for the non-terminal.

A recursive descent parser (in a PASCAL-like notation) based on the example grammar used above to demonstrate starter and follower set is given as an example.

```
(* next_token is a lookahead to the current next token *)
(* read_token consumes the current next token from the input stream *)
(* this recognises all rules which define A *)
procedure A
begin
 case next_token  of
'*': ;
'0': read_token;
'1': begin
     read_token;
     A;
   end
 else
   error;
 end
end

(* this recognises all rules which define S *)
procedure S
begin
 A
 if next_token = '*' then
   read_token
 else
   error;
end
```

Figure 6 - Recursive descent parser

This algorithm does not check if the grammar is unambiguous and may require extensive manual re-work to alter the parser should the grammar be modified.

1.3.2.3 A Top-down Algorithm using one Lookahead Symbol

This algorithm is based on a finite-state automata with stack using a top-down (or predictive) approach and reading the source symbols from left to right, using one symbol of look-ahead to help decide on the appropriate state transition or rule recognition. The parse trees generated are traversed using a pre-order (or left hand) traversal.

This algorithm is usually known as the LL(1) parsing algorithm, where :-

L - read the text from Left to right

L - use a Left-hand (or pre-order) traversal of the parse tree

(1) - always use one look-ahead symbol

The LL(1) algorithm can be generalised to use more than one look-ahead symbol when it is known as LL(n) (or LL), where n is the number of look-ahead symbols required. The LL algorithm requires the placing of constraints on the productions of the grammar which mean that many grammars cannot use this algorithm.

The algorithm has two parts, the state table generation and the parser which uses the state table. The first and follow referred to in the algorithm are the first and follow sets for a non-terminal symbol. These sets will not have had the epsilon symbol removed.

```
Input: Grammar G
Output: Parse table (or array) M

Note $ is used to denote end of input.

Algorithm:

For each production A -> rhs of the grammar
1) for each terminal a in first(rhs), add A = rhs to the table M[A,a]
2) If epsilon is in first(rhs), add A = rhs for each terminal b in follow(A).
   If epsilon is in first(rhs) and $ is in follow(A), add A = rhs to table M[A,$]

Finally mark each undefined entry in M as error.

If there is more than one entry in any M[A,x] then the grammar is not LL(1)
```

Figure 7 - LL(1) Table Generation

The following is the LL(1) parse table generated for the example grammar used to indicate recursive descent.

Note:

first(A '*') = { epsilon, '0', '1' }, follow(A '*') = { $ }

	Terminals			
Non-terminals	'0'	'1'	'*'	$
S	S = A '*'	S = A '*'	S = A '*'	S = A '*'
A	A = '0' A	A = '1'	A =	

Table 4 - LL(1) Parse Table

The generated parse table M is used in the parser. This LL(1) parser is described using a pseudo-PASCAL notation.

```
(* let X be the top stack symbol *)
(* let a be the next input symbol *)
push goal symbol G onto the stack
repeat
  if X is a terminal or $ then
  begin
    if X = a then
      pop X from stack and consume a from input
    else
      error
  end
  else
  begin
    if M[X,a] = X -> Y1 Y2 ... Yk then
    begin
      pop X from stack
      push Yk,Yk-1, ... Y1 onto stack (Y1 new stack top)
    end
    else
      error
  end
until stack is empty
```

Figure 8 - LL(1) parser

For more details of this algorithm refer to [Aho 1977].

1.3.2.4 Parsing from the Bottom Up using Lookahead symbols

This algorithm (like the LL(1) algorithm) is also based on a finite-state automata with stack but it uses a bottom-up approach. It reads the source from left to right and generates a right-hand (or post-order) traversal of the parse tree using a number of look-ahead symbols to determine state transitions.

The algorithm is known as LR(1), where :-

L - read source from **L**eft to right

R - use a **R**ight-hand traversal of the parse tree

(1) - always use one look-ahead symbol

The LR(1) algorithm can be generalised to use more than one look-ahead symbol when it is known as LR(n) (or LR), where n is the number of look-ahead symbols required.

The LR algorithm was first described in the paper [Knuth 1966] "On the Translation of Languages from Left to Right" and extended in the paper [DeRemer 1971] "Simple LR(k) Grammars".

A variant of the LR algorithm which only uses one look-ahead symbol where necessary is known as the LALR(1) algorithm, where :-

L - Look

A - Ahead

L - read source from Left to right

R - use a Right-hand traversal of the parse tree

(1) - with at most one look-ahead symbol

The mechanism for generating LALR(1) parse tables is best shown in [Aho 1977] "The Principles of Compiler Design". The paper [Pager 1977] "The Lane-Tracing Algorithm for Constructing LR(k) Parsers and Ways of Enhancing Its Efficiency" improves on the algorithm in [Aho 1977].

It can be shown that a grammar which is LL(1) is also LALR(1). The converse is not true. Therefore LALR(1) can recognise more languages than LL(1). It can also be shown that the LALR(1) algorithm will detect ambiguous grammars. The LALR(1) parse table contains shift and reduce instructions.

The example LALR(1) parse table, shown below, is generated from the grammar used to demonstrate the recursive descent parser.

	States					
Symbols	1	2	3	4	5	6
$					accept	
S						
A	s2			s6		
*	r"A = “	s5	r“A = 1”	r“A = “		r“A = 0 A”
1	s3			s3		
0	s4			s4		

Table 5 - LALR(1) Parse Table

Entries in the LALR(1) parse table M[X,t] contain shift, reduce or accept instructions, where X is a state and t is a symbol (terminal or non-terminal). Blank entries represent parse errors.

The following is the pseudo PASCAL code used to operate the LALR(1) parsing algorithm.

```
Let X be top of state stack, S next state,  t next token

repeat
 case M[X,t].instruction of
shift:
   begin
     S = M[X,a].data
     push S onto state stack
     consume t from input queue
   end
reduce:
   begin
     reduce rule = M[X,a].data
     R -> R1 R2 ... Rk
     pop k items from state stack
     insert R into input queue as head of queue
   end
error:
   begin
     error
   end
 end
until t = Goal Symbol
```

Figure 9 - LALR(1) Algorithm

This algorithm is based on a table driven state machine with SHIFT and REDUCE instructions.

1.4 Lexical Analysis Methods

Each parser will have an associated lexical analyser which reads the source text combining character sequences to form the parse tokens.

1.4.1 Dictionary Lookup and Names

As indicated earlier, lexical analysis is akin to looking up a word in a dictionary. For a language there is a finite set of parse tokens possible. However some of these parse tokens can have many forms. Keywords such as **while** or **do** have a fixed form. Examples of tokens that can have many forms are variable identifiers (akin to proper nouns in English) and also integers. These types of tokens can be defined by sets of simple rules.

Most computer languages have an overlap in the rules for defining an identifier and a keyword. That is, a keyword could be regarded as an identifier. There are two alternate mechanisms to resolve this problem. When a character sequence is found which matches a keyword either accept the keyword or continue the sequence whilst the sequence obeys the format of an identifier. The first method does not allow an identifier to start with the text of a keyword, the second method does.

1.4.2 Finite State Automata

The concept of tokens being words in a dictionary leads to the possibility that each token can be detected by a finite state automata. The dictionary can be implemented as a non-deterministic automata q.v. which has a single start state and multiple end states, one for each token.

A **non-deterministic automata** is a state machine which allows transitions to multiple states from a state for the same trigger (or character). That is, it may not be possible to determine the next state to reach if the trigger (or character) has a transition to more than one state.

A **deterministic automata** is a state machine which has at most one transition from each state for a trigger.

It is possible to describe the format of tokens and have mechanisms to convert the token definitions into a format which represents a non-deterministic automata. Also, there are algorithms which can convert non-deterministic automata into deterministic automata. These algorithms are shown in [Aho 1977]. Both mechanisms can be implemented as computer software taking as input the set of token definitions for a language and outputting a description of a deterministic finite state automata.

A deterministic finite state automata is another example of a state machine. This has two instructions, SHIFT and ACCEPT. Each SHIFT instruction knows of the transition trigger and the target state. Each ACCEPT instruction knows of the token just recognised for use by the associated parser.

1.5 Compiler-Compilers

The section on parsing mentioned that mechanisms existed which could automatically generate parsers from the language re-write rules, subject to constraints on the rules. A software tool known as a **compiler-compiler** is an example of such a mechanism. This type of tool has three mechanisms :-

- one for generating parsers from grammars
- one to allow semantics to be hooked in
- one for generating a lexical analyser from the token definitions

A compiler-compiler could generate recursive descent parser from a grammar but would need to ensure the grammar is unambiguous. However, most compiler-compilers use either the LL or LR algorithms to validate the re-write rules and to also generate the appropriate parse tables. Using either the LL or LR algorithms means that a general purpose table-driven parser routine can be used. A compiler-compiler (based on the LALR(1) algorithm which is a variant of the LR algorithm) which was designed and implemented by the author was used as a background tool within the research. The implementation of the run-time parsing and lexical kernel formed the basis for the architecture of the hardware developed as part of the research.

1.6 Objectives of the Research

The research had three objectives.

Firstly, to investigate the possibility of accelerating the parsing and lexical analysis by using specialised hardware. This was to determine if it was possible to have hardware (specific to the recognition of languages) which was sufficiently general purpose to recognise most computer languages. An alternative was to have hardware specific to each individual language. Part of this work was to investigate the suitability of the various parsing and lexical analysis algorithms for implementation as hardware.

Secondly, to verify if it was possible to design an appropriate instruction set which could be used for the lexical analysis and parsing processes in combination.

Lastly, to implement a VLSI chip set capable of executing designed instruction set.

This research is based on the authors own software implementation of a simple compiler-compiler. This compiler-compiler uses the LALR(1) algorithm to both validate the input grammar and to generate the parsing tables automatically.

1.7 Overview of the Thesis

This thesis describes current work in the field, and the results of the work carried out to fulfil the research objectives.

Chapter 2 indicates the current status of work in the field.

Chapter 3 describes the steps taken to design the instruction set. The use of the compiler-compiler system to validate the various designs of instruction sets is indicated.

Chapter 4 describes the hardware design of the RISC processor to execute the instruction set. The further use of the compiler-compiler system to both derive, simulate and generate test vectors for the logic design is also shown.

Chapter 5 indicates the potential applications of the hardware.

Finally, Chapter 6 indicates the scope for further research based on the implemented hardware.

Also, there are a number of appendices attached which describe the logic design, its validation and a simple language that can be recognised by the processor.

Appendix A describes the steps taken to validate the hardware design by using the software implementation of the parser to generate test-vectors.

Appendix B details some of the logic design for the bit-slice device.

Appendix C provides an example of a simple language (with grammar) which can be used to synthesise logic design from regular expressions.

2. Hardware Implementations

The practical implementation of hardware to accelerate the parsing and lexical processes has not been widely considered or described in the literature. Most papers describe theoretical hardware implementations of either lexical analysers or parsers which use the LL(1) algorithm. The LALR(1) parsing algorithm which allows a wider range of languages to be recognised does not appear to have been considered.

What follows is a discussion of the paper [Evans 1985] "Architectures for Language Recognition" which describes two hardware architectures, one used to implement recognisers for regular expressions suitable for a lexical analyser, and the second to implement an LL(1) parser. Also discussed is another paper [Kazuo et al. 1983] "Design and Evaluation of Parsing Chip" which describes the implementation of the LL(1) algorithm but is specifically targeted at implementing a parser for PASCAL.

2.1 Recognising Regular Expressions in Hardware

A regular expression is formed from mixing tokens with indicators showing token repetition. Thus the notation which uses "+", "*", "?" , "," and also "[" "]" (briefly described in the previous chapter) can be used to form regular expressions. Thus, in this notation, [a](d, b c) is an example of a regular expression. It could alternatively be written as a* (d, b c). This notation can be used to specify both lexical analysers or parsers after transformation into re-write rules.

2.1.1 Logic Design

The following describes an architecture to recognise regular expressions where the individual tokens (for parsers) or characters (for lexical analysers) are input at regular clocked intervals.

2.1.1.1 Recognising a Token

As shown in Figure 10 - Token recogniser cell, this logic cell accepts as input signals the next character or token, the clock and the current result. It will output the new result for input to the next stage. The token to be recognised (or token reference) is compared for equality with the token input. The result of the comparison is logically anded with the current result to form the value to be latched into a flip-flop to form the next result.

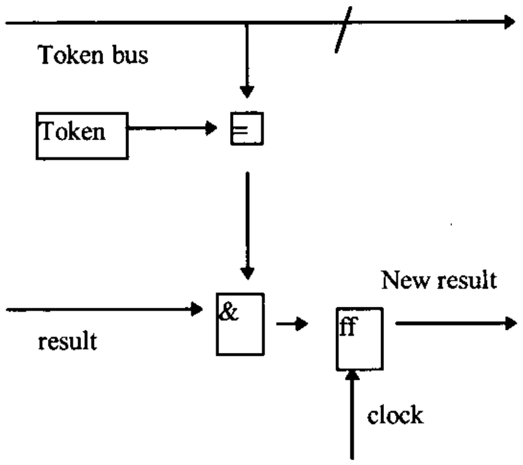

Figure 10 - Token recogniser cell

This logic could be implemented as a parameterised VLSI custom cell where the parameter is the token reference.

2.1.1.2 String concatenation

If two regular expressions E1 and E2 are to be recognised where E2 follows after E1 then this can be implemented as follows. The output result of E1 becomes the input result for E2.

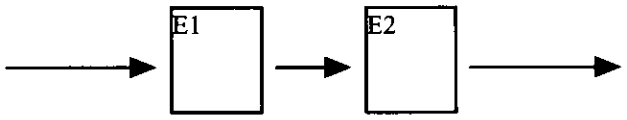

Figure 11 - String Concatenation Cell

If E1 and E2 are composite VLSI cells then the use of cell abutment should automatically provide the required logic and power connections.

2.1.1.3 String union

String union is defined to be the new string formed by selecting either E1 or E2 where they are both regular expressions. This is simply the OR-ing together of the result signals of E1 and E2.

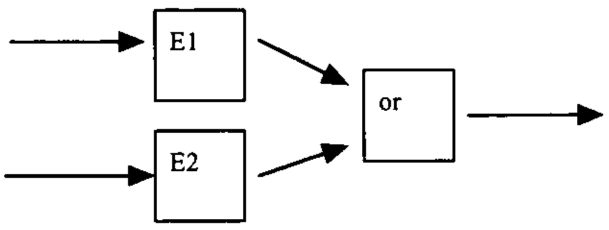

Figure 12 - String union cell

The above diagram shows the logical connections needed to form the "E1 , E2" expression. The diagram shows a layout which would create irregular shaped areas for cell layout. In VLSI semi-custom cell design a linear layout is preferable to allow logic connections by cell abutment. The following layout method (adapted from the paper under discussion) enables a uniform height to be used for the logic cells.

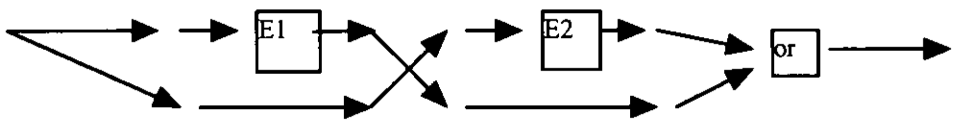

Figure 13 - Linear layout of string union cell

This layout requires an extra wiring channel for E1 and E2, with three extra types of cell. The first cell type splits the result signal channel to form an extra bypass routing channel. The second cell type switches the result and extra wiring channels over. The third cell type or's the result channel and the extra wiring channel to form the new result signal. The logic is equivalent to the previous layout except that the E1 and E2 macro cells have been modified to be capable of being placed in lines.

The paper [Evans 1985] describes the use of two wiring channels and two OR gates for the same end result, thus wasting silicon. The modified cell design, as shown in "Figure 13 - Linear layout of string union cell", is an improvement since the redundant or-gate and wiring channel are eliminated.

2.1.1.4 Repetition

Noting that the regular expressions A* and (A+)? are equivalent then we need only consider mechanisms for A+ and A?. As A? means A is optional, then the new result is formed from the OR-ing of the current result with the result output from A (implying A has been detected).

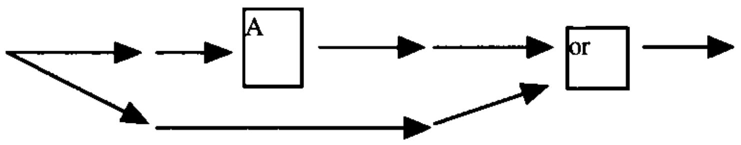

Figure 14 - Optional Cell

Notice that this uses the "split cell" and the "or cell" to select the extra wiring channel with the result channel.

A+ means that A is concatenated at least once. Thus the input for A is either the prefix or the result output from A.

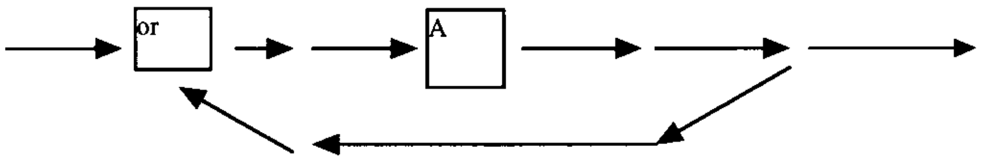

Figure 15 - Repeat 1..n cell

At first glimpse this seems to require two extra types of cells to cope with the reversed direction of use of the extra wiring channel. However, the original split and the or cells can both be used if the extra wiring channel extends the full cell width in both cases. For the split cell there is an internal link joining the extra wiring channel to the result wiring channel. For the or cell the second input to the or gate is taken from extra wiring channel. It is important to note that all cells will need wiring channels for the token bus, clock and result signals. There may be many extra wiring channels required depending on the depth of nesting caused by use of the , * and ? operators.

All cells will need to have an associated parameter to indicate the number of extra wiring channels, with the or, split and switch cells having an extra parameter to indicate which extra channel is being used by the cell.

2.1.2 Logic Synthesis from Regular Expressions

The following set of re-write rules define a language which contains regular expressions, using the "*", "+", "?" and "," notation which also uses brackets. The language describes the definition of a single regular expression.

```
RegRule = LeftName '=' Exp ';' ;
LeftName = identifier ;
Exp = Factor ;
Exp = Exp ',' Factor ;
Factor = Term ;
Factor = Factor Term ;
Term = Primary ;
Term = Primary '+' ;
Term = Primary '*' ;
Term = Primary '?' ;
Primary = identifier ;
Primary = '(' Exp ')' ;
```

The following rules can have semantic actions attached so that the correct logic cells are generated to form the appropriate regular expression recogniser.

Re-write Rule	Semantic Action To Apply
RegRule = LeftName '=' Exp ';'	write code for expression
LeftName = identifier	initialise and note expression name
Exp = Exp ',' Factor	code for A , B
Factor = Factor Term	code for A B
Term = Primary '+'	code for A+
Term = Primary '*'	code for A* (== A+ ?)
Term = Primary '?'	code for A?
Primary = identifier	recognise token

Table 6 - Regular Expression Semantic Actions

These semantic actions will generate code or layout information for a single regular expression.

2.1.3 Critique

The architecture described above will successfully generate hardware to recognise regular expressions. A potential disadvantage of this approach is that each recogniser however can only recognise one expression which is defined at time of manufacture. The use of FPGA's only cuts down the time between design and implementation. Also, the architecture has no mechanism whereby a lexical string (defined as a regular expression) can be remembered and passed to parsing hardware. This is needed when passing on the value of parse tokens such as identifiers or numbers.

A lexical analyser recognises a number of regular expressions and has to indicate which one has been found. This requires the architecture to be able to generate a mechanism to detect which of a number of hardware recognisers is first to detect a token. There is no such mechanism available.

In most computer languages there is a potential clash between the use of keywords such as "begin" or "end" and the form of an identifier. Some languages resolve this by not allowing an identifier to start with a reserved keyword, such as BASIC. The remaining languages allow identifiers to start with a keyword.

The hardware architecture described will always signal that a keyword has been recognised in preference to an identifier. This is a severe constraint on the range of computer languages this system can be used with.

Overall the architecture is not practical for recognising lexical tokens in hardware given the above problems.

2.2 An Architecture to recognise LL(1) Grammars

The paper [Evans 1985] "Architectures for Language Recognition" also describes a hardware architecture which recognises languages defined by LL(1) grammars. This architecture depends on the theorem that, if for each non-terminal Z and terminal a, then there is at most one re-write rule which takes one of three forms :-

- Z = a
- Z = a X
- Z = a X Y

where X, Y, Z are non-terminals and a is a terminal then the grammar is LL(1).

The action to be taken by the three types of rule is shown in the following table.

Rule Format	Current Phase (Test)	Next Phase (Action)
Z = a	rule = Z, token = a	rule' = POP
Z = a X	rule = Z, token = a	rule' = X
Z = a X Y	rule = Z, token = a	rule' = X, PUSH Y

Table 7 - LL(1) Actions

The re-write rules can be placed in a table of terminal versus non-terminal symbols, where each entry will be either a re-rewrite rule (in one of the three forms) or no entry indicating an error. The parsing algorithm works as follows :-

Repeat the following until either both the stack and token input stream are empty or an error is detected.

Using the current combination of rule and token access the table to see which re-write rule is being recognised.

Depending on the table entry take the appropriate action. The POP and PUSH actions refer to the associated stack and rule' represents the next value of rule. In all cases the token from the input stream is consumed.

Parse errors can arise in a number of ways. If there is no entry this represents a parse error. A POP command on an empty stack is also a parse error.

The next section describes the logic to implement each of the three types of cell.

2.2.1 Logic Design

There are three types of cell corresponding to the three types of re-write rule. The three logic cells all have a common sub-unit which is used to recognise which rule and token combination triggers the action for that specific cell. This sub-unit is shown below.

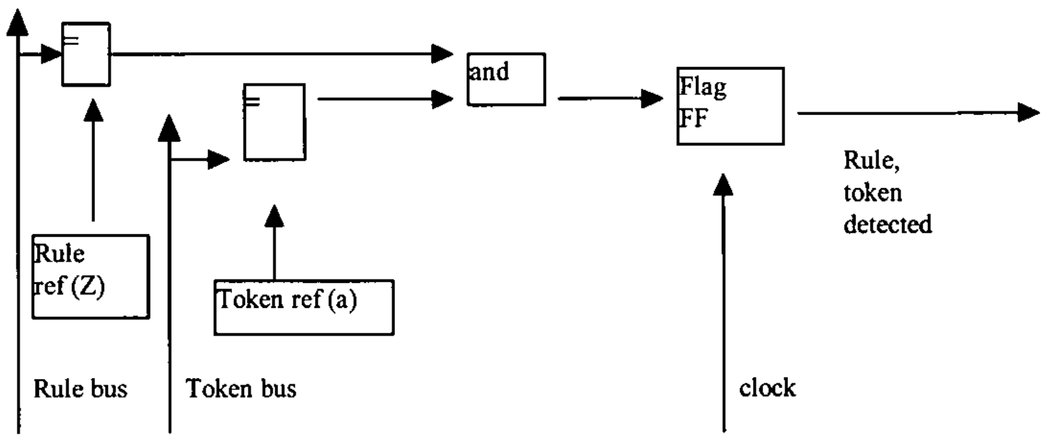

Figure 16 - Rule and Token detection cell

The rule and token detection cell uses a synchronous clock to latch the fact that the rule, token combination has been found for that particular clock cycle. The next clock cycle will clear the flag (unless the same rule, token combination is present). The required action depends on the rule type and is carried out in the next cycle.

The three different action cells corresponding to the three rule types are shown below.

2.2.1.1 Cell for rule "Z = a"

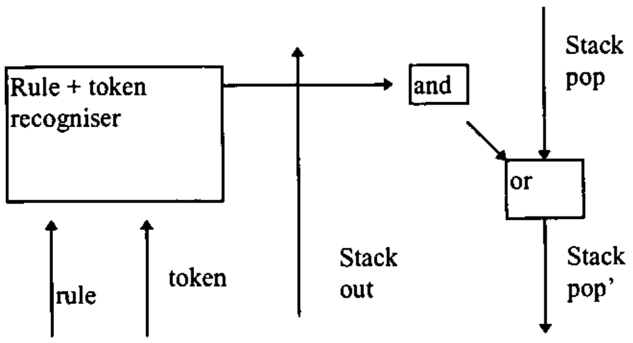

Figure 17 - Cell for rule "Z = a"

For the rule "Z = a", the stack is popped to give the next value for the rule bus. The "stack pop" command is rippled through the cells. An alternative mechanism would be to use a tri-state buffer to "or" the value onto the stack control signal.

2.2.1.2 Cell for rule "Z = a X"

For this cell the action is to place the value of the X non-terminal onto the rule bus for the next clock cycle. The value of X is tri-stated onto the rule bus as shown below. This allows the rule bus to have the value X for the next clock.

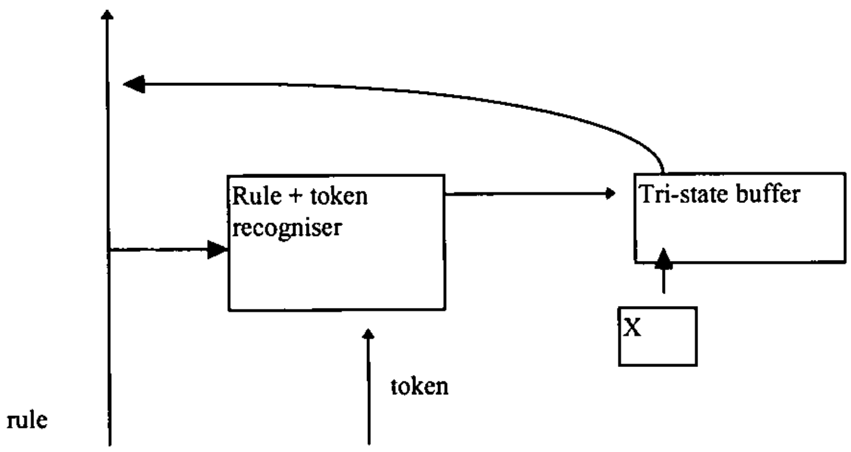

Figure 18 - Cell for rule "Z = a X"

2.2.1.3 Cell for rule "Z = a X Y"

For this cell the action is to place the value of the X non-terminal onto the rule bus and also to push onto the stack the value of the Y non-terminal. This stack value will eventually be popped when a rule of the form "Z = a" is detected.

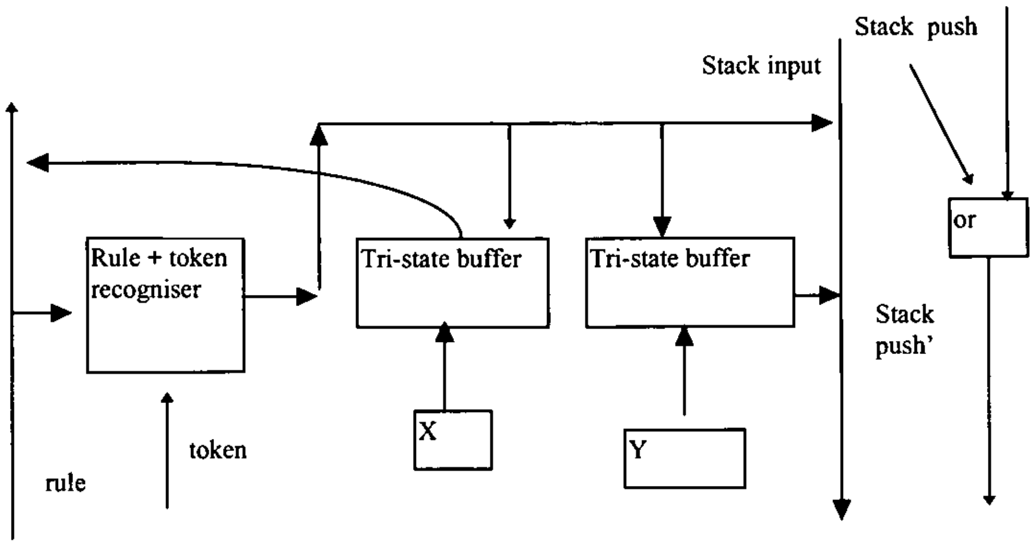

Figure 19 - Cell for rule "Z = a X Y"

2.2.1.4 Combined Cell

The paper combines these three cells into one combined cell. The combined cell uses tri-state buffers extensively instead of the ripple-through logic as shown in the three cells described above. The combined cell uses two extra flags to indicate if non-terminals X, Y are present. Extra logic is used to ensure that the tri-state buffers are correctly activated. The paper [Evans 1985] shows a logic diagram for the combined cell.

2.2.2 Logic Synthesis

The complete LL(1) recogniser is formed by instancing all rules present in the grammar as the appropriate logic cells. The cells are joined together so that the rule, token, stack input, stack output busses and the stack control signals are connected. A stack is also required which has its output connected to the rule bus and input connected to the tri-stated stack input bus from the rule cells.

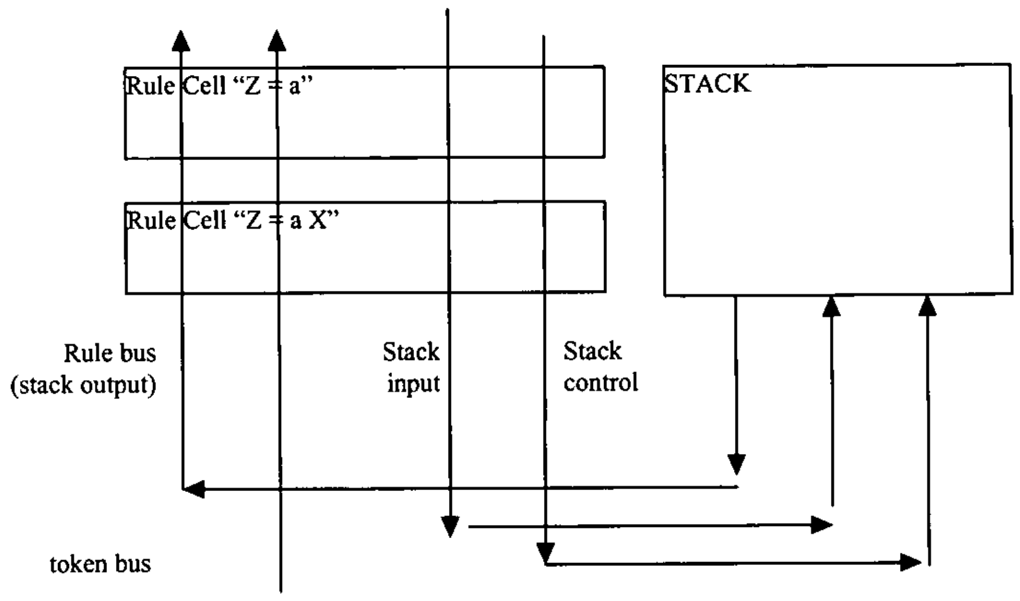

Figure 20 - Complete LL(1) recogniser

2.2.3 Critique

This architecture can be used to generate a large range of LL(1) parsers. Unfortunately most languages are not LL(1). Therefore this architecture will be unable to recognise a large range of computer languages of interest to programmers and computer scientists.

This architecture fixes the implementation of the parsing hardware at time of manufacture, thus preventing rapid modification of a parser. Specialised FPGA's could be designed which contained the three (or single combined cell) as the basic logic element and therefore allow for device re-use.

Also the architecture has no error recovery mechanism to allow the hardware to continue from a parse error. The hardware just reports the first error found. Most users would regard this as a serious failing.

Thirdly, there is no mechanism to use the sequence of rules predicted (LL(1) is a top down parsing technique). This is needed to allow hooks for the semantic actions to be called. Associated with this is the problem that there is no mechanism to pass token strings from the lexical analyser hardware to the semantic hardware at the appropriate parse state.

Overall, the use of the LL(1) algorithm by this architecture is the main stumbling block to its practical use.

2.3 Other Implementations

The paper [Kazuo 1983] "Design and evaluation of parsing chip" describes the implementation of an LL(1) parser for PASCAL. The design used a number of PLA's to implement the parse state engine. The design was not able to perform lexical analysis. This design was successfully validated by being fabricated. The design, however, was not an example of a general purpose architecture able to synthesise general purpose parsers. Therefore this was of limited interest to this thesis.

2.4 Summary

To summarise, the LL(1) algorithm has been the main focus of research into the implementation of parsing techniques as hardware. The next chapters will describe the practical implementation of the LALR(1) algorithm with built-in support for the process of lexical analysis.

3. Instruction Set Design

In the previous chapters, it is shown that the LALR(1) parsing algorithm is based on a finite state machine (with an associated state stack) which only uses shift and reduce actions. This strongly suggests that these two actions could be implemented as instructions for a VLSI RISC processor. This proposed processor would have a very specialised instruction set which could implement both the shift and reduce actions as required by the LALR(1) algorithm. Each computer language would be implemented as a different program to be run by the processor. Also, the parsing of a sentence for a given language would be carried out by the running of the appropriate program on the processor.

It must also be noted that the lexical analysis algorithm also uses a finite state machine (without a state stack) where this machine also has shift and accept actions. These actions could be implemented by extending the processor instruction set with extra instructions to implement the shift and accept actions.

The derivation of an instruction set able to implement both the LALR(1) and lexical analysis algorithms will now be described. The design and simulation of the combined instruction set was carried out using a compiler-compiler. This software was developed as part of the research.

3.1 Parse Instructions

The use of only shift and reduce actions by the LALR(1) algorithm indicates that a minimum of two instructions is required. Accordingly a description of the required behaviour of the shift and reduce actions as instructions will be given. Also a simple error recovery mechanism is described.

3.1.1 States and Instruction Sequences

As stated earlier the LALR(1) parsing algorithm is based on a finite state machine with state stack. The algorithm operates with shifts (or transitions) from one state to another state being triggered by the recognition of the next parse token (or word).

Each state may also recognise one or more rules of the grammar, where the recognition of the rule will again depend on the next parse token. If the current combination of state and token has no defined action then this indicates the detection of a parse error.

One possible representation of a single parse state would be as a large array of entries implemented as a case statement. The current token would be used as the index to select the appropriate entry. Each entry would be one of the shift, reduce or error actions. In this state representation each state would need an entry for all possible tokens. This would create large sparsely populated arrays. This state representation has the benefit of ensuring all state transitions took the same time. This representation was investigated but quickly rejected as being an inefficient use of memory.

An alternative state representation would implement a state as a list of conditionally triggered instructions. Thus, each shift or reduce action is a single conditional instruction which would be triggered by comparing the current token with the token needed to trigger the action. The final instruction for each state would implement the default action to take for that state (usually the error detection action). This representation would be more efficient in memory usage. It's main disadvantage is that each state would take a variable amount of time caused by the need to examine a number of instructions for the appropriate token. This state representation was chosen as the starting point for the design of the instruction set.

3.1.2 Parser Registers

The parser processor will require a number of registers to hold temporary data and to indicate which instruction is being executed. Also the parser processor uses a stack to hold states. The main parser registers are shown in the following table.

Register	Purpose
Program Counter	Points to current Parse Instruction
Instruction Register	Contains current instruction
Top Symbol	Indicates immediate next token symbol in token input queue
Next Symbol	Indicates next symbol after "Top Symbol" (Could be undefined)
Error Flags	Used to record parse errors
State Stack Pointer	Points to Top of State Stack.

Table 8 - Parser Registers

As the individual states each consist of sequences of instructions, then each state can be represented by the address of the first instruction in the state. Thus the state stack can actually store addresses, where each stack entry (representing a state) is actually the address of the first instruction in a state.

The initial instruction set is shown in the following table.

Instruction	Parameter 1	Parameter 2	Parameter 3	Parameter 4
shift	OnToken	ToState	UNUSED	UNUSED
reduce	OnToken	ByRule	RuleToken	RuleCount
shift-reduce	OnToken	ByRule	RuleToken	RuleCount

Table 9 - Initial Parse Instruction Set

The following sections describe the actions of the instructions in further detail.

3.1.3 Shift Action

The shift action represents the transition from one state to another state, where the transition is triggered by the recognition of a parse token. As an instruction this would be shown using a pseudo-assembler notation as :-

```
On <Token> Shift To <State>
```

3.1.3.1 Parameters

The <Token> parameter denotes the triggering token for the shift action.

The <State> parameter denotes the new state to go to. The parameter value is actually the address of the first instruction in the state.

3.1.3.2 Instruction Actions

Firstly, the current token is compared with the <Token> parameter. If there is no match then the processor executes the next instruction, otherwise the following steps are performed.

The <State> parameter is pushed onto the state stack.

The current token (which must be identical to the <Token> parameter) is consumed.

If there is no value held in the next token buffer then the next token must be read. This is the appropriate time to activate the lexical analyser which scans the raw input text to recognise the next token.

3.1.4 Reduce Action

The reduce action represents the occasion when a grammar rule for the language has been recognised. As an instruction this would be shown using a pseudo-assembler notation as :-

```
On <Token> Reduce By <Rule> New Token <Token> Pop <Count>
```

3.1.4.1 Parameters

The <Token> parameter denotes the triggering token for the reduce action.

The <Rule> parameter denotes which rule has been recognised. This is used to indicate which semantic routine should be called. (See the section on "Semantic Interrupts").

The second <Token> parameter denotes the non-terminal token which is defined by the rule.

The <Count> parameter indicates the number of states to pop from the state stack.

3.1.4.2 Instruction Actions

As with the shift instruction, the current token is compared with the <Token> parameter. If no match is detected then the next instruction is executed, otherwise the following steps are performed.

Firstly the <Rule> parameter is used to indicate which semantic actions should be performed. See the following section on "Semantic Interrupts".

Next, the rule's left-hand token symbol (the <Token> parameter) is inserted into the token input queue as the next token to be read.

Finally, a number of states must be "popped" from the state stack. The <Count> parameter is used to determine how many states should be popped from the state stack. For each item on the right-hand of the rule, one pop is done. Thus the <Count> parameter will equal to the number of items on the right-hand of the rule.

3.1.4.3 Semantic Interrupts

The reduce action provides the opportunity or hook to allow semantic actions to be performed. Each individual rule can be regarded as an interrupt generated by the processor. Each interrupt will trigger an action or sequence of actions within additional hardware to implement the semantics of the particular language. Each language has its own associated semantic actions and therefore will require different hardware to implement these actions.

It is important to note that the processor cannot have two semantic actions in progress simultaneously. That is each semantic action must complete before the next one can be started. To guarantee this, the processor should be constrained to only be able to continue with the parsing process when the current semantic interrupt has been completed.

3.1.5 Halting

The LALR(1) parsing process should halt when the unique rule, with the goal symbol as it's left-hand symbol, is recognised or reduced. Therefore, if the goal rule has a fixed value such as 1 then this can be detected by the reduce instruction and the processor halted accordingly. An alternative is to add a halt instruction to the instruction set.

3.1.6 Error Handling

The LALR(1) algorithm is able to detect a parsing error at the first possible opportunity and should halt at that point flagging the fact. However, this mechanism of halting for each error is not acceptable, since each parse would only reveal the first error detected and no more. The processor architecture needed a mechanism whereby a form of error recovery could be attempted.

In most software implementations of the LALR(1) algorithm this is achieved by a combination of popping the state stack and skipping tokens until it is possible to continue the parse process and successfully recognise a rule in the language grammar.

The concept of a special token to assist in error-handling, denoted by $error, was examined. The $error token would be used to denote the presence of a parse error in the token input stream. The detection of a parse error would cause the $error token to be inserted in the token input stream. The error-handling mechanism would then have the task of removing the $error token and a limited number of tokens following the $error token. Also, the $error token could then be used as the trigger for a shift instruction. This would allow a grammar to be augmented with special error rules each of which would contain the $error token followed by one or more tokens as the rule's right-hand. The sequence of tokens following the $error token would allow the parser to re-synchronise itself with that token sequence after an error was detected. The extra error rules would cause some parse states to contain shift instructions which would be triggered by the $error token. Thus the error recovery mechanism would be to pop the state stack until the state at the top of the state stack contained a shift instruction triggered by the $error token. If there were no state found which satisfied that criteria before emptying the state stack then this would imply that no recovery was possible. The detection of the case that no recovery was possible should cause the processor to halt and the reason for the halt to be flagged.

This mechanism will allow both tokens and states to be skipped until a valid rule can be recognised. Additionally the associated semantic hardware needs to be informed that a parse error has been found. This can be solved by having a special rule which has as its left-hand token the $error token and an empty right-hand side.

Thus the instruction set can be extended to allow a reduce instruction to trigger the rule "$error = ".

3.1.7 Default Actions for a Parse State

The default action for a state will normally be the error action which is a modified reduce action. This error action, if reached, must be triggered irrespective of the next token. The set of tokens can be extended by adding the concept of a wild-card token which matches any token and is denoted by $lambda. Thus the default action for a state can be triggered by the $lambda token. This ensures that all parse actions are triggered by a token match, even if the token to match is a wild-card. The $lambda token could be used to trigger either shift or reduce actions.

3.1.8 State Table Minimisation

The use of the $lambda token enables some minimisation of state tables to be achieved.

3.1.8.1 Replacing the default action

If a state contains at least one reduce action then one of these reduce actions can be chosen to replace the default error action. The reduce action could be triggered by many different tokens where each token requires one reduce instruction. Therefore this minimisation replaces all occurrences of the reduce instruction by a single instance of the reduce instruction. This replacement reduce instruction is triggered by the $lambda token and will be the final instruction for a given state.

3.1.8.2 Single Reduce States

Another possible minimisation occurs when a state consists of only one reduce action. That is, the state has no shift actions and contains one reduce action with the default error token. This type of state will only be reached by shift actions contained within other states.

It is possible to eliminate the state and all its instructions by adding an instruction (shift-reduce) which combines the effects of the shift and reduce actions. Those shift instructions which point to the state being eliminated are replaced by the new shift-reduce instruction. This shift-reduce instruction is described in the next section.

3.1.9 Shift-Reduce Action

The shift reduce action represents the combination of a shift action immediately followed by a reduce action. This action is triggered by the recognition of a parse token. As an instruction this would be shown using a pseudo-assembler notation as :-

```
On <Token> Shift Reduce <Rule> New Token <Token> Pop <Count>
```

3.1.9.1 Parameters

The <Token> parameter denotes the triggering token for the shift action.

The <Rule> parameter denotes which rule has been recognised.

The second <Token> parameter denotes the non-terminal token which is the left-hand symbol being defined by the rule.

The <Count> parameter indicates the number of states to pop from the state stack.

3.1.9.2 Instruction Actions

Firstly, the current token is compared with the <Token> parameter. If there is no match then the processor executes the next instruction, otherwise the following steps are performed.

Next, the current token (which must be identical to the <Token> parameter) is consumed.

The <Rule> parameter is used to indicate which semantic actions should be performed. See the section on "Semantic Interrupts".

Next, the rule's left-hand token symbol (given by the <Token> parameter) is inserted into the token input queue as the next token to be read.

Finally, a number of states must be "popped" from the state stack. The <Count> parameter is used to determine how many states should be popped from the state stack. For each item on the right-hand of the rule, one pop is done. The value of the <Count> parameter is one less than the number of items on the rule right-hand side since the shift half of the shift-reduce action would normally push a state onto the state stack. This push to the state stack is not needed.

3.2 Lexer Instructions

A lexical analyser is another example of a finite state machine but does not use an associated state stack. However, the instructions for the lexical state machine will correspond to the parser shift and reduce actions. Also a simple mechanism which can recover from lexical errors such as incorrectly spelt language keywords is described.

3.2.1 States and Instruction Sequences

The algorithm for a lexical analyser is based on a finite state machine which has transitions (or shifts) from state to state which are triggered by the next character present in the input stream. Some states will be "accept" states when a lexical token has been detected. For the lexical algorithm used by this research, it must be noted that each accept state only detects one token. This ensures that a character string can be an example of just one lexical token. Thus for the computer language PASCAL the string "begin" will be regarded as the **begin** keyword and not as a variable identifier.

Each state will have a default action to be performed should there be no shift transition be defined for the next character present. As the lexical algorithm used by this research forces an accept state to have a single token this can be used to determine the default action for a state. Thus the default action for an accept state is the accept action for the token detected, and for a non-accept state it is the error action.

The selected representation for each lexical state follows the representation used within the parser. That is, a lexical state is a list of conditionally triggered instructions. Each shift action is a single conditional instruction triggered by comparing the current character with the character (or range of characters) needed to trigger the instruction. The final instruction for the state implements the default action (accept or error) for that state.

3.2.2 Lexical Analyser Registers

The lexical analyser will also use some of the parser registers. These are shown in the following table.

Register	**Purpose**
Program Counter	Points to current lexical instruction
Instruction Register	Contains current lexical instruction
Top Symbol	Indicates next character in input

Table 10 - Lexical Registers

The lexical analyser must also store the character strings for the previous token, current token and next possible token that it is trying to recognise. These strings are used by the semantic actions associated with the parser. These character strings could be stored within three individual areas of memory where each character string needs a pair of start and end pointers. Both the software and hardware implementations actually used a cyclic buffer to hold the three character strings.

Instruction	Parameter 1	Parameter 2	Parameter 3
shift	MinChar	MaxChar	ToState
accept	AcceptToken	UNUSED	UNUSED
test	TestRoutine	NextState (if passed)	NextState AcceptToken

Table 11 - Initial Lexical Instruction Set

3.2.3 Lexical Shift

The shift action represents the transition from one state to another, where the transition is triggered by the recognition of a lexical character. As an instruction this would be shown using a pseudo-assembler notation as :-

```
Shift <ToState> On Char Range <lo> <hi>
```

3.2.3.1 Parameters

The <lo> and <hi> parameters indicate the contiguous range of characters which will trigger the lexical shift action. <lo> being the minimum and <hi> the maximum.

The <ToState> parameter denotes the new state to go to. The parameter value is actually the address of the first instruction in the state.

3.2.3.2 Instruction Actions

Firstly, the input character buffer is examined to see if it is empty. If it is empty then a request to read the next character in the input stream is made. This causes the processor to wait until a character is supplied.

If the input character buffer is not empty then the current next character (which is read from the buffer) is examined to see if it is in the range given by the <lo> and <hi> parameters. If there is no match then the processor will continue at the next instruction.

If there is a match then the character is added to the buffer holding the next token character string and the program counter with the value of the <ToState> parameter.

3.2.4 Lexical Accept

The lexical accept action is taken when a lexical token has been recognised. This action is triggered as the default action for some states. As an instruction this would be shown using a pseudo-assembler notation as :-

```
Accept <Token>
```

3.2.4.1 Parameters

The <Token> parameter represents the token just recognised and corresponds to the value used by the parser for the language.

3.2.4.2 Instruction Actions

Firstly, the <Token> parameter is stored in the TopSymbol register. This provides a return link to the parser to indicate which token has been found.

Next, the token string buffer pointers are adjusted so that the token strings are updated. Thus, the current token becomes the previous token, the possible next token becomes the current token.

Finally, the Program Counter register is reset to point to the next parse instruction. As the lexical analyser is only entered from a parse shift instruction then the top of the state stack will contain the appropriate address.

3.2.5 Error Handling

It cannot be assumed that the input stream of lexical characters will be free from "spelling mistakes". That is the character input stream could contain sub-sequences of characters which cannot be matched with the definitions of any lexical token. This could occur when attempting to find the longest matching sequence of characters that can be recognised as a token.

For example :-

In PASCAL a real number will contain a decimal point ('.') and an integer could be followed by the sub-range token '..'. Thus the sequence '12..' could be a miss-spelt decimal number or it could be twelve ('12') followed by the sub-range token ('..'). If the lexical analyser has the strategy of recognising the shortest sequence it will easily recognise '12' followed by '..' but will find it hard to recognise a decimal number, since the digits before the decimal point will be regarded as an integer and the token sent to the parser. If the lexical analyser has the strategy of recognising the longest sequence it will attempt to recognise '12..' as starting a decimal number of the form '12.D' (where D is a non-empty sequence of digits) and regard the next '.' as an error.

This type of error will be detected when a lexical state has no transition defined for the next input character and the state does not default to recognising a token. Thus the lexical parsing algorithm implicitly uses the strategy of trying to recognise the longest string.

Therefore a mechanism to detect and correct both genuine errors and errors caused by the maximal string strategy is needed. This mechanism will be triggered as the default action for those states which do not accept a token.

Noting that,

- each state has an associated default token (which could be the error token)
- each character of the string corresponds to a state (the first character maps to the initial lexical state)

then each character of the string will have a matching token. The sequence of tokens defined will be a mixture of legal and error tokens, where the last token should be an error token. For the example given above (of '12..') this would be the token sequence "integer integer error error". (This is assuming that a decimal number cannot end with a decimal point). Using the maximal string strategy, then the next token should be the maximal sub-sequence of characters which has a legal token corresponding to the last character. Thus, for the example given, the next token will be '12' which is the maximal string forming a legal token (an integer). The surplus characters following the maximal string will form the start point for the next token to be recognised. If there is no maximal string this implies that the first character seen cannot start any token and that it should be ignored (after flagging the fact by means of an error interrupt). The remaining characters should be used as the start point for the next token. Also, the lexical analyser should be restarted from its initial state.

Thus the error action would become an instruction having the form,

Error <ErrorRoutine>

The instruction actions would be as outlined above.

Also each lexical state would need a new instruction to note the default token (possibly the error token) for that state. As an instruction this would be shown using the pseudo-assembler notation as :-

```
Default <Token>
```

where <Token> would indicate the value of the default token.

This instruction would push the <Token> parameter onto a stack which would be initialised to be empty whenever the lexical analyser was started either from the parser or by the **Default** instruction.

The combination of Error and Default instructions meant that the Accept instruction was redundant. Therefore all occurrences of the accept instruction were replaced by the error instruction which was renamed to be accept.

3.2.6 Test Action

Many computer languages allow identifiers (or names) to belong to different classes (or types) such as procedure identifiers, record identifiers or variable identifiers. These identifiers could be different types of token yet have the same lexical definition. Therefore a mechanism was required which could be used to split strings of the same format into different tokens. It would be a special variant of the shift instruction. This test instruction would raise an interrupt routine (able to read the current character sequence) and return a logical value whether to shift or not to another lexical state.

Also most languages allow comments (which may be nested) which are not tokens but must be allowed as legal noise or whitespace. A mechanism to check for nesting levels and allow the external semantic engine to clear the whitespace was required. To satisfy both requirements the test instruction was implemented. As an instruction this would be shown using a pseudo-assembler notation as :-

```
Test <Routine> Goto <GotoState>
```

3.2.6.1 Parameters

The <Routine> parameter indicates which routine is to be called.

The <GotoState> parameter indicates which state to go to depending on the status returned by the interrupt routine indicated by the <Routine> parameter.

3.2.6.2 Instruction Actions

First, the <Routine> parameter is used to generate an interrupt to the required semantic routine. This routine may need to read the value of the string which is the possible next token. This will allow the routine to match the token string (a possible identifier) with other known token strings. The routine could return a status value formed from two flags. The subsequent actions depend on the values of the flags. One flag indicates if the current token string should be reset and the lexer reset to its initial state. This flag takes precedence over the second flag. The second flag indicates if the test was successful and the <GotoState> parameter can be used to indicate the next lexical state. If the test was not successful then the next instruction in the state is performed.

3.3 Review of Initial Instruction Set

The initial instruction set as described in the previous sections used an instruction size of 32 bits. Address 0 of the instruction address space was used to hold the address of the first lexical instruction (all lexical instructions followed the parser instructions) and also the address of the first parse state.

The initialisation of the combined parser and lexer processor read address 0 to stack the first parser address and to start execution at the first lexical state. The initial instruction set design required every instance of the lexical test and parse shift instructions to need the address of the first lexical state. Storing it at address 0 reduced the number of parameters needed for those instruction.

The following table shows the initial instruction set and indicates the instruction and parameter locations as bit string locations in the 32-bit instruction.

Inst'n	Inst'n <31:29>	Param 1 <28:16>	Param 2 <12:0>	Param 3 <15:8>	Param 4 <7:0>
Shift	0	Token	State		
Call	1	Token	Routine		
ShiftCall	2	Token	Routine		
Reduce	3	Left Symbol	Count		
LexShift	4	State		LoChar	HiChar
Test	5	State	Routine		
Default	6		State		
Accept	7		Routine		

Table 12 - Initial Combined Instruction Set

The original decision to use both a 32-bit instruction and 8-bit characters constrained the lexshift instruction to have a 13-bit address space.

The introduction of the 16-bit UNICODE standard for characters (an extension of the ASCII code) and also noting that some of the combined parse and lexical tables for languages could be larger than 8192 words (13-bit address range) forced the development of a new version of the instruction set which would avoid these limitations. This new version is describe in the next sections of this chapter.

3.4 Micro-Instructions

The similarities in behaviour of the lexical and parser instructions from the original design led to the concept of lexical and parser actions as macros. The new instruction set would therefore consist of a number of micro-instructions that could be combined to form the required actions for the parser and lexer.

Another design goal of the new instruction set was to have a larger addressing range (bigger than 8192) and also to be able to use 8-bit or 16-bit characters. These goals implied that the instruction parameter size should be increased.

The mixture of parser and lexer actions as macros of micro-instructions having variable parameter width forced the adoption of the following instruction architecture.

Each micro-instruction would consist of two parts, 3-bits describing the instruction and n-bits for the parameter. Each address location would consist of 4 micro-instructions, a phase 0, phase 1, phase 2 and phase 3 micro-instruction. This would make each lexical and parser action into a Very Long Instruction Word (VLIW) format.

The 3-bit instruction would have different actions depending on the phase of the instruction. Thus the complete instruction set would actually comprise of 32 micro-instructions, 8 for each phase. The final micro-instruction set has some instructions which are identical in behaviour, but are in different phases. These instructions may not have identical values.

The n-bit parameter would have to represent a state address (both parser and lexical), a parser token, a parser semantic action, a lexical test routine and also a lexical character. The value of n for the software emulation of the processor was set to be 13. The actual hardware implementation used a bit-slice architecture, where the parameters could be multiples of 8 bits. Thus the software emulation (using 13-bits) would require two bit-slices to allow 16-bit parameters.

3.4.1 Registers and Flags

The processor architecture uses a number of registers and flags to hold information about the progress of the parse and lexical state machines.

3.4.1.1 Program Counter

The current address is held in two registers. PC indicates the address of the current VLIW instruction and Phase indicates which of the 4 micro-instructions is being executed. P1 is used to hold the parameter value of the current micro-instruction pointed to by PC and Phase.

3.4.1.2 Token Queue

The token queue is used to store the parse token values. It is implemented as two registers, TopSymbol and LookAheadSymbol and an associated flag ValidQueue.

TopSymbol is used to store the value of the next parse token. This is either read from the input stream of parse tokens, the LookAheadSymbol register or the left-hand symbol of a parse rule recognised by a parse reduce action. It is also available for use by the lexical instructions to return the next parse token detected by the lexical machine.

LookAheadSymbol is also used to store the value of the next parse token. It is written to (from the TopSymbol) when a parse reduce inserts the rule left-hand symbol into the head of the parse token queue. The LookAheadSymbol register value (if it is valid) is returned to the TopSymbol register after a parse shift instruction has been executed.

The ValidQueue flag indicates that the LookAheadSymbol register holds a valid token value.

3.4.1.3 Lexical Buffer

The lexical buffer is used to store the lexical values (as character strings) of the previous token and current token recognised. It also holds the lexical characters which should form the next token.

The buffer is implemented in software as a cyclic buffer. It comprises of an area of memory and four pairs of registers. Each pair of registers acts as pointers to the start and end of the token character string. The fourth pair of registers is used when outputting one of the token strings for use by a semantic or lexical test action. The use of a cyclic buffer enables the memory space to be re-used but requires a decision on the appropriate size of memory to hold all the token strings. Each memory address location must be able to store a lexical character (either 8-bit ASCII or 16-bit

UNICODE). Also each register pair represents a start and end address (of the buffer memory) and so must be compatible with the buffer memory address range.

The three main register pairs are TokenBuffer, TokenIs and TokenWas.

- TokenWas points to the previous token string.
- TokenIs points to the current token string.
- TokenBuffer points to the string which may form the next token.

Finally the TokenRam register pair is used to point to one of the TokenWas, TokenIs or TokenBuffer strings as required.

3.4.1.4 Stacks

The software implementation of the processor architecture also uses two stacks. These being the parse state stack and the token stack. Both stacks require an area of memory and a stack pointer.

The parse stack holds the parse state values which are implemented as instruction addresses. Thus each stack location must be able to store a processor instruction address. The maximum stack size required depends on the language grammar and the source input.

The token stack holds the token values for return by the lexical machine. Thus each stack location must be able to store a parse token value. The maximal size of this stack is identical to the size of the longest token string. If the language grammar allows large size comments then a large token state buffer is required.

3.4.1.5 Flags

Most of the flags used by the processor architecture are used to report on the status of the parser/lexical processes. These types of flags are initialised as false and may only be set to true.

The EOIFound flag is used to indicate that the end of input token has been seen. This token is akin to a full-stop in an English sentence.

The ParseDone flag indicates that the parsing process has finished. It could be caused by a successful parse of the input or by a fatal (and unrecoverable) error being detected.

Also, a number of flags are used to indicate warnings and errors detected in the running of the parse and lexical processes.

A warning flag indicates a fault that can be recovered from. These are :-

- ParseSyntax - a parse syntax error
- ParseSemantic - a semantic action error
- LexSyntax - a lexical syntax error

An error flag indicates that the fault cannot be recovered from. These are :-

- SourceExhausted - attempting to read the input stream after the end of input token was seen.
- NoErrorHandler - no error handler rule has been specified
- BufferOverflow - the lexical buffer has overflowed (caused by a very long token)
- StackOverflow - the state or token stack has overflowed
- StackUnderflow - attempting to pop from an empty stack
- IllegalInstruction - attempting to execute one of the undefined micro-instructions

If any error flag is set then this will cause the processor to halt.

The remaining flags, which can be both set and reset, are the SynchroniseMode, SysResult and SysNullToken flags.

The SynchroniseMode flag is set to show that a parse error has been detected and cleared when a non-error rule parse reduce has been performed.

The SysResult flag is cleared before the start of each parse semantic action (or lexical test). It is set or cleared by the action routine to indicate success or failure. This is then used to either set other error flags or to select the next instruction address.

The SysNullToken flag is cleared before the start of each lexical test. It is set by the test routine to indicate that the possible next lexical token is a comment and can be ignored. It causes the possible token buffer to be emptied and restarted with the next character in the character input stream.

3.4.2 The Micro-Instruction Set

The following table lists the micro-instructions for each phase, giving the instruction name, code and parameter. Each phase can potentially have eight micro-instructions defined. Only phase one defines all eight micro-instructions.

Name	Phase	Code <15:13>	Parameter Usage <12:0>
ifequal	0	0	token
lambda	0	1	not used
illegal	0	2	not used
nomatch	0	3	not used
lexchar	0	4	character
lexerror	0	5	token
lexeoi	0	6	token
lexaccept	0	7	token
shift	1	0	state
shift-reduce	1	1	rule
reduce	1	2	rule
lambda	1	3	not used
lexshift	1	4	character
lerror	1	5	rule
lextest	1	6	rule
perror	1	7	rule
lpush	2	0	token
assign	2	1	token
push	2	2	token
loadchar	2	3	not used
halt	2	4	not used
lambda	2	5	not used
illegal	2	6	not used
illegal	2	7	not used
goto	3	0	state
pop	3	1	not used
readstack	3	2	not used
illegal	3	3	not used
illegal	3	4	not used
illegal	3	5	not used
illegal	3	6	not used
illegal	3	7	not used

Table 13 - Micro-Instructions

The table indicates that some instructions have the same name despite having different values and different phases, in particular the lambda (or no-op) instruction. The use of the same name indicates that the behaviour of the instructions is identical. The following sub-sections describe each named micro-instruction detailing its purpose, parameter and actions performed. Each instruction action is described using a PASCAL-like notation.

3.4.2.1 ifequal

Parameter Token
Purpose used to check if the current top symbol triggers a parse shift or reduce action.
Action

```
if (P1 = TopSymbol) then
begin
  Phase := Phase + 1;
end
else
begin
  PC := PC + 1;
  Phase := 0;
end;
```

3.4.2.2 lambda

Parameter unused
Purpose no-op used to jump to next phase
Action

```
Phase := Phase + 1;
```

3.4.2.3 illegal

Parameter unused
Purpose undefined instruction
Action

```
IllegalInstructionFlag := true;
```

This will cause the processor to halt.

3.4.2.4 nomatch

Parameter unused
Purpose no-op jump to next instruction address
Action

```
PC := PC + 1;
Phase := 0;
```

3.4.2.5 lexchar

Parameter Character
Purpose test if the character in the input is greater than or equal to the parameter character.
Action

```
if (P1 <= TopSymbol) then
begin
  Phase := Phase + 1;
end
else
begin
  PC := PC + 1;
  Phase := 0;
end;
```

3.4.2.6 lexerror

Parameter Token
Purpose Use the token stack to find the longest token possible from the lexical text just read.
Action

```
repeat
  TopSymbol := POP_TOKEN_STACK
  TokenIsEnd:= Dec13(TokenIsEnd);
until (TokenIsStart = TokenIsEnd) or (TopSymbol <> 0);

if (TopSymbol = 0) then
begin
  Phase := Phase + 1;
end
else
begin
  TokenBufferStart := TokenIsEnd;
  PC := StateStack[StateSP];
  Phase := 0;
end;
```

3.4.2.7 lexeoi

Parameter Token
Purpose Note that the token denoting the end of lexical input has been seen.
Action

```
EOIFoundFlag := true;
TopSymbol := P1;
TokenBufferStart := TokenIsEnd;
PC := StateStack[StateSP];
Phase := 0;
```

3.4.2.8 lexaccept

Parameter Token
Purpose Note that a valid token has been seen.
Action

```
TopSymbol := P1;
TokenBufferStart := TokenIsEnd;
PC := StateStack[StateSP];
Phase := 0;
```

3.4.2.9 shift

Parameter State
Purpose Note the parse state to be shifted to, depending on if the token queue is empty then goto that state otherwise next phase (ready to start the lexical machine).
Action

```
PUSH_STATE_STACK(P1)
if ValidQueueFlag or EOIFoundFlag then
begin
  TopSymbol := LookAheadSymbol;
  ValidQueueFlag := false;
  PC := StateStack[StateSP];
  Phase := 0;
end
else
begin
  { get the lexical token }
  TokenWasStart := TokenIsStart;
```

```
 TokenWasEnd := TokenIsEnd;
 Phase := Phase + 1;
end;
```

3.4.2.10 shift-reduce

Parameter Rule
Purpose Call up the rule specified as parameter to perform the associated semantic actions. (Remember to use the current token string)
Action

```
SynchroniseModeFlag := false;
TheAction := cActionA;
SysNullToken := false;
SysResult := SemanticAction(TheParsePtr,TheSyntaxPtr,P1);
if not SysResult then
begin
  ParseSemanticFlag := true;
end;
Phase := Phase + 1;
```

3.4.2.11 reduce

Parameter Rule
Purpose Call up the rule specified as parameter to perform the associated semantic actions. (Remember to use the previous token string)
Action

```
LookAheadSymbol := TopSymbol;
ValidQueueFlag := true;
SynchroniseModeFlag := false;
TheAction := cActionB;
SysNullToken := false;
SysResult := SemanticAction(TheParsePtr,TheSyntaxPtr,P1);
if not SysResult then
begin
  ParseSemanticFlag := true;
end;
Phase := Phase + 1;
```

3.4.2.12 lexshift

Parameter Character
Purpose Check that the current lexical character is less than or equal to the expected character.
Action

```
if (TopSymbol <= P1) then
begin
  TokenIsEnd := Inc13(TokenIsEnd);
  Phase := Phase + 1;
end
else
begin
  PC := PC + 1;
  Phase := 0;
end;
```

3.4.2.13 lerror

Parameter Rule
Purpose Call up a special semantic action to indicate that a lexical error has been detected.
Action

```
TheAction := cActionD;
SysNullToken := false;
SysResult := SemanticAction(TheParsePtr,TheSyntaxPtr,P1);
LexSyntaxFlag := true;
TokenBufferStart := Inc13(TokenBufferStart);
Phase := Phase + 1;
```

3.4.2.14 lextest

Parameter Rule
Purpose Call up a lexical test routine to check on the possible token string.
Action

```
TheAction := cActionC;
SysNullToken := false;
SysResult := SemanticAction(TheParsePtr,TheSyntaxPtr,P1);
if SysNullToken then
begin
 TokenBufferStart := TokenIsEnd;
 PC := PC + 1;
 Phase := Phase + 1;
end
else if SysResult then
begin
 Phase := Phase + 1;
end
else
begin
 PC := PC + 1;
 Phase := 0;
end;
```

3.4.2.15 perror

Parameter Rule
Purpose Depending on if the end of input has been seen or attempting to re-synchronise caused by previous errors then possible call up a special semantic action to indicate that a new parse error has been seen.
Action

```
if EOIFoundFlag then
begin
 SourceExhaustedFlag := true;
end
else if SynchroniseModeFlag then
begin
 ValidQueueFlag := false;
 Phase := Phase + 1;
end
else
begin
 LookAheadSymbol := TopSymbol;
 ValidQueueFlag := true;
 SynchroniseModeFlag := true;
 ParseSyntaxFlag := true;
 TheAction := cActionA;
 SysNullToken := false;
 SysResult:=SemanticAction(TheParsePtr,TheSyntaxPtr,P1);
 if not SysResult then
 begin
```

```
  ParseSemanticFlag := true;
 end;
end;
Phase := Phase + 1;
```

3.4.2.16 lpush

Parameter Token
Purpose Initialise the lexical engine, clear the TokenIs string but point to the first possible character for the token string.
Action

```
PUSH_TOKEN_STACK(P1);
TokenIsStart := TokenBufferStart;
TokenIsEnd := TokenBufferStart;
Phase := Phase + 1;
```

3.4.2.17 assign

Parameter Token
Purpose After a parse rule has been recognised then note the left-hand token of the rule. If attempting to re-synchronise input after a parse syntax error then pop the state stack to find a state which has a shift on the $error (=0) token.
Action

```
if SynchroniseModeFlag then
begin
 { find an error handler }
 StateSP := Inc13(StateSP);
 repeat
  PC := POP_STATE_STACK;
  Phase := 0;
  TopSymbol := ($1fff and ReadTable(4*PC+Phase));
 until EMPTY_STATE_STACK or (TopSymbol = 0);
 if EMPTY_STATE_STACK then NoErrorHandlerFlag := true;
end
else
begin
 TopSymbol := P1;
 Phase := Phase + 1;
end;
```

3.4.2.18 push

Parameter Token
Purpose Push the specified token value onto the token stack.
Action

```
PUSH_TOKEN_STACK(P1);
Phase := Phase + 1;
```

3.4.2.19 loadchar

Parameter not used
Purpose Read the next character from the lexical input stream (only if the buffer is empty).
Action

```
TokenRamStart := TokenIsEnd;
TokenRamEnd := TokenBufferEnd;
if (TokenRamStart = TokenRamEnd) then
begin
 { Read next char into buffer }
```

```
  LexRam[TokenRamStart] := READ_NEXT_CHAR;
  TokenBufferEnd := Inc13(TokenBufferEnd);
  if (TokenBufferStart = TokenBufferEnd) then
  begin
    BufferOverflowFlag := true;
  end;
end;
TopSymbol := LexRam[TokenRamStart];

PC := PC + 1;
Phase := 0;
```

3.4.2.20 halt

Parameter not used
Purpose Halt the processor
Action

```
ParseDoneFlag := true;
Phase := 0;
```

3.4.2.21 goto

Parameter State
Purpose Goto the specified instruction address.
Action

```
PC := P1;
Phase := 0;
```

3.4.2.22 pop

Parameter not used
Purpose pop a single value from the state stack.
Action

```
if EMPTY_STATE_STACK then StackUnderflowFlag := true;
StateSP := Dec13(StateSP);
PC := PC + 1;
Phase := 0;
```

3.4.2.23 readstack

Parameter not used
Purpose goto the address specified by the top of the state stack.
Action

```
PC := StateStack[StateSP];
Phase := 0;
```

3.5 Combined Macros

The individual micro-instructions for the new instruction set can be combined to form macro-instructions. These macro-instructions implement the parser and lexer instructions of the initial instruction set. As mentioned earlier these macros could be regarded as a form of Very Long Instruction Word (VLIW).

3.5.1 State Table Macro

The state table macro or layout for the combined parse and lex states has the following structure (using the regular expression notation for grammars).

Table = Initialisation ParseState+ ParsePop LexState+

The Initialisation, ParseState, ParsePop and LexState macro entities are defined in the next sections. The above indicates that there must be an initialisation macro, at least one parse state, that there must be a parse pop section and at least one lex state. The processor does not verify that a language table has the correct structure. The correct table structure should always be generated by the associated compiler-compiler system.

3.5.2 Initialisation

The processor starts with all registers and flags initialised to zero or false. For the parser machine to be initialised it needs a token value to be input. This is provided by the macro placed at address 0.

Phase 0 = lambda0

Phase 1 = shift <parse state0>

Phase 2 = lpush <lexstate0 default>

Phase 3 = goto <lexstate0>

3.5.3 ParseState

The compiler-compiler used with the processor generates the processor code so that the parse states start at address 1. Each parse state will have the following structure (using the regular expression notation for grammars).

ParseState = (Shift, Reduce, ShiftReduce)* Default

The Shift, Reduce, ShiftReduce and Default entities mentioned correspond to the ParseShift, ParseReduce, ParseShiftReduce and ParseDefault macros.

3.5.4 ParsePop

This is used to store the popping from the parse state stack which is required by the Parse Reduce action. It has the following structure.

ParsePop = Pop+ ReadStack

Pop is a macro such that :-

Phase 0 = lambda

Phase 1 = lambda

Phase 2 = lambda

Phase 3 = pop

ReadStack is a macro such that :-

Phase 0 = lambda

Phase 1 = lambda

Phase 2 = lambda

Phase 3 = readstack

The number of pop macros is given by the number of tokens in the rule with the largest number of tokens in the right-hand side of the grammar rule.

3.5.5 LexState

Each lexical state will have the following structure (using the regular expression notation for grammars).

LexState = TestArc* (LexLoadChar CharArc+)? LexAccept

3.5.6 Parse Shift

The ParseShift macro splits down to

Phase 0 = IfEqual <Token>

Phase 1 = Shift <ParseState>

Phase 2 = lpush <lexstate 0 default token>

Phase 3 = goto <lexstate 0>

3.5.7 Parse Reduce

The ParseReduce macro has three variants depending on the rule to be reduced.

If the rule is the goal rule then

    Phase 0 = IfEqual <Token>
    Phase 1 = lambda
    Phase 2 = halt
    Phase 3 = readstack

else if the rule has no tokens on its right-hand side then

    Phase 0 = IfEqual <Token>
    Phase 1 = reduce <Rule>
    Phase 2 = assign <rule left-hand token>
    Phase 3 = readstack

otherwise

    Phase 0 = ifequal <Token>
    Phase 1 = reduce <rule>
    Phase 2 = assign <rule left-hand symbol>
    Phase 3 = goto <address of pop = right-hand rule count>

3.5.8 Parse Shift-Reduce

The ParseShiftReduce macro has three variants depending on the rule to be reduced.

If the rule is the goal rule then

    Phase 0 = ifequal <Token>
    Phase 1 = lambda
    Phase 2 = halt
    Phase 3 = readstack

else if the rule has no tokens on its right-hand side then

    Phase 0 = ifequal <Token>
    Phase 1 = shiftreduce <Rule>
    Phase 2 = assign <rule left-hand token>

Phase 3 = readstack

otherwise

Phase 0 = ifequal <Token>

Phase 1 = shiftreduce <rule>

Phase 2 = assign <rule left-hand symbol>

Phase 3 = goto <address of pop = right-hand rule count>

3.5.9 Parse Default Reduce

The ParseDefaultReduce macro has four variants depending on the default rule being recognised and the number of tokens in the rule right-hand side.

If the rule is the error rule then

Phase 0 = lambda

Phase 1 = perror <Error Rule>

Phase 2 = assign <Error Token>

Phase 3 = readstack

else if the rule is the goal rule then

Phase 0 = lambda

Phase 1 = lambda

Phase 2 = halt

Phase 3 = readstack

else if the rule has no tokens on its right-hand side then

Phase 0 = lambda

Phase 1 = reduce <Rule>

Phase 2 = assign <rule left-hand token>

Phase 3 = readstack

otherwise

Phase 0 = lambda

Phase 1 = reduce <rule>

Phase 2 = assign <rule left-hand symbol>

Phase 3 = goto <address of pop = right-hand rule count>

3.5.10 Lex Test

Each instance of the LexTest macro takes up two address locations.

Address0:

Phase 0 = lambda

Phase 1 = lextest <test routine>

Phase 2 = push <next lex state default token>

Phase 3 = goto <next lex state>

Address1:

Phase 0 = nomatch

Phase 1 = lambda

Phase 2 = lpush <lex state 0 default token>

Phase 3 = goto <lex state 0>

3.5.11 Lex Load Char

The LexLoadChar macro splits down to

Phase 0 = lambda

Phase 1 = lambda

Phase 2 = loadchar

Phase 3 = goto <next address = current address + 1>

3.5.12 Lex Shift

The LexShift macro splits down to

Phase 0 = lexchar <lo character>

Phase 1 = lexshift <hi character>

Phase 2 = push <next lex state default token>

Phase 3 = goto <next lex state>

3.5.13 Lex Accept

The LexAccept macro has three variants depending on the default token being recognised.

If the recognised token is the error token (indicating a lexical syntax error) then

Phase 0 = lexerror <ErrorToken>

Phase 1 = lerror <Error Rule>

Phase 2 = lpush <lex state 0 default token>

Phase 3 = goto <lex state 0>

else if the token is the end of input token then

Phase 0 = lexeoi <EOI Token>

Phase 1 = lerror <error rule>

Phase 2 = lpush <lex state 0 default token>

Phase 3 = goto <lex state 0>

otherwise

Phase 0 = lexaccept <token>

Phase 1 = lerror <error rule>

Phase 2 = lpush <lex state 0 default token>

Phase 3 = goto <lex state 0>

3.5.14 State Size Reduction

It can be observed that some language grammars have duplicated instructions in some parse states and also some lex states. Therefore, one further optimisation is to merge the overlap into a new state, removing the overlap from the two original states. The two original states would then each terminate with a new macro, the Continue macro. This would be implemented as,

Phase 0 = lambda

Phase 1 = lambda

Phase 2 = lambda

Phase 3 = goto <new merged state>

For a pair of parse states the merged state must contain the ParseDefault macro.

For a pair of lex states the merged state must contain the LexAccept macro.

This optimisation has been included in the compiler-compiler software suite which generates the instruction tables for a language.

3.6 Sample Language Table Sizes

The following table gives examples of the sizes of parse tables generated by the compiler-compiler using the original and new instruction sets. For a list of the original parse instructions refer to "Table 9 - Initial Parse Instruction Set", and for a list of the original lexer instructions refer to "Table 11 - Initial Lexical Instruction Set". For a list of the final instruction set refer to "Table 13 - Micro-Instructions". The following table also compares the size of parse tables for a number of computer language grammars. Also included is a comparison of the count of instructions executed to read the language grammars.

It can be seen that the new micro-instruction count is less than double the old instruction count. This should not imply that the new instruction set will have longer execution times, since each new micro-instruction is simpler (and presumably faster) than the old instruction.

Note also that the count of phase0 micro-instructions executed is consistently less than the number of instructions executed from the old instruction set. This may be caused by a different ordering of the triggers for parse and lex shift actions between the tables generated for the two instruction sets.

Language	Old Table Max Address	Old Instruction Count	New Table Max Address	New Instruction Count	New Phase0 Count
ACE	818	37509	701	60597	29723
BASIC	1260	30658	1105	48912	23723
M2	4177	107536	3326	177459	84728
M2V	3915	97245	3134	158835	76489
PCPASCAL	3924	100399	3470	169863	80133

Table 14 - Comparison of Table Sizes

The following table shows some example counts of instruction executed for source text written in a range of languages, where :-

- ACE is a simple BASIC-like language.
- BASIC is a grammar defining a variant of the original BASIC language.
- M2 and M2V are both language grammars for MODULA-2. M2V is a grammar which was defined for use on the DEC VAX/VMS operating system.
- PCPASCAL is a grammar derived from the PASCAL definition used for the Borland Turbo Pascal compiler.

Language	Input	Instruction Count (Old Instruction Set)	Instruction Count (New Instruction Set)	Phase0 Count (New Instruction Set)
ACE	bad	248	5910	2771
ACE	bad1	364	5508	2566
ACE	jdm	3167	5964	2789
ACE	test	6644	12348	5820
ACE	test1	7366	13771	5456
BASIC	bad	1019	1885	935
BASIC	test	788	1476	704
M2	deb	6812	13689	6202
M2	example	7450	15262	6708
M2	example1	9656	23720	9584
M2	example2	3537	6690	3141
M2V	deb	6718	13521	6114
M2V	example	7357	15096	6621
M2V	example1	9630	23688	9564
M2V	example2	3484	6604	3094
PCPASCAL	test	12364	25007	11352

Table 15 - Comparison of Parse Input

The examples given within the table show that the new instruction set roughly doubles the number of instructions executed compared with the original instruction set. However due to the simpler actions for the new micro-instructions, the execution time of each new instruction should be less than that of the old instruction.

4. Hardware Design

The instruction set architecture, which was designed to implement the combined LALR(1) and lexical analyser algorithms, did not impose any major constraints on the physical implementation of the processor. The software implementation of the processor, used within the compiler-compiler system suggested the main functional blocks to be implemented as hardware. These functional blocks are shown in the following diagram.

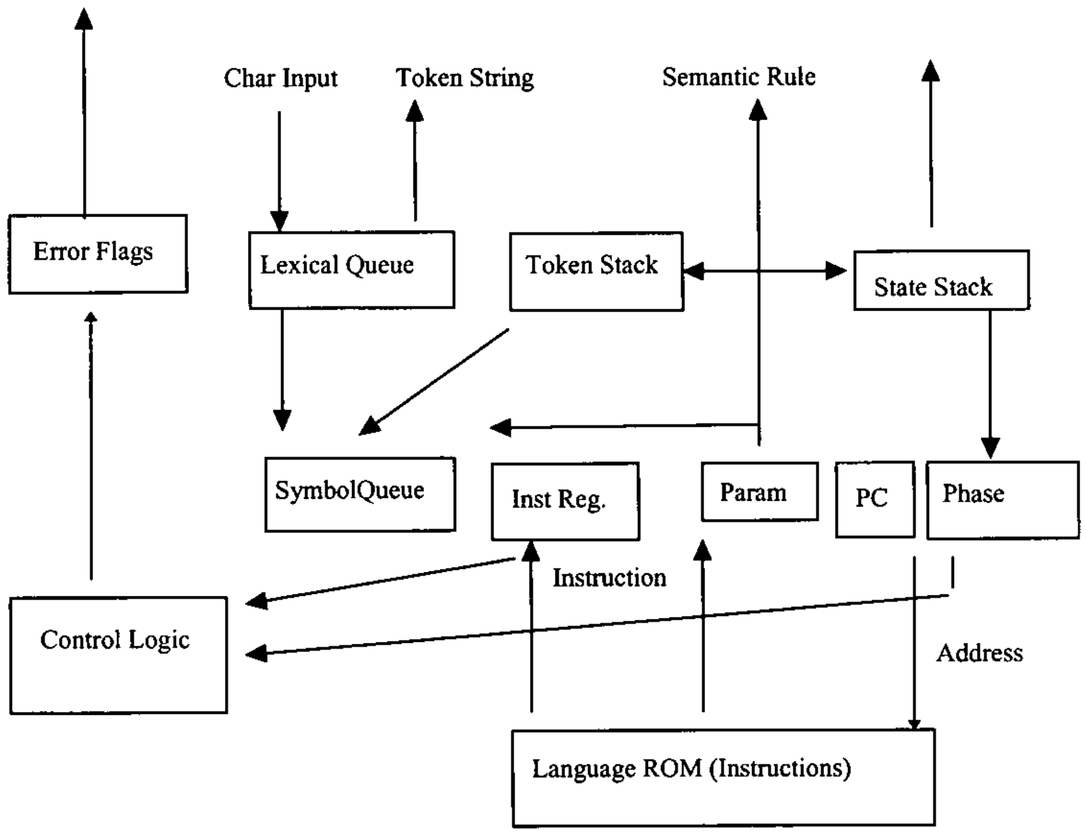

Figure 21 - Processor Functional Blocks

The PASCAL source code for the software emulation of the processor was interpreted as being a register transfer model for the hardware implementation.

The following sections discuss the implementation of the various logic blocks within the processor.

4.1 State, Token Stacks and Token Queues

The original idea was to implement the two token and state stacks and also the lexical queue as individual devices which would be controlled from the main processor device. This would have required the design of one stack device and a lexical character buffer device. The stack device would have been used twice, once for the state stack and once for the token stack.

4.1.1 State and Token Stack

The stack device was designed to have the architecture as shown in the diagram below.

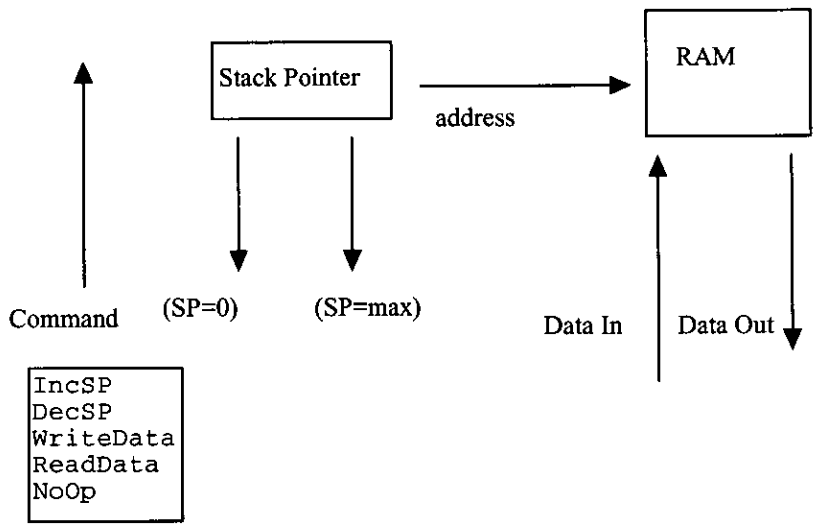

Figure 22 - Stack Device

As the software version of the processor used a 13-bit parameter this required the stack RAM data width to also be 13-bits. The silicon design tools were only able to provide memory holding up-to 8192 bits. Several attempts were made to have multiple memory blocks on a single device (so that a stack depth > 512) but due to restrictions on internal wire-lengths these attempts were unsuccessful.

The stack chip was however fabricated and contained a RAM block of 13-bits width and address range of 512 locations. This device, using a 2 micron CMOS technology, was 4772 by 4656 microns in size and used a 40 pin dual in line package.

4.1.2 Token Char Queue

The token character (or lexical) buffer queue was designed to have the hardware implementation as shown in the following diagram. This implementation was also derived from the processor software emulation, which acted as a register transfer model. The lexical buffer queue was not implemented as a standalone device. After the difficulties with the stack chip, the concept of incorporating internal RAM for the stacks and queues was abandoned.

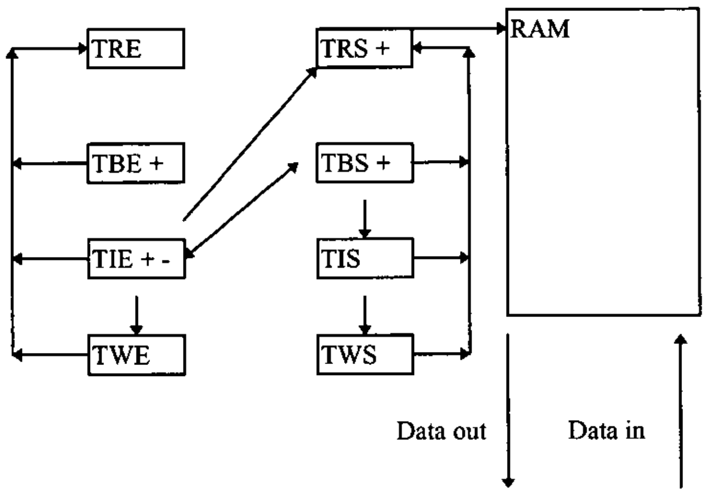

Figure 23 - Lexical Buffer

Key :-
+ implies that the register can be incremented by 1. (i.e. TRS, TBE, TBS, TIE)
- implies the register can be decremented by 1. (i.e. TIE)

Hardware Register	*Source Code Variable*
TRS	TokenRamStart
TRE	TokenRamEnd
TBS	TokenBufferStart
TBE	TokenBufferEnd
TIS	TokenIsStart
TIE	TokenIsEnd
TWS	TokenWasStart
TWE	TokenWasEnd

Figure 24 - Lexical Buffer Registers

The lexical buffer block also incorporated control logic to ensure the defined register to register transfers were possible, also to enable the increment and decrement by 1 of the specified registers. Finally the control logic ensured data could be written to and read from the attached RAM.

The lexical buffer queue was implemented as a cyclic buffer. This is shown in the following diagram.

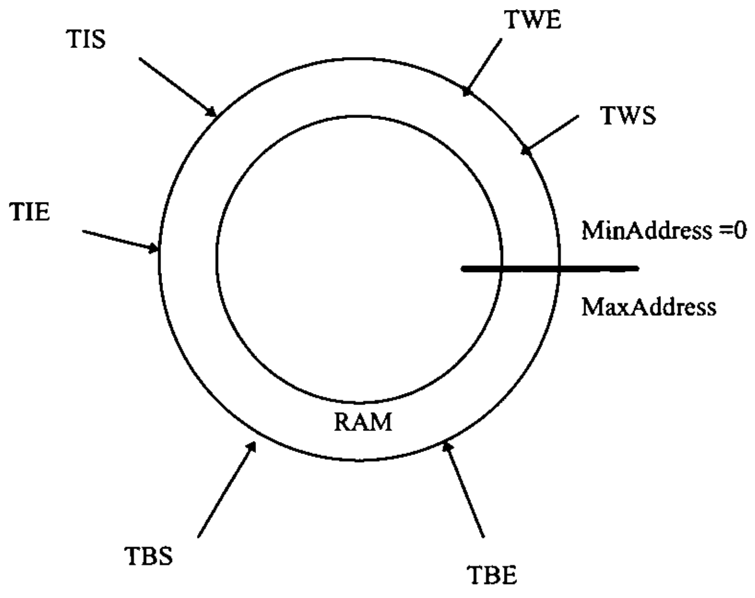

Figure 25 - Lexical Queue as Cyclic Buffer

If any pair of associated registers (i.e. TxS, TxE) have the same value (that is, point to the same memory address) then the corresponding token string is empty. The TRS, TRE registers are only used when either reading from or writing to, the buffer RAM. The TRS register is also used as the current address pointer for the lexical memory.

When the processor is "parsing", the values held in the TWS, TWE, TIS, TIE, TBS and TBE registers are not modified and are in the order shown in the diagram.

When the processor is "lexical analysing", the values held in the TIS, TIE, TBS and TBE registers are being modified. Additionally, the value in the TIS and TBS registers are identical and TIE is guaranteed to have an inclusive value in the range between TBS + 1 and TBE. The source input supplies characters which are read into the lexical queue and appended into the memory location indicated by TBE (TokenBufferEnd).

4.1.3 Token Symbol Queue

The Token symbol queue is primarily used to store the next parse token. This has either been read from the source (by the lexical instructions) or been inserted from the left-hand side symbol of a grammar rule that has been recognised.

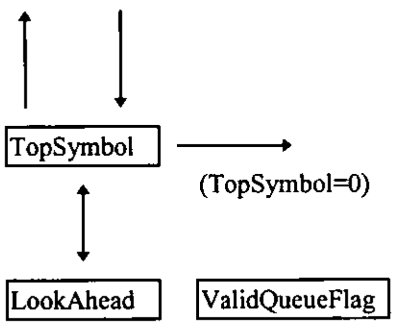

Figure 26 - Symbol Queue

The diagram shows the register to register data-paths. The comparison signal (TopSymbol = 0) is defined from the TopSymbol register value. The diagram omits the comparison signals (TopSymbol=P1), (TopSymbol>=P1) and (TopSymbol<=P1) where P1 represents the value of the instruction parameter register. The comparison signals are used by some of the Phase0 micro-instructions.

4.1.4 Using Internal or External Memory

The difficulties in having multiple memory blocks on a device, caused by the excessive wire-lengths involved forced a re-evaluation of the design of the processor hardware architecture. It was decided to move all memory required by the stacks and buffer queues off the processor to become external to the processor design.

The use of external memory would enable the size of the stacks and buffer queues to be increased. It was noted that the size of the memory available for use by stack and queue logic units imposed some constraints on the run-time use of the processor. The size of the token stack memory constrained the size of tokens (especially comment tokens). Also the buffer queue had to contain three token strings, which would also restrict the size of tokens.

The need for external RAM memory imposed a requirement for memory addressing additional to the instruction memory (assumed to be in a ROM). The use of three separate address spaces for the two stacks and one buffer queue was immediately discarded, since this would have required three address signal busses. The concept of one address space (and hence one address bus) which combined all address spaces (token stack, state stack, lexical queue and instruction memory) was utilised.

The software implementation of the processor (and hence the hardware version) also has to read from the source input stream the values of the lexical characters. The processor also is required to output (on demand) a token character string when a semantic action is in progress. That is, an I/O address space was also required.

The various address spaces were therefore combined into one address space which was partitioned into eight segments. These segments are indicated in a following table.

Segment	Page	Use/Purpose of Segment
0	0	Phase 0 Instruction (ROM)
1	1	Phase 1 Instruction (ROM)
2	2	Phase 2 Instruction (ROM)
3	3	Phase 3 Instruction (ROM)
4	0	State Stack RAM
5	1	Token Stack RAM
6	2	Lexical Buffer RAM
7	3	Unused (I/O space)

Table 16 - Memory Segment Definition

Segments 0 to 3 are used by the instruction memory (usually ROM) and correspond to the instructions for phase 0 to 3 respectively. The other four segments 4 to 7 are used by the various RAM address spaces, including the Input/Output (I/O) space.

The segmented memory address space, which is able to address ROM, RAM and I/O, forced some constraints on the legal combinations of the various memory enable and write signals. To enforce the legal combinations, the use of an internal 4-bit "IOMode" control bus (giving 16 legal combinations) was adopted. This is shown in the table below. The low three wires of the bus are input to the bit-slice device so the correct address register can be used.

The instruction memory segments could be implemented as either ROM or RAM memory. If the instruction memory is RAM then a mechanism is needed to load the parse and lex instructions from an external source using a fixed message protocol. InstFlag is an internal signal used to indicate that the instruction memory segments are being written to (if RAM) and therefore is indirectly set and cleared by the protocol. This protocol will be described later.

IOMode	Meaning	Enable		Write	Wanted			Address	Page
		Inst	Data		Inst	Data	DMA		
0	NoOp	0	0	0	0	0	0	PC	0
1	DataWanted	0	0	0	0	1	0	PC	0
2	DataDMA	0	1	0	0	0	1	TRS	2
3	InstWanted	0	0	0	1	0	0	PC	0
4	ReadInst0	1	0	InstFlag	InstFlag	0	0	PC	0
5	ReadInst1	1	0	InstFlag	InstFlag	0	0	PC	1
6	ReadInst2	1	0	InstFlag	InstFlag	0	0	PC	2
7	ReadInst3	1	0	InstFlag	InstFlag	0	0	PC	2
8	ReadData0	0	1	0	0	0	0	StateSP	0
9	ReadData1	0	1	0	0	0	0	TokenSP	1
10	ReadData2	0	1	0	0	0	0	TRS	2
11	ReadData3	0	1	0	0	0	0	TRS	3
12	WriteData0	0	1	1	0	0	0	StateSP	0
13	WriteData1	0	1	1	0	0	0	TokenSP	1
14	WriteData2	0	1	1	0	0	0	TRS	2
15	WriteData3	0	1	1	0	0	0	TRS	2

Table 17 - IOMode Definition

4.2 New Processor Architecture

The use of external memory and also having the bit width of the instruction parameter the same size as the state address (state stack data), the token size, the character size and the rule size enabled the processor to be split into a control logic block and a data logic block. The control and data blocks have the connections as shown in the following diagram.

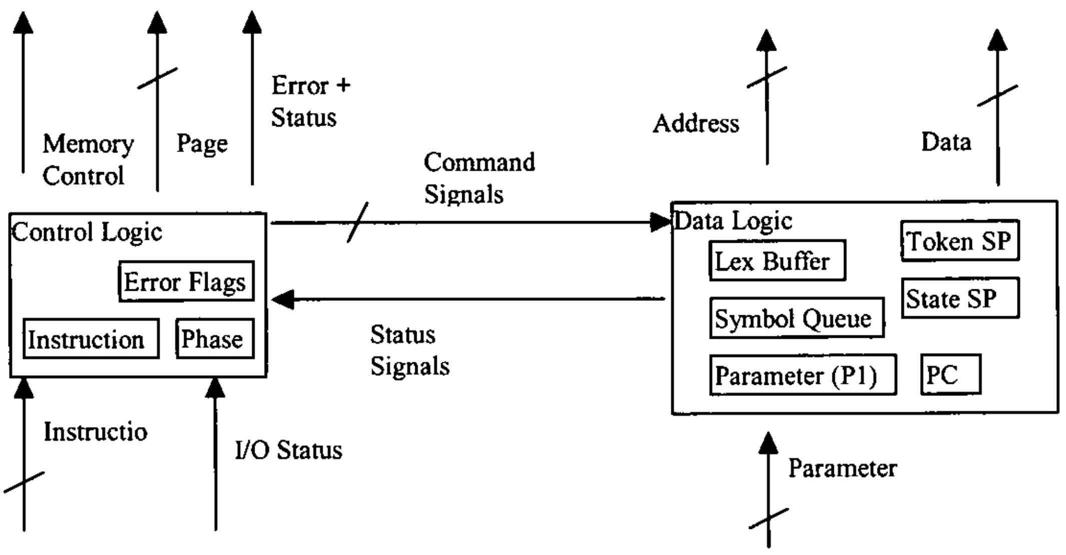

Figure 27 - New Processor Architecture

All registers and logic circuitry which depend on the width of the parameter, address, token and character buses are localised within the data block. Thus the data block could be implemented with different register widths (i.e. as a bit-slice). Also, the fixed size busses such as the instruction bus (3-bits) are embedded within the Control logic block.

4.2.1 Cycle-based Simulation

The modifications to the processor architecture (that is, the splitting of the control and data-path logic) required further changes to be made to the compiler-compiler software to emulate the new processor. Primarily the software was altered to provide cycle based simulation, unlike the previous version of software which only simulated the execution of instructions. Emulation results of the instruction execution variant of the software are indicated in a table of results in the previous chapter. Results for the cycle based emulation are listed in a table in a following section in this chapter.

The software emulation of the processor (written in PASCAL) then formed the design specification for the two types of logic block. The PASCAL language has similar constructs to those present in the logic synthesis language LOLA which is part of the SOLO 1400 software tool set used to design and layout the logic. For more details about the design suite refer to the Solo 1400 User Guide [European Silicon Structures 93]. The similarity of constructs (such as the case statement) enabled the PASCAL source to be quickly converted into logic via the logic synthesis tool, once the PASCAL source was manually transformed into LOLA.

The software emulation of the processor was also able to generate test-vectors (and the expected signal outputs) to validate the logic design. It was found that the signal outputs from the SOLO 1400 logic simulator, MADS and the expected results from the software emulation were in agreement. Refer to "Appendix **A - Software Simulation**" for part of an example simulation run.

4.2.2 Processor Physical Implementation

The data-path and control blocks were both designed and fabricated using 1.5 micron CMOS gate-array technology. The data-path block (logic and registers) was implemented as an 8-bit bit-slice device. Both devices are described in the next sections.

4.3 Data-Path Bit Slice

The bit slice device used a repeated logic cell with ripple-through logic. The top level block incorporated logic to decode the command bus signals and the repeated slice logic.

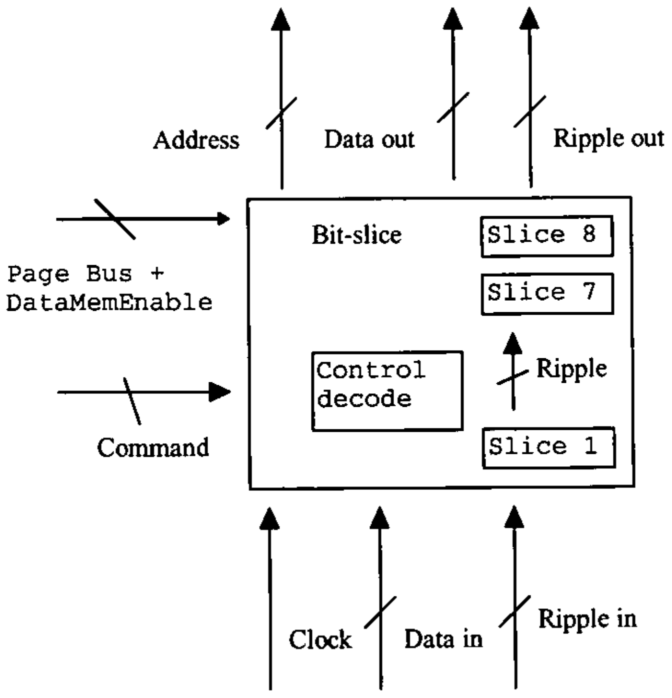

Figure 28 - Bit-slice I/O

The operation of the bit-slice is controlled by the combined EngineCommand bus, the Page bus and the DataMemEnable signal. The combination of Page bus and DataMemEnable signal is used to select which register should be used to form the memory address register, one of PC, TokenSP, StateSP or TokenRAMStart. The EngineCommand bus controls the register to register transfers and also the increment or decrement (by one) of some of the registers.

The devices can be combined in series as shown in the next diagram.

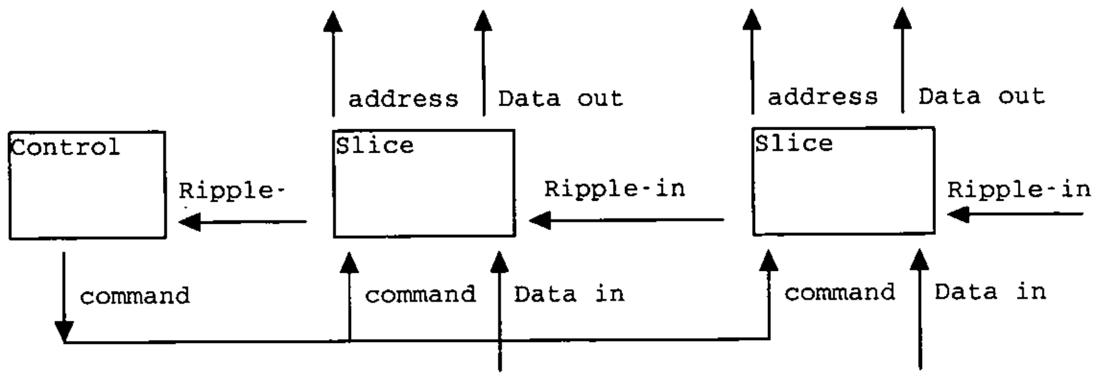

Figure 29 - Control and Bit-slice blocks

Ripple-through logic was extensively used for simplicity of design. As the main objective of implementing the processor in silicon was to prove the concept, the penalty of long delay paths for signals (forcing a slower clock) was accepted. The list of ripple-through signals is listed in the following table.

Ripple Signal	Signal Purpose
Carry	Carry for Increment/Decrement by 1
TokenRamEqual	(TRS = TRE)
TokenBufferEqual	(TBS = TBE)
TokenIsEqual	(TIS = TIE)
TokenSPIsZero	(TokenSP = 0)
StateSPIsZero	(StateSP = 0)
P1IsZero	(P1 = 0)
SymbolGreaterOrEqual	(TopSymbol >= P1)
SymbolEqual	(TopSymbol = P1)

Table 18 - Bit-Slice Ripple Signals

The logic for the individual ripple signals is of 4 basic types. These are :-

- to test if the register is zero
- to compare two registers for equality
- to propagate a carry/borrow signal for the increment/decrement by 1 of a register
- to test if a register is greater than or equal to another register.

Using the MODEL hardware description language this becomes,

or[and[a,not[b]],and[eqv[a,b],r_in]] -> r_out

Thus one slice will require an or gate, two and gates, an equivalence gate and an inverter.

4.3.1.4 Carry/Borrow for Increment/Decrement by 1

If a[i] represents the i'th bit for register a, and c[i] is the i'th carry/borrow input signal and inc represents a signal indicating the number is to be incremented (if at logic 1) and indicates a decrement (if at logic 0) then we have the following table :-

inc (+1 if 1, -1 if 0)	c[i] carry/borrow in	a[i] register	a'[i] New value of a[i]	c[i+1] carry/borrow out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	0	0
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Table 19 - Carry/Borrow for +1/-1

a'[i] is (a[i] <> c[i]), i.e. the exclusive-or of a[i] with c[i]

and one expression for c[i+1] can be optimised to be,

c[i+1] is (inc = a[I]) and (c[I] = 1)

The carry/borrow value of the most significant bit of an increment/decrement is ignored by the control logic since underflow or overflow is permitted.

Using the MODEL hardware description language this becomes,

eqv[a,c_in] -> a_new

and[eqv[inc,a],c_in] -> c_out

4.3.2 Register To Register Transfer

The register to register transfer is carried out using a number of internal busses. An increase in silicon area (caused by wiring and extra gates needed to form individual increment and decrement logic for the TRS, TBS, TBE and TIE registers) was avoided by using a single increment/decrement logic block with the internal busses. This is shown in the next diagram.

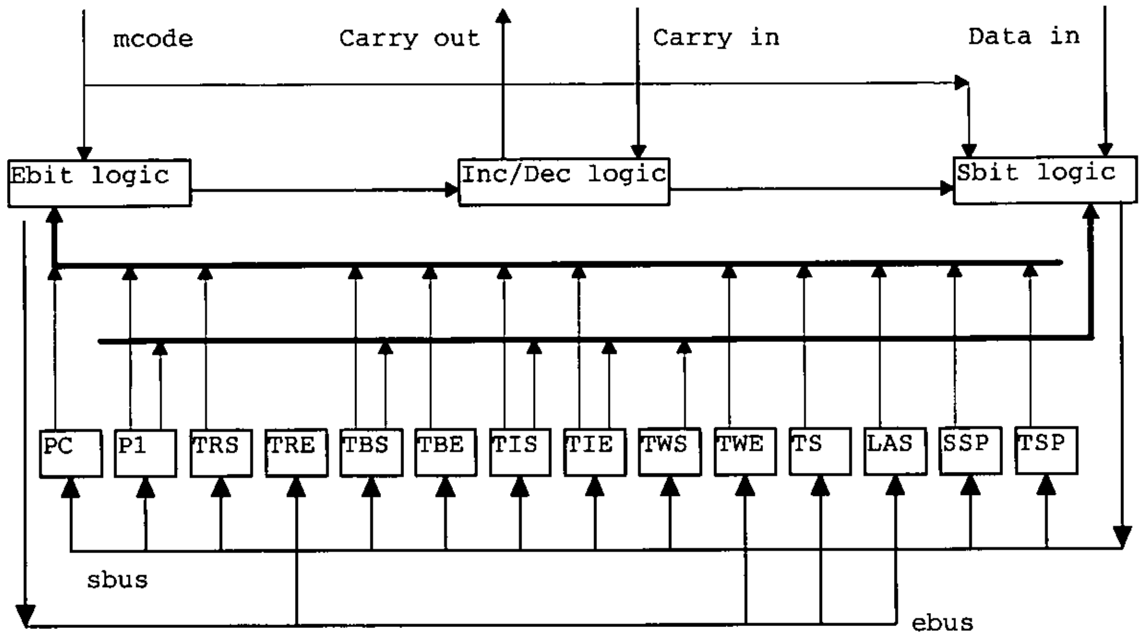

Figure 30 - Bit-slice Register Transfer

The individual registers each have an associated command signal indicating when to load data values (from either the sbus or ebus) or to reload the current register value. The load command is decoded from the command bus input to the bit-slice at the top level of the device.

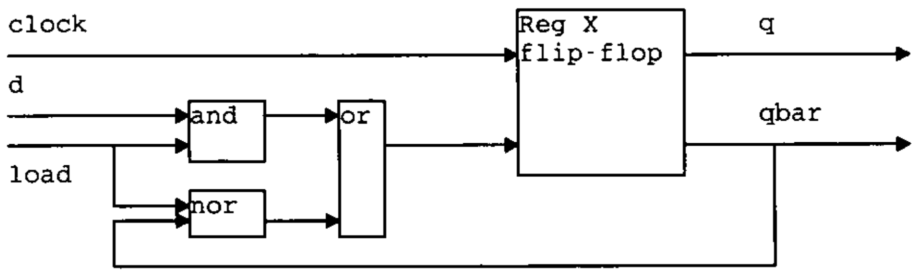

Figure 31 - Register Transfer

The q and qbar register outputs are both used so as to minimise both logic used and path delay times.

4.3.3 Bit-Slice Commands

The operation of each bit-slice device is decided by the EngineCommand bus. This is 5-bits wide giving a total of 32 possible commands. The individual commands are listed in the following table.

Command Bus	Command Name	Command Actions
0	NoOp	None
1	TokenSetBusMayBe	TRS := TIE, TRE := TBE
2	TokenSetBusBuffer	TRS := TBS, TRE := TBE
3	TokenSetBusIs	TRS := TIS, TRE := TIE
4	TokenSetBusWas	TRS := TWS, TRE := TWE
5	TokenSetIsEmpty	TIS := TBS, TIE := TBS
6	TokenLoadWasIs	TWS := TIS, TWE := TIE
7	Zero	Clear all registers
8	TokenSPZero	TokenSP := 0
9	SymbolLoadPC	PC := P1
10	TokenAccept	TBS := TIE
11	SymbolP1Load	P1 := MEMORY (DataIn)
12	SymbolLoadLAS	LookAhead := 0
13	SymbolPop	TopSymbol := LookAhead
14	SymbolPush	LookAhead := TopSymbol
15	SymbolLoadTS	TopSymbol := P1
16	StateSPInc	StateSP := StateSP + 1
17	TokenSPInc	TokenSP := TokenSP + 1
18	TokenIncRamStart	TRS := TRS + 1
19	SymbolInc	PC := PC + 1
20	TokenIncBufferStart	TBS := TBS + 1
21	TokenIncBufferEnd	TBE := TBE + 1
22	TokenIncIsStart	TIS := TIS + 1
23	TokenIncIsEnd	TIE := TIE + 1
24	StateSPDec	StateSP := StateSP - 1
25	TokenSPDec	TokenSP := TokenSP - 1
26	TokenDecRamStart	TRS := TRS - 1
27	SymbolDec	PC := PC - 1
28	TokenDecBufferStart	TBS := TBS - 1
29	TokenDecBufferEnd	TBE := TBE - 1
30	TokenDecIsStart	TIS := TIS - 1
31	TokenDecIsEnd	TIE := TIE - 1

Table 20 - Bit-Slice Commands

All the commands are implemented and available for use, however the control device uses a subset of the commands to implement the instruction set for the processor.

4.3.4 Fabrication Details

The bit-slice device was fabricated using a 1.5 micron CMOS gate array technology. The actual device size was 3777 by 3244 microns. It used 6664 stages where each stage consisted of a pair of NFET and PFET devices and so the bit-slice logic was implemented using 13328 transistors. The device was packaged in an 84 pin grid array where 16 pins were reserved for power and ground connections (8 power and 8 ground) and 15 pins were unconnected.

The diagram on the next page shows the physical design layout for the bit-slice device.

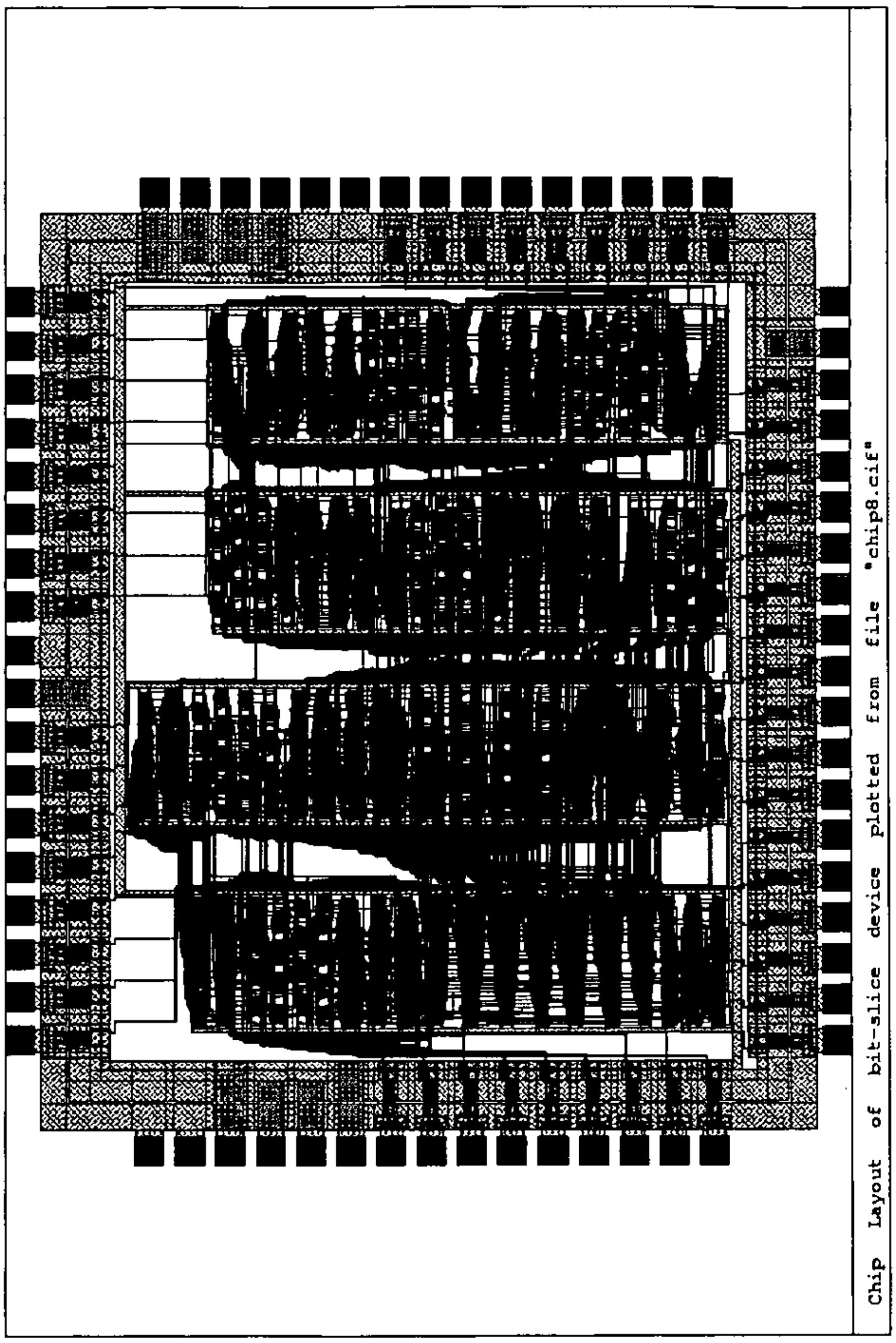
Chip Layout of bit-slice device plotted from file "chip8.cif"

Figure 32 - Bit-Slice Device Chip Layout

4.3.5 Device Pinout

The diagram shows the pin layout of the 84 pin grid array used by the bit-slice device. The package has 85 pins where pin C9 is used for alignment or package orientation when inserting onto a circuit board.

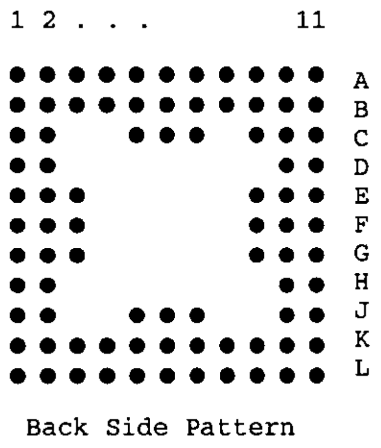

Figure 33 - PGA Pin Layout

The pinout of the device is given by the following table. NC indicates that the pin is Not Connected to the encapsulated chip. Also the Pad number indicates the internal pin for connection to the device bond pads.

Pad	Pin	Signal	Pad	Pin	Signal	Pad	Pin	Signal	Pad	Pin	Signal
1	B2	NC	22	K2	NC	43	K10	NC	64	B10	NC
2	C2	address8	23	K3	NC	44	J10	clock	65	B9	NC
3	B1	address7	24	L2	NC	45	K11	mcode0	66	A10	NC
4	C1	address6	25	L3	gnd	46	J11	mcode1	67	A9	seout
5	D2	address5	26	K4	gnd	47	H10	mcode2	68	B8	sgeout
6	D1	address4	27	L4	gnd	48	H11	mcode3	69	A8	plizout
7	E3	address3	28	J5	vdd	49	F10	mcode4	70	B6	sspizout
8	E2	address2	29	K5	vdd	50	G10	iomcode0	71	B7	tspizout
9	E1	address1	30	L5	vdd	51	G11	iomcode1	72	A7	tieout
10	F2	gnd	31	K6	carryin	52	G9	iomcode2	73	C7	tbeout
11	F3	vdd	32	J6	trein	53	F9	datain1	74	C6	treout
12	G3	dataout8	33	J7	tbein	54	F11	datain2	75	A6	carryout
13	G1	dataout7	34	L7	tiein	55	E11	datain3	76	A5	gnd
14	G2	dataout6	35	K7	tspizin	56	E10	datain4	77	B5	gnd
15	F1	dataout5	36	L6	sspizin	57	E9	datain5	78	C5	gnd
16	H1	dataout4	37	L8	plizin	58	D11	datain6	79	A4	vdd
17	H2	dataout3	38	K8	sgein	59	D10	datain7	80	B4	vdd
18	J1	dataout2	39	L9	sein	60	C11	datain8	81	A3	vdd
19	K1	dataout1	40	L10	NC	61	B11	vdd	82	A2	NC
20	J2	NC	41	K9	NC	62	C10	gnd	83	B3	NC
21	L1	NC	42	L11	NC	63	A11	NC	84	A1	NC

Table 21 - Bit Slice Pinout

4.4 Control Device

The logic for the control device is implemented as a finite state machine, whose state changes depend on internal and external signals and also on the current state. Each state also outputs a number of signal values to operate external logic such as the bit-slice devices (via the EngineCommand signals), memory (via the enable, write, address and data bus signals) and the semantic logic (via the irq and dataout bus signals).

4.4.1 Internal Logic

The top level logic design of the control device is outlined in the next diagram. This diagram shows that the major logic unit is the 'next state and commands' block where each connection from this block represents a signal bus sending commands to the associated logic.

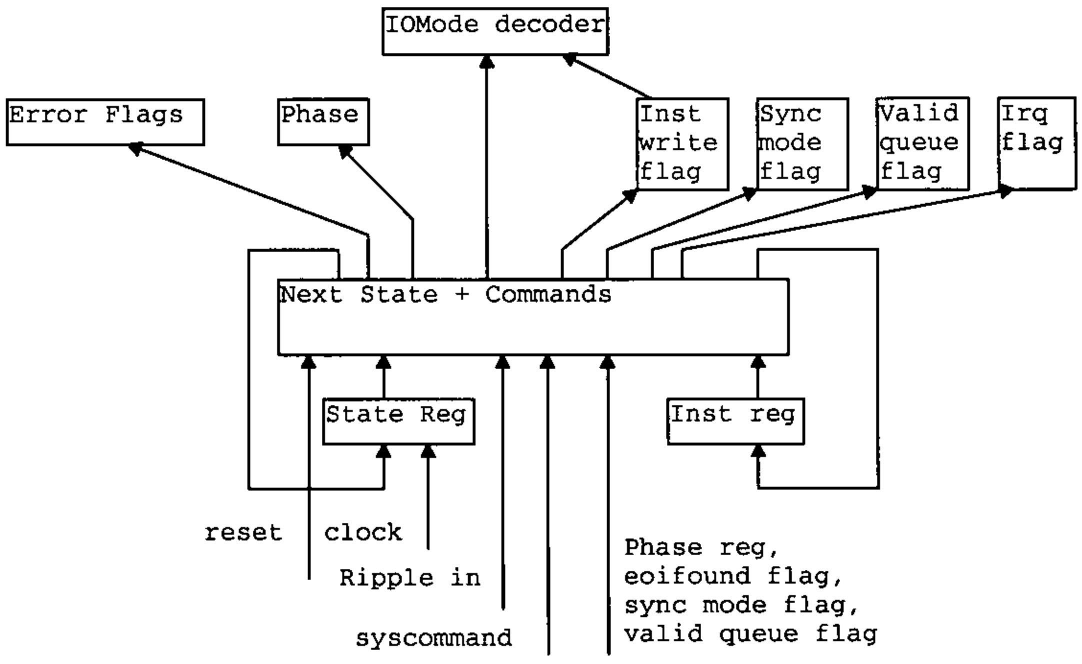

Figure 34 - Control Logic Internals

The possible commands that can be sent to the IOMode decoder are defined in "Table 17 - IOMode Definition". Note that the InstWrite flag also is an input to the decoder. The implementation of the InstWrite, Sync Mode, Valid Queue and IRQ flags is detailed in the following section 4.4.1.1 titled "Flag Logic".

4.4.1.1 Flag Logic

All flags use a dual wire command bus which allows 4 possible commands to be defined. Using Flag to denote the current flag value and NewFlag to denote the next value for Flag, the commands and actions are :-

Command	Value	Command1	Command0	Action
FlagNoOp	0	0	0	NewFlag := Flag
FlagUnused	1	0	1	NewFlag := Flag
FlagClear	2	1	0	NewFlag := false
FlagSet	3	1	1	NewFlag := true

Table 22 - Flag Commands

This set of commands could be implemented in two ways. One method is to "gate the clock" such that the FlagSet and FlagClear commands are gated with the system clock to give the clock input to the latch. The other method is to always ensure that the latch data input has a legal value and that the system clock directly feeds the latch clock.

The technique of "Gating the clock" causes extra loading on the clock wiring which then slows down the clock. This method was therefore not used.

The second method imposes extra loading on the command signals. This extra loading can be ignored as it is comparatively local, and not global like the system clock. It is implemented as shown in the next diagram.

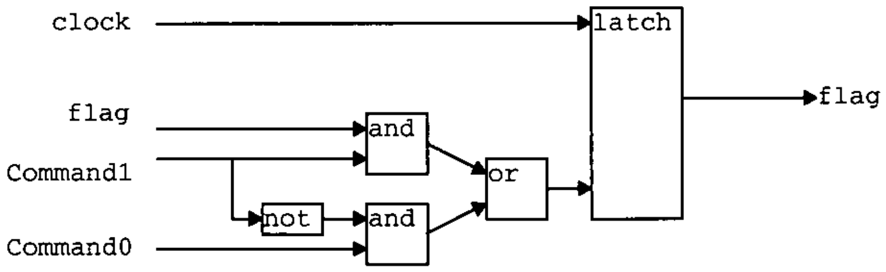

Figure 35 - Control Flag Logic

4.4.1.2 Error Flags

The error flags block contains the 9 error flags and also includes the EOIFound flag. All of these flags are cleared at system reset and only set individually when the error condition has been detected. Thus the command bus need only have 4 wires allowing 16 possible commands which are listed in the next table.

Name	Value	Action
ErrorNoOp	0	None
ErrorSourceExhausted	1	SourceExhausted := true
ErrorParseSyntax	2	ParseSyntax := true
ErrorParseSemantic	3	ParseSemantic := true
ErrorLexSyntax	4	LexSyntax := true
ErrorStackUnderflow	5	StackUnderflow := true
ErrorStackOverflow	6	StackOverflow := true
ErrorIllegalInstruction	7	IllegalInstruction := true
ErrorNoErrorHandler	8	NoErrorHandler := true
ErrorBufferOverflow	9	BufferOverflow := true
ErrorUnused10	10	
ErrorUnused11	11	
ErrorUnused12	12	
ErrorUnused13	13	
ErrorFoundEOI	14	EOIFound := true
ErrorReset	15	All flags set to false

Table 23 - Commands for Error Logic

Each flag was implemented as described in section 4.4.1.1 titled "Flag Logic", so each value of the ErrorCommand bus is able to set values for the individual control busses for the ten flags.

4.4.1.3 Phase Register

The phase register is two bits wide and can be left unchanged or set to a value from 0 to 3. This requires 5 possible commands which mandates the use of 3 command wires. The commands to modify this register are given below. Each command then generated a flag command to leave unchanged, clear or set the two latches forming the phase register.

Command	Value	Action	HighBit Flag Action	LowBit Flag Action
SemNoOp	0		FlagNoOp	FlagNoOp
SemUnused1	1		FlagNoOp	FlagNoOp
SemUnused2	2		FlagNoOp	FlagNoOp
SemUnused3	3		FlagNoOp	FlagNoOp
SemSet0	4	Phase := 0	FlagClear	FlagClear
SemSet1	5	Phase := 1	FlagClear	FlagSet
SemSet2	6	Phase := 2	FlagSet	FlagClear
SemSet3	7	Phase := 3	FlagSet	FlagSet

Table 24 - Phase Register Commands

4.4.2 Processor Internal States

The control device executes each micro-instruction of the processor as a sequence of steps, where each step will perform some actions in the bit-slice, associated memory, source input logic or semantic logic. Each step is a single state which in combination form the state machine that is the processor.

The processor control logic uses 48 states. The state machine could be implemented using the concept of "one hot" encoding with 48 latches (one per state). In "one hot" encoding only one latch should ever be set (representing the current state) and all others latches are clear. This requires the next state logic to ensure that there is no possibility of more than one state latch being set simultaneously. Also extensive re-design is needed using this approach if extra states need to be added. The actual implementation used 6 latches to form a state register which allows 64 possible states. As only 48 states are used this left sufficient unused states for later expansion of the state machine.

A simplified version of the state machine is shown in the next diagram.

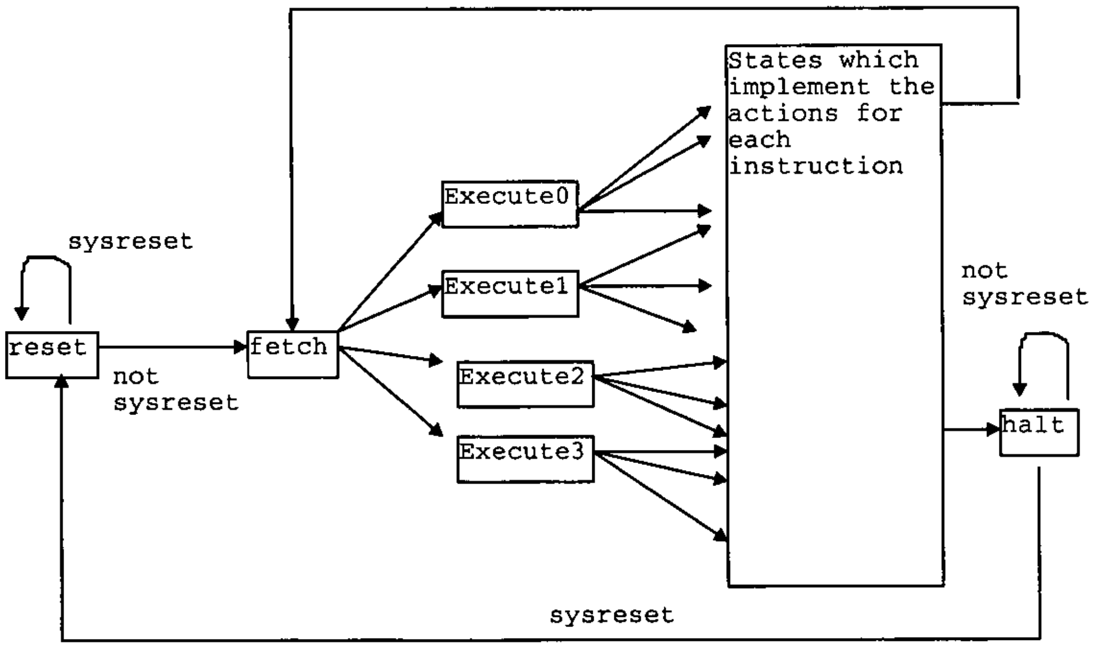

Figure 36 - Processor State Machine

After the processor has reset, the processor loops through a fetch and execute cycle using many states until it halts either because the parse and lexical analysis has completed or an error has been found. The processor then goes to the halt state. In this state the error flags are available for use by external hardware. The halt state is only exited by applying the external reset.

Each micro-instruction starts with the fetch state. This latches in the instruction and parameter to be executed next. (The memory control signals InstMemEnable will be cleared at the end of this state). Depending on which phase 0 .. 3 is being executed, the next state will be one of Execute0 .. Execute3. The ExecuteX states decode the micro-instruction to be executed and jump to the sequence of states needed by each micro-instruction to implement the required actions. Some instructions have common actions and therefore have states in common.

4.4.3 Interfaces and Protocols

The control device has a number of input and output signals which are used to interface to external logic. The interfaces are implemented using a number of protocols for the following purposes,

- To read or write to memory
- Logic to implement language semantics
- Logic to input source text
- Logic to read Token character strings

The protocols and signals used are described in the next sections.

Some of the protocols use the SysCommand input bus to send status information back to the processor from the external logic. Each SysCommand bus value is used and is indicated in the next table.

SysCommand	SysCommand2	SysCommand1	SysCommand0
SysNoOp	0	0	0
StateWanted	0	0	1
TokenWanted	0	1	0
IRQ_NullToken	0	1	1
IRQ_OK	1	0	0
IRQ_Err	1	0	1
DataAvailable	1	1	0
InstAvailable	1	1	1

Table 25 - SysCommand Bus Definitions

4.4.3.1 System Reset

The processor could power up into any of the possible internal states. The logic of the state machine has been designed so that if the external reset (sysreset) is true then the state machine will goto the reset state. While in the reset state the processor internal registers will be initialised, usually to zero. The reset state is only exited when the external reset goes low.

4.4.3.2 Accessing Memory

The processor memory space is split into instruction memory or data memory for the stacks and lexical buffer. Refer to "Table 17 - IOMode Definition" for more information.

Memory access is controlled by the InstMemEnable, DataMemEnable and MemoryWrite output signals. The instruction memory is accessed when InstMemEnable is high and data memory when DataMemEnable is high. The combination of InstMemEnable, DataMemEnable, MemoryPage0, MemoryPage1 help to select which of the eight memory segments is being accessed.

Memory access takes two clock cycles. The first clock cycle sets the memory enable signals, memory page signals and memory write signal to a legal combination to read or write to a memory segment. The second clock cycle will clear the enable signal.

4.4.3.3 Reading the Source Text

The processor needs to read the source text to be able to parse it. A request for the next character in the source input is indicated by SysDataWanted going high. This output stays high until the external logic has a character available which is indicated by SysCommand having the value SysDataAvailable. At this point the character is loaded into the bit-slice and SysDataWanted will go low from the next clock cycle. It must be noted that each character of source text is read individually and only requested when the processor needs it.

4.4.3.4 Interrupts (Rule Recognition and Test Routines)

The processor needs to indicate an interrupt to the external logic which handles language semantics and lexical tests that a grammar rule or test must be handled. This is signalled by the SysIRQ output going high and staying high until the external logic acknowledges interrupt completion. For a parse semantic action, the grammar rule being recognised is output on the dataout bus. For a lexical text routine, the lexical test being checked is also output on the dataout bus. Interrupt completion is indicated by the SysCommand input bus having one of the values :-

- SysIRQ_OK
- SysIRQ_Err
- SysIRQ_NullToken.

Whilst the interrupt is in progress, which could take many clock-cycles, the SysCommand bus must have the value of SysIRQ_NoOp. This value is used to indicate that the external logic has not completed.

The legal values to indicate completion are :-

- SysIRQ_OK indicates that the interrupt has been successful.
- SysIRQ_Err indicates that the interrupt detected a parse semantic error. This should only be used when the interrupt triggered is one to recognise a grammar rule. That is the interrupt is **not** a lexical test routine.
- SysIRQ_NullToken should only be used by a lexical test routine to indicate that the potential next token being recognised can be discarded, probably since it is a whitespace (or comment) token.

4.4.3.5 Outputting Current Parse State

During the interrupt raised by the processor (i.e. SysIRQ is high), the external logic may need to request the value of the current parse state for use in error reporting. This is indicated by SysCommand having the value SysStateWanted for a single clock-cycle. The value of the current parse state is output on the bit-slice data output bus for the next clock cycle for use by the external logic.

4.4.3.6 Outputting Tokens

During the interrupt raised by the processor, the external logic may need to request the complete value of one of the token strings held in the lexical buffer.

This is triggered by SysCommand having the value SysTokenWanted for one clock-cycle whilst SysIRQ is true. From the next clock-cycle the individual characters of the token string are output from the data memory, one character per clock-cycle until the complete token string has been send. Valid characters are indicated by the values being both true for the output SysDataDMA and of the TREIn input.

4.4.4 Error Detection and Handling

The control device also has a limited capability to detect and handle errors. The control device has 9 internal latches which are used to indicate a range of detected errors and warnings (3 warnings and 6 errors). The latch outputs are connected to chip bond pads for use by external logic. The ParseDone flag (available as an output pin) indicates when the parsing process has terminated. At that moment the error flags can be examined to determine if the parse was successful.

The three warnings detected are ParseSemanticError (pin 7), LexSyntaxError (pin 8) and ParseSyntaxError (pin 32). Detection of one of these warnings will not cause the processor to halt. LexSyntax and ParseSyntax errors indicate that the lexical tokens and parse tokens respectively do not follow the structure given by the language grammar. ParseSyntaxError could trigger a NoErrorHandlerError if the grammar does not contain any error rules (or error handler routines).

The six errors detected will cause the processor to halt since continuation could cause unexpected behaviour.

The SourceUsedError (pin 31) flags the situation that an attempt has been made to read more source input after the end of input token has been recognised.

The IllegalInstructionError (pin 13) flags the situation when an illegal or undefined instruction has been read and the processor is attempting to execute it.

The NoErrorHandlerError (pin 14) flags the situation when a parse error has been detected and the state stack contains no state which has a shift instruction triggered by the error token. Error handlers can only be defined by adding error rules to the grammar definition.

The BufferOverflowError (pin 15) flags the situation when appending a character from the source input to the TokenBuffer (hence incrementing TokenBufferEnd) it is found that TokenBufferStart and TokenBufferEnd have the same value. This indicates the buffer has overflowed. The TBE signal from the bit-slice device indicates when TokenBufferStart and TokenBufferEnd are identical, the control logic uses this signal at the instance when this becomes a fatal error.

Bounds checks are also performed on the two stack pointers (TokenSP and StateSP). Underflow and overflow of these stacks are fatal errors causing unexpected processor behaviour and are flagged by StackUnderflowError (pin 11) and StackOverflowError (pin 12).

The TSPIZ (TokenStackPointerIsZero) and SSPIZ (StateStackPointerIsZero) signals from the bit-slice are used to detect these occurrences by the control logic. Underflow is detected when the StackPointer (SP) is zero and a StackPop (or SP := SP - 1) command is requested. Overflow is detected when the StackPointer is zero and a StackPush (or SP := SP + 1) has just been executed.

4.4.5 Fabrication Details

The control device was also fabricated using a 1.5 micron CMOS gate-array technology. The actual device size was 3317 by 3076 microns. It used 5201 stages which is equivalent to 10402 transistors. The device was packaged into a 48 pin dual in line where 8 pins were reserved for power and ground connections (4 power and 4 ground).

The diagram on the next page shows the physical design layout for the control device.

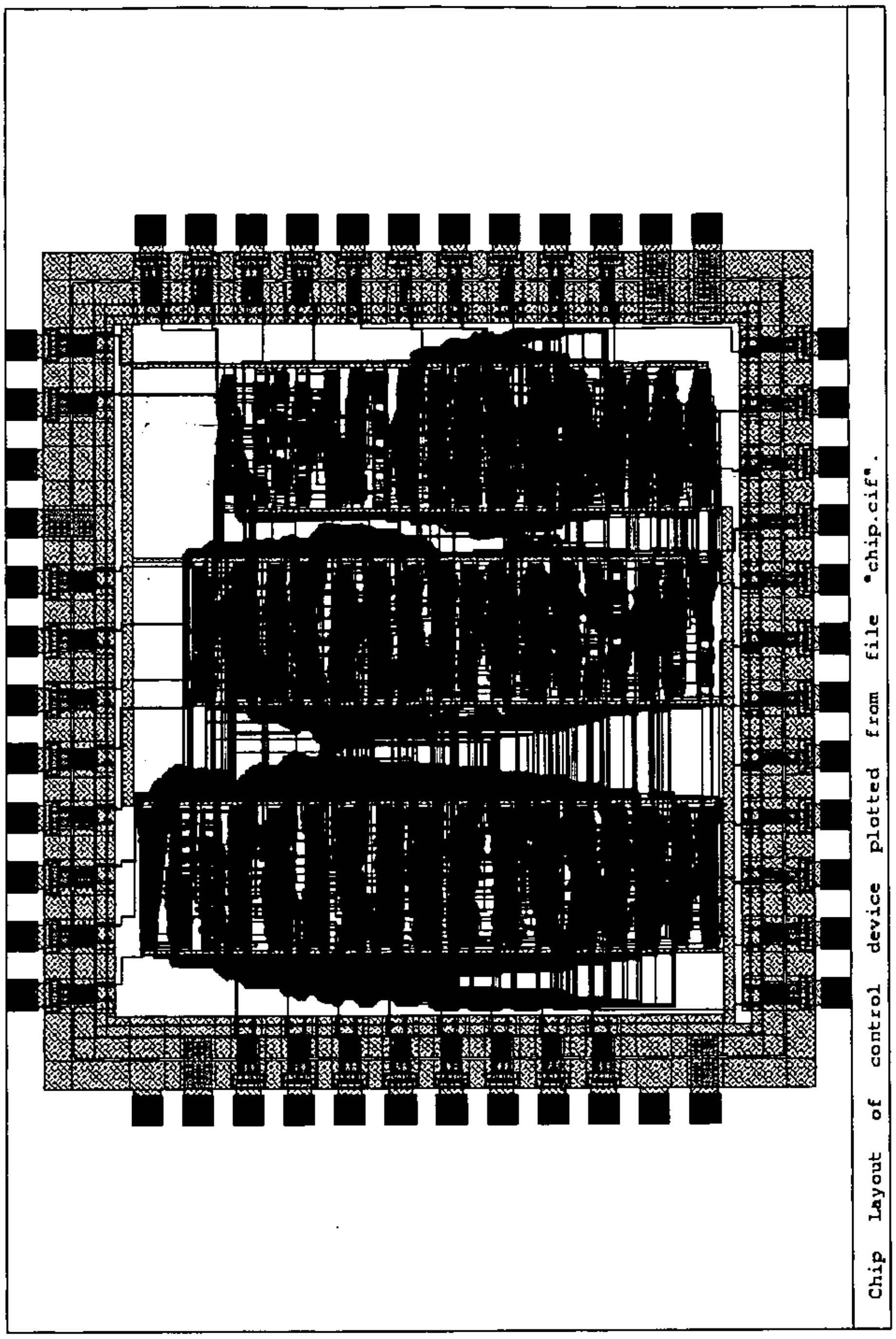

Figure 37 - Control Device Chip Layout

4.4.6 Device Pinout

Items marked with * are input signals from the most significant bit-slice device. Items marked ** indicate those signals which are outputs and go to all bit-slice devices. Items marked + indicate those signals which are output to all bit-slice devices and also the external memory devices.

Pin Number	Signal	Pin Number	Signal
1	syscommand1	48	syscommand0
2	syscommand2	47	inst2
3	datawanted	46	inst1
4	datadma	45	inst0
5	parsedone	44	vdd
6	irq	43	gnd
7	parsesemanticerror	42	sysreset
8	lexsyntaxerror	41	clock
9	gnd	40	*tbe
10	vdd	39	*tre
11	stackunderflowerror	38	*tie
12	stackoverflowerror	37	*pliz
13	illegalinstructionerror	36	*sspiz
14	noerrorhandlererror	35	*tspiz
15	bufferoverflowerror	34	*symbolge
16	instmenenable	33	*symbolequal
17	datamemenable	32	parsyntaxerror
18	instwanted	31	sourceusederror
19	gnd	30	vdd
20	vdd	29	gnd
21	**engcommand4	28	memorywrite
22	**engcommand3	27	+memorypage0
23	**engcommand2	26	+memorypage1
24	**engcommand1	25	**engcommand0

Table 26 - Control Device Pinout

The signals named XXXerror (e.g. stackunderflowerror) indicate the error flags which show the final parse status.

4.5 Testing and Emulation Results

The software emulation of the processor was not only able to generate the test vectors (and the expected results) for the two types of devices but it was also able to estimate the number of clock cycles required for a parse and lexical analysis. Accordingly no detailed analysis of the required minimal set of test vectors to validate the processor was deemed to be required. The two chip designs were validated by using the test vectors generated from the software emulation runs and comparing the actual results from the MADS hardware simulation software (provided as part of the SOLO 1400 chip design suite) with the expected results provided by the software emulation.

The following table details some sample runs. The table is used to indicate the range of tests performed and also to help indicate processor performance. A number of source files (in different languages) were used as input to the processor. The instruction counts match those recorded for the emulation runs used to compare the original 32-bit instruction set with the new 4-phase instruction set, with the exception of those for the LALR language. The LALR language is a language definition to define language definitions (and is therefore self-referential). This language was changed slightly between the two sets of test runs and therefore the run-times and instructions counts were different. The individual language grammars were not affected by the LALR grammar alteration and were not altered. Thus no change was expected in the count of instructions executed for the language test runs.

Language	File	Clock Cycle Count	Instruction Count	Chars Read	Line Count	Cycles/Line	Interrupts
LALR	ACE	165579	60621	1758	47	3523	1670
LALR	BASIC	136128	48931	1568	51	2669	1546
LALR	M2	497166	177537	6429	144	3453	4907
LALR	M2V	442626	158906	5552	130	3405	4334
LALR	PCPASCAL	478590	169947	6615	144	3324	4331
ACE	bad	16274	*5910	147	8	2034	197
ACE	bad1	15192	*5508	143	8	1899	173
ACE	jdm	16408	5964	148	8	2051	199
ACE	test	33871	12348	294	16	2117	410
ACE	test1	37852	13771	345	18	2103	548
BASIC	bad	5353	*1885	66	5	1071	53
BASIC	test	4238	*1476	63	4	1060	35
M2	deb	36572	13689	452	16	2286	284
M2	example	41695	15262	546	27	1544	412
M2	example1	68487	23720	1538	69	993	25
M2	example2	17714	6690	193	12	1476	159
M2V	deb	36200	13521	452	16	2263	283
M2V	example	41332	15096	546	27	1531	411
M2V	example1	68370	23688	1538	69	991	24
M2V	example2	17522	6604	193	12	1460	158
PCPASCAL	test	76122	25007	1241	67	1136	793

Table 27 - Clock cycles for Parse Input

The examples marked with an asterisk (in the instruction count column) represent those parse runs used to test the processors ability to detect invalid input.

It must be noted that a pseudo-random number generation was used to add in estimated delays caused by semantic actions and characters being read from the source stream.

Noting the variations caused by the random numbers and using the results from the larger source files tested, this gives a range of 2669 to 3523 clock cycles per line of source. The clock cycle depends on the worst case delay times for the bit-slice and control devices. These were given as being 25ns and 40ns respectively. For the fabricated devices and using two bit-slices this would give a clock cycle of 90ns. Best case delay times were given as 16ns and 25ns for the bit-slice and control devices respectively, giving a best case clock cycle time of 57ns.

Accordingly, the processor can compile an estimated 3154 to 4163 lines per second (using worst case delay) and 4980 to 6573 lines per second (using best case delays).

5. Real Applications

The processor can be used in most situations where there is a need for communication using a formal language. This does imply that the processor is restricted to compiling computer languages. The next sections will briefly outline some possible applications which are not implementations of compilers in hardware but do involve language recognition. The first, second and third sections describe potential uses of the processor which has been investigated using the software simulation. The later sections describe other possible applications which have been investigated in less detail.

5.1 Logic Synthesis

One possible application is to use the processor to parse regular expressions so that a logic block which recognises the regular expression can be synthesised. The "Appendix C - Synthesis Software" provides details of an appropriate grammar, the corresponding processor instruction table, and the required semantic actions needed to convert the regular expression into MODEL source code.

The first example expression, A = a b+ c will be used to demonstrate the synthesis process.

The parse tree for the example expression using the grammar from the appendix is shown below. The following table relates the actions attached to the grammar rules to the PASCAL functions which implement the semantics of the actions.

Rule Action	PASCAL Routine
A1	leftnameis
A2	primaryisid
A3	repeatisstar
A4	factoragain
A5	ruleis

Table 28 - Logic Synthesis Routines

The source code for the PASCAL routines can be seen in the appendix.

Note that the order in which the routines are called is given by the left-right post-traversal of the parse tree, which is implied by the LALR(1) algorithm.

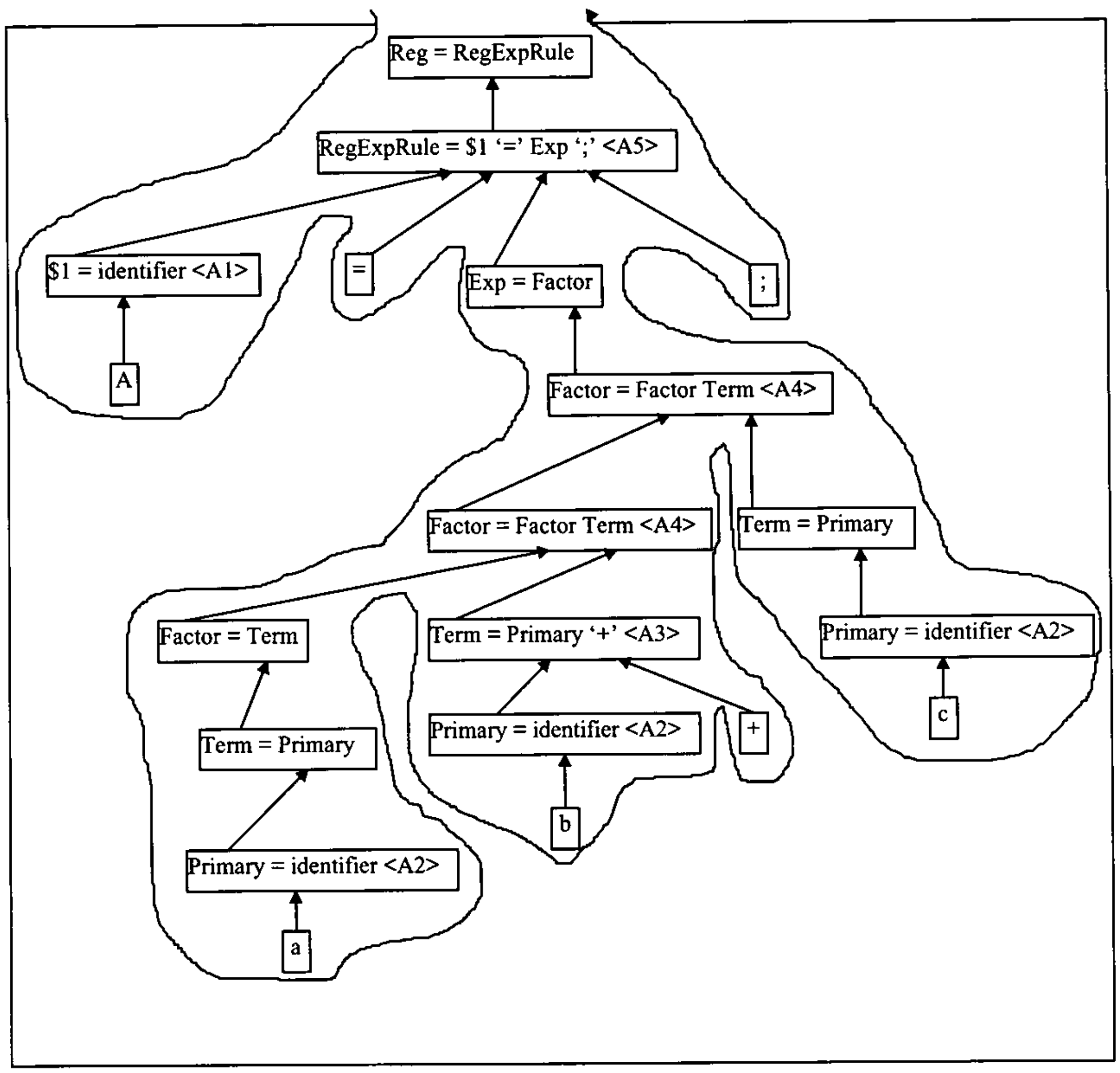
Reg = RegExpRule
RegExpRule = $1 '=' Exp ';' <A5>
$1 = identifier <A1>
A
=
Exp = Factor
;
Factor = Factor Term <A4>
Factor = Factor Term <A4>
Term = Primary
Primary = identifier <A2>
c
Factor = Term
Term = Primary
Primary = identifier <A2>
a
Term = Primary '+' <A3>
Primary = identifier <A2>
b
+

Figure 38 - Logic Synthesis Parse Tree

Using the parse table for the grammar (given in the appendix), the input source text and referring to the definitions for the micro-instructions given in Chapter 3 "Instruction Set Design" it is possible to determine the sequence of calling the semantic actions.

For the given example the sequence of semantic routines will generate the following MODEL source code.

```
Part A [ clk,tokin ] -> res
  Signal n1;
  Signal n2;
```

```
 Signal n3;
 Signal n4;
 Signal n5;
 Signal n6;
 Signal n7;
 ONE -> n1
 token("a")[ clk,tokin,n1 ] -> n2
 n2 -> n5
 or[ n5,n4 ] -> n3
 token("b")[ clk,tokin,n3 ] -> n4
 n4 -> n6
 token("c")[ clk,tokin,n6 ] -> n7
 n7 -> res
End
```

After eliminating wires with duplicate names, this can be written as :-

```
Part A [ clk,tokin ] -> res
 Signal n3;
 Signal n5;
 Signal n6;

 token("a")[ clk,tokin,ONE ] -> n5
 or[ n5,n6 ] -> n3
 token("b")[ clk,tokin,n3 ] -> n6
 token("c")[ clk,tokin,n6 ] -> res
End;
```

This can be represented by the following logic diagram.

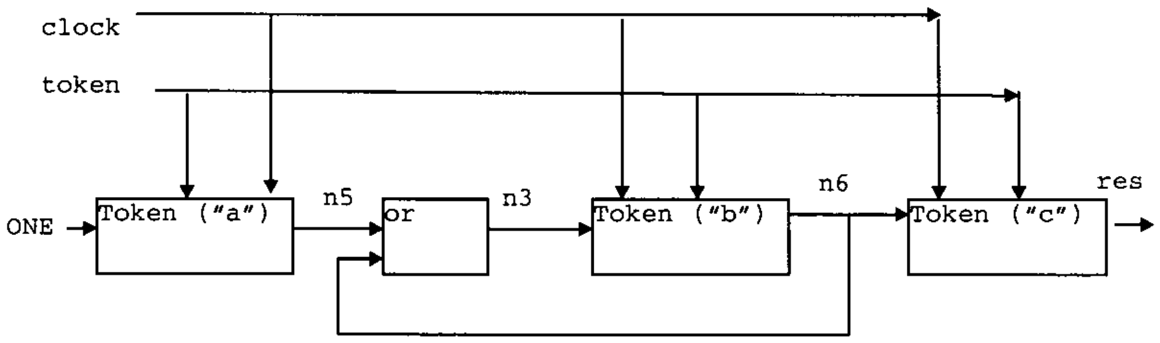

Figure 39 - Synthesised Logic

Identical logic to recognise the expression A = a b+ c could also have been generated using the algorithm discussed in Chapter 2 "Hardware Implementations". Thus it is possible that a simple logic synthesis tool could be implemented using the processor.

5.2 Device Mask Generation

The manufacture of most integrated circuits depends on the use of photo-lithography to generate the masks describing the physical layout of integrated circuits. Each mask consists of a collection of geometric shapes, where the shapes can be formed from the combination of primitive geometric shapes such as a circle, rectangle or trapezium. This physical layout can be described using a number of specially designed languages. One of these languages is the Caltech Intermediate Form (CIF), which is widely used.

CIF can be easily defined using an LALR(1) grammar and thus can be parsed by the processor using the tables generated by the compiler-compiler. For a simple example CIF file, the processor took an estimated 24110 clock-cycles whilst executing 10112 instructions and 208 semantic actions when reading 105 characters.

The processor could control the photo-lithography machine, by operating the photo-lithography camera aperture size and location directly from the semantic actions thus generating the appropriate shapes.

The use of direct write X-ray etching machines instead of photo-lithography for some masks could also be enhanced by the use of this processor. In this case the semantic actions control the operation of the X-ray beam directly.

5.3 JAVA

The use of the processor to accelerate the compilation of Java would also be an example of the potential for this device. The Java Language Specification described within [Gosling et al. 1996] includes an LALR(1) grammar for the Java language. This was converted into the regular grammar notation used for input to the Compiler-Compiler system. Initially, the size of the resulting JAVA grammar caused problems for the MS-DOS based compiler-compiler. This was caused by the memory required to store the parse and lex states being greater than that available under MS-DOS. The compiler-compiler was modified to use a different run-time environment which provided a larger memory range than MS-DOS. This modification enabled the compiler-compiler to generate a table of instructions for the processor which would allow JAVA source to be recognised.

The use of the RISC processor to recognise JAVA source code would reduce or possibly remove the need to transfer large files of pre-compiled JAVA byte code over the Internet. Noting that JAVA source code is smaller in size than the corresponding

JAVA byte code, this would reduce the data bandwidth needed by the Internet to support JAVA.

Further improvements in the speed of recognition of computer source code such as JAVA or PASCAL could be gained by extending the INTEL Pentium instruction set with the instruction set described by this thesis. Merging the instruction sets would not be expensive in terms of silicon area, since the first implementation of the RISC processor was in 1.5 micron and current Pentium processors use 0.25 micron technology. Using a system architecture of a single control device (3317 by 3076 microns) with two bit-slice devices (each 3777 by 3244 microns), where the given sizes are for a geometry of 1.5 micron would give an approximate increase of 1600 by 580 microns for a Pentium implemented in 0.25 micron geometry.

5.4 Pen Plotters

Pen plotters are examples of devices which can have a simple language to control their operation. A pen plotter has a range of simple commands which are used in combination to draw pictures (including text). Some of these commands are :-

- Pen Up
- Pen Down
- MoveTo
- Reset
- Home
- EndOfInput
- SelectPen
- LoadPaper
- EjectPaper

The structure of allowed command sequences could then be defined by an appropriate language LALR(1) grammar. Some of the re-write rules of the grammar will require actions to be performed. These actions, in turn, will interact with the physical world; such as, causing the movement of the pen from one location to another location. For the pen plotter, the recognition of a rule will cause an interrupt which will set/clear the signal values on the dataout bus of the processor. Thus the source text describing the diagram can interact with the plotter mechanisms and its logic circuitry via the processor.

5.5 Disk Controllers

Disk controllers provide an interface between a computer and the electronic hardware used to read and write the digital data on magnetic media. The behaviour of a disk controller is similar to that of the pen plotter described previously. The disk controller operates the movement across the disk surface of the disk read/write heads usually via a stepper motor. This corresponds to the MoveTo, HeadUp and HeadDown pen plotter commands.

5.6 Machine Tools (DNC)

Machine tools in engineering are used to drill holes and grind and route surfaces for sheet materials such as steel, titanium, tin or even plastic. The operation of a modern machine tool is usually controlled by a computer-like device with files written in a special Numerical Control (NC) language being used as source. The NC source data (usually referred to as a "tape") describes the tool operations in a similar notation to that used by a pen plotter. Each tool can be regarded as being equivalent to a pen, which can be selected, moved to a given set of co-ordinates (x, y, z) and have the speed of the tool also selected.

6. Concluding Remarks and Future Work

The research project reported on in this thesis was dedicated to the problem of accelerating the process of parsing and lexical analysis. Almost all parsing and lexical analysis is performed on general purpose computers which add time overheads to the joint processes. The main objective of this PhD research was to develop new mechanisms by which the parsing and lexical processes could be accelerated. The research was carried out in two main areas, namely the investigation of appropriate algorithms to form the basis of a hardware accelerator and the physical implementation of the hardware accelerator. In this chapter of the thesis, a summary of the original contributions of the thesis is presented. Also, possible research areas in which future work could be carried out are discussed.

6.1 Summary of Contributions Made by This Thesis

This thesis has presented the following contributions to the field of parsing and lexical analysis:

1) A novel processor instruction set (containing 24 instructions) has been defined which has sufficient instructions to be able to execute a combined parsing and lexical analysis. A novel feature of the instruction set is its ability to extend the size of instructions parameters. The design of the instruction set involved investigation of the LALR(1) parsing algorithm and finite state machine lexical analysis algorithms to determine the primitive operations which could be implemented as instructions.

2) A VLSI chip set has been fabricated which is able to execute the defined instruction set. A novel feature of this chip set is its ability to activate the semantic actions (required by a language) directly from the hardware. The chip set is implemented in 1.5 micron CMOS technology with the data-paths implemented using the bit-slice technique.

6.2 Future Work

In this PhD project, a specialised processor has been implemented which can be used to accelerate the combined process of parsing and lexical analysis. However parsing and lexical analysis are front-end mechanisms used to trigger the correct sequence of semantic actions. The author believes that further research opportunities could result from investigations into the design of general purpose logic able to perform special semantic actions in co-operation with the processor. Research areas which could be worth investigation are outlined below.

6.2.1 Semantic Hardware

One feature of the processor is its ability to directly trigger semantic actions using the combination of SysIRQ signal and DataOut bus signals and also its ability to output the relevant token string as a character sequence. This provides opportunities to investigate the design and use of specific hardware able to work in co-operation with the processor.

6.2.1.1 Symbol Tables

A symbol table is used by a compiler to hold information about the identifiers or variables defined by a program, where the information usually includes the identifier name, its type (whether integer, character, record, etc.) and scope of visibility.

Possible research would investigate if a general purpose symbol table was possible and if so, to then design appropriate logic to implement it.

The symbol table implemented as hardware would be an example of a sub-unit of semantic hardware directly controlled by the processor.

6.2.1.2 Code Generators

Most compilers are used to convert source text expressed in a given language into executable code for a target machine. Research has already been carried out into general purpose mechanisms for converting intermediate code generated by a language parser into machine specific executable code.

Possible further research could involve investigations using the processor to generate the intermediate code. Further research could investigate if hardware could be implemented to transform intermediate code into true machine code.

6.2.2 Software

Although the processor is a hardware device it depends heavily on the compiler-compiler software to generate the parse and lexical analysis tables for each formal language being recognised. The following areas of research relate to the software aspects of the processor.

6.2.2.1 Optimisations

The state tables which form the processor instructions are generated without regard for optimisation. The ordering of the instructions within the tables could be altered by analysis of the grammar taking into account the range of possible language sentences, or possible programs. The re-ordering would be focused on the possible optimisations which would reduce the time taken by the processor to parse a range of sentences.

Another possible optimisation would involve the reduction of the table sizes.

One possible future research topic could investigate this possibility.

6.2.2.2 Re-ordering the Rules

The grammar rules for a language are defined in numerical order and the values passed out to the DataOut bus when SysIRQ is active reflect that fact. Thus the associated semantic hardware has to decode the complete set of DataOut signals to determine which logic sub-block is to be activated. A possible re-ordering of the rules by associated function, such as grouping all rules which refer to the symbol table logic could be performed.

Investigating this possibility could be a further research topic carried out in conjunction with the research into symbol table hardware.

6.2.2.3 Allowing larger grammars

Further work is needed on the current version of the compiler-compiler software to overcome the memory limitations imposed by MS-DOS which limits the size of grammars which can be read. As the compiler-compiler uses a software emulation of the processor, instead of re-writing the software, it may be possible to research the conversion of the compiler-compiler software into hardware, thus providing a universal compiler-compiler system.

References

1. Aho, A. V. and Ullman, J. D., 1977, "Principles of Compiler Design", (Addison-Wesley)

2. Altera Corporation, 1995, "MAX+PLUS II: Getting Started", (Altera)

3. Altera Corporation, 1995, "MAX+PLUS II: AHDL", (Altera)

4. Altera Corporation, 1996,"1996 Data Book", (Altera)

5. Ayres, R., 1983, "VLSI Silicon Compilation and the Art of Automatic MicroChip Design", (Prentice-Hall)

6. Brown, P. J., 1981, "Writing Interactive Compilers and Interpreters", (John Wiley and Sons)

7. Chinitz, M., 1981, "The Logic Design of Computers", (Howard Sams)

8. Chu, Y., ed., 1975, "High Level Language Computer Architecture", (Academic Press)

9. Denning, P., Dennis, J., and Qualitz J., 1978, "Machines, Languages, and Computation", (Prentice-Hall)

10. DeRemer, F., 1971, "Simple LR(k) Grammars",Communications of the ACM, **14**, P453-460

11. Downs, T., and Shultz, M., 1988, "Logic Design with Pascal, Computer-Aided Design Techniques",(Van Nostrand Reinhold)

12. European Silicon Structures, 1993, "Solo 1400 User Guide", (European Silicon Structures)

13. Evans, R. A. and Morrison J. D., 1985, "Architectures for Language Recognition", Integration, the VLSI journal, **3**, P175-187, (North-Holland)

14. Fischer, C. N. and LeBlanc, R. J., 1991, "Crafting a Compiler with C", (Benjamin/Cummings Publishing)

15. Gosling, J., Joy, B. and Steele, G., 1996, "The Java Language Specification", (Prentice-Hall)

16. Harbison, S. P. and Steele, G., 1984, "C, A Reference Manual", (Prentice-Hall)

17. Hunter, R., 1981, "The Design and Construction of Compilers", (John Wiley and Sons)

18. Iliffe, J. K., 1982, "Advanced Computer Design", (Prentice-Hall)

19. Johnson, S., and Lesk, M., 1978, "Unix Time-Sharing System: Language Development Tools", The Bell System Technical Journal, **57**, P2155-2175

20. Kazuo Seo, Masaharu Hirayama and Akira Fusaoka, 1983, "Design and Evaluation of Parsing Chip", VLSI-83, P317-326, (North-Holland, Amsterdam)

21. Knuth, D., 1966, "On the Translation of Languages from Left to Right", Information and Control, **8**, P607-639

22. Lewin, D., 1974, "Logical Design of Switching Circuits", (Nelson)

23. McCluskey, E. J., 1986, "Logic Design Principles: with emphasis on testable semicustom circuits", (Prentice-Hall)

24. McGettrick, A. D., 1980, "The Definition of Programming Languages", (Cambridge University Press)

25. McMullin, J. D., 1996, "Implementing a Parser and Lexical Analyser using FPGA technology", Electronic Engineering, **68**, P44.

26. Mead, C. and Conway, L., 1980, "Introduction to VLSI Systems", (Addison-Wesley)

27. Miczo, A., 1987, "Digital Logic Testing and Simulation", (John Wiley)

28. Minsky, M., 1972, "Computation: Finite and Infinite Machines", (Prentice-Hall)

29. Pager, D., 1977, "The Lane-Tracing Algorithm for Constructing LR(k) Parsers and Ways of Enhancing Its Efficiency", Journal of Information Sciences, **12**, P19-42, (North-Holland)

30. Patterson, D., and Hennessy, J., 1990, "Computer Architecture. A Quantative Approach", (Morgan Kaufmann)

31. Pollard, L., 1990, "Computer Design and Architecture", (Prentice-Hall)

32. Preston, J. V. and Lofgren, J. D., 1994, "FPGA Macros Simplify State Machine Design", Electronic Design, December, P109-118.

33. Pucknell, D. A., 1990, "Fundamentals of Digital Logic Design: with VLSI Circuit Applications", (Prentice-Hall)

34. Staken, P., 1996, "A Practitioner's Guide to RISC Microprocessor Architecture", (John Wiley)

35. Sze, S. M., ed., 1983, "VLSI Technology", (McGraw-Hill)

36. van Eekelen, M., Huitema, H., Nocker, E., Smetsers, S. and Plasmeijer, R., 1993, "Concurrent Clean, Language Manual", Technical Report 93-13, (Katholieke Universteit Nijmegen)

37. Weste, N., and Eshraghian K., 1985, "Principles of CMOS VLSI Design: A Systems Perspective", (Addison-Wesley)

7. Appendix A - Software Simulation

Each run of the software emulation of the processor is capable of generating a set of test vectors and expected outputs. A number of sets of these test vectors were used to drive the SOLO 1400 MADS logic simulator. The waveform results from each run (only examining the values at the time of clock rise and fall) were compared with the expected results. The simulator results and the predicted results from the processor emulation were found to match, giving a high level of confidence in the implemented logic design. Part of a sample simulation run for the control chip is shown below.

7.1 Main Simulation File - The Template

The following is the contents of the main file used to drive the MADS simulator. The include file "control.vec" contains the output from the processor emulation providing test vectors and predicted outputs. This file will be different for each run of the processor emulation. The include file "sim.h" contains utility code to convert the data in the control.vec file into commands which can drive the simulator.

```
#include "sim.h"
#include "control.vec"

main()
{

 vector_step = 1000;
 tick = 0;

 Set_Cycle(vector_step);

 extclock = 1;
 extsysreset = 1;
 extsyscommand0 = 0;
 extsyscommand1 = 0;
 extsyscommand2 = 0;
 extsymbolequal = 0;
 extsymbolge = 0;
 exttspiz = 0;
 extsspiz = 0;
 extp1iz = 0;
 exttie = 0;
 exttre = 0;
 exttbe = 0;
 extinst0 = 0;
 extinst1 = 0;
 extinst2 = 0;
 testdevice;

 Simulate;
}
```

7.2 Simulator Utility Code

This file contains utility functions to convert the data from a processor simulation run into commands to drive the MADS simulator. The file also holds information about the names of the external pins, if they are inputs or outputs, and the legal values of the various command signals.

```
Input  extclock;
Input  extsysreset;
Input  extsyscommand0;
Input  extsyscommand1;
Input  extsyscommand2;
Input  extsymbolequal;
Input  extsymbolge;
Input  exttspiz;
Input  extsspiz;
Input  extp1iz;
Input  exttie;
Input  exttre;
Input  exttbe;
Input  extinst0;
Input  extinst1;
Input  extinst2;
Output extengcommand0;
Output extengcommand1;
Output extengcommand2;
Output extengcommand3;
Output extengcommand4;
Output extmemorypage0;
Output extmemorypage1;
Output extinstmemen;
Output extdatamemen;
Output extmemorywrite;
Output extinstwanted;
Output extdatawanted;
Output extdatadma;
Output extparsedone;
Output extirq;
Output extsourceused;
Output extparsyntax;
Output extparsemantic;
Output extlexsyntax;
Output extstackunder;
Output extstackover;
Output extillegalinst;
Output extnoerrhandle;
Output extbufferover;

int vector_step;
int tick;

int cSysNoOp        = 0;
int cSysStateWanted  = 1;
int cSysTokenWanted  = 2;
int cSysIRQ_NullToken = 3;
int cSysIRQ_OK       = 4;
int cSysIRQ_ERR      = 5;
int cSysDataAvailable = 6;
int cSysInstAvailable = 7;

int cEngineNoOp              = 0;
int cEngineTokenSetBusMayBe   = 1;
int cEngineTokenSetBusBuffer  = 2;
int cEngineTokenSetBusIs      = 3;
int cEngineTokenSetBusWas     = 4;
int cEngineTokenSetIsEmpty    = 5;
int cEngineTokenLoadWasIs     = 6;
int cEngineZero               = 7;
int cEngineTokenSPZero        = 8;
int cEngineSymbolLoadPC       = 9;
int cEngineTokenAccept        = 10;
int cEngineP1Load             = 11;
int cEngineSymbolLoadLAS      = 12;
int cEngineSymbolPop          = 13;
int cEngineSymbolPush         = 14;
int cEngineSymbolLoadTS       = 15;
int cEngineStateSPInc         = 16;
int cEngineTokenSPInc         = 17;
int cEngineTokenIncRamStart   = 18;
int cEngineSymbolInc          = 19;
int cEngineTokenIncBufferStart = 20;
int cEngineTokenIncBufferEnd  = 21;
int cEngineTokenIncIsStart    = 22;
int cEngineTokenIncIsEnd      = 23;
int cEngineStateSPDec         = 24;
int cEngineTokenSPDec         = 25;
int cEngineTokenDecRamStart   = 26;
int cEngineSymbolDec          = 27;
int cEngineTokenDecBufferStart = 28;
int cEngineTokenDecBufferEnd  = 29;
int cEngineTokenDecIsStart    = 30;
int cEngineTokenDecIsEnd      = 31;

/* simple simulation step */
void simstep()
{
 Toggle(extclock);
 Next_Cycle;

 Toggle(extclock);
 Next_Cycle;
}

/* General purpose command line set-up */
void setcommand( int syscommand )
{
 extsyscommand0 = ((syscommand     ) & 1);
 extsyscommand1 = ((syscommand >> 1) & 1);
 extsyscommand2 = ((syscommand >> 2) & 1);
}
```

```
/* General purpose mode line set-up */
void setinst( int inst)
{
 extinst0 = ((inst     ) & 1);
 extinst1 = ((inst >> 1) & 1);
 extinst2 = ((inst >> 2) & 1);
}

/* General purpose command line set-up */
void docommand( int syscommand,
            int inst,
            int sysreset,
            int symbolequal,
            int symbolge,
            int tokenspiszero,
            int statespiszero,
            int p1iszero,
            int tokenisequal,
            int tokenramequal,
            int tokenbufferequal )
{

 tick = tick + 1;
 setcommand( syscommand );
 setinst( inst );
 extsysreset    = ((sysreset) & 1);
 extsymbolequal = ((symbolequal) & 1);
 extsymbolge    = ((symbolge) & 1);
 exttspiz       = ((tokenspiszero) & 1);
 extsspiz       = ((statespiszero) & 1);
 extp1iz        = ((p1iszero) & 1);
 exttie         = ((tokenisequal) & 1);
 exttre         = ((tokenramequal) & 1);
 exttbe         = ((tokenbufferequal) & 1);

 simstep;
}

int SigToInt( Output a )
{
 return a;
}

void cb( char *mess, Output actual, int expected )
{
 int actualbar;
/* code doe NOT correctly check the expected v actual signal values */
/* dom't know if signal values are set up correctly or what */
/* hence this code is commented out */

/*
 actualbar = (~(SigToInt(actual) & 1) & 1);

 if (actualbar == expected)
 {
  printf("Mis-match at tick %d for ",tick);
  printf(mess);
  printf("\n");
 }
*/
}

/* Test Vector Check Routines */
void checkvectors(int enginecommand,
            int memorypage,
            int instmemenable,
            int datamemenable,
            int memorywrite,
            int instwanted,
            int datawanted,
            int datadma,
            int parsedone,
            int irq,
            int sourceexhausted,
            int parsesyntax,
            int parsesemantic,
            int lexsyntax,
            int stackunderflow,
            int stackoverflow,
            int illegalinstruction,
            int noerrorhandler,
            int bufferoverflow )
{
 cb("enginecommand0",extengcommand0, ((enginecommand     ) &1) );
 cb("enginecommand1",extengcommand1, ((enginecommand >> 1) &1) );
 cb("enginecommand2",extengcommand2, ((enginecommand >> 2) &1) );
 cb("enginecommand3",extengcommand3, ((enginecommand >> 3) &1) );
 cb("enginecommand4",extengcommand4, ((enginecommand >> 4) &1) );

 cb("memorypage0",extmemorypage0, ((memorypage     ) &1) );
 cb("memorypage1",extmemorypage1, ((memorypage >> 1) &1) );

 cb( "InstMemEnable", extinstmemen, instmemenable );
 cb( "DataMemEnable", extdatamemen, datamemenable );
 cb( "MemoryWrite", extmemorywrite, memorywrite );
 cb( "InstWanted", extinstwanted, instwanted );
 cb( "DataWanted", extdatawanted, datawanted );
 cb( "DataDMA", extdatadma, datadma );
 cb( "ParseDone", extparsedone, parsedone );
 cb( "IRQ", extirq, irq );
 cb( "SourceExhausted", extsourceused, sourceexhausted );
 cb( "ParseSyntax", extparsyntax, parsesyntax );
 cb( "ParseSemantic", extparsemantic, parsesemantic );
 cb( "LexSyntax", extlexsyntax, lexsyntax );
 cb( "StackUnderflow", extstackunder, stackunderflow );
 cb( "StackOverflow", extstackover, stackoverflow );
 cb( "IllegalInstruction", extillegalinst, illegalinstruction );
 cb( "NoErrorHandler", extnoerrhandle, noerrorhandler );
 cb( "BufferOverflow", extbufferover, bufferoverflow );

}
```

7.3 Processor Emulation Data

The include file "control.vec" contains the actual test vector and expected results.

The following is a small fragment of an actual file.

```
/* exercise the device */
void testdevice()
{
/* At tick 1 */
 docommand( cSysNoOp,0,1,1,1,0,0,1,0,0,0 );
 checkvectors( cEngineP1Load,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 2 */
 docommand( cSysNoOp,0,1,1,1,1,1,1,1,1,1 );
 checkvectors( cEngineP1Load,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 3 */
 docommand( cSysNoOp,0,1,1,1,1,1,1,1,1,1 );
 checkvectors( cEngineP1Load,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 4 */
 docommand( cSysNoOp,0,1,1,1,1,1,1,1,1,1 );
 checkvectors( cEngineP1Load,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 5 */
 docommand( cSysNoOp,0,1,1,1,1,1,1,1,1,1 );
 checkvectors( cEngineP1Load,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 6 */
 docommand( cSysNoOp,0,0,1,1,1,1,1,1,1,1 );
 checkvectors( cEngineNoOp,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 7 */
 docommand( cSysNoOp,0,0,1,1,1,1,1,1,1,1 );
 checkvectors( cEngineP1Load,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 8 */
 docommand( cSysNoOp,0,0,1,1,1,1,1,1,1,1 );
 checkvectors( cEngineStateSPInc,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 9 */
 docommand( cSysNoOp,0,0,0,0,1,1,0,1,1,1 );
 checkvectors( cEngineNoOp,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 10 */
 docommand( cSysNoOp,0,0,0,0,1,0,0,1,1,1 );
 checkvectors( cEngineTokenLoadWasIs,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 11 */
 docommand( cSysNoOp,0,0,0,0,1,0,0,1,1,1 );
 checkvectors( cEngineP1Load,2,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );
/* At tick 12 */
 docommand( cSysNoOp,0,0,0,0,1,0,0,1,1,1 );
 checkvectors( cEngineTokenSPZero,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 );

/* Values after tick 12 have been deleted from the file */
}
```

7.4 Simulator results

The following is a fragment from the MADS simulator waveform output file which corresponds to the control.vec file fragment given in the previous section.

The MADS simulator is an event driven simulator and hence the output file shows when the output signals changed. However the information provided by the processor emulation software is based on a cycle based emulation. Therefore only those lines where the clock edge goes high or low will be of interest.

Note that the column headings have been removed and are indicated in the following table.

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
1	extclock	11	exttie	21	extengcommand4	31	extirq
2	extsysreset	12	exttre	22	extmemorypage0	32	extsourceused
3	extsyscommand0	13	exttbe	23	extmemorypage1	33	extparsyntax
4	extsyscommand1	14	extinst0	24	extinstmemen	34	extparsemantic
5	extsyscommand2	15	extinst1	25	extdatamemen	35	extlexsyntax
6	extsymbolequal	16	extinst2	26	extmemorywrite	36	extstackunder
7	extsymbolge	17	extengcommand0	27	extinstwanted	37	extstackover
8	exttspiz	18	extengcommand1	28	extdatawanted	38	extillegalinst
9	extsspiz	19	extengcommand2	29	extdatadma	39	extnoerrhandle
10	extp1iz	20	extengcommand3	30	extparsedone	40	extbufferover

```
Timing diagram of loaded/ecpd10/ind/max/chip.trc


                           1                   2                   3                   4
           1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0

   0.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 X X X X X X X X X X X X X X X X X X X X X X X X
   1.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 X X X X X X X X X X X X X X X X X X X X X X X X
    ...    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 X X X X X X X X X X X X X X X X X X X X X X X X
  12.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 X X X X X X X X X X X X X 0 X X X X X X X X X X
  13.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 X X X X X X X X X X X X X 0 X X X X X X X X X X
    ...    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 X X X X X X X X X X X X X 0 X X X X X X X X X X
  20.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 X 1 X X X X X X X X X X X 0 X X X X X X X X X X
  21.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 X 1 X X X X X X X X X X X 0 X X X X X X X X X X
    ...    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 X 1 X X X X X X X X X X X 0 X X X X X X X X X X
  24.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 X X X X X X X X X X 0 X X X X X X X X X X
  25.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 X X X X X X X X X X 0 X X X X X X X X X X
    ...    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 X X X X X X X X X X 0 X X X X X X X X X X
  27.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 X 0 X X X X X X X X 0 X X X X X X X X X X
  28.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 X X X X X X X X 0 X X X X X X X X X X
  29.00    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 X X X X X X X X 0 X X X X X X X X X X
    ...    0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 X X X X X X X X 0 X X X X X X X X X X
1000.00    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 X X X X X X X X 0 X X X X X X X X X X
1001.00    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 X X X X X X X X 0 X X X X X X X X X X
    ...    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 X X X X X X X X 0 X X X X X X X X X X
1011.00    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 X X X X X X X X 0 X X X 0 0 X X X X X
1012.00    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 X X X X X X X X 0 0 X X 0 0 0 0 0 0 0
1013.00    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 X X X X X X X X 0 0 0 0 0 0 0 0 0 0 0
1014.00    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 X X X 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1015.00    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 X X 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1016.00    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1017.00    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    ...    1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2000.00    0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
 2001.00   0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 3000.00   1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 3001.00   1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 4000.00   0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 4001.00   0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 5000.00   1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 5001.00   1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 6000.00   0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 6001.00   0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 7000.00   1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 7001.00   1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 8000.00   0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 8001.00   0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9000.00   1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9001.00   1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10000.00   0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10001.00   0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 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 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10020.00   0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10021.00   0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 X 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 1 1 1 1 1 1 1 1 0 0 0 1 1 1 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10025.00   0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10026.00   0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1 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 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11000.00   1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11001.00   1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11025.00   1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11026.00   1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ...     1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11028.00   1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11029.00   1 0 0 0 0 1 1 1 1 1 1 1 1 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 0 0
11030.00   1 0 0 0 0 1 1 1 1 1 1 1 1 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 0 0
   ...     1 0 0 0 0 1 1 1 1 1 1 1 1 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 0 0
12000.00   0 0 0 0 0 1 1 1 1 1 1 1 1 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 0 0
```

8. Appendix B - Processor Implementation

The logic for the control device was generated using the logic synthesis tools provided within the Solo 1400 design system. The PASCAL source code for the control state machine was modified to become valid input text for the logic synthesis tool LOLA.

The following MODEL code defines an individual bit-slice element which is repeated eight times within the actual bit-slice device. For reasons of clarity, the code has been altered to remove any buffer logic which was only added to reduce excessive gate loading. Also not shown is the decode logic which generates the appropriate loadXXX signals from the mcode signal bus.

```
Part loadbit[clk,d,load] -> q
  bdff[clk,or[and[load,d],and[not[load],q]]] -> q
End

Part slice[clock,
        mcode0,mcode1,mcode2,mcode3,mcode4,
        iomcode0,iomcode1,iomcode2,
        loadtrs,
        loadtre,
        loadtbs,
        loadtbe,
        loadtis,
        loadtie,
        loadtws,
        loadtwe,
        loadpc,
        loadp1,
        loadtopsymbol,
        loadlookaheadsymbol,
        loadstatesp,
        loadtokensp,
        carryin,
        trein,tbein,tiein,tspizin,sspizin,
        p1izin,sgein,sein,
        memorydata] -> carryout,
                treout,tbeout,tieout,
                tspizout,
                sspizout,
                p1izout,sgeout,seout,
                dataout,address
Signal trs,tre,
     tbs,tbe,
     tis,tie,
    tws,twe,
    pc,
    p1,
    topsymbol,
    lookaheadsymbol,
    tokensp,
    statesp,
    ebus,
    cbus,
    sbus,
    sumbus,
    datainbus,
    topsymbolmatch,
    tokenismatch,
    tokenbuffermatch,
    tokenrammatch

data[mcode0,mcode1,mcode2,mcode3,mcode4,memorydata] -> datainbus
ebit[p1, mcode0, mcode1, mcode2, mcode3, mcode4,
    lookaheadsymbol, pc, statesp, tbe,
    tbs, tie, tis, trs,
    tokensp, twe, topsymbol] -> ebus
cbit[mcode3, mcode4, ebus] -> cbus
exor[carryin,ebus] -> sumbus

sbit[p1, mcode0, mcode1, mcode2, mcode3, mcode4,
    datainbus, sumbus, tbs, tie, tis,tws] -> sbus

loadbit[clock,sbus,loadtrs] -> trs
loadbit[clock,ebus,loadtre] -> tre
loadbit[clock,sbus,loadtbs] -> tbs
loadbit[clock,sbus,loadtbe] -> tbe
loadbit[clock,sbus,loadtis] -> tis
loadbit[clock,sbus,loadtie] -> tie
loadbit[clock,sbus,loadtws] -> tws
loadbit[clock,ebus,loadtwe] -> twe
loadbit[clock,sbus,loadpc] -> pc
loadbit[clock,sbus,loadp1] -> p1
loadbit[clock,ebus,loadtopsymbol] -> topsymbol
loadbit[clock,ebus,loadlookaheadsymbol] -> lookaheadsymbol
loadbit[clock,sbus,loadstatesp] -> statesp
loadbit[clock,sbus,loadtokensp] -> tokensp

{ Ripple through logic }
and[carryin,cbus] -> carryout

eqv[topsymbol,p1] -> topsymbolmatch
eqv[tis,tie] -> tokenismatch
eqv[tbs,tbe] -> tokenbuffermatch
eqv[trs,tre] -> tokenrammatch

and[not[p1],p1izin] -> p1izout
and[not[statesp],sspizin] -> sspizout
and[not[tokensp],tspizin] -> tspizout
or[and[topsymbol,not[p1]],and[topsymbolmatch,sgein]] -> sgeout
and[topsymbolmatch,sein] -> seout
and[tokenismatch,tiein] -> tieout
and[tokenbuffermatch,tbein] -> tbeout
and[tokenrammatch,trein] -> treout
```

```
 iocode[iomcode0,iomcode1,iomcode2,
      pc,statesp,trs,tokensp] -> address

 p1 -> dataout
End

Endoffile
```

9. Appendix C - Synthesis Software

This appendix contains further details about the logic synthesis language which was referenced in section 5.1 “Logic Synthesis”.

9.1 Examples of Regular Expressions

The following are examples of regular expressions which have been used as input to the software emulation of the processor.

Note :-

the + operator is used to indicate repetition at least once,

the ? operator is used to denote optional inclusion,

the * operator is equivalent to the *? operators in combination,

the , operator indicates choice between two alternatives.

Also the use of brackets () is used to change priority of operations.

9.1.1 Example 1

```
A = a b+ c ;
```

This describes a regular expression A which is shorthand for

A = { a b c , a b b c, a b b b c, ...}

The b+ represents the repetition of b at least once.

9.1.2 Example 2

```
A = a b* c ;
B = c m+ (a,d+)? ;
```

This describes two separate regular expressions A and B which are shorthand for

A = { a c, a b c, a b b c, ...}

B = { c m, c m a, c m d, c m d d, ... , c m m, c m m a, c m m d, c m m d d, ...}

9.2 Grammar for Synthesis Language

The examples of regular expressions in the previous section can be described by an LALR(1) grammar. The following text, using the regular expression notation, defines such a grammar. This grammar can and has been input to the compiler-compiler software to generate tables for use with the processor.

Note that the parse grammar is defined by the section commencing $parser and terminated by $lexer. The lexical structure of tokens is described from the $lexer to the end of file.

```
!{ ************************************************************** }
!{ *                                                            * }
!{ * Copyright (c) 1996 J.D.McMullin. All rights reserved. * }
!{ *                                                            * }
!{ ************************************************************** }
$parser Reg
Reg = RegExpRule + ;
RegExpRule = identifier <leftnameis> '=' Exp ';' <ruleis> ;
Exp = Factor (',' Factor <expagain>)* ;
Factor = Term (Term <factoragain>)* ;
Term = Primary ,
     Primary '+' <repeatisplus> ,
     Primary '*' <repeatisstar> ,
     Primary '?' <repeatisquery> ;
Primary = identifier <primaryisid> ,
       '(' Exp ')' ;
!
! now to define the lexical items
! remember $eoi MUST be present and also $whitespace MAY be present
!
$lexer
eof = 26 ;
rs = 30 ;
tab = 9 ;
$eoi = eof ;
$whitespace = (space, '!' commentchar* rs) <commentfound> ;
space = (' ', tab, rs)+ ;
commentchar = tab, [' '..'~'] ;
identifier = idchar (idchar, "_", ["0".."9"])* ;
idchar = ["A".."Z"], ["a".."z"] ;
```

9.3 Semantic Actions

The following text is the full Turbo Pascal source used to implement the semantic actions referenced in the grammar defined in the previous section. This code was generated as a template by the compiler-compiler. The semantic actions have been manually added.

The semantic actions are used to convert regular expressions into MODEL logic descriptions. This MODEL code can then be input to the SOLO 1400 design suite (as was used to design the processor) to form a logic design.

```
{ ************************************************************ }
{ * This software was generated by J.D.McMullin as an * }
{ * integral part of his M.Phil, Ph.D research.        * }
{ ************************************************************ }
unit regAct;
interface
uses share,parser;
type
 ErrStringfn = function ( errno:integer16 ): lexstring;

 function RegGetChar( h:pointer ): char;
 function RegParseFile(
       FileName,Extension : lexstring;
       var ParseHandle : pointer;
       ReadTable : Readfn;
       ReadTableMax : ReadMaxfn;
       DoAction : SemanticActionfn;
       ReadChar : GetCharfn;
       ReadErrString : ErrStringfn ) : boolean;
 function RegParseError( h,h1:pointer ):boolean;
 function RegLexError( h,h1:pointer ):boolean;
 function LeftNameIs( h,h1:pointer ):boolean;
 function RuleIs( h,h1:pointer ):boolean;
 function ExpAgain( h,h1:pointer ):boolean;
 function FactorAgain( h,h1:pointer ):boolean;
 function RepeatIsPlus( h,h1:pointer ):boolean;
 function RepeatIsQuery( h,h1:pointer ):boolean;
 function RepeatIsStar( h,h1:pointer ):boolean;
 function PrimaryIsId( h,h1:pointer ):boolean;

 function CommentFound( h,h1:pointer ):boolean;

implementation
type
 cType = (cToken,cOr,cRename);
 RegPtr = ^RegParseState;
 RegParseState = record
            { lexical input/output stream variables }
            StreamPos,
            StreamLineNumber : integer32;
            StreamInName,
            StreamInExt,
            StreamOutName,
            StreamOutExt,
            StreamBuffer : lexstring;
            StreamIn,
            StreamOut : text;
            ErrorLineNumber : integer32;
            { Error Flags to indicate Parse Error has occurred }
            ErrorDetected : boolean;
            { "class method" }
            FetchErrString : ErrStringfn;
            { User-defined variables below }
            lexsp : integer;
```

```
        lexstack : array [1..100] of record
                              n1,
                              n2,
                              c1,
                              c2 : integer32;
                             end;
        cellcount : integer;
        cellstack : array [1..200] of record
                              t : cType;
                              r : integer32;
                              n : lexstring;
                              i1,
                              i2,
                              o1 : integer32;
                             end;
        TheGlobalNode : integer32;
        TheGlobalCell : integer32;
        LeftSymbol : lexstring;
       end;

{ Global Action Routine variables }

{ routine to read lexical token string }
procedure CurrentToken( h:pointer; var s:lexstring);
var
 ch : char;
begin
 clearstring( s );
 TokenWanted( h );

 while ValidTokenChar( h, ch ) do
 begin
  { DMA read of LexCache to form LexToken }
  appendstringchar( s, ch );
 end;
 TokenAccepted( h );
end;

{ Routine to read individual chars from the input file }
function RegGetChar( h:pointer ): char;
var
 p : RegPtr;
 ch : char;
begin
 p := h;
 with p^ do
 begin
  while (stringlength(StreamBuffer) < 1) do
  begin
   StreamLineNumber := StreamLineNumber + 1;
   readstring( StreamIn, StreamBuffer );
   readln( StreamIn );
   StreamPos := 1;
   WriteString( StreamOut, '[' );
   WriteInteger( StreamOut, StreamLineNumber , 4 );
   WriteString( StreamOut, '] ' );
   WriteString( StreamOut, StreamBuffer );
   WriteLn( StreamOut );
   appendstringchar( StreamBuffer , rs);
   if eof( StreamIn ) then
   begin
```

```
        appendstringchar( StreamBuffer , eoi);
      end;
    end;

    ch := StreamBuffer[StreamPos];
    if (ch <> eoi) then
    begin
      StreamPos := StreamPos + 1;
      if (StreamPos > stringlength(StreamBuffer)) then
        clearstring( StreamBuffer );
    end;
  end; { with }
  RegGetChar := ch;
end;

function RegParseFile(
        FileName,Extension : lexstring;
        var ParseHandle : pointer;
        ReadTable : Readfn;
        ReadTableMax : ReadMaxfn;
        DoAction : SemanticActionfn;
        ReadChar : GetCharfn;
        ReadErrString : ErrStringfn ) : boolean;
var
  p : RegPtr;
  status : integer16;
  flag : boolean;
begin
  new( p );
  ParseHandle := p;

  with p^ do
  begin
    { User defined initialisation code for this parse pass }

    { Parser system initialisation code }
    { MODIFY WITH CAUTION }
    FetchErrString := ReadErrString;
    ErrorDetected := false;
    ErrorLineNumber := 0;

    clearstring( StreamBuffer );
    StreamLineNumber := 0;

    StreamInName := FileName;
    StreamInExt := Extension;
    StreamOutName := FileName;
    StreamOutExt := 'deb';

    openoldfile( StreamIn, StreamInName, StreamInExt );
    reset( StreamIn );

    opennewfile( StreamOut, StreamOutName, StreamOutExt );
    rewrite( StreamOut );

    Status := SyntaxEngine( ParseHandle, ReadTable, ReadTableMax, DoAction, ReadChar);
    if (Status = 0) then
    begin
      if ErrorDetected then
      begin
        write('**** Warning **** File ');
      writestring(output,StreamInName);
      writeln(' contained a semantic or lexical error' );
      flag := false;
    end
    else
    begin
      write('File ');
      writestring(output,StreamInName);
      writeln(' parsed OK' );
      flag := true;
    end;
  end
  else
  begin
    write('**** WARNING **** File ');
    writestring(output,FileName);
    writeln(' contained at least one Syntax Error' );
    writeparsestatus( output, Status );
    flag := false;
  end;

  closefile( StreamIn );
  WriteString( StreamOut, '**** EOF ****' );
  WriteLn( StreamOut );
  closefile( StreamOut );
 end; { with }

 RegParseFile := flag;
end;

function RegParseError( h,h1:pointer ):boolean;
var
 p : RegPtr;
 s : lexstring;
 i : integer32;
begin
 p := h;
 CurrentToken( h1,s );
 with p^ do
 begin
  ErrorDetected := true;

  WriteString( StreamOut, '      ' );
  for i := 1 to StreamPos - 1 do
  begin
   if (StreamBuffer[i] = tab) then
    WriteString( StreamOut, tab )
   else
    WriteString( StreamOut, ' ' );
  end;

  WriteString( StreamOut,'^ **** ERROR **** Found token ' );
  WriteString( StreamOut,s);
  WriteString( StreamOut,' but expecting ');
  WriteString( StreamOut,fetcherrstring( CurrentState(h1) ) );
  WriteLn( StreamOut );

  if (ErrorLineNumber = 0) then
  begin
   ErrorLineNumber := StreamLineNumber;
  end;
  end; { with }

  RegParseError := true;
end;

function RegLexError( h,h1:pointer ):boolean;
var
  p : RegPtr;
  s : lexstring;
  i : integer32;
begin
  p := h;
  CurrentToken( h1,s );
  with p^ do
  begin
    ErrorDetected := true;

    WriteString( StreamOut, '     ' );
    for i := 1 to StreamPos - 1 do
    begin
      if (StreamBuffer[i] = tab) then
        WriteString( StreamOut, tab )
      else
        WriteString( StreamOut, ' ' );
    end;

    WriteString( StreamOut,'^ **** ERROR **** Found token ' );
    WriteString( StreamOut,s);
    WriteString( StreamOut,' but expecting a legal token');
    WriteLn( StreamOut );

    if (ErrorLineNumber = 0) then
    begin
      ErrorLineNumber := StreamLineNumber;
    end;
  end; { with }

  RegLexError := true;
end;

{ Semantic Actions to be coded below }

{ Dont forget to add each one to the Turbo Pascal  }
{ interface definition                             }
{ Also, all semantic routines are defined as follows }

{ function xx( h,h1:pointer ):boolean;   }
{ var                                     }
{   p : LALRPointer;                      }
{   s : lexstring;                        }
{   flag : boolean;                       }
{ begin                                   }
{   p := h;                               }
{   CurrentToken( h1,s );                 }
{   flag := true;                         }
{                                         }
{   User-developed Code (may alter flag/p^ values)  }
{                                         }
{   xx := flag;                           }
{ end;                                    }
{ Also remember about WhiteSpace(h:pointer) when    }
{ removing $whitespace / comments                  }

function newcell( h:pointer ):integer32;
var
 p : RegPtr;
begin
 p := h;
 with p^ do
 begin
  TheGlobalCell := TheGlobalCell + 1;
  newcell := TheGlobalCell;
 end;
end;

function newnode( h:pointer ):integer32;
var
 p : RegPtr;
begin
 p := h;
 with p^ do
 begin
  TheGlobalNode := TheGlobalNode + 1;
  newnode := TheGlobalNode;
 end;
end;

function tokencell( h:pointer; s : lexstring; n1,n2 : integer32 ):integer32;
var
 p : RegPtr;
begin
 p := h;
 with p^ do
 begin
  CellCount := CellCount + 1;
  with CellStack[CellCount] do
  begin
   t := cToken;
   r := CellCount;
   n := s;
   i1 := n1;
   i2 := 0;
   o1 := n2;
  end;
  tokencell := CellCount;
 end;
end;

function orcell( h:pointer; n1,n2,n3 : integer32 ): integer32;
var
 p : RegPtr;
begin
 p := h;
 with p^ do
 begin
  CellCount := CellCount + 1;
  with CellStack[CellCount] do
  begin
   t := cOr;
   r := CellCount;
   n := '';
```

```
    i1 := n1;
    i2 := n2;
    o1 := n3;
   end;
   orcell := CellCount;
  end;
end;

function renamecell( h:pointer; n1,n2 : integer32 ): integer32;
var
  p : RegPtr;
begin
  p := h;
  with p^ do
  begin
   CellCount := CellCount + 1;
   with CellStack[CellCount] do
   begin
    t := cRename;
    r := CellCount;
    n := '';
    i1 := n1;
    i2 := 0;
    o1 := n2;
   end;
   renamecell := CellCount;
  end;
end;

procedure push( h:pointer; n1,n2,c1,c2 : integer32 );
var
  p : RegPtr;
begin
  p := h;
  with p^ do
  begin
   lexsp := lexsp + 1;
   lexstack[lexsp].n1 := n1;
   lexstack[lexsp].n2 := n2;
   lexstack[lexsp].c1 := c1;
   lexstack[lexsp].c2 := c2;
  end;
end;

procedure pop( h:pointer; var n1,n2,c1,c2 : integer32 );
var
  p : RegPtr;
begin
  p := h;
  with p^ do
  begin
   n1 := lexstack[lexsp].n1;
   n2 := lexstack[lexsp].n2;
   c1 := lexstack[lexsp].c1;
   c2 := lexstack[lexsp].c2;
   lexsp := lexsp - 1;
  end;
end;

function LeftNameIs( h,h1:pointer ):boolean;
var
```

```
  p : RegPtr;
  s : lexstring;
 begin
  p := h;
  with p^ do
  begin
   CurrentToken( h1, s );
   { initialise all data for this regular expression }
   LeftSymbol := s;
   TheGlobalNode := 0;
   TheGlobalCell := 0;
   CellCount := 0;
   lexsp := 0;
  end;
  LeftNameIs := true;
 end;

 procedure writecell( h : pointer; c : integer32 );
 var
  p : RegPtr;
 begin
  p := h;
  with p^ do
  begin
   with CellStack[c] do
   begin
    write(StreamOut,'  ');
    case t of
  cOr:  write(StreamOut,'or[ n',i1:1,',n',i2:1,' ]');
cRename:  write(StreamOut,'n',i1:1);
cToken:  write(StreamOut,'token(''',n,''')[ clk,tokin,n',i1:1,' ]');
    end;
    writeln(StreamOut,' -> n',o1:1);
   end;
  end;
 end;

 function RuleIs( h,h1:pointer ):boolean;
 var
  p : RegPtr;
  n1,n2 : integer32;
  c1,c2,c : integer32;
  i : integer32;
 begin
  p := h;
  with p^ do
  begin
   pop( h, n1,n2,c1,c2 );
   writeln(StreamOut,'Part ',LeftSymbol,' [ clk,tokin ] -> res');
   for i := 1 to TheGlobalNode do
   begin
    writeln(StreamOut,'  Signal n',i:1,';');
   end;
   writeln(StreamOut,'  ONE -> n',n1:1);
   c := c1;
   while (c <> c2) do
   begin
    writecell(h,c);
    c := cellstack[c].r;
   end;
   if (c1 <> c2) then writecell(h,c2);
```

```
    writeln(StreamOut,'  n',n2:1,' -> res');
    writeln(StreamOut,'End;');
    { reset the system for the next expression }
    TheGlobalNode := 0;
    TheGlobalCell := 0;
    CellCount := 0;
    lexsp := 0;

  end;
  RuleIs := true;
end;

function ExpAgain( h,h1:pointer ):boolean;
var
  p : RegPtr;
  n1,n2,n3,n4,n5 : integer32;
  c1,c2,c3,c4,c5,c6 : integer32;
begin
  p := h;
  with p^ do
  begin
    { E = F ',' F }
    pop( h, n3,n4,c3,c4 );
    pop( h, n1,n2,c1,c2 );
    n5 := newnode(h);
    c5 := renamecell(h,n1,n3);
    c6 := orcell(h,n2,n4,n5);

    cellstack[c5].r := c1;
    cellstack[c2].r := c3;
    cellstack[c4].r := c6;

    push( h, n1,n5,c5,c6 );
  end;
  ExpAgain := true;
end;

function FactorAgain( h,h1:pointer ):boolean;
var
  p : RegPtr;
  n1,n2,n3,n4 : integer32;
  c1,c2,c3,c4,c5 : integer32;
begin
  p := h;
  with p^ do
  begin
    { F = T T }
    pop( h, n3,n4,c3,c4 );
    pop( h, n1,n2,c1,c2 );
    c5 := renamecell(h,n2,n3);

    cellstack[c2].r := c5;
    cellstack[c5].r := c3;

    push( h, n1,n4,c1,c4 );
  end;
  FactorAgain := true;
end;

function PrimaryIsId( h,h1:pointer ):boolean;
var
```

```
  p : RegPtr;
  n1,n2 : integer32;
  c1 : integer32;
  s : lexstring;
begin
  p := h;
  with p^ do
  begin
    CurrentToken( h1,s );
    n1 := newnode(h);
    n2 := newnode(h);
    c1 := tokencell(h,s,n1,n2);
    push( h, n1,n2,c1,c1 );
  end;
  PrimaryIsId := true;
end;

function RepeatIsPlus( h,h1:pointer ):boolean;
var
  p : RegPtr;
  n1,n2,n3 : integer32;
  c1,c2,c3 : integer32;
begin
  p := h;
  with p^ do
  begin
    { F = T + }
    n1 := newnode(h);
    pop( h, n2,n3,c2,c3 );
    c1 := orcell(h,n1,n3,n2);
    cellstack[c1].r := c2;
    push( h, n1,n3,c1,c3 );
  end;
  RepeatIsPlus := true;
end;

function RepeatIsQuery( h,h1:pointer ):boolean;
var
  p : RegPtr;
  n1,n2,n3 : integer32;
  c1,c2,c3 : integer32;
begin
  p := h;
  with p^ do
  begin
    n1 := newnode(h);
    pop( h, n2,n3,c2,c3 );
    c1 := orcell(h,n2,n3,n1);
    cellstack[c3].r := c1;
    push( h, n2,n1,c2,c1 );
  end;
  RepeatIsQuery := true;
end;

function RepeatIsStar( h,h1:pointer ):boolean;
var
  p : RegPtr;
  n1,n2,n3,n4 : integer32;
  c1,c2,c3,c4 : integer32;
begin
  p := h;
  with p^ do
  begin
    n1 := newnode(h);
    n2 := newnode(h);
    pop( h, n3,n4,c3,c4 );
    c1 := orcell(h,n1,n4,n3);
    c2 := orcell(h,n3,n4,n2);
    cellstack[c1].r := c3;
    cellstack[c4].r := c2;
    push(h,n1,n2,c1,c2);
  end;
  RepeatIsStar := true;
 end;

 function CommentFound( h,h1:pointer ): boolean;
 begin
  WhiteSpace( h1 );
  CommentFound := true;
 end;

end.
```

9.4 LALR(1) Parser and Lexer Tables

If the grammar definition is LALR(1) then the compiler-compiler automatically generates the combined parse and lex tables for use with the processor software emulation.

The example table, given next, has been generated from the grammar describing regular expressions. Note that the individual parse and lexical states are indicated. Also note that each entry consists of four micro-instructions and has an associated comment.

```
{ ************************************************************* }
{ * This software was generated by J.D.McMullin as an * }
{ * integral part of his M.Phil, Ph.D research.          * }
{ ************************************************************* }

unit Regtab;

interface
type
 integer16 = integer;
 function Regtableread( a:integer16) : word;
 function Regtablemax:integer16;
implementation

const
 addrmax = 152;
 instmin = 0;
 instmax = 611;
 plt : array [instmin..instmax] of word =
 (
$2000,$0001,$0000,$003D, { 0 PSHIFT $lambda 1 ** lex reset 61 push $error }
```

```
{ Parse State 1 }
$0003,$2005,$2013,$4000, { 1 PSC identifier  _$2 = identifier <leftnameis>  }
$000F,$0007,$0000,$003D, { 2 PS Reg 7 }
$0010,$2003,$2012,$4000, { 3 PSC RegExpRule  _$1 = RegExpRule  }
$0012,$0009,$0000,$003D, { 4 PS _$1 9 }
$0013,$000D,$0000,$003D, { 5 PS _$2 13 }
$2000,$E000,$2000,$4000, { 6 ELSE  **** PARSE ERROR HANDLER **** }

{ Parse State 3 }
$0002,$6001,$8001,$003B, { 7 PSC $eoi  $goal = Reg $eoi  }
$2000,$E000,$2000,$4000, { 8 ELSE  **** PARSE ERROR HANDLER **** }

{ Parse State 5 }
$0003,$2005,$2013,$4000, { 9 PSC identifier  _$2 = identifier <leftnameis>  }
$0010,$2002,$2012,$003B, { 10 PSC RegExpRule  _$1 = _$1 RegExpRule  }
$0013,$000D,$0000,$003D, { 11 PS _$2 13 }
$2000,$4004,$200F,$003B, { 12 ELSE  Reg = _$1  }

{ Parse State 6 }
$000A,$000F,$0000,$003D, { 13 PS '=' 15 }
$2000,$E000,$2000,$4000, { 14 ELSE  **** PARSE ERROR HANDLER **** }

{ Parse State 9 }
$0003,$2014,$200E,$4000, { 15 PSC identifier  Primary = identifier <primaryisid>  }
$0004,$0016,$0000,$003D, { 16 PS '(' 22 }
$000C,$001D,$0000,$003D, { 17 PS Exp 29 }
$000D,$001F,$0000,$003D, { 18 PS Factor 31 }
$000E,$0021,$0000,$003D, { 19 PS Primary 33 }
$0011,$0025,$0000,$003D, { 20 PS Term 37 }
$2000,$E000,$2000,$4000, { 21 ELSE  **** PARSE ERROR HANDLER **** }

{ Parse State 11 }
$0003,$2014,$200E,$4000, { 22 PSC identifier  Primary = identifier <primaryisid>  }
$0004,$0016,$0000,$003D, { 23 PS '(' 22 }
$000C,$0027,$0000,$003D, { 24 PS Exp 39 }
$000D,$001F,$0000,$003D, { 25 PS Factor 31 }
$000E,$0021,$0000,$003D, { 26 PS Primary 33 }
$0011,$0025,$0000,$003D, { 27 PS Term 37 }
$2000,$E000,$2000,$4000, { 28 ELSE  **** PARSE ERROR HANDLER **** }

{ Parse State 12 }
$0009,$2006,$2010,$0039, { 29 PSC ';'  RegExpRule = _$2 '=' Exp ';' <ruleis>  }
$2000,$E000,$2000,$4000, { 30 ELSE  **** PARSE ERROR HANDLER **** }

{ Parse State 13 }
$0016,$0029,$0000,$003D, { 31 PS _$4 41 }
$2000,$4009,$2016,$4000, { 32 ELSE  _$4 =  }

{ Parse State 14 }
$0006,$2010,$2011,$003B, { 33 PSC '*'  Term = Primary '*' <repeatisstar>  }
$0007,$2011,$2011,$003B, { 34 PSC '+'  Term = Primary '+' <repeatisplus>  }
$000B,$200F,$2011,$003B, { 35 PSC '?'  Term = Primary '?' <repeatisquery>  }
$2000,$4012,$2011,$003B, { 36 ELSE  Term = Primary  }

{ Parse State 15 }
$0017,$002C,$0000,$003D, { 37 PS _$6 44 }
$2000,$400D,$2017,$4000, { 38 ELSE  _$6 =  }

{ Parse State 16 }
$0005,$2013,$200E,$003A, { 39 PSC ')'  Primary = '(' Exp ')'  }
```

```
$2000,$E000,$2000,$4000, { 40 ELSE  **** PARSE ERROR HANDLER **** }

{ Parse State 18 }
$0008,$0032,$0000,$003D, { 41 PS ',' 50 }
$0014,$2008,$2016,$003B, { 42 PSC _$3  _$4 = _$4 _$3  }
$2000,$400A,$200C,$003A, { 43 ELSE  Exp = Factor _$4  }

{ Parse State 22 }
$0003,$2014,$200E,$4000, { 44 PSC identifier  Primary = identifier <primaryisid>  }
$0004,$0016,$0000,$003D, { 45 PS '(' 22 }
$000E,$0021,$0000,$003D, { 46 PS Primary 33 }
$0011,$200B,$2015,$4000, { 47 PSC Term  _$5 = Term <factoragain>  }
$0015,$200C,$2017,$003B, { 48 PSC _$5  _$6 = _$6 _$5  }
$2000,$400E,$200D,$003A, { 49 ELSE  Factor = Term _$6  }

{ Parse State 24 }
$0003,$2014,$200E,$4000, { 50 PSC identifier  Primary = identifier <primaryisid>  }
$0004,$0016,$0000,$003D, { 51 PS '(' 22 }
$000D,$2007,$2014,$003B, { 52 PSC Factor  _$3 = ',' Factor <expagain>  }
$000E,$0021,$0000,$003D, { 53 PS Primary 33 }
$0011,$0025,$0000,$003D, { 54 PS Term 37 }
$2000,$E000,$2000,$4000, { 55 ELSE  **** PARSE ERROR HANDLER **** }

{ Max Symbols in a rule = 4 }
$2000,$6000,$A000,$2000, { 56 POP }
$2000,$6000,$A000,$2000, { 57 POP }
$2000,$6000,$A000,$2000, { 58 POP }
$2000,$6000,$A000,$2000, { 59 POP }
$2000,$6000,$A000,$4000, { 60 REDUCE }

{ Lex State 1 }
$2000,$6000,$6000,$003E, { 61 LC }
$8009,$8009,$4000,$005C, { 62 IF 9 .. 9 LS 92 push $error }
$801A,$801A,$4002,$0056, { 63 IF 26 .. 26 LS 86 push $eoi }
$801E,$801E,$4000,$005C, { 64 IF 30 .. 30 LS 92 push $error }
$8020,$8020,$4000,$005C, { 65 IF   ..   LS 92 push $error }
$8021,$8021,$4000,$0057, { 66 IF ! .. ! LS 87 push $error }
$8028,$8028,$4004,$004E, { 67 IF ( .. ( LS 78 push '(' }
$8029,$8029,$4005,$004F, { 68 IF ) .. ) LS 79 push ')' }
$802A,$802A,$4006,$0050, { 69 IF * .. * LS 80 push '*' }
$802B,$802B,$4007,$0051, { 70 IF + .. + LS 81 push '+' }
$802C,$802C,$4008,$0052, { 71 IF , .. , LS 82 push ',' }
$803B,$803B,$4009,$0053, { 72 IF ; .. ; LS 83 push ';' }
$803D,$803D,$400A,$0054, { 73 IF = .. = LS 84 push '=' }
$803F,$803F,$400B,$0055, { 74 IF ? .. ? LS 85 push '?' }
$8041,$805A,$4003,$0064, { 75 IF A .. Z LS 100 push identifier }
$8061,$807A,$4003,$0064, { 76 IF a .. z LS 100 push identifier }
$A000,$A000,$0000,$003D, { 77 LA $error ** lex reset 61 push $error }

{ Lex State 13 }
$E004,$A000,$0000,$003D, { 78 LA '(' ** lex reset 61 push $error }

{ Lex State 12 }
$E005,$A000,$0000,$003D, { 79 LA ')' ** lex reset 61 push $error }

{ Lex State 11 }
$E006,$A000,$0000,$003D, { 80 LA '*' ** lex reset 61 push $error }

{ Lex State 10 }
$E007,$A000,$0000,$003D, { 81 LA '+' ** lex reset 61 push $error }
```

```
{ Lex State 9 }
$E008,$A000,$0000,$003D, { 82 LA ',' ** lex reset 61 push $error }

{ Lex State 8 }
$E009,$A000,$0000,$003D, { 83 LA ';' ** lex reset 61 push $error }

{ Lex State 7 }
$E00A,$A000,$0000,$003D, { 84 LA '=' ** lex reset 61 push $error }

{ Lex State 6 }
$E00B,$A000,$0000,$003D, { 85 LA '?' ** lex reset 61 push $error }

{ Lex State 5 }
$C002,$A000,$0000,$003D, { 86 LA $eoi ** lex reset 61 push $error }

{ Lex State 4 }
$2000,$6000,$6000,$0058, { 87 LC }
$8009,$8009,$4000,$0057, { 88 IF 9 .. 9 LS 87 push $error }
$801E,$801E,$4000,$005C, { 89 IF 30 .. 30 LS 92 push $error }
$8020,$807E,$4000,$0057, { 90 IF   .. ~ LS 87 push $error }
$A000,$A000,$0000,$003D, { 91 LA $error ** lex reset 61 push $error }

{ Lex State 3 }
$2000,$C015,$4000,$006A, { 92 LT commentfound to 106 push $error }
$6000,$6000,$0000,$003D, { 93 ** lex reset 61 push $error }
$2000,$6000,$6000,$005F, { 94 LC }
$8009,$8009,$4000,$005C, { 95 IF 9 .. 9 LS 92 push $error }
$801E,$801E,$4000,$005C, { 96 IF 30 .. 30 LS 92 push $error }
$8020,$8020,$4000,$005C, { 97 IF   ..   LS 92 push $error }
$8021,$8021,$4000,$0057, { 98 IF ! .. ! LS 87 push $error }
$A000,$A000,$0000,$003D, { 99 LA $error ** lex reset 61 push $error }

{ Lex State 2 }
$2000,$6000,$6000,$0065, { 100 LC }
$8030,$8039,$4003,$0064, { 101 IF 0 .. 9 LS 100 push identifier }
$8041,$805A,$4003,$0064, { 102 IF A .. Z LS 100 push identifier }
$805F,$805F,$4003,$0064, { 103 IF _ .. _ LS 100 push identifier }
$8061,$807A,$4003,$0064, { 104 IF a .. z LS 100 push identifier }
$E003,$A000,$0000,$003D, { 105 LA identifier ** lex reset 61 push $error }

{ Lex State 14 }
$2000,$6000,$6000,$006B, { 106 LC }
$8009,$8009,$4000,$0080, { 107 IF 9 .. 9 LS 128 push $error }
$801A,$801A,$4002,$0056, { 108 IF 26 .. 26 LS 86 push $eoi }
$801E,$801E,$4000,$0080, { 109 IF 30 .. 30 LS 128 push $error }
$8020,$8020,$4000,$0080, { 110 IF   ..   LS 128 push $error }
$8021,$8021,$4000,$007B, { 111 IF ! .. ! LS 123 push $error }
$8028,$8028,$4004,$004E, { 112 IF ( .. ( LS 78 push '(' }
$8029,$8029,$4005,$004F, { 113 IF ) .. ) LS 79 push ')' }
$802A,$802A,$4006,$0050, { 114 IF * .. * LS 80 push '*' }
$802B,$802B,$4007,$0051, { 115 IF + .. + LS 81 push '+' }
$802C,$802C,$4008,$0052, { 116 IF , .. , LS 82 push ',' }
$803B,$803B,$4009,$0053, { 117 IF ; .. ; LS 83 push ';' }
$803D,$803D,$400A,$0054, { 118 IF = .. = LS 84 push '=' }
$803F,$803F,$400B,$0055, { 119 IF ? .. ? LS 85 push '?' }
$8041,$805A,$4003,$0064, { 120 IF A .. Z LS 100 push identifier }
$8061,$807A,$4003,$0064, { 121 IF a .. z LS 100 push identifier }
$A000,$A000,$0000,$003D, { 122 LA $error ** lex reset 61 push $error }

{ Lex State 16 }
$2000,$6000,$6000,$007C, { 123 LC }
```

```
$8009,$8009,$4000,$007B, { 124 IF 9 .. 9 LS 123 push $error }
$801E,$801E,$4000,$0080, { 125 IF 30 .. 30 LS 128 push $error }
$8020,$807E,$4000,$007B, { 126 IF   .. ~ LS 123 push $error }
$A000,$A000,$0000,$003D, { 127 LA $error ** lex reset 61 push $error }

{ Lex State 15 }
$2000,$C015,$4000,$0093, { 128 LT commentfound to 147 push $error }
$6000,$6000,$0000,$003D, { 129 ** lex reset 61 push $error }
$2000,$6000,$6000,$0083, { 130 LC }
$8009,$8009,$4000,$0080, { 131 IF 9 .. 9 LS 128 push $error }
$801A,$801A,$4002,$0056, { 132 IF 26 .. 26 LS 86 push $eoi }
$801E,$801E,$4000,$0080, { 133 IF 30 .. 30 LS 128 push $error }
$8020,$8020,$4000,$0080, { 134 IF   ..   LS 128 push $error }
$8021,$8021,$4000,$007B, { 135 IF ! .. ! LS 123 push $error }
$8028,$8028,$4004,$004E, { 136 IF ( .. ( LS 78 push '(' }
$8029,$8029,$4005,$004F, { 137 IF ) .. ) LS 79 push ')' }
$802A,$802A,$4006,$0050, { 138 IF * .. * LS 80 push '*' }
$802B,$802B,$4007,$0051, { 139 IF + .. + LS 81 push '+' }
$802C,$802C,$4008,$0052, { 140 IF , .. , LS 82 push ',' }
$803B,$803B,$4009,$0053, { 141 IF ; .. ; LS 83 push ';' }
$803D,$803D,$400A,$0054, { 142 IF = .. = LS 84 push '=' }
$803F,$803F,$400B,$0055, { 143 IF ? .. ? LS 85 push '?' }
$8041,$805A,$4003,$0064, { 144 IF A .. Z LS 100 push identifier }
$8061,$807A,$4003,$0064, { 145 IF a .. z LS 100 push identifier }
$A000,$A000,$0000,$003D, { 146 LA $error ** lex reset 61 push $error }

{ Lex State 17 }
$2000,$6000,$6000,$0094, { 147 LC }
$8009,$8009,$4000,$005C, { 148 IF 9 .. 9 LS 92 push $error }
$801E,$801E,$4000,$005C, { 149 IF 30 .. 30 LS 92 push $error }
$8020,$8020,$4000,$005C, { 150 IF   ..   LS 92 push $error }
$8021,$8021,$4000,$0057, { 151 IF ! .. ! LS 87 push $error }
$A000,$A000,$0000,$003D  { 152 LA $error ** lex reset 61 push $error }
 );

 function Regtableread( a:integer16) : word;
 begin
  if (instmin<=a) and (a<=instmax) then
  begin
   Regtableread := plt[a];
  end
  else
  begin
   write('Illegal Instruction Address ',a);
   writeln(' legal range [',instmin,'..',instmax,']');
   Regtableread := plt[a mod 4];
  end;
 end;

 function Regtablemax:integer16;
 begin
  Regtablemax := addrmax;
 end;

end.
```

9.5 Examples of Parses

The section "Examples of Regular Expressions" in this chapter gave some regular expressions suitable for input to the software emulation of the processor. The resulting output is given in the corresponding sections. The output also includes MODEL source code intended to generate logic to recognise the expression. The MODEL code generated follows the algorithm described in section 2 "Hardware Implementations". The logic to recognise a given token, such as "a" is referenced but not defined. It will be similar to that defined in section 2.1.1.1 "Recognising a Token".

9.5.1 Example 1

The regular expression A = a b+ c generated the following debug information which also included some MODEL source code for logic to recognise the expression. The logic to detect tokens a, b and c is not defined.

```
[  1] A = a b+ c ;
Part A [ clk,tokin ] -> res
 Signal n1;
 Signal n2;
 Signal n3;
 Signal n4;
 Signal n5;
 Signal n6;
 Signal n7;
 ONE -> n1
 token("a")[ clk,tokin,n1 ] -> n2
 n2 -> n5
 or[ n5,n4 ] -> n3
 token("b")[ clk,tokin,n3 ] -> n4
 n4 -> n6
 token("c")[ clk,tokin,n6 ] -> n7
 n7 -> res
End;
[  2]
[  3]
**** EOF ****
```

9.5.2 Example 2

The regular expressions A = a b* c and B = c m+ (a, d+)? Gave the following fragments of MODEL source code.

```
[  1] A = a b* c ;
Part A [ clk,tokin ] -> res
 Signal n1;
 Signal n2;
 Signal n3;
 Signal n4;
 Signal n5;
 Signal n6;
 Signal n7;
 Signal n8;
 ONE -> n1
 token("a")[ clk,tokin,n1 ] -> n2
 n2 -> n5
 or[ n5,n4 ] -> n3
```

```
  token("b")[ clk,tokin,n3 ] -> n4
  or[ n3,n4 ] -> n6
  n6 -> n7
  token("c")[ clk,tokin,n7 ] -> n8
  n8 -> res
 End;
 [  2] B = c m+ (a,d+)? ;
 Part B [ clk,tokin ] -> res
  Signal n1;
  Signal n2;
  Signal n3;
  Signal n4;
  Signal n5;
  Signal n6;
  Signal n7;
  Signal n8;
  Signal n9;
  Signal n10;
  Signal n11;
  Signal n12;
  ONE -> n1
  token("c")[ clk,tokin,n1 ] -> n2
  n2 -> n5
  or[ n5,n4 ] -> n3
  token("m")[ clk,tokin,n3 ] -> n4
  n4 -> n6
  n6 -> n10
  token("a")[ clk,tokin,n6 ] -> n7
  or[ n10,n9 ] -> n8
  token("d")[ clk,tokin,n8 ] -> n9
  or[ n7,n9 ] -> n11
  or[ n6,n11 ] -> n12
  n12 -> res
 End;
**** EOF ****
```