D E S I G N   A U T O M A T I O N
O F   A
M A C H I N E - I N D E P E N D E N T
C O D E   G E N E R A T O R

Thesis submitted by

PETER GRAHAM CLAYTON

in fulfilment of the requirements

for the degree of

MASTER OF SCIENCE

RHODES UNIVERSITY

DECEMBER 1983

A C K N O W L E D G E M E N T S

My thanks are due to Professor Pat Terry, for help and encouragement which has extended way beyond the limits of this thesis, and for permission to make use of much of his software, and to Louisa for her support and help in preparing this report.

A B S T R A C T

As both computer languages and architectures continue to proliferate, there is a continuing need for new compilers. Researchers have attempted to ease the work of producing compilers by developing methods to automate compiler writing. While much work has been done (and considerable success achieved) in writing parsers which can handle a variety of source languages (using mainly table-driven analysis methods), less progress has been made in formalizing the code generation end of the compiler. Nevertheless, some of the more recent publications in code generation stress portability or retargetability of the resulting compiler. A number of code generator synthesisers have been developed, some of which produce code that can be compared in quality with that produced by a conventional code generator. However, because of the complexity of generalizing the mapping from source language to target machine, and the need for efficiency of various kinds, code generator synthesisers are large, complicated programs. Consequently, the person who develops a code generator using one of these tools invariably needs to be a code generation specialist himself.

Many compilers follow a pattern of having a front end which generates intermediate code, and a back end which converts intermediate code to machine code. The intermediate code is effectively machine independent, or can be designed that way.

With these points in mind, we have set out to write a system of programs which

1. will allow the generation of such a back end in a reasonably short time, for a general intermediate code, and for a general machine code, and
2. can be used by anyone who has a sound knowledge of the target machine's architecture and associated assembler language, but is not necessarily a specialist compiler writer.

The system consists of a series of friendly, interactive programs by means of which the user sets up tables defining the architecture and assembly level instructions for the target machine, and the code templates onto which intermediate codes produced by a parser have been mapped. A general notation has been developed to represent machine instructions using the same format as the target assembler. Thus the code generator writer is able to write code sequences to perform the effects of the intermediate codes, using assembly mnemonics familiar to him. The resultant table-driven code generator simply replaces a sequence of intermediate codes by their respective code-templates, relocating them in memory and filling in addresses known only at code-generation time.

This thesis describes the use and implementation details of this generalized code generation system. As an example, the implementation of a code generator for a CLANG [23] parser on an 8Ø8Ø processor is described. The discussion also includes guide-lines on how to implement a loader and associated run-time routines for use in executing the object code.

The results of a number of bench-marks have shown, as expected, that code produced by a code generator developed in this manner is larger and slower than that from a special purpose optimizing code generator, but is still several times faster than interpreting the intermediate code. The major benefit to be gained from using this system lies in the shorter development time by a less skilled person.

C O N T E N T S

1. INTRODUCTION

...let's see if you can read my code:

```
V D D L    R B T    C L J K S
  N T     P S D   K L H S M
            A T F O
```

If you can read it, make it rain tomorrow
so I will know.

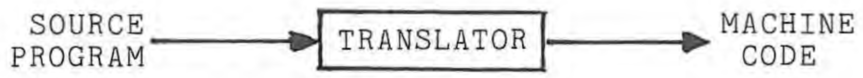
from CHILDREN'S LETTERS TO GOD[1]

In the early days of computers, the esoteric nature of internal machine code did much to inhibit the computer revolution. While most humans find it difficult to understand the "language" of the central processing unit, most computer processors are completely devoid of conventional linguistic skills. The non-specialist was able to enjoy the use of a computer only once high-level languages had been developed.

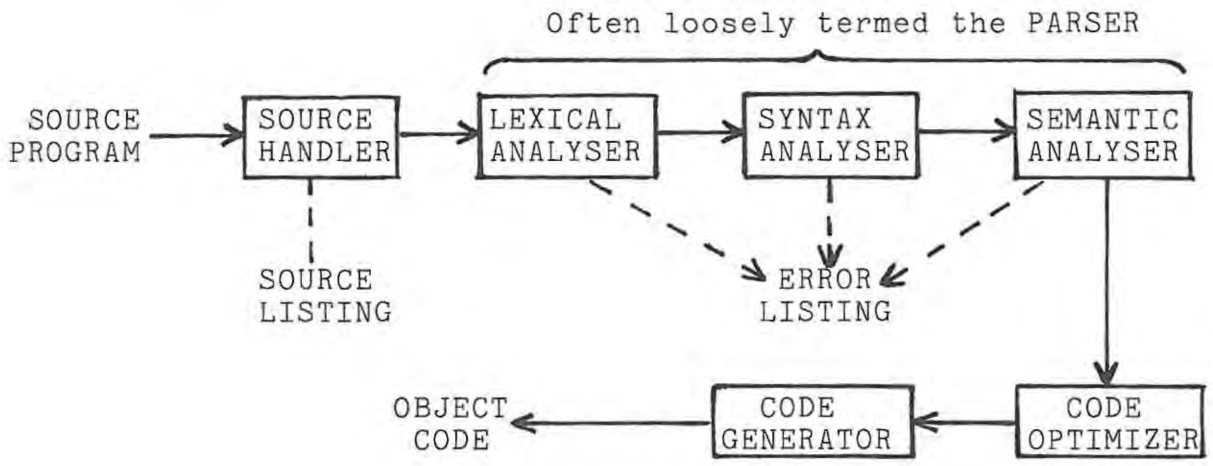
In the last decade, many new general-purpose programming languages have emerged, along with a large number of more specialised languages. Since no major breakthrough has been achieved in "humanizing" the computer processor's communicative ability, the design and implementation of compilers for programming languages has become an essential part of systems software development. Although the construction of new compilers is being made easier by the use of various language development tools, the implementation of a compiler for a major high-level language remains a non-trivial task.

1.1 COMPILERS IN BRIEF

A compiler is a computer program which translates programs, written in a particular **source** programming language, into executable code for a target computer of a particular design.



The compilation process is usually subdivided into a sequence of simpler **phases**. It is convenient to think of each phase as an operation which takes, as input, one representation of the source program and produces, as output, another representation. Most compilers consist of at least the following phases:



The boundaries between phases are blurred. Two or more of the phases above may often be combined into one phase. The modern trend towards "intelligent" editors, has seen some of the initial stages of the compiler incorporated into complementary editors. In some interpretive systems, the compilation process may be completely transparent to the user.

A brief description of the operation of each phase appears below. More information on the historical development of compilers can be found in articles by Knuth [4], Feldman and Gries [5] (which also includes a comprehensive bibliography up to that time) and Aho [6] and in many modern textbooks on compilers [7]-[11].

1.1.1 Lexical Analysis (or scanning)

The name is derived from LEXICON [Greek] meaning DICTIONARY. This initial parsing phase sees the source program as a sequence of characters. These characters are grouped into **tokens**, which are substrings of the input stream which logically belong together, such as identifiers and constants. The lexical analyser is usually also responsible for recognising comments, blanks and separators in the source program. The output of the lexical analysis phase is a stream of tokens.

1.1.2 Syntax Analysis (or true parsing phase)

This phase groups tokens emitted by the lexical analyser into syntactic structures such as expressions and statements. It is the syntax analyser that determines whether the source program satisfies the formation rules of the programming language. If the source program is not syntactically correct, this phase must be able to produce appropriate error diagnostics, and recover from each error, so as to catch any subsequent errors that might be present.

The syntax analyser usually invokes some semantic action as each syntactic construct is recognised. The semantic action may be to enter some information into a table, emit some error diagnostics, or generate some output. The output of the syntax analyser is some intermediate language representation of the source program such as postfix Polish notation, a tree structure, quadruples or codes for a hypothetical stack machine.

1.1.3 Semantic Analysis

This phase has to do with the actual meaning of words and constructs. Type checking, determining that functions are called with the appropriate number of arguments, and verifying that identifiers have been declared are typical of what takes place during semantic analysis.

1.1.4 Code Optimization

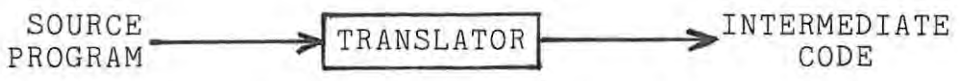
An optimizing compiler attempts to transform an intermediate language representation of the source program into one from which a faster or smaller object program can be generated. The term "optimization" is something of a misnomer, as it is theoretically impossible for a compiler to produce the best possible object program for every source program. A more accurate term might be "code improvement" as suggested by Aho & Ullman [7]. Tradition, however, has given us the term "code optimization".

A number of optimizations can be safely performed only by knowing information gathered from the entire program. One needs to make use of a technique known as Global Data-flow Analysis [2] [3]. Other forms of optimization are static, for example the well known **peephole** optimization technique [7]. This attempts to improve the generated code, by viewing a small range of instructions (peephole) and trying to improve upon them. It is a characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. Thus, repeated passes over the code are necessary to get the maximum benefit. Peephole optimization can be applied at both the intermediate and final code stages.

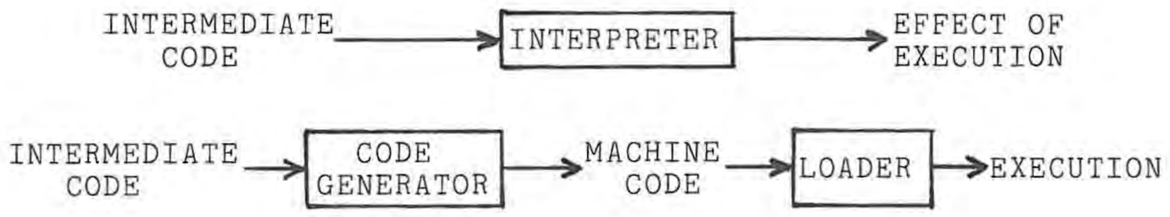
1.1.5 Code Generation

The code generator is the final stage of the compiler and usually incorporates the optimization algorithms. This phase produces machine code directly, and is therefore the non-portable component of a compiler.

To promote portability and decrease the complexity of the translation algorithms, the code generation phase in a modern compiler is usually written as a completely separate module. Communication with the more easily transportable parts of the compiler is by means of a clearly defined set of intermediate codes.



The compiler writer then has the option of either interpreting the intermediate codes, or of converting them to machine code.



1.2 AUTOMATED CODE GENERATOR WRITING

Recently. much interest has been shown in tools that reduce the effort needed to construct a good compiler. These go by names such as **compiler-compilers**, **translator writing systems** or **compiler generators**.

To date, more work has been done on automating the front end of the compiler (possibly since this can be done without regard to target machine architecture) than the code generation end (which is target machine dependent). Considerable success has been achieved, using mainly table-driven analysis methods [7], towards making this aspect of compiler writing not only easier, but more reliable. It is easier to check that a grammar is an accurate syntax description than it is to check the implicit description embodied in the logic of a syntax analysis program.

While less progress has been made in formalizing the code generation end of the compiler, a number of successful code generator synthesisers have been developed [14] - [21]. The emphasis here lies in the **portability** or **retargetability** of the resulting compiler.

In attempting to provide tools for automating code generation, most researchers have again turned to table driven methods. By using a table-driven syntax analyser, we can handle the analysis of a new language by giving a new grammar to the table building program and then providing the new table to the analyser. Similarly, by presenting a description of a new target machine to the code generator's table-building program, we can retarget the code generator to produce code for the new machine.

A portable C compiler has been developed at Bell Laboratories, Murray Hill [14]. This compiler incorporates programs such as LEX, a lexical analyser generator, and YACC (Yet Another Compiler-Compiler), which generates parsers from an input specification language that describes the desired syntax, both well known to users of the UNIX [37] system. The code generator uses a table consisting of a collection of templates. Each template contains a pattern to search for in a previously generated intermediate program form. This demands a lot of work from the implementor, who must specify each template. The portable C compiler provides code of reasonable quality, and has been retargeted successfully on over a dozen machine types.

Another table-driven code generator is under development at the University of California, Berkeley [15] [21], in which the templates are generated automatically, instead of being written by the implementor. The added generality is both a strength and a weakness. There is less work involved in developing a code generator using this tool, but it is claimed [15] that both the quality of the code generated and the speed of the code generator are significantly reduced when compared with the C compiler.

An ambitious compiler-compiler project has been undertaken by a large group of researchers at Carnegie-Mellon University [16] [19]. The goal of this project is to develop a compiler-compiler that is capable of producing high-quality optimizing compilers from formal machine and language descriptions. Known as the PQCC (Production-Quality Compiler-Compiler), this enormous project boasts good code that can be compared with that of a conventional code generator.

A number of other fruitful investigations into compiler-building methods are being conducted at various research institutes around the world. A recently revised bibliography of the subject has been published by Ganapathi and Fisher [20].

Because of the complexity of generalizing the mapping from source language to target machine, and the need for efficiency of various kinds, code generator synthesisers are large, complicated programs. Consequently, the person who develops a code generator using one of these tools invariably needs to be a code generation specialist himself.

1.3 WHY ANOTHER MACHINE-INDEPENDENT CODE GENERATOR ?

All the code generator generators studied in the literature satisfy, to varying degrees, the aims of their authors. Without exception, they reduce the amount of work required in developing new compilers or transporting existing compilers to a new system. However, they are also, without exception, large programs. Very few code generator synthesisers can be run on any machine smaller than a mainframe. In addition, all the systems studied require of the implementor some knowledge of code generation and optimization techniques.

At Rhodes University, microcomputers of various manufacture are widely used. The most popular operating system is the UCSD p-System, which interprets its object programs. Since most undergraduate students spend far more time compiling their programs than running them, the slow interpreter does not prove to be a handicap. However, interpretation becomes an impeding factor if new systems programs are developed on the UCSD p-System. An example of such a systems program is the CLANG compiler [23], developed at Rhodes University on the UCSD p-System. The entire compiler is interpreted and, since the CLANG system also interprets the codes it produces, the effect of executing object programs is accomplished by two levels of interpretation. An obvious gain in speed would be achieved if at least one (but preferably both) of the levels of interpretation was replaced by direct execution.

Since new machine architectures are acquired from time to time, a transportable compiler would also reduce the amount of effort needed by supervisors and teachers before students could write programs in familiar languages on new systems.

A project was undertaken at Rhodes University to write a system of programs which would generate code generators for both large and small computers. The resultant code generators would take their input from parsers which produce some suitable form of intermediate code. At the same time, an attempt was made to satisfy the following criteria:

1. the code generator should be up and going in a reasonably short time, and

2. the person developing the code generator would not have to be a specialist compiler writer. In fact, a sound knowledge of the target machine's architecture and associated assembler language should be sufficient.

1.4 OUTLINE OF THE REMAINDER OF THE THESIS

Much of the work described in this thesis is of a highly technical nature, and it becomes difficult, if not tedious, to discern the intent from the details. Accordingly, the remainder of this report is divided into three sections. The first describes the system in fairly broad detail, gives an example of an actual system produced, and discusses briefly refinements which have been made to the system in the closing stages of the project. The second describes how the system might be used to construct further code generators, both like that described in the first section, and for other architectures. The third and most technical section has, due to its bulk, been confined to a set of appendices. This section discusses implementation details of the system, which will be of interest only to those wishing to extend its basic philosophy. Code for all programs appear with the implementation details in the appendices.

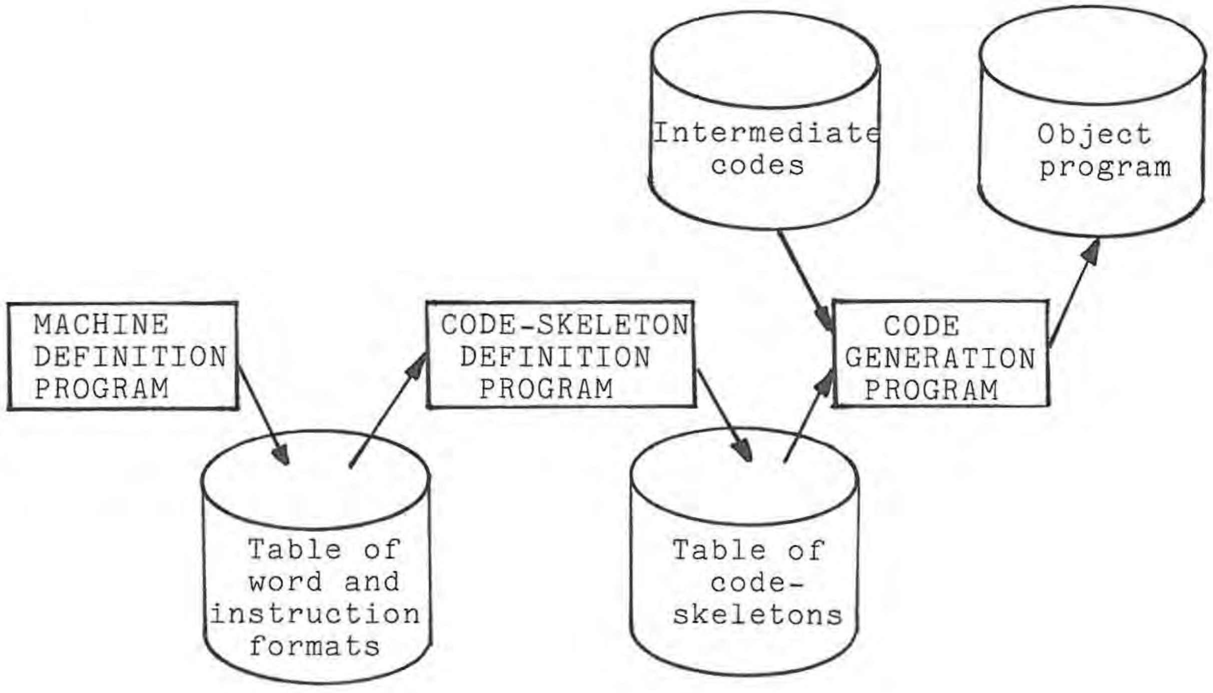
PART 1

INTENT AND ACCOMPLISHMENT

2. OVERVIEW OF THE THE GENERALIZED CODE GENERATOR

Faced with the major problem of trying to design a truly general machine representation, it was decided that code optimization would have to be sacrificed in the interests of ease of use. The most successful retargetable compilers have been those with minimal processing at the code generation stage, while the most successful optimizing compilers have been designed around a particular target machine. A general compiler can achieve both goals only if it relies on the compiler writer to provide the optimization information.

The system to be described consists of two definition programs, and a general table-driven code generation program.


Intermediate codes
Object program
MACHINE DEFINITION PROGRAM
CODE-SKELETON DEFINITION PROGRAM
CODE GENERATION PROGRAM
Table of word and instruction formats
Table of code-skeletons

This system has been written for use with any parser which produces intermediate codes which are effectively machine instructions for a hypothetical stack machine [11] [12] [23]. Two parsers of this type have been used to test the code generator. One for a relatively simple system [23], the other for a system similar to the Pascal P4 compiler [39]. Other forms of intermediate codes, such as three-address codes [7], require slightly different assumptions, but the notation described below should prove effective for these codes as well without major alteration.

2.1 THE MACHINE DEFINITION PROGRAM

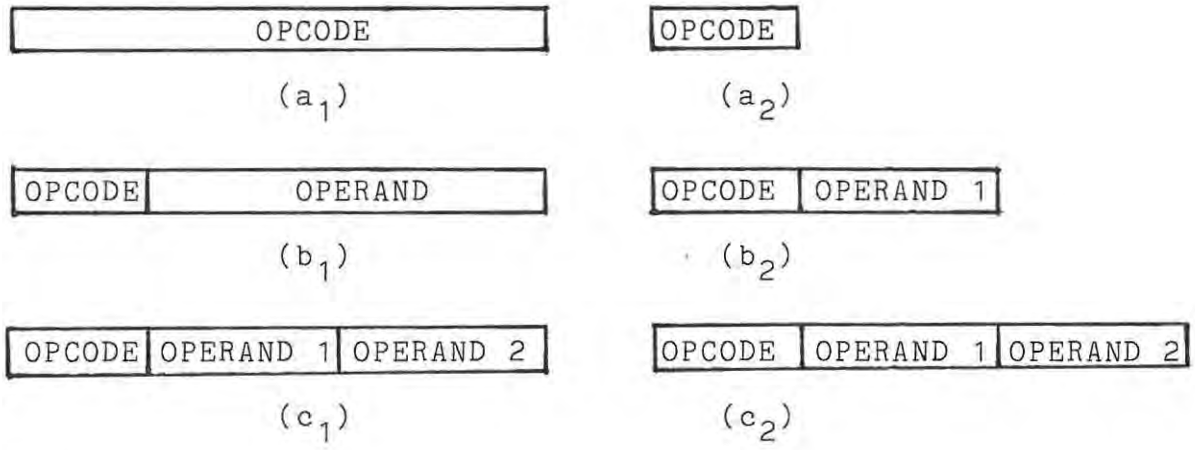
This table construction program takes as input a description of the target machine. The nature of the description differs considerably from the corresponding table construction programs of other well known compiler-compilers. All table-driven code generators demand certain details of the target machine, notably the bit-patterns of instruction opcodes. Several well known products, for example the Berkeley project [15] [21], require details of the way in which registers are designated and the selection strategy to be used in allocating registers and in choosing which ones to save and re-use if there are not enough. Since these projects attempt to reorder computations to produce a smaller object code, other parts of the machine description indicate which operators are commutative and whether frequently-used constants are given permanent register or memory storage. The code generator described in this thesis is driven by a template matching algorithm and, since the compiler-writer is required to define the templates, the criteria for using registers, constants and operands are decided by him, not by a program. In place of this information, the table construction program of this thesis collects details of assembly mnemonics which are used to write the templates.

2.1.1 Overview of Machine Instruction Formats

Although computer processors vary greatly in architecture, they do not disagree in their underlying principles so widely as to detract from a concept of code retargetability. In particular, all processors are capable of recognising a binary pattern as representing a particular operation, and of producing the effect of that operation.

To realise a retargetable code generator, one must be able to produce a set of instructions (the bit patterns) which the processor will recognise. This involves knowing such information as the length of each bit pattern and its value. In addition, if one is to allow non-sequential flow of control, one needs to know how many bits there are in the machine's smallest addressable unit. This enables an accurate **Program Counter** to be maintained, and consequently, jumping and looping instructions to be produced correctly.

All machine level instructions must contain an operation code, or **opcode** for short, which indicates what action is to be performed. In addition, many instructions contain, or specify the location of, data used by the instruction. The fields used to specify data are known as **operand** or **argument** fields. These operand fields may be immediate data, the address of the data, a register name or the addressing mode which the instruction is to use.

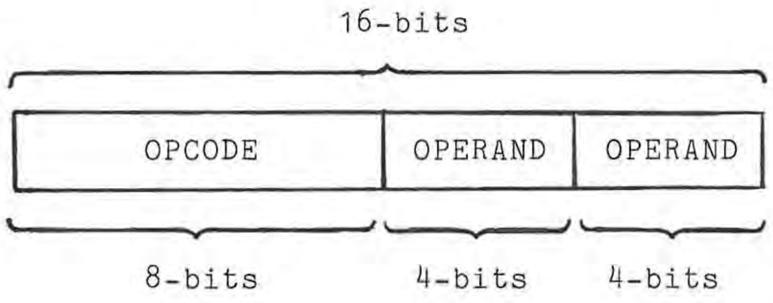


On some machines, all instructions have the same length; on others, there may be several lengths. Moreover, instructions may be shorter, the same length as, or longer than the word (i.e. smallest addressable unit) length. Within a particular instruction length, various instruction formats may exist. Opcodes and operand fields may vary both in number and length.

In attempting to design a truly general definition for an instruction, taking the enormous variety of possible instruction formats into account, a bit-by-bit definition appears to be the only solution.

2.1.2 A Notation for describing Instruction Formats

A notation has been developed for describing an instruction in terms of its length and the number and width of its fields. Each field is described by an alphabetic letter, which is repeated as many times as there are bits in the field. For example, an instruction format



might be defined as

FFFF FFFF AAAA BBBB

F represents a field which is eight bits wide. A and B represent fields which are each four bits wide. Spaces may be inserted for clarity and are ignored. The choice of alphabetic letter is completely arbitrary, but must be unique for the instruction in question.

Most processors have a small number of unique instruction formats, typically between 3 and 15. Several individual instructions may share the same format. All the fields will agree in number and width, but the operand field will contain a different binary pattern for each instruction, representing a different operation. The size of the instruction set differs from computer to computer, typically within the range 60 to 300.

2.1.3 Input for the Machine Definition Program

The machine definition program allows a user to define all formats for a particular architecture (termed word formats for the purpose of this program) in the notation already described. Individual instructions may then be defined which conform to one of the word formats. This prevents unnecessary typing and reduces errors.

The aim of the machine definition program is to produce a set of instruction mnemonics which agree as closely as possible with those with which an assembler programmer of the target machine will be familiar.

Each instruction's definition must include the name, or **mnemonic**, by which the instruction will be known. The mnemonic is really representative of the binary patterns of the opcode fields (there may be more than one opcode field). Therefore the implementor is also required to furnish the name (this will be a character which marks the position of the field in the word format description, for example F, A or B) and value of the field (or fields) which will be representative of the instruction mnemonic.

Any remaining fields are regarded as argument (operand) fields. These fields will also eventually contain a binary pattern, but their values are only filled in when the instruction is used during the code-skeleton definition phase. In the meantime however, the program collects information about valid actual values for each argument field. The user is required to specify the valid binary range (positive binary, 2's complement or restricted range) for the field, so that errors may be trapped when the instruction is used. Alternatively, the user may choose an "unrestricted" range to indicate that no validation is to be performed on a particular field.

In the interests of making the instructions resemble an assembler format as closely as possible, the user is able to define a set of non-numeric symbols which will be used to represent numeric values in an argument field. Thus one is able to refer to a register by its name, rather than by its numeric value. Memory addresses, too, may be replaced by alphabetic names representing an address. In this way, the readability is improved for code segments which perform calls to run-time subroutines.

To emulate the assembly format even further, the user is asked to specify valid separating characters which will precede each argument.

Finally, the user must specify the order in which the arguments are to appear in the assembly-like version of the instruction. This is important for ensuring correct validation.

2.1.4 An Example

The best explanation of this notation might be by means of an example. Let us define the 8080 assembler instruction

```
MVI   reg, immediate value
```

This is a two-byte instruction of the form:

Byte 1: 00RRR110 where RRR is a three-bit integer value representing one of the registers

```
B -> 000
C -> 001
D -> 010
E -> 011
H -> 100
L -> 101
M -> 110
A -> 111
```

Byte 2: an immediate value in the positive integer range 0 .. 255.

The definitions required to record this instruction are:

(a) Define the word form:

Word format number: 1

Description:	Immediate instruction
Format:	FF RRR GGG IIII IIII

The fields are specified by distinct alphabetic letters, one for each bit.

(b) Define an instruction which uses the format description of (a) above:

Instruction uses word form number:	1
Mnemonic which will represent instruction:	MVI
Field to associate with mnemonic:	F
Value of mnemonic field:	ØØ
Another field to associate with mnemonic:	G
Value of this field:	11Ø
Legal values for argument field R:	Ø .. 7
where Ø is represented by the symbol	B
1 is represented by the symbol	C
.	
.	
.	
7 is represented by the symbol	A
Separator to precede argument field R:	space
Argument position for field R:	1
Legal values for argument field I:	Ø .. 255
Separator to precede argument field I:	space or comma
Argument position for field I:	2

(Both definitions are required unless definition (a) has already been entered to describe a previous instruction of the same format, in which case only (b) is necessary).

In this way, target machine instructions are defined for use in later stages of the system. When an instruction such as

MVI A,25 is used,

1. 'MVI' fills in the fields associated with the mnemonic

F	F	R	R	R	G	G	G	I	I	I	I	I	I	I	I
Ø	Ø				1	1	Ø								

2. 'A' fills in the field associated with the first argument (symbol A represents the value 7)

F	F	R	R	R	G	G	G	I	I	I	I	I	I	I	I
Ø	Ø	1	1	1	1	1	Ø								

3. '25' fills in the field associated with the second argument

F	F	R	R	R	G	G	G	I	I	I	I	I	I	I	I
Ø	Ø	1	1	1	1	1	Ø	Ø	Ø	Ø	1	1	Ø	Ø	1

Thus it can be seen that the machine definition program essentially provides the information for a table-driven assembler for a particular target machine.

2.1.5 Defining Specific Machines

The Machine Definition Program has been thoroughly tested on the Z-8Ø based North Star Advantage computer. The example in chapter 3 shows a machine definition for this computer using 8Ø8Ø assembly mnemonics and bears witness to the fact that all aspects of the 8Ø8Ø assembly language can be implemented using the current machine definition program. Implementing a compiler on the 8Ø8Ø processor has the advantage of showing up a "worst case" condition. The 8Ø8Ø is probably one of the simplest processors available which can still be classed as a computer and its deficiencies at the conventional machine level place extra effort on the assembly level programmer. As one moves up the scale towards more powerful computers, the effort required to implement an assembly level system decreases. Time efficient routines become less important as processors get faster, increased memory size relieves the problem of overlaying, and other additional features, such as special types of registers and floating-point hardware, take the place of soft routines. In this particular case, however, the 8Ø8Ø example does have a disadvantage in that, since the 8Ø8Ø language mnemonics are very simple (e.g. each opcode has only one true addressing mode), it does not show up some of the deficiencies of the notation described in this thesis. The examples below are intended to illustrate the application of the notation to other assembler languages.

In contrast to the 8Ø8Ø assembly language, the PDP-11 series of computers has a highly sophisticated assembly language, called MACRO-11 [31]. Instructions are 16, 32 and 48 bits long. Although the smallest addressable unit is the byte, the word (two bytes) is addressed at all times, so that only even addresses are allowed. The PDP-11 uses an expanding opcode scheme. What appear to be 32- and 48-bit instructions are actually 16-bit instructions followed by one or two data words. What makes this assembly language so powerful are its advanced instruction formats and addressing techniques. Source and destination fields generally contain two sub-fields, one specifying the addressing mode and the other specifying the address (e.g. a register name). By way of an example, consider the MOV instruction:

$$\overbrace{\hspace{20em}}^{\text{16 bits}}$$

MOV	source mode M	source register R	destination mode N	destination register D
4	3	3	3	3

The 3-bit register fields take on one of the values Ø to 7 to represent one of the 16-bit general registers RØ, R1, R2, R3, ... , R7. The 3-bit mode field also takes on a value in the range Ø to 7 to represent one of the following addressing modes:

MODE	ADDRESS	EXAMPLE
Ø	Register Addressing Operand is in R	Move R3 to R4 **MOV R3,R4**
1	Register Indirect R points to the operand	Move memory word pointed to by R3 to R4 **MOV @R3,R4**
2	Autoincrement R is used to point to the operand and is then incremented	Move memory word pointed to by R3 to R4 and add 2 to R3 **MOV (R3)+,R4**
3	Autoincrement indirect R is used to point to the operand address and is then incremented	Move to R4 the memory word addressed by the word R3 points to; add 2 to R3 **MOV @(R3)+,R4**
4	Autodecrement R is decremented, then used to point to the operand	Decrement R3 by 2, then load R4 from the address R3 points to **MOV -(R3),R4**
5	Autodecrement indirect R is decremented, then used to point to operand address	Decrement R3 by 2, then load R4 indirectly from the address R3 points to **MOV @-(R3),R4**
6	Indexing operand address = contents of R + 16-bit offset in next word	Load R4 with the memory word at (R3) + 24 **MOV 24(R3),R4**
7	Indexing & indirect pointer to operand address = contents of R + 16-bit offset in next word	Load R4 with the memory word pointed to by word (R3) + 24 **MOV @24(R3),R4**

The word format description for this instruction is straightforward:

```
FFFF MMM RRR NNN DDD
```

Field F will be associated with the mnemonic and will have the value ØØØ1. The register fields, R and D, can both be defined as taking on the symbolic names of the registers.

```
"RØ" represents the value ØØØ
"R1" represents the value ØØ1
"R2" represents the value Ø1Ø
  .                         .
  .                         .
  .                         .
"R7" represents the value 111
```

However, specifying the mode fields presents a problem. Let us ignore modes 6 and 7 for the mean time, and assume we can define the mode fields using the symbolic names:

```
" "  represents the value ØØØ
"@"  represents the value ØØ1
"+"  represents the value Ø1Ø
"@+" represents the value Ø11
"-"  represents the value 1ØØ
"@-" represents the value 1Ø1
```

Using our defined instruction, the examples for modes Ø to 5 will be written:

```
Ø     MOV    R3,  R4
1     MOV @ R3,  R4
2     MOV + R3,  R4
3     MOV @+ R3,  R4
4     MOV - R3,  R4
5     MOV @- R3,  R4
```

We can make these instructions look more like the MACRO-11 originals by defining the characters "(" and ")" as the appropriate separators, so that they may also be included in our instructions. However, the problem of splitting a symbol (as mode 3 requires), @(R3)+ instead of @+(R3), cannot be solved using the current notation. In addition, the argument position must be the same for all versions of the instruction, forcing mode 2 to position the mode before the register, +(R3), instead of the more realistic (R3)+. Both these problems can be overcome by defining these non-conforming addressing modes as separate instructions. However, this would still require some unique identifying feature, e.g. using a different instruction mnemonic might allow mode 3 to read:

```
MOVAI @(R3)+, R4
```

The above mentioned problems are easily solved (at the expense of a tedious definition) by grouping the mode and register fields together as a single 6-bit field.

MOV	SSSSSS	DDDDDD
4	6	6

where each argument field may take on one of 2^6 symbols:

```
    "RØ"  represents the value  ØØØ ØØØ
    "R1"  represents the value  ØØØ ØØ1
     .                              .
     .                              .
     .                              .
    "R7"  represents the value  ØØØ 111
   "@RØ"  represents the value  ØØ1 ØØØ
     .                              .
     .                              .
     .                              .
   "@R7"  represents the value  ØØ1 111
 "(RØ)+"  represents the value  Ø1Ø ØØØ
```

```
      .                           .
      .                           .
      .                           .
"@(RØ)+"  represents the value  Ø11 ØØØ
      .                           .
      .                           .
      .                           .
 "-(RØ)"  represents the value  1ØØ ØØØ
      .                           .
      .                           .
      .                           .
"@-(RØ)"  represents the value  1Ø1 ØØØ
"@-(R1)"  represents the value  1Ø1 ØØ1
      .                           .
      .                           .
      .                           .
```

This definition allows our defined instruction to imitate the original PDP-11 assembly language exactly.

Modes 6 and 7 call for expanding opcodes. Since the current notation is not designed to handle this feature, a solution to this problem cannot be found which matches the original assembly language instruction. Thus a separate MOV-INDEX instruction will have to be defined.

The COMPASS assembly language of the CDC CYBER series of processors [32] also presents an interesting case. The smallest addressable unit on these machines is the 6Ø-bit word and address fields are up to 17 bits in length. The Cyber has three instruction lengths: 15, 3Ø and 6Ø bits. Thus it is possible to squeeze several instructions into a single word. This presents no problem as far as the transportable code generator is concerned, since the CDC's decoding microcodes will unpack the instructions in the appropriate way. However, instructions may not be split across word boundaries. This restriction demands that that the no-operation instruction (this 15-bit NO instruction has the octal code 46000) be used within code skeletons to pad unused bits at the end of a word. Alternatively, the code generation program could be tailored to emit padding bits where necessary.

The CDC assembly level instructions also include some unique addressing features. For example, when a memory address is put into the 18-bit register A1, the contents of that address are automatically loaded into the 60-bit register X1. However, the assembly language mnemonics for these instructions are very basic and are easily implemented using the machine definition program. For example

```
        SA1  25
```

will set register A1 to 25, thereby loading the contents of word 25 into register X1.

Finally, the **ICL 1900** assembly language, PLAN [33], shows up the system's behaviour for instruction mnemonics which use fixed formats. A normal 24-bit ICL 1900 instruction has the format:

XXX	FFFFFFF	MM	NNNNNNNNNNNN
3	7	2	12

The assembly language allocates specific fields to fixed columns in the source language input line.

```
Columns  1 to  6  =  Label field
Columns  7 to 12  =  Opcode field
Columns 13 to 15  =  Accumulator field
Columns 16 to 72  =  Operand field
```

The name of a register used for indexing (termed "modifier" in ICL jargon) may optionally follow the operand address enclosed in parentheses. For example, add to accumulator 7 the word at address [3Ø27 + contents of index register 2]:

```
|1        |7        |13  |16
|         |ADX      | 7  | 3Ø27(2)
```

The order of operands as mapped onto the machine instruction is:

```
|1        |7        |13  |16
|         |F        | X  | N(M)
```

The machine definition program imposes a free format on the assembly language, forcing the user to declare his separation characters or use the space character as a separator. However, the basic structure of the ICL PLAN language is maintained as long as at least one space is typed between the mnemonic, accumulator and operand fields, and as long as the user declares the parentheses about the modifier field as separators. Since blanks are ignored, the user is still able to type the instructions in fixed format if he so wishes.

2.2 THE CODE-SKELETON DEFINITION PROGRAM

The table driven code generator of this thesis uses the template-matching idea of Bell Laboratories' portable C compiler [14]. The nature of the templates are somewhat different from those of the C compiler. The front end of the portable C compiler produces an intermediate form which is a tree structure (termed the Abstract Syntax Tree [7]). The tree structure makes it easy to distinguish operands (leaf nodes) and operators (internal nodes), and to vary the order in which the computation is performed. Temporary values (at the internal nodes) are stored in registers and register allocation algorithms are needed. The code generator uses a table containing a collection of templates. Each template contains a pattern to search for in the tree. The templates for the code generator described in this theses are far simpler. They are written in the assembler notation of section 2.1 and, since the intermediate code used in this case is very similar to machine code, simply take the place of the intermediate code. The program described in this section is a table driven assembler/editor which constructs the table of templates.

A number of compilers exist which produce assembler language as their output, notably some well known C compilers [41] [42]. While certain features of the code generation system are similar in concept to this type of compiler, the output of a synthesised code generator is final machine code, and its advantage over such compilers lies in its retargetability and the absence of an assembly stage in the final compilation sequence.

2.2.1 Intermediate Codes

"Intermediate codes for a hypothetical stack machine" are machine-like function codes for a make-believe computer whose primary data structure is a stack. This is not to say that a machine could not be built to directly execute such codes. Certain models of Burroughs computers and the Western Digital Micro Engine (which is microcoded to execute UCSD p-Code, level 3) are examples of machines which are microcoded to do this. However, in real machine terms, intermediate codes of this type tend to be rather powerful.

Most conventional machine level instructions specify the movement of one item of data, or a single operation on one or two items of data. Since many intermediate codes for a hypothetical machine originated for the express purpose of being interpreted (i.e. simulating the action of the machine by means of a piece of software), they incorporate fewer design constraints than their microprogram controlled or sequential logic hardware controlled counterparts. It is not difficult to write an interpreter which can perform several move and arithmetic/logic operations as a single pseudo-operation.

In addition, intermediate codes tend to be designed around the language of the source program. Thus the set of intermediate codes for a Pascal compiler might include a CALL code, which sets up a procedure stack frame entry on the stack so that recursion might be implemented before jumping to the code for that procedure; on the other hand, a set of intermediate codes for a Forth system might include a DEF code to create a dictionary entry for a new defined word. At a lower level, however, a piece of processing hardware must be designed to execute programs written in a wide range of source languages; so in the interests of generality, its machine level instructions need to be kept simple.

Although most intermediate codes take their operands from the stack, it is sometimes more convenient for an argument to be included, in much the same way as some opcodes in a conventional processor require operands. As an example, an intermediate code to perform a conditional jump

```
JMPF  Y
```

might be interpreted as

```
if TOS = FALSE then PC := Y
pop (TOS)
```

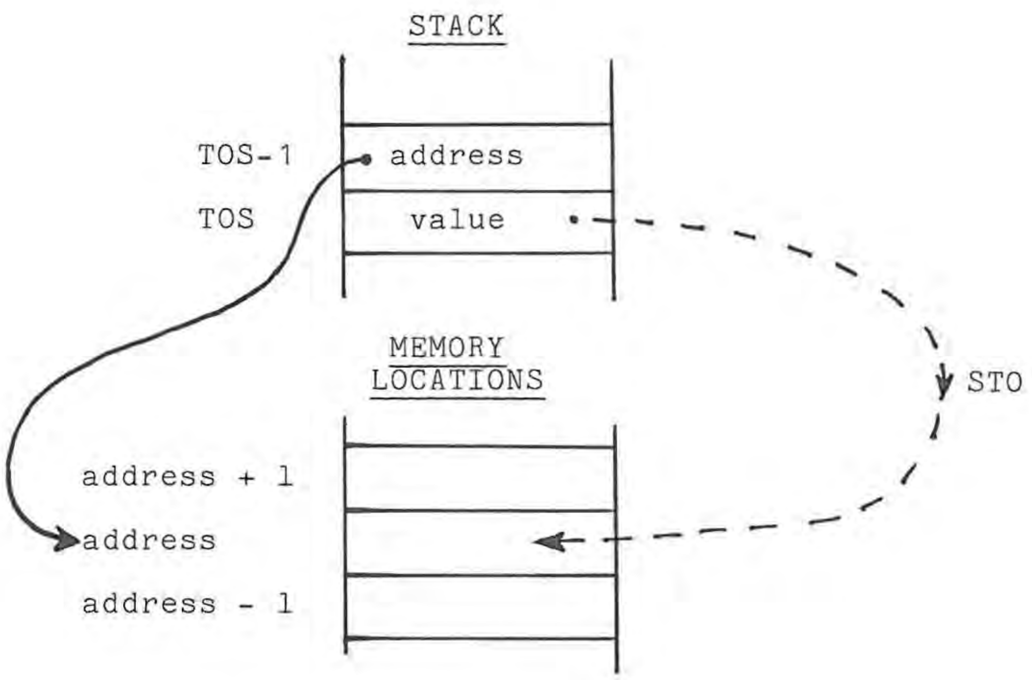
Each intermediate code may take zero or more arguments (the current implementation allows a maximum of 2), whose values are known at code generation time.

2.2.2 Conversion of Intermediate Code to Machine Code

For the reasons above, a one-to-many mapping is usual when writing conventional machine code segments to produce the same effect as a single intermediate code.

$$\begin{tikzcd}\text{INTERMEDIATE CODE} \arrow[r] & \boxed{\text{TRANSLATION}} \arrow[r] & \begin{array}{l}\text{SEVERAL}\\ \text{MACHINE}\\ \text{INSTRUCTIONS}\end{array}\end{tikzcd}$$

For example, suppose an intermediate code STO stores the value at the top of the stack (TOS) in the memory location whose address is at TOS-1 (next from top).



A code segment written in 8Ø8Ø assembler code to produce the effect of STO might be:

```
POP  D    ; get value
POP  H    ; get address
MOV  M,E  ; store low order byte
INX  H    ; increment address
MOV  M,D  ; store high order byte
```

The body of machine instructions, which produces the same effect as the intermediate code, is called a **code-skeleton.**

The code-skeleton definition program is the combination of an editor and an assembler. It allows the user to define a code-skeleton for each of the intermediate codes using conventional assembler mnemonics.

As each code skeleton is entered, it is checked against the table of word and instruction formats for validity. Help messages and meaningful error diagnostics are constantly available, making this an easy program to drive.

The user is able to perform the usual types of editing operations on his code skeletons. The editor is essentially a line-at-a-time editor, which performs operations on the "current" instruction. Control keys are available for listing the entire code skeleton, printing the current code skeleton, stepping through the code-skeleton either forwards or backwards, jumping to the top of the code skeleton, and deleting and inserting instructions.

2.2.3 Enhancements

If several code skeletons use a similar instruction sequence, then the implementor can make use of a facility which exists for writing run-time subroutines. These are really just code-skeletons which have a name associated with them. They will be included only once in the object program, and can be called from any code-skeleton, simply by including a referencing instruction with the name of the run-time subroutine as its address argument.

In addition, a number of standard run-time subroutines must be written, usually as part of the loader (see chapter 4), and included in every program. These are subroutines for performing very common operations such as input/output, arithmetic and logic operations, and standard function calls. A number of reserved names represent these run-time subroutines, and are recognised when used in the address field of a referencing instruction.

Finally, it is necessary to be able to define memory addresses relative to some other address values known at compile-time. For example, the reserved name PC represents the code-generation time address of the current instruction.

Reserved names also exist to represent the values of the two arguments which an intermediate code might have. Table 8.2.1 and its associated text provide more information on this aspect.

These reserved and user-declared subroutine names are the only valid non-numeric values which may appear in an address field. They may optionally be followed by a "+" or "-" sign and a numeric value, to represent an address (or value) relative to the value represented by the given name. The numeric offset is in "smallest addressable units".

For example, the following code skeleton reads a positive integer:

```
CALL IREAD     ; Call run-time subroutine
POP  D         ; Get result of read
PUSH D         ; but leave a copy on the stack
MOV  A,D       ; Test result < Ø
ANI  128       ; Set zero flag if sign-bit off
JZ   PC+6      ; Cond jump to current PC + 6 bytes
CALL ERR       ; Report an error
```

Here IREAD is a reserved run-time subroutine name (see table 8.2.1) and ERR a user defined run-time subroutine skeleton, whose actual addresses will be filled in at code generation time. PC will be the current instruction address, not known at code-skeleton definition time, but only at code generation time.

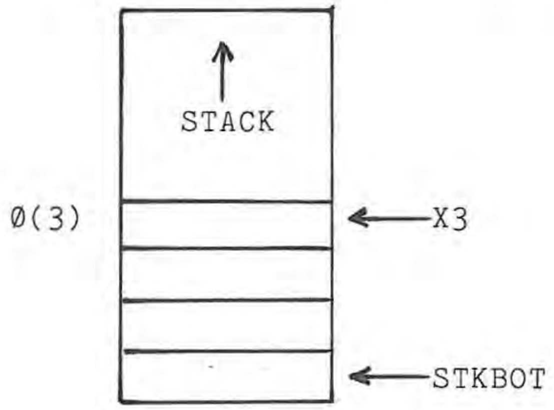
Code-skeletons are represented in such a way that they can be relocated to any address in memory, and are stored on a disc file for use by the code generator.

It can be seen that the user defining the code generator merely has to be able to write an assembly language code sequence to produce the desired effect of each intermediate code.

2.1.4 Code-Skeletons on Specific Machines

Since both the PDP-11 and the 8Ø8Ø computers have built-in stack facilities, they are ideal target machines onto which to map intermediate codes for a hypothetical stack machine. However, non stack-oriented machines, such as the ICL 19ØØ, show up some inhibiting features. This does not mean that it is impossible to write code-skeletons to produce the effects of the intermediate language codes on this machine, the implementation will just be a little more clumsy than it is on stack-oriented machines.

On the ICL 19ØØ, words are 24 bits long. To implement a stack for pushing and popping single length words, one might designate one of the registers to point to the top of the stack. The most convenient method might be to set aside one of the index registers. Let us use register 3 (denoted X3), and make use of indexed addressing to refer to the stack. Assume that the stack begins at the address stored in STKBOT (used to test for underflow) and moves upwards in memory towards higher addresses. At the start of execution, X3 should be set by the instruction LDX 3 STKBOT.



The push instruction would have to be implemented as:

```
|1    |7      |13  |16
|     |STO    | 6  | Ø(3)     [ push X6
|     |ADN    | 3  | 1        [ increment SP
```

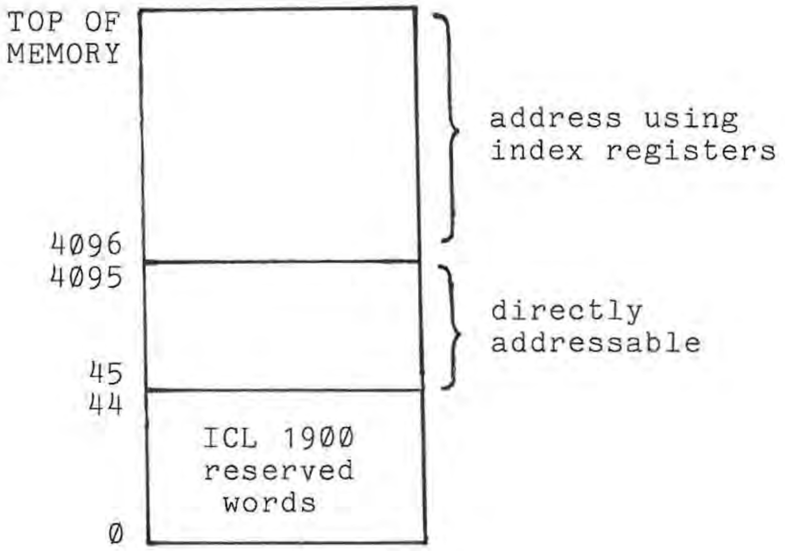
and the pop instruction would be:

```
|1    |7      |13  |16
|     |LDX    | 6  | Ø(3)     [ pop X6
|     |SBN    | 3  | 1        [ decrement SP
```

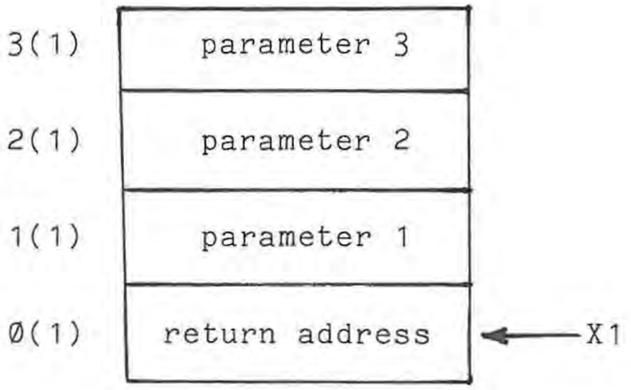
The ICL 19ØØ shows up another addressing feature which deserves special mention. The normal instruction form described in section 2.1.5 has a 12-bit address field.

XXX	FFFFFFF	MM	NNNNNNNNNNNN
3	7	2	12

This restricts the direct addressing to memory locations below address 4Ø96 only (the first 45 of which are reserved by the ICL 19ØØ operating system for use as registers and work space). Higher memory locations must be addressed using index registers.



Indexed addressing turns out to be useful for implementing procedure stack frames. One simply sets an index register to the address of the bottom of the stack frame and uses an absolute offset to reference each word. For example



However, having to use some indirect addressing method to access ordinary variables can become costly.

The ICL 1900 places another restraint on the size of executable code, by the format of its branch instructions.

XXX	FFFFFF	NNNNNNNNNNNNNNN
3	6	15

Since the address field of the branch instruction contains an absolute address, a 15-bit field means that the executable code should not extend beyond address 32767 (An extended branch mode is, however, available).

Whereas the generalized code generator described in this document favours a processor which can address all memory locations with equal ease, a more efficient run-time system could be produced for the ICL 1900 series if data were stored below address 4096, code below address 32768, and procedure stack frames in high memory. It should be stressed that a viable code generator could be developed for an ICL 1900 computer using the code generator synthesiser, but it would not be able to match the efficiency of a tailor-made compiler. Fortunately, most modern computers are able to address all their memory (within reasonable limits) with equal ease.

2.3 THE CODE GENERATION PROGRAM

This program simply replaces a sequence of intermediate codes by their respective code skeletons, relocating the skeletons in memory, and filling in addresses known only at code generation time. Because this is a relatively simple process, the code generation time is short.

A few simple optimizations are included at present, but for the reasons already mentioned, these are not extensive.

2.4 WHEN TO USE THE PROGRAMS

To implement any compiler using the machine-independent code generator, one has to run the machine definition program first. It should only be necessary to run this program once for a new computer. Subsequent runs might be necessary to debug the instruction definition table or to add new definitions to the table. Since some machine instructions have more obscure uses than others, one might initially use only a subset of the target machine's instruction set to implement a new compiler.

Once the machine definition table has been established, the implementor will run the code-skeleton definition table once for each compiler he wishes to implement on the target computer. Again, subsequent runs might be necessary to debug the code-skeleton table or add new code-skeletons to the table.

Whereas the previous two programs are run only once for a particular compiler implementation, the general table-driven code generator (and its associated parser) is run every time a program is compiled. The output is a machine representation of the particular source program.

3. A SAMPLE IMPLEMENTATION

This chapter describes the implementation of a code generator for the CLANG parser on an 8080 processor using the automated code generation system. Some knowledge of the CLANG intermediate language codes [23] and the assembly language and architecture of the 8080 processor [24] - [28] is assumed. Readers not familiar with these topics are advised to consult the references listed.

3.1 THE CLANG LANGUAGE

The CLANG language is a Pascal-like one used at Rhodes University as the basis of a course in compiler writing, and for testing the principles of concurrent programming. It exists as a set of subsets, of increasing complexity. The subset used here allows simple control structures (if-then-else, while) nested procedure declarations, recursion, simple parameter passing by reference and by value, but supports integers, and arrays of integers, as its only data type. The higher facilities for data abstraction [23] have not been included at this stage. A sample program is given below in section 3.5.

A CLANG Parser/Interpreter system has been written in Pascal and implemented on various machines, notably using UCSD Pascal, by P.D. Terry. The system was adapted to run under Pascal MT+ by the author, and a listing of the parser and interpreter sections of this appears in full in Appendix E.

3.2 THE MACHINE DEFINITION

The following subset of the 8Ø8Ø assembly instructions is required to produce the effect of the CLANG intermediate codes.

Arithmetic:	ADC r	ADD r	SBB r	SUB r
	DCR r	INR r		
	DCX rp	INX rp	DAD rp	
Branching:	CALL addr	RET	PCHL	
	JMP addr	JZ addr	JNZ addr	
Logical:	ANA r	ANI i	ORA r	
Data movement:	MOV r,r	MVI r,i	LXI rp,i	
	LHLD addr	SHLD addr	XCHG	
Stack:	PUSH rp	POP rp	SPHL	
Miscellaneous:	NOP			

where "r" is a register (A,B,C,D,E,H,L,M), "rp" is a register pair (B,D,H,PSW), "i" is an immediate value and "addr" a 2-byte address.

The word and instruction definitions for these instructions are shown below (and more fully in the example of Appendix A5). Since the assembler mode of the skeleton definition program has no directive for setting aside a word of store for use as data, the instruction, ASCII, has been defined as an 8-bit word which has the first bit always zero and subsequent bits variable. This enables the implementor to set aside a byte of memory and initialise it to the value of an ASCII character code.

e.g. ASCII 65 for the character "A".

ASCII is not used in the example below, but proved to be useful when implementing a code generator on the 8Ø8Ø processor for the enhanced Pascal-s [39] compiler (see chapter 6). Setting aside other areas of data within a code skeleton can be achieved using similar clumsy definitions. Assembler directives would be an obvious future enhancement to the system.

Word Definitions

1.	8-bit register form	FFFF F RRR
2.	8-bit move form	FF RRR SSS
3.	Immediate move	FF RRR GGG IIII IIII
4.	8-bit immediate	FFFF FFFF IIII IIII
5.	3-BYTE form	FFFF FFFF AAAA AAAA AAAA AAAA
6.	Double-reg form	FF DD GGGG
7.	Inr-dcr form	FF RRR GGG
8.	No-operand form	FFFF FFFF
9.	3-BYTE double-reg	FF DD GGGG AAAA AAAA AAAA AAAA
1Ø.	ASCII byte fiddle	O CCCC CCC

Instruction Definitions

1.	ADC	Machine format:	10ØØ1 RRR	
		Assembler format:	ADC R	
		Arg 1. field R:	B	(Ø , ØØØ)
			C	(1 , ØØ1)
			D	(2 , Ø1Ø)
			E	(3 , Ø11)
			H	(4 , 1ØØ)
			L	(5 , 1Ø1)
			M	(6 , 11Ø)
			A	(7 , 111)
2.	ADD	Machine format:	1ØØØØ RRR	
		Assembler format:	ADD R	
		Arg 1. field R:	B	(Ø , ØØØ)
			C	(1 , ØØ1)
			D	(2 , Ø1Ø)
			E	(3 , Ø11)
			H	(4 , 1ØØ)
			L	(5 , 1Ø1)
			M	(6 , 11Ø)
			A	(7 , 111)
3.	ANA	Machine format:	1Ø1ØØ RRR	
		Assembler format:	ANA R	
		Arg 1. field R:	B	(Ø , ØØØ)
			C	(1 , ØØ1)
			D	(2 , Ø1Ø)
			E	(3 , Ø11)
			H	(4 , 1ØØ)
			L	(5 , 1Ø1)
			M	(6 , 11Ø)
			A	(7 , 111)
4.	MOV	Machine format:	Ø1 RRR SSS	
		Assembler format:	MOV R,S	
		Arg 1. field R:	B	(Ø , ØØØ)
			C	(1 , ØØ1)
			D	(2 , Ø1Ø)
			E	(3 , Ø11)
			H	(4 , 1ØØ)
			L	(5 , 1Ø1)
			M	(6 , 11Ø)
			A	(7 , 111)
		Arg 2. field S:	B	(Ø , ØØØ)
			C	(1 , ØØ1)
			D	(2 , Ø1Ø)
			E	(3 , Ø11)
			H	(4 , 1ØØ)
			L	(5 , 1Ø1)
			M	(6 , 11Ø)
			A	(7 , 111)

```
5.   ANI   Machine format:     111ØØ11Ø IIIIIIII
           Assembler format:   ANI I
           Arg 1. field I:     Ø..255     (ØØØØØØØØ..11111111)

6.   CALL  Machine format:     11ØØ11Ø1 AAAAAAAAAAAAAAAA
           Assembler format:   CALL A
           Arg 1. field A:     NO ARGUMENT CHECKS (width=16)

7.   JMP   Machine format:     11ØØØØ11 AAAAAAAAAAAAAAAA
           Assembler format:   JMP A
           Arg 1. field A:     NO ARGUMENT CHECKS (width=16)

8.   JZ    Machine format:     11ØØ1Ø1Ø AAAAAAAAAAAAAAAA
           Assembler format:   JZ A
           Arg 1. field A:     NO ARGUMENT CHECKS (width=16)

9.   JNZ   Machine format:     11ØØØØ1Ø AAAAAAAAAAAAAAAA
           Assembler format:   JNZ A
           Arg 1. field A:     NO ARGUMENT CHECKS (width=16)

1Ø.  DAD   Machine format:     ØØ DD 1ØØ1
           Assembler format:   DAD D
           Arg 1. field D:     B      ( Ø , ØØØ )
                               D      ( 1 , ØØ1 )
                               H      ( 2 , Ø1Ø )
                               SP     ( 3 , Ø11 )

11.  INX   Machine format:     ØØ DD ØØ11
           Assembler format:   INX D
           Arg 1. field D:     B      ( Ø , ØØØ )
                               D      ( 1 , ØØ1 )
                               H      ( 2 , Ø1Ø )
                               SP     ( 3 , Ø11 )

12.  DCX   Machine format:     ØØ DD 1Ø11
           Assembler format:   DCX D
           Arg 1. field D:     B      ( Ø , ØØØ )
                               D      ( 1 , ØØ1 )
                               H      ( 2 , Ø1Ø )
                               SP     ( 3 , Ø11 )

13.  LHLD  Machine format:     ØØ1Ø1Ø1Ø AAAAAAAAAAAAAAAA
           Assembler format:   LHLD A
           Arg 1. field A:     NO ARGUMENT CHECKS (width=16)

14.  SHLD  Machine format:     ØØ1ØØØ1Ø AAAAAAAAAAAAAAAA
           Assembler format:   SHLD A
           Arg 1. field A:     NO ARGUMENT CHECKS (width=16)
```

```
15.  DCR   Machine format:     00 RRR 101
           Assembler format:   DCR R
           Arg 1. field R:     B      ( 0 , 000 )
                               C      ( 1 , 001 )
                               D      ( 2 , 010 )
                               E      ( 3 , 011 )
                               H      ( 4 , 100 )
                               L      ( 5 , 101 )
                               M      ( 6 , 110 )
                               A      ( 7 , 111 )

16.  INR   Machine format:     00 RRR 100
           Assembler format:   INR R
           Arg 1. field R:     B      ( 0 , 000 )
                               C      ( 1 , 001 )
                               D      ( 2 , 010 )
                               E      ( 3 , 011 )
                               H      ( 4 , 100 )
                               L      ( 5 , 101 )
                               M      ( 6 , 110 )
                               A      ( 7 , 111 )

17.  ORA   Machine format:     10110 RRR
           Assembler format:   ORA R
           Arg 1. field R:     B      ( 0 , 000 )
                               C      ( 1 , 001 )
                               D      ( 2 , 010 )
                               E      ( 3 , 011 )
                               H      ( 4 , 100 )
                               L      ( 5 , 101 )
                               M      ( 6 , 110 )
                               A      ( 7 , 111 )

18.  NOP   Machine format:     00000000
           Assembler format:   NOP

19.  PCHL  Machine format:     11101001
           Assembler format:   PCHL

20.  POP   Machine format:     11 DD 0001
           Assembler format:   POP D
           Arg 1. field D:     B      ( 0 , 000 )
                               D      ( 1 , 001 )
                               H      ( 2 , 010 )
                               PSW    ( 3 , 011 )

21.  PUSH  Machine format:     11 DD 0101
           Assembler format:   PUSH D
           Arg 1. field D:     B      ( 0 , 000 )
                               D      ( 1 , 001 )
                               H      ( 2 , 010 )
                               PSW    ( 3 , 011 )

22.  RET   Machine format:     11001001
           Assembler format:   RET
```

```
23.  SPHL  Machine format:     11111001
           Assembler format:   SPHL

24.  XCHG  Machine format:     11101011
           Assembler format:   XCHG

25.  SBB   Machine format:     10011 RRR
           Assembler format:   SBB R
           Arg 1. field R:     B      ( 0 , 000 )
                               C      ( 1 , 001 )
                               D      ( 2 , 010 )
                               E      ( 3 , 011 )
                               H      ( 4 , 100 )
                               L      ( 5 , 101 )
                               M      ( 6 , 110 )
                               A      ( 7 , 111 )

26.  SUB   Machine fomat:     10010 RRR
           Assembler format:   SUB R
           Arg 1. field R:     B      ( 0 , 000 )
                               C      ( 1 , 001 )
                               D      ( 2 , 010 )
                               E      ( 3 , 011 )
                               H      ( 4 , 100 )
                               L      ( 5 , 101 )
                               M      ( 6 , 110 )
                               A      ( 7 , 111 )

27.  MVI   Machine format:     00 RRR 110 IIIIIIII
           Assembler format:   MVI R,I
           Arg 1. field R:     B      ( 0 , 000 )
                               C      ( 1 , 001 )
                               D      ( 2 , 010 )
                               E      ( 3 , 011 )
                               H      ( 4 , 100 )
                               L      ( 5 , 101 )
                               M      ( 6 , 110 )
                               A      ( 7 , 111 )
           Arg 2. field I:     0..255     (00000000..11111111)

28.  LXI   Machine format:     00 DD 0001 AAAAAAAAAAAAAAAA
           Assembler format:   LXI D,A
           Arg 1. field D:     B      ( 0 , 000 )
                               D      ( 1 , 001 )
                               H      ( 2 , 010 )
                               SP     ( 3 , 011 )
           Arg 2. field A:     NO ARGUMENT CHECKS (width=16)

29.  ASCII Machine format:     0 CCCCCCC
           Assembler format:   ASCII C
           Arg 1 field C:      0..127     (0000000..1111111)
```

3.3 CODE-SKELETON DEFINITIONS

The level 2 CLANG parser generates 27 intermediate language codes, one of which is not used by the code generator (STK is used only in aiding students to understand the stack frame concept). Codes take zero, one or two arguments. The current Code-Skeleton definition program allows a maximum of two arguments per code, called X and Y. CLANG generally uses the first argument to indicate the level at which a declaration was made, and the second to indicate an address or offset.

The CLANG pseudo-machine has a set of "display" registers, used to point to the most deeply nested procedure stack frame for a particular declaration level (useful in handling variable addressing) and a "base register", pointing to the base of the top stack frame on the stack. The most convenient method of implementing these pointers in the target program would undoubtedly be to use the processor's own registers. Unfortunately, the 8080 does not have enough register pairs for the implementor to dedicate some of them permanently to a particular task, so memory locations have to be used instead. In order to be able to refer to these memory locations by name, they are "defined" as run-time subroutines (simply a block of memory which has a name).

Run-time subroutine BASERE is a 2-byte pointer.

```
BASERE:   NOP
          NOP
```

Run-time subroutine DISPLA allows a maximum declaration nesting of 5 (i.e. five 2-byte pointers).

```
DISPLA:   NOP
          NOP
DISPLA+2: NOP
          NOP
DISPLA+4: NOP
          NOP
DISPLA+6: NOP
          NOP
DISPLA+8: NOP
          NOP
```

Four more run-time subroutines are defined to be called from code-skeletons.

GETDIS is called with a value X in the H-L register pair to return the value of the Xth display register in the D-E pair.

```
GETDIS:   DAD  H          ; X * 2 for 16-bit addressing
          LXI  B,DISPLA
          DAD  B          ; + DISPLA
          MOV  E,M        ; get value from
          INX  H          ; DISPLA  + (2*X)
          MOV  D,M
          RET
```

ADDRCD is called after a call to GETDIS to use the display value in D-E and a displacement in H-L to calculate the address of a local variable.

```
ADDRCD:   DAD  H          ; * 2 for 16-bit addressing
          DAD  D          ; + base address
          RET             ; address in H-L
```

STORE is a subroutine to store TOS (top of stack) value at TOS-1 address.

```
          POP  B          ; return address
          POP  D          ; get value
          POP  H          ; get address
          MOV  M,E        ; store low order byte
          INX  H          ; increment address
          MOV  M,D        ; store high order byte
          PUSH B          ; restore return address
          RET
```

The last run-time subroutine, START, has a special initialization status. It stores the stack's first word in DISPLA[1] and BASERE.

```
          LXI  D,-2       ; D,E = -2
          LXI  H,Ø        ;
          DAD  SP         ; H,L = SP
          DAD  D          ; subtract 2
          SHLD DISPLA
          SHLD BASERE
```

The CLANG Intermediate Language Codes (ILC's) are defined as follows:

```
ILC Ø:    LIT Y  -  push integer literal.

          LXI  D,Y        ; literal Y
          PUSH D
```

ILC 1: LDA X Y - Push address of variable with level X, offset Y.

```
LXI  H,X-1
CALL GETDIS     ; display value
LXI  H,YN       ; negative offset
CALL ADDRCD     ; address of variable
PUSH H
```

ILC 2: CAL X Y - Call a procedure - set up a procedure stack frame and jump to the procedure code. X is the level and Y the address.

```
LXI  H,X        ; get old display
CALL GETDIS     ;   in D-E
LHLD BASERE     ; get 'base reg'
XCHG            ; store in D,E
PUSH H          ; store old display
                ; SP now points to 1st entry on
                ; new stack frame.
LXI  H,Ø        ; H,L = SP
DAD  SP         ;
SHLD BASERE     ; new base reg
PUSH D          ; old base reg
XCHG            ; SP value in D-E
LXI  H,X        ; store SP value in
DAD  H          ;    DISPLA  + (2 * X)
LXI  B,DISPLA   ;         .
DAD  B          ;         .
MOV  M,E        ;         .
INX  H          ;         .
MOV  M,D        ;
LXI  B,PC+1Ø    ; get return address
PUSH B          ;    onto stack
POP  H          ; reduce SP so that INT will
POP  H          ;    increase it by the
POP  H          ;    correct amount in the
                ;    called procedure.
JMP  YA         ; goto new procedure
```

```
ILC 3:    RT X Y  -  Return from a procedure - discard stack
                     frame.  X is the level, Y indicates the
                     number   of   words   of   parameter
                     information (= Y-1).

               LHLD BASERE     ; get 'base reg'
               LXI  D,-4
               DAD  D          ; reduce base by 2 entries
               SPHL            ; new value for SP
               POP  D          ; ret addr
               POP  H          ; back link
               SHLD BASERE     ; reset current base
               LXI  H,X-1      ; display address
               DAD  H          ;
               LXI  B,DISPLA
               DAD  B          ; (in H-L)
               POP  B          ; display copy
               MOV  M,C
               INX  H
               MOV  M,B
                               ; remove Y-1 parameters
               LXI  H,Y-1
               DAD  SP
               SPHL
               XCHG            ; Return to the calling
               PCHL            ;              procedure

ILC 4:    STK - not implemented.


ILC 5:    INT Y  -  Increment stack pointer by Y.
               LXI  H,YN
               DAD  H          ; 16-bit words
               DAD  SP         ; H-L = SP
               SPHL            ; new value for SP


ILC 6:    IND Y  -  Evaluate   address   of  a   subscripted
                    variable.   At this point,  the top two
                    entries on the stack will be:
                         TOP  : subscript value
                         TOP-1: address of Øth element.

               LXI  D,Ø        ; low bound is always zero
               PUSH D
               LXI  D,Y        ; Y-value gives upper bound
               PUSH D
               CALL CHKSUB     ; Run time subroutine to check
                               ; the range of the subscript
               POP  H          ; Get subscript value
               DAD  H          ;   and double it to allow for
               PUSH H          ;   16 bit addressing
               CALL ISUB       ; Subtract subscript value from
                               ; address of element zero - the
                               ; array is stored on the stack
```

```
ILC 7:    BRN Y  -  Unconditional branch to Y.

               JMP  YA

ILC 8:    BZE Y  -  Branch to Y if top of stack is zero.

               POP  D
               MOV  A,D     ;
               ORA  E       ; set z-flag if both Ø
               JZ   YA      ; Issue branch order

ILC 9:    NEG  -  Negate top of stack.

               POP  D       ; get integer
               LXI  H,Ø     ; H-L = Ø
               PUSH H       ; Ø - INTEGER
               PUSH D       ;
               CALL ISUB    ; integer subtract

ILC 1Ø:   ADD  -  Add top two integers on stack.

               CALL IADD    ; call to monitor routine.

ILC 11:   SUB  -  Subtract top two integers on stack.

               CALL ISUB    ; call to monitor routine.

ILC 12:   MUL  -  Multiply top two integers on stack.

               CALL IMUL    ; call to monitor routine.

ILC 13:   DIV  -  Divide top two integers on stack.

               CALL IDIV    ; call to monitor routine.

ILC 14:   OD   -  Replace TOS by 1 if it is odd, by Ø
                  otherwise.

               POP  D          ; get integer
               MOV  A,E        ; get low order byte
               ANI  1          ; retain only bit Ø
               MOV  E,A        ; low order byte = 1 if odd
               MVI  D,Ø        ; high order byte = Ø
               PUSH D          ;
```

```
ILC 15:   EQL  -  Test top two stack elements for equality.
                  Return 1 or Ø to indicate true or false.

               CALL ISUB    ; subtract
               POP  D       ; get result
               MOV  A,D     ;
               ORA  E       ; set z-flag
               LXI  D,Ø     ; does not alter z-flag
               JNZ  PC+4
               INR  E       ; set result to 1
               PUSH D       ;

ILC 16:   NEQ  -  Test top two elements for inequality.
                  Return 1 or Ø to indicate true or false.

               CALL ISUB    ; subtract
               POP  D       ; get result
               MOV  A,D     ;
               ORA  E       ; set z-flag
               LXI  D,Ø     ; does not alter z-flag
               JZ   PC+4
               INR  E       ; set result to 1
               PUSH D       ;

ILC 17:   LSS  -  Test top two elements for TOS-1 < TOS.
                  Return 1 or Ø.

               CALL ISUB    ; subtract
               POP D        ; get result
               MOV A,D      ; test result < Ø
               ANI 128      ; set z if sign-bit off
               LXI D,Ø      ; does not alter z-flag
               JZ  PC+4
               INR E        ; set result to 1
               PUSH D       ;

ILC 18:   GEQ  -  Test top two elements for TOS-1 >= TOS.
                  Return 1 or Ø.

               CALL ISUB    ; subtract
               POP D        ; get result
               MOV A,D      ; test result < Ø
               ANI 128      ; set z if sign-bit off
               LXI D,Ø      ; does not alter z-flag
               JNZ PC+4
               INR E        ; set result to 1
               PUSH D       ;
```

```
ILC 19:  GTR  -  Test top two elements for TOS-1 > TOS.
                 Return 1,Ø

               CALL ISUB     ; subtract
               POP  D        ; get result
               PUSH D        ; but don't destroy it
               MOV  A,D      ; first test for zero
               ORA  E        ; set z-flag
               LXI  D,Ø      ; does not alter z-flag
               JZ   PC+14    ; return a Ø if equal
                             ; now test the sign
               POP  D        ;
               MOV  A,D      ; test result > Ø
               ANI  128      ; set z if sign-bit off
               LXI  D,Ø      ; does not alter z-flag
               JNZ  PC+4
               INR  E        ; set result to 1
               PUSH D        ;


ILC 2Ø:  LEQ  -  Test top two elements for TOS-1 <= TOS.
                 Return 1,Ø

               CALL ISUB     ; subtract
               POP  D        ; get result
               PUSH D        ; but don't destroy it
               MOV  A,D      ; first test for zero
               ORA  E        ; set z-flag
               LXI  D,Ø      ; does not alter z-flag
               POP  H        ; does not alter z-flag
               JZ   PC+9     ; return a 1 if equal
                             ; now test the sign
               MOV  A,H      ; test result < Ø
               ANI  128      ; set z if sign-bit off
               JZ   PC+6
               LXI  D,1      ; set result to 1
               PUSH D        ;


ILC 21:  STO  -  Store TOS value at TOS-1 address.

               CALL STORE


ILC 22:   HLT   -  Stop execution, return to operating
                   system. For a CP/M based implementation
                   this is just a "warm boot".

               JMP  Ø
```

ILC 23: INN - Read an integer value. TOS = address of variable for value to be read into.

```
LXI  D,Ø
PUSH D      ; channel
CALL IREAD  ; value returned on stack
CALL STORE
```

ILC 24: PRN - Write out integer on TOS.

```
POP  H       ; get value
LXI  D,Ø
PUSH D       ; channel
PUSH H       ; push value
PUSH D       ; field width=Ø
CALL IWRTE
```

ILC 25: NL - New line on output.

```
LXI  D,Ø
PUSH D       ; channel
CALL WRTLN
```

ILC 26: PRS - Print a string. The characters will already have been pushed onto the stack followed by the string length.

```
LXI  D,Ø
PUSH D       ; channel
CALL STRWRT
```

ILC 27: LDX - Substitute Address on TOS with its value.

```
POP  H       ; get address
MOV  E,M     ; load low order byte
INX  H       ; increment address
MOV  D,M     ; load high order byte
PUSH D       ; resultant value
```

3.4 THE CODE-GENERATION DRIVING TABLE

Once the definitions of sections 3.2 and 3.3 have been entered into the appropriate tables and stored on disc files and the loader and monitor routines (described in chapter 4) have been written, the penultimate task of the implementor (the final task is to test the system thorough/ly) is to set up the table which drives the general code generator. The format of this table is detailed in section 9.3. The table may be set up using an editor or by running the program written for this purpose.

For the CLANG system on the 8080 processor, using the monitor routines listed in Appendix D, the table would appear as:

8080.MAC	(name of machine definition file)
CLANG.SKL	(name of code skeleton file)
8	(bits in smallest addressable unit)
4	(bits per digit in output file)
2	(number of words in an address field)
4352	(initial PC = 1100 hex)
259	(address of RTS WRTLN)
262	(address of RTS IWRTE)
265	(address of RTS FPWRTE)
268	(address of RTS STRWRT)
271	(address of RTS IREAD)
274	(address of RTS FPREAD)
277	(address of RTS FLOAT)
280	(address of RTS FIX)
283	(address of RTS IADD)
286	(address of RTS ISUB)
289	(address of RTS IMUL)
292	(address of RTS IDIV)
295	(address of RTS FPADD)
298	(address of RTS FPSUB)
301	(address of RTS FPMUL)
304	(address of RTS FPDIV)
307	(address of RTS ERROR)
310	(address of RTS CHKSUB)

3.5 A SAMPLE PROGRAM

The program listed in table 3.5.1 might not appeal to the Pascal precisian, but it serves well in demonstrating some features of CLANG and its implementation.

TABLE 3.5.1

A SAMPLE CLANG PROGRAM

```
{$O+ Compiler directive requesting object code listing}
program REVERSE; { Write a list of numbers in reverse }
var TERMINATOR;

    procedure GO;
    var LOCAL;

        procedure REORDER;
        var D;
        begin
          READ(D);
          if TERMINATOR <> D then REORDER;
          WRITE(D)
        end; { REORDER }

    begin { GO }
      REORDER
    end;  { GO }

begin { REVERSE }
  TERMINATOR := 0;
  GO
end.  { REVERSE }
```

The intermediate language code version of this program (as generated by the CLANG parser listed in Appendix E, which produces a file in the format required by the general code generator), is displayed in table 3.5.2 and the the object program generated by the code generator defined in this chapter is shown in table 3.5.3. The loader of Appendix D is able to load and execute the final target code. Although they are not used by the CLANG system, Appendix D includes a full complement of floating-point, type conversion and file handling routines. This enables it to be used with the enhanced Pascal-s compiler [39].

TABLE 3.5.2

CLANG INTERMEDIATE LANGUAGE CODES

```
             ADDRESS  MNEMONIC  FUNCTION LEVEL ADDRESS

                   Ø     BRN        7      Ø     21----------+
+---------->       1     BRN        7      Ø     18-------+  |
|    +--+--->      2     BRN        7      Ø      3---+   |  |
|    |  |              (start of procedure REORDER)   |   |  |
|    |  |          3     INT        5      Ø      4 <-+   |  |
|    |  |          4     LDA        1      3      3       |  |
|    |  |          5     INN       23                     |  |
|    |  |          6     LDA        1      1      3       |  |
|    |  |          7     LDX       27                     |  |
|    |  |          8     LDA        1      3      3       |  |
|    |  |          9     LDX       27                     |  |
|    |  |         1Ø     NEQ       16                     |  |
|    |  |         11     BZE        8      Ø     13---+   |  |
|    |  +---      12     CAL        2      2      2   |   |  |
|    |            13     LDA        1      3      3 <-+   |  |
|    |            14     LDX       27                     |  |
|    |            15     PRN       24                     |  |
|    |            16     NL        26                     |  |
|    |            17     RET        3      3      1       |  |
|    |                 (start of procedure GO)            |  |
|    |            18     INT        5      Ø      4 <-----+  |
|    +------      19     CAL        2      2      2          |
|                 2Ø     RET        3      2      1          |
|                      (start of program REVERSE)            |
|                 21     INT        5      Ø      4 <--------+
|                 22     LDA        1      1      3
|                 23     LIT        Ø      Ø      Ø
|                 24     STO       21
+---------        25     CAL        2      1      1
                  26     HLT       22
```

TABLE 3.5.3

8080 ABSOLUTE TARGET CODE

```
              1100  NOP      BASERE
              1101  NOP
              1102  NOP      DISPLA
              1103  NOP
              1104  NOP
              1105  NOP
              1106  NOP
              1107  NOP
              1108  NOP
              1109  NOP
              110A  NOP
              110B  NOP
              110C  DAD   H  GETDIS
              110D  LXI   B,1102
              1110  DAD   B
              1111  MOV   E,M
              1112  INX   H
              1113  MOV   D,M
              1114  RET
              1115  DAD   H  ADDRCD
              1116  DAD   D
              1117  RET
              1118  POP   B  STORE
              1119  POP   D
              111A  POP   H
              111B  MOV   M,E
              111C  INX   H
              111D  MOV   M,D
              111E  PUSH  B
              111F  RET
              1120  LXI   D,FFFE  START
              1123  LXI   H,0000
              1126  DAD   SP
              1127  DAD   D
              1128  SHLD  1102
              112B  SHLD  1100
      BRN 21  112E  JMP   1247 ----> REVERSE
 G -> BRN 18  1131  JMP   11F8 ----> GO
 R -> BRN 3   1134  JMP   1137 --+
      INT 4   1137  LXI   H,FFFC <-+ REORDER
              113A  DAD   H
              113B  DAD   SP
              113C  SPHL
   LDA 3 3    113D  LXI   H,0002
              1140  CALL  110C
              1143  LXI   H,FFFD
              1146  CALL  1115
              1149  PUSH  H
   INN        114A  LXI   D,0000
              114D  PUSH  D
              114E  CALL  010F
              1151  CALL  1118
   LDA 1 3    1154  LXI   H,0000
              1157  CALL  110C
              115A  LXI   H,FFFD
              115D  CALL  1115
              1160  PUSH  H
   LDX        1161  POP   H
              1162  MOV   E,M
              1163  INX   H
              1164  MOV   D,M
              1165  PUSH  D
   LDA 3 3    1166  LXI   H,0002
              1169  CALL  110C
              116C  LXI   H,FFFD
              116F  CALL  1115
              1172  PUSH  H
```

```
   LDX        1173  POP   H
              1174  MOV   E,M
              1175  INX   H
              1176  MOV   D,M
              1177  PUSH  D
   NEQ        1178  CALL  011E
              117B  POP   D
              117C  MOV   A,D
              117D  ORA   E
              117E  LXI   D,0000
              1181  JZ    1185 --+
              1184  INR   E      |
              1185  PUSH  D <----+
   BZE 13     1186  POP   D
              1187  MOV   A,D
              1188  ORA   E
              1189  JZ    11B5 ---------+
   CAL 2 2    118C  LXI   H,0002        |
              118F  CALL  110C          |
              1192  LHLD  1100          |
              1195  XCHG                |
              1196  PUSH  H             |
              1197  LXI   H,0000        |
              119A  DAD   SP            |
              119B  SHLD  1100          |
              119E  PUSH  D             |
              119F  XCHG                |
              11A0  LXI   H,0002        |
              11A3  DAD   H             |
              11A4  LXI   B,1102        |
              11A7  DAD   B             |
              11A8  MOV   M,E           |
       R      11A9  INX   H             |
       ^      11AA  MOV   M,D           |
       |      11AB  LXI   B,11B5        |
       |      11AE  PUSH  B             |
       |      11AF  POP   H             |
       |      11B0  POP   H             |
       |      11B1  POP   H             |
       +----- 11B2  JMP   1134          |
   LDA 3 3    11B5  LXI   H,0002 <------+
              11B8  CALL  110C
              11BB  LXI   H,FFFD
              11BE  CALL  1115
              11C1  PUSH  H
   LDX        11C2  POP   H
              11C3  MOV   E,M
              11C4  INX   H
              11C5  MOV   D,M
              11C6  PUSH  D
   PRN        11C7  POP   H
              11C8  LXI   D,0000
              11CB  PUSH  D
              11CC  PUSH  H
              11CD  PUSH  D
              11CE  CALL  0106
   NL         11D1  LXI   D,0000
              11D4  PUSH  D
              11D5  CALL  0103
```

(TABLE 3.5.3 continued...)

```
RET 3 1   11D8  LHLD 1100
          11DB  LXI  D,FFFC
          11DE  DAD  D
          11DF  SPHL
          11E0  POP  D
          11E1  POP  H
          11E2  SHLD 1100
          11E5  LXI  H,0002
          11E8  DAD  H
          11E9  LXI  B,1102
          11EC  DAD  B
          11ED  POP  B
          11EE  MOV  M,C
          11EF  INX  H
          11F0  MOV  M,B
          11F1  LXI  H,0000
          11F4  DAD  SP
          11F5  SPHL
          11F6  XCHG
          11F7  PCHL
INT 4     11F8  LXI  H,FFFC <-(GO)
          11FB  DAD  H
          11FC  DAD  SP
          11FD  SPHL
CAL 2 2   11FE  LXI  H,0002
          1201  CALL 110C
          1204  LHLD 1100
          1207  XCHG
          1208  PUSH H
          1209  LXI  H,0000
          120C  DAD  SP
          120D  SHLD 1100
          1210  PUSH D
          1211  XCHG
          1212  LXI  H,0002
          1215  DAD  H
          1216  LXI  B,1102
          1219  DAD  B
          121A  MOV  M,E
          121B  INX  H
 (R)      121C  MOV  M,D
  ^       121D  LXI  B,1227
  |       1220  PUSH B
  |       1221  POP  H
  |       1222  POP  H
  |       1223  POP  H
  +-------1224  JMP  1134
RET 2 1   1227  LHLD 1100
          122A  LXI  D,FFFC
          122D  DAD  D
          122E  SPHL
          122F  POP  D
          1230  POP  H
          1231  SHLD 1100
          1234  LXI  H,0001
          1237  DAD  H
          1238  LXI  B,1102
          123B  DAD  B
          123C  POP  B
          123D  MOV  M,C
          123E  INX  H
          123F  MOV  M,B
          1240  LXI  H,0000
          1243  DAD  SP
          1244  SPHL
          1245  XCHG
```

```
INT 4     1247  LXI  H,FFFC <-(REVERSE)
          124A  DAD  H
          124B  DAD  SP
          124C  SPHL
LDA 1 3   124D  LXI  H,0000
          1250  CALL 110C
          1253  LXI  H,FFFD
          1256  CALL 1115
          1259  PUSH H
LIT 0     125A  LXI  D,0000
          125D  PUSH D
STO       125E  CALL 1118
CAL 1 1   1261  LXI  H,0001
          1264  CALL 110C
          1267  LHLD 1100
          126A  XCHG
          126B  PUSH H
          126C  LXI  H,0000
          126F  DAD  SP
          1270  SHLD 1100
          1273  PUSH D
          1274  XCHG
          1275  LXI  H,0001
          1278  DAD  H
          1279  LXI  B,1102
          127C  DAD  B
          127D  MOV  M,E
          127E  INX  H
 (G)      127F  MOV  M,D
  ^       1280  LXI  B,128A
  |       1283  PUSH B
  |       1284  POP  H
  |       1285  POP  H
  |       1286  POP  H
  +-------1287  JMP  1131
HLT       128A  JMP  0000
```

4. WRITING A LOADER AND STANDARD RUN-TIME SUBROUTINES

It would be misleading to leave the reader with the impression that the development of a code generator using the tool described in this document is as simple as the preceding description might lead one to believe. It is certainly true that the steps in the preceding description can be used to develop a code generator capable of translating intermediate codes to machine codes. However, such object code is useless without a loader to load it into memory for execution. In addition, run-time subroutines are required, which can be called from the object program to perform input/output and arithmetic operations. Development time for this essential part of the compilation tool varies from days to weeks depending on the architecture of the target machine and the complexity of the loader/run-time subroutines.

It is always worth the effort of finding out what linkers, loaders and standard routines are already available on the target machine. The implementor will save himself considerable effort if he writes some code to get his target program into the correct format for an existing linking loader. Similarly, a search of existing system libraries might save one the task of writing standard subroutines, such as those for input/output and floating point operations.

4.1 IMPLICATIONS OF THE LOADING PROCESS

A simple **absolute loader** is easier to implement than one which links modules, relocates code, or handles overlays. An absolute loader need only be furnished with the address at which to start loading binary code, and the binary code itself. Absolute loaders are useful on single user machines, and on larger machines whose user areas are addressed via an operating system controlled base register. For example, ICL 1900 systems use a base-limit protection mechanism, with the effect that the code (which an individual user sees as absolute) is really relative to a base register. However, absolute loaders are of limited use on machines capable of supporting a relocatable user slot type operating system. What is needed in this case is a loader which can load a program into an area of memory which is dictated by the operating system rather than by the program being loaded. This is a **relocatable loader.** Relocatable code contains address references which depend upon a particular "relativizer". In choosing a particular type of loader, the code generator implementor will need to be familiar with the requirements of his target machine.

The provision of inter-module communication facilities causes additional work for the loader. If a module is to have more than one entry point, it is usual to precede it with a jump table. It might in fact prove more convenient for the implementor to write a separate linking program which combines several modules into a suitable form for a simpler loader, rather than to use a **linking loader** to perform the entire task at once.

4.2 RUN-TIME SUPPORT PACKAGES

Implementing run-time subroutines on a processor as simple as the 8Ø8Ø, whose greatest arithmetic boasts are its 8-bit integer add, subtract and shift instructions, is a considerable task. However, the same routines on a CDC mainframe, with its hard-wired floating-point routines and built-in multiply and divide instructions, become trivial by comparison. On systems which allow access to library facilities from an assembler level, it is desirable to store standard run-time procedures in a library, which is selectively searched so that only those which are referenced by the application program are included in the final task.

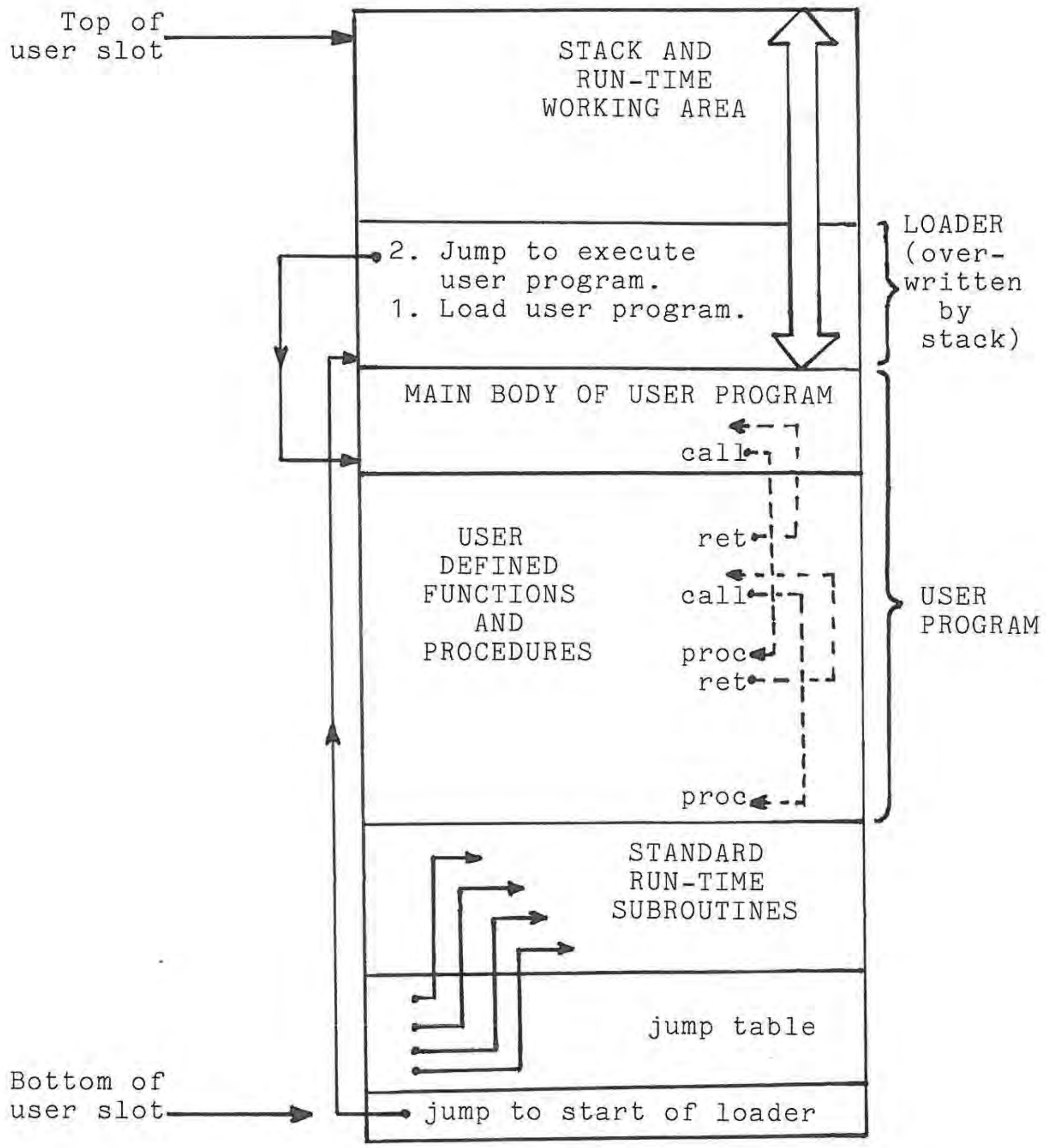
The implementor is responsible for writing both the run-time subroutines and their calling sequences from within code-skeletons, so details of data representation and parameter passing are left up to his personal design preference. Since the entire system is based on a hypothetical stack machine, however, data storage and parameter passing is most easily achieved using a stack mechanism. It is possible that not all standard run-time subroutines will be needed for some implementations (e.g. a simple compiler might not include real arithmetic). Nevertheless, dummy addresses should be provided for these unused standard routines in the code generator parameter file to keep the sequence of run-time addresses correct.

4.3 A SAMPLE IMPLEMENTATION

Figure 4.3.1 shows the basic load time organization of a loader and its associated run-time subroutines for a target implementation of a hypothetical stack machine. This diagram shows up two points worth noting. Firstly, once the loader has loaded and passed control to the user program, it is of no further use and can be overwritten during the running of the program. Secondly, the standard run-time subroutines are always referenced by means of a jump table. This allows a run-time subroutine to be moved about in memory without having to alter its referencing address in the user program.

FIGURE 4.3.1

Load time memory organization


Top of user slot
STACK AND RUN-TIME WORKING AREA
LOADER (over-written by stack)
2. Jump to execute user program.
1. Load user program.
MAIN BODY OF USER PROGRAM
call
ret
call
proc
ret
proc
USER DEFINED FUNCTIONS AND PROCEDURES
USER PROGRAM
STANDARD RUN-TIME SUBROUTINES
jump table
Bottom of user slot
jump to start of loader

Appendix D lists the loader and monitor routines necessary for running a program on the 8080 processor with Pascal features, such as might have been produced by the CLANG system of chapter 3 or the enhanced Pascal-s [39] system. The loader is based on the design of Figure 4.3.1 and includes backpatching so that the target code file can be read using the format of table 9.3.2. The run-time subroutines include all those standard functions listed in table 8.2.1. Implementation details of these routines will not be discussed as the source listing of Appendix D has been perspicuously commented.

In addition to the 8080 implementation of Appendix D, the author has written similar run-time packages for the ICL 1900 and PDP-11 series computers. These routines are not based on any particular algorithms, and no doubt better have been written. However, they have been well tested and have shown themselves to be reliable. A glance through Appendix D should convince the reader that the writing of a run-time package for a meaningful language is not a trivial task. The 8080 loader and run-time package is the result of several weeks of programming and testing. The ICL 1900 and PDP-11 implementations required less effort because of the additional programming features available on those computers. In particular, the ICL 1900 GEORGE system allows access to a useful library of trigonometric and type conversion routines.

5. THE MARK 3 SYSTEM

During the implementation of the mark 2 system (the programs listed in the Appendices), several improvements made themselves apparent. The mark 2 system with these added improvements constitutes the mark 3 system. The simpler system has been retained for the purposes of this document since, as a working system, it demonstrates all the necessary principles without excess frills. The paragraphs below outline some of the advantages of the mark 3 enhancements, and how they were implemented. All the facilities discussed have been realized, unless otherwise stated, but space does not allow for a detailed discussion. Interested readers can obtain listings of mark 3 from the author.

5.1 MEMORY ALLOCATION AT CODE GENERATION TIME

In the earliest version of the code generator (mark 1), all intermediate codes, all generated target codes and all code-skeleton definitions were held in memory. This arrangement was very convenient for three reasons:

1. With no disc transfers, the code generator was extremely fast,
2. with all intermediate codes in memory, address association (intermediate address to target equivalent) was trivial, and
3. with all the target code in memory, all forward

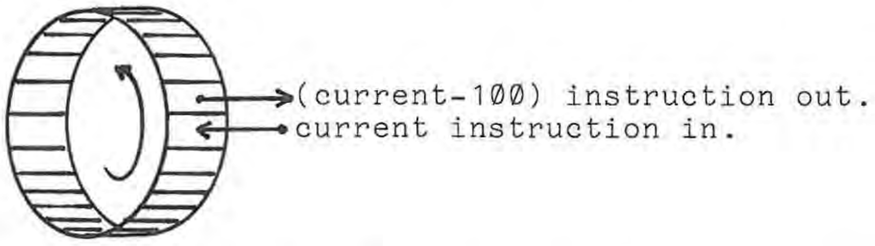
references could be resolved in a single pass at code-generation time.

However, this arrangement was most unsatisfactory, because each table was limited in its memory size. It was only possible to compile toy programs which generated a small number and variety of intermediate codes.

The mark 2 system overcomes the program size and intermediate code range restriction by storing the code-skeletons on external disc (at the expense of a slower code-generator due to file searches) and by writing out target codes as they are generated (at the expense of having to perform backpatching at load time). The advantage of point 2 above is retained and with the intermediate code array extended to 5ØØØ entries, Pascal programs in excess of 1ØØØ source lines are able to be processed. The slower code generation time is not regarded as a major disadvantage as the system was never intended for use in compilation intensive environments (e.g. student practicals).

The mark 3 system attempts to improve on the memory allocation of the current system in two ways. So as to decrease the number of disc accesses, up to ten code-skeletons may be held in memory at a time. A usage count associated with each indicates which one is used least often. The most seldom used skeleton is overwritten when a new skeleton needs to be read into memory. Naturally, this strategy only succeeds if not all intermediate codes are produced with equal frequency. A facility exists which enables the implementor to override this strategy by selecting certain code-skeletons to be permanently memory-resident.

The second improvement involves the output of target codes. Instead of writing target codes to disc as soon as they are generated, they are stored in a circular list which is able to hold 1Ø1 target instructions. An instruction is only written out once 1ØØ subsequent instructions have been generated, or at the end of the program segment.



This feature enables forward references to be resolved as long as they do not span more than 1ØØ instructions. More importantly, however, some measure of peephole optimization may be implemented (see section 5.3). If instruction sequences are found which can be deleted or replaced by cheaper sequences, the surrounding code can be relocated.

5.2 IMPROVED CODE-SKELETON DEFINITIONS

An obvious improvement to the assembly format of instructions would be the inclusion of a comment field. This enhancement was effected, but soon rejected, since the size of the data structure was more than doubled for each code-skeleton if one allowed for a fair sized comment on each instruction.

A second code-skeleton definition improvement did prove to be beneficial, however. This enhancement allows argument fields to be specified using expressions involving compile time constants. The mark 2 system allows very limited expression savings by allowing an argument field to be expressed in several ways, e.g. X, XN and XA denote the value of argument X, the negative of the argument X value, and the target address corresponding to argument X (more details can be found in section 8.2). The mark 3 system allows both constants and compile time names to be combined, using the simple arithmetic operators +, -, and *. The benefit to be gained from this enhancement is best demonstrated by means of the following example. Consider the run-time subroutine GETDIS from section 3.2:

```
GETDIS:   DAD  H          ; X * 2 for 16-bit addressing
          LXI  B,DISPLA
          DAD  B          ; + DISPLA
          MOV  E,M        ; get value from
          INX  H          ; DISPLA  + (2*X)
          MOV  D,M
```

In this example, constant values known at compile-time are used to calculate the address only at run-time. On the mark 3 system using expression evaluation, the whole of GETDIS can be replaced by a single instruction:

```
LHLD  2*X+DISPLA    ; value into H-L
```

5.3 OPTIMIZATION

If the reader has studied the target code of table 3.5.3, he will probably have noticed that due to the mechanical nature of the code generator, some rather clumsy code has crept into the object program. In particular, a lot of redundant storing and reloading of variables occurs across the boundaries of 8080 code bodies for consecutive intermediate codes.

```
e.g. PUSH H    ; last instruction for intermediate code n
     POP  H    ; first instruction for intermediate code n+1
```

As long as the code body for intermediate code n+1 is not the destination of a branch instruction, such code can be deleted (as in the above example) or improved:

```
     PUSH D    or   PUSH H    ; replace with XCHG
     POP  H         POP  D
```

Note that such improvements will normally only be made across the boundaries of 8Ø8Ø code bodies, as a construct such as

```
POP  D
PUSH D    ; still need the value of D on the stack
```

within a code body is usually intentional. This is an obvious application of peephole optimization.

Another obvious peephole optimization is the removal of multiple jump statements. This will become obvious if one studies the intermediate code program of table 3.5.2. All JMP PC+1 type instructions can be removed (e.g. the BRN at intermediate code address 2) as these serve no useful purpose (but simply makes the CLANG intermediate code generator far simpler). Jumps to other jump statements can be altered to jump to the final destination (e.g. the call at intermediate code address 25 can be altered to branch directly to the code body at address 18).

A general specification for optimization techniques is not easy to formulate. These optimizations obviously become machine dependent. In addition, it is debatable whether such facilities should be included in a system intended for use by a non compiler specialist. The mark 3 system reaches a compromise by allowing the compiler writer to specify simple optimizations of the type mentioned above, or to switch this option off.

The mark 3 notation for specifying peephole optimization is very poor, and this enhancement, if it is to be retained, requires much refinement. Since it has been said that this feature's strongest point is its ability to be switched off and ignored, its scope will be described only briefly.

If the implementor supplies the function number and address argument of each intermediate code type which performs a branch, the mark 3 code generator makes a pass over the intermediate code program building up a table of addresses which are the destinations of branches. Since the entire intermediate code program is held in memory, the program is able to alter the addresses of branch instructions which are redirected by another branch instruction. At the same time, JMP PC+1 intermediate codes can be replaced by a dummy intermediate code which produces a null code-skeleton, effectively eliminating it without having to relocate the rest of the intermediate code program.

In addition, the user can specify three groups of target code sequences c_1, c_2 and c_3 meaning: "if the current code-skeleton begins with the sequence c_1, and if the previous code-skeleton ended with c_2, then replace both c_1 and c_2 with c_3 in the final target code". This replacement is suppressed if the current code-skeleton is the destination of a branch instruction.

Table 5.3.1 compares the code generated by the mark 3 code generator, with and without the simple optimization feature, using the following simple CLANG program:

```
program TEST;
{$O+}

var I;

begin
  I := 1;
  WRITE(I);
end.
```

whose intermediate code program is:

```
ADDRESS  MNEMONIC  FUNCTION LEVEL ADDRESS
   Ø       BRN         7      Ø      1
   1       INT         5      Ø      4
   2       LDA         1      1      3
   3       LIT         Ø      Ø      1
   4       STO        21
   5       LDA         1      1      3
   6       LDX        27
   7       PRN        24
   8       NL         26
   9       HLT        22
```

The code skeletons for this example differ slightly from those of chapter 3 because the expression evaluation enhancement of the mark 3 system enables one to detail some of the code-skeletons more concisely.

TABLE 5.3.1

Code generated by the mark 3 code generator

	without optimization			with optimization		
BRN Ø 1	0A1D	JMP	0A20	0A1D	LXI	D,FFF8
INT Ø 4	0A20	LXI	D,FFF8	0A20	LXI	H,0000
	0A23	LXI	H,0000	0A23	DAD	SP
	0A26	DAD	SP	0A24	DAD	D
	0A27	DAD	D	0A25	SPHL	
	0A28	SPHL		0A26	LHLD	0A05
LDA 1 3	0A29	LHLD	0A05	0A29	LXI	D,FFFA
	0A2C	LXI	D,FFFA	0A2C	DAD	D
	0A2F	DAD	D	0A2D	PUSH	H
	0A30	PUSH	H	0A2E	LXI	D,0001
LIT Ø 1	0A31	LXI	D,0001	0A31	POP	H
	0A34	PUSH	D	0A32	MOV	M,E
STO	0A35	POP	D	0A33	INX	H
	0A36	POP	H	0A34	MOV	M,D
	0A37	MOV	M,E	0A35	LHLD	0A05
	0A38	INX	H	0A38	LXI	D,FFFA
	0A39	MOV	M,D	0A3B	DAD	D
LDA 1 3	0A3A	LHLD	0A05	0A3C	MOV	E,M
	0A3D	LXI	D,FFFA	0A3D	INX	H
	0A40	DAD	D	0A3E	MOV	D,M
	0A41	PUSH	H	0A3F	XCHG	
LDX	0A42	POP	H	0A40	LXI	D,0000
	0A43	MOV	E,M	0A43	PUSH	D
	0A44	INX	H	0A44	PUSH	H
	0A45	MOV	D,M	0A45	PUSH	D
	0A46	PUSH	D	0A46	CALL	0106
PRN	0A47	POP	H	0A49	LXI	D,0000
	0A48	LXI	D,0000	0A4C	PUSH	D
	0A4B	PUSH	D	0A4D	CALL	0103
	0A4C	PUSH	H	0A50	JMP	0000
	0A4D	PUSH	D			
	0A4E	CALL	0106			
NL	0A51	LXI	D,0000			
	0A54	PUSH	D			
	0A55	CALL	0103			
HLT	0A58	JMP	0000			

5.4 COMPILING SEPARATE MODULES

Although the mark 2 code generator is able to handle a reasonably large source program, it is always useful not to have to recompile an entire program when only a small section has been altered. A simple system for compiling separate modules has been experimented with. This presently puts the onus on the end-user to supply the run-time base address of the module. The code generator will supply the end address of the module for use in estimating the start address of the next module. Transfer of control between modules is via names whose final addresses are specified at code-generation time. An improvement on this approach is for the implementor to allow the end user to compile his modules as normal. It is possible, by carefully selecting the range of target instructions used, as well as the approach to addressing (e.g. all addresses relative to a base value), to generate code which can be relocated to any address in memory [29]. The implementor must then write a linking loader (chapter 4) to combine several relocatable modules. Once again, this feature is within the realm of the code-generation specialist, and should probably be avoided in systems to be implemented by a novice.

An obvious extension to generating separate modules is to overlay the modules which are generated. Again, a crude implementation simply generates two modules of absolute code beginning at the same address. However, the writer of the run-time routines must now write an extra code sequence which can be called upon to load an overlay as required. This system is not too difficult to implement as long as the overlaid code does not alter itself in any way (necessitating the writing out of overlays as well). The transfer of data via variables does not constitute a problem, since a hypothetical stack machine stores all its data in stack frames, separate from the overlaid code.

The current mark 3 system is able to handle a crude form of separate module compilation, but overlays have not yet been successfully implemented.

6. HISTORICAL REMARKS AND CONCLUSIONS

The original efforts of implementing a generalized code generator grew out of a project to implement a code generator for an experimental Pascal/Basic teaching language, BPL [4Ø], in 1979 on the ICL 19Ø4. The author has subsequently implemented a Pascal code generator for the PDP-11 series of computers. Both projects have supplied valuable contibutions towards this thesis, particularly in the form of reliable run-time packages, leading eventually to the current generalized code generator. An implementation of the system on the North Star Advantage Microcomputer was outlined in a paper given at the RU Symposium on Computer Science, Theory and Practice, in November 1983, and subsequently described more fully in this document.

Implementations on an 8Ø8Ø processor using the CLANG parser of appendix E and a bigger, more realistic Pascal-s+ parser have been well tested. Pascal-s+ [39] is an enhancement by P.D. Terry of Wirth's original Pascal-s [38] system. It has been adapted by the author for use with a synthesised code generator to provide a compiler which falls short of standard Pascal only in its lack of the "with" statement. Both implementations show the system to be fairly successful within the limitations imposed by its original objectives.

A number of bench-marks were carried out to compare the execution of code produced by the generalized CLANG implementation with code produced by a special purpose optimizing code generator (Pascal MT+ [35] was used), and with intermediate codes interpreted by the CLANG interpreter. Only execution time was considered (the MT+ compiler produces huge code files, making the compar⁄ison of target code size rather meaningless). As expected, code produced by the synthesised code generator is slower than that produced by a special purpose optimizing code generator, but still executes several times faster than an interpreter written to produce the same effect. The greatest difference in execution time is noticed in programs which perform many calculations and little input/output. Compilation times for the synthesised and special purpose systems are comparable, but are much shorter on the interpreted system.

There can be little doubt that the time taken to develop a code generator using this tool is less than the time required to write a special purpose code generator. However, the resultant code generator is more useful for programs which will be run more times than they will be compiled than vice versa. The expedience of the system lies chiefly in its ability to replace multi-level interpretation with direct execution, and in its use as a step in bootstrapping a compiler onto a new computer.

PART 2

A USER MANUAL

7. USING THE MACHINE DEFINITION PROGRAM

The current program has been implemented in Pascal MT+ on a CP/M based system (North Star Advantage), but could be easily altered to run on other Pascal systems, with only minor modifications needed to the machine definition program and to the following description of how to use it.

7.1 CURRENT IMPLEMENTATION REQUIREMENTS

The following seven files are required to run the machine definition program on a CP/M system.

```
MACDEF.COM      -    Executable Root Program
MACDEF.ØØ1      }
MACDEF.ØØ2      }
MACDEF.ØØ3      }
MACDEF.ØØ4      }    Overlays for the program
MACDEF.ØØ5      }
MACDEF.ØØ6      }
```

Also useful, but not mandatory, are the following two message files, which improve the friendliness of the program.

```
MACDEF.HLP      -    Help message file
MACDEF.MES      -    Error message file
```

To run the program, type

```
MACDEF <RET>
```

and answer the questions or make selections from menus which follow.

The machine definition table will be saved on a user named file whose default extension is ".MAC".

The disc space required to store the root program and overlays is 48k, with an additional 14k required if the message files are included. The ".MAC" file typically requires 4k to 6k of disc storage.

7.2 DRIVING THE PROGRAM

Provided the help message file (MACDEF.HLP) is present, the user may request help at any time by responding to an invitation for input by typing

```
? <RET>
```

Help relevant to the current operation will be supplied. In some cases, additional help or an example will be offered. If the character "?" is a valid part of a mnemonic, separator or argument field in the assembly language for the machine being defined, then the user should choose another character as his help request character. This can be done when invited to do so at the start of the program.

The sequence <ESC> <RET> can be used to abandon any operation. Execution returns to the main menu or to the appropriate sub-menu. Once an operation has been abandoned, the machine definition table is returned to its state prior to commencement of the operation.

The main menu is displayed automatically. The user selects one of seven options by typing a representative character.

```
W - Word Definition Mode
I - Instruction Definition Mode
D - Edit Word Definitions
E - Edit Instruction Definitions
R - Read Definitions from a File
L - List Definitions
X - Exit
```

7.2.1 Word Definition Mode (Option W on main menu)

Word formats are used to group together a set of instructions which have the same length and field format. To describe a new word format, three items of information are required.

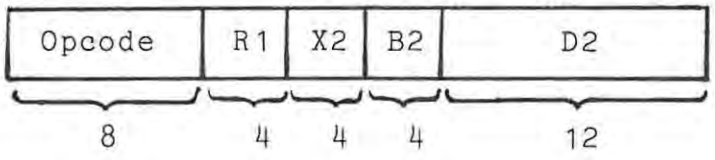
The **first** item of information is a textual description of the word format for easy identification by the user at a later time. This description may be up to 2Ø characters in length, or may be left blank by just pressing the <RET> key when the description is requested.

The **second** value which the program requests is the length of the word in bits. This value, used for validation during field definition, should include all fields, including expanded operands.

Finally, the user is requested to type the **field definitions** in the notation described in Section 2.1. Each field in the word form is represented by a unique alphabetic letter, which is repeated as many times as there are bits in the field. It should be noted that only upper case alphabetic letters may be used. Seperating spaces may appear both between and within fields, and are ignored.

For example, the three items of information required to define an IBM 37Ø Series "RX" wordform would be:

Format of the RX word form



where R1 is a register,
X2 is a register used as an index,
B2 is a base register,
D2 is a positive displacement (Ø..4Ø95)

DESCRIPTION: Register-Index Form

LENGTH: 32 bits

FIELD DEF: OOOO OOOO RRRR XXXX BBBB DDDDDD DDDDDD

The word definition routines are called in a loop. To define several word definitions, answer "Y" (yes) to the question "Another word definition ?".

Since instruction definitions refer to defined word forms, word definition mode is usually entered before instructions are defined.

7.2.2 <u>Instruction Definition Mode</u> (Option I on main menu)

All instructions conform to a word format which has already been defined. For example, suppose the instruction

```
          LDA  R1,address
```

conforms to the word format

```
          FFFFFF  RR  AAAAAAAA AAAAAAAA
```

This instruction will be used as a continuing example.

To define an instruction, one has to specify the function and value of all the fields of the word format.

The program will prompt the user to type the index number of the word format description to which the instruction conforms. (As each word format is defined, in word definition mode, it is recorded in the definition table. For convenience, a list of all defined word formats and their index values may be viewed by responding with the character "*").

Mnemonic Fields

A textual mnemonic for the instruction must be supplied next. A mnemonic is a short word, representative of what the instruction does, e.g. LDC for an instruction which loads a constant into an accumulator. Later, one is able to refer to the instruction by its mnemonic. The mnemonic may be made up of any printable characters (no control characters are allowed), but may not begin with the help request character. The current implementation allows a maximum of six significant characters. In addition, each instruction mnemonic must be unique. It will be advisable to follow the mnemonics of one's target machine's assembly language as closely as possible.

For example, an obvious mnemonic for the instruction

```
LDA R1,address
```

is "LDA".

The program now displays the word format for the instruction, and requests that the user specify which field will represent a binary pattern unique to this particular instruction. This is essentially the field which will be recognised as representing the mnemonic.

A field is selected by typing the character marking the field in the word format description. Naturally, only defined field names may be selected.

The binary bit pattern for the chosen field may now be entered. If the user prefers, the field value may be prefixed:

(1) with the character "D", and entered as a decimal number (e.g. D35),

(2) with the character "H", and entered as an hexadecimal number (e.g. H23)

or (3) with the character "O", and entered as an octal number (e.g. O43).

A check is made to ensure that the value will fit into the chosen field. Negative values are converted according to 2's complement rules, and must be input using a minus sign.

To continue with the example

```
          LDA R1,address
     FFFFFF RR AAAAAAAAAAAAAAAA

  FIELD TO BE ASSOCIATED WITH MNEMONIC: F
  BIT PATTERN FOR FIELD F: 10011
                      or   O23
                      or   D19
                      or   H13
```

More than one field may be associated with the mnemonic, by stipulating further fields in the same manner as the first (reply "Y" to the question "Should another field be associated with the mnemonic ?"). As a further prompt, after defining each mnemonic field the program displays the word format for the instruction, with binary patterns filled in for mnemonic fields. The definition for field F in the above example would yield:

```
010011 RR AAAAAAAAAAAAAAAA
```

Argument Fields

Any field not designated as being associated with the instruction mnemonic is assumed to be an argument (variable) field. Argument fields accompany the mnemonic in the assembly language format and are assigned a value at a later time (e.g. fields R and A in our example). The program steps through each argument field in turn, prompting the user to supply validation information for this field.

Argument Ranges

Firstly, the user is required to define the legitimate range of arguments which the named field may hold. These will be checked against the actual arguments used at the code generator definition stage. A field generally takes an integer value as an argument, but can also be specified by a symbol representing an integer. To choose a valid range of arguments for the field, a menu is displayed, from which the user should select one of five options. Selections 1 to 4 (type the appropriate digit to make the selection) define various integer ranges:

1. A positive integer in the range Ø .. biggest integer which can fill the field. If the field is bigger than the maximum integer length for the implementation, selection 4 is chosen instead.

2. A negative or positive integer in the 2's complement range for the field. Again, if the field is bigger than the maximum integer length for the implementation, selection 4 is chosen instead.

3. This selection requires the user to specify the lower and upper bounds for the field. The numbers should be typed in decimal (the default number base for this input, e.g. 26), octal (preceded by O, e.g. O32), binary (preceded by B, e.g. B11Ø1Ø), or hexadecimal (preceded by H, e.g. H1A). A check is made to ensure that the values will fit into the

field. Negative values are converted according to 2's complement rules, and must be typed using a minus sign.

For example, if a 5-bit field were to take an integer in the range Ø to 8, the definition might be

```
LOWER BOUND: Ø
UPPER BOUND: 8
```

whereas if it were to take on the range -8 to +7, the definition might be

```
LOWER BOUND: B-1ØØØ
UPPER BOUND: B111
```

(Note that lower = B11ØØØ upper = BØ1ØØØ will not be acceptable).

Selection 4 is preferred if the field exceeds the maximum integer length for the implementation.

4. This option is used if no validation is to be performed at the code generator specification stage. This selection must be made if the field is larger than the maximum integer length for the computer upon which the machine definition program is implemented.

Option 5 is selected if the user wishes to represent integer values by symbolic names.

We continue with our sample definition for instruction

```
              LDA R1,address
          Ø1ØØ11 RR AAAAAAAAAAAAAAAA

     VALID ARGUMENTS FOR FIELD R: 5  (I want to  use
                                      symbolic names)
     VALID ARGUMENTS FOR FIELD A: 1  (integer range
                                      Ø to 65535)
```

Symbolic Names for Arguments

If option 5 is chosen, as it was for field R in our example, the user is required to define a set of symbolic names and their corresponding values for the field. However, before the user is allowed to do this, the program asks him whether an identical field already exists in a previously defined instruction. Since the same symbolically named field often appears in several instructions (for example an argument which takes on register names), both time and memory/disc space can be saved if all such instructions refer to the same set of symbolic names instead of each having its own identical set.

If the user chooses to refer to a symbolic name list set up during a previous definition, the program will prompt him to supply the index number of an instruction which has an identical field to the one being defined, i.e. same identifying letter and same length. As was the case for word format specifications, each instruction is recorded in a definition table as it is determined. For convenience, a list of all instructions defined so far and their index values may be viewed by responding with the character "*".

On selecting a previous instruction, the user will be shown the instruction and its matching field (if one exists) so that he can verify that it in fact has the argument range which he desires.

When specifying a new set of symbolic names for an argument field, one is required to supply the number, N, of valid symbols which the argument is to allow. The program will then prompt the user N times for a symbolic name. The rules for making up symbolic names for arguments are the same as those for instruction mnemonics. One may use the same names as for the existing mnemonics, but all symbols for a particular argument field must be unique. Each symbolic name is given a corresponding integer which it represents. The default base in which these values are typed is decimal but, as usual, the value may be typed in binary, octal or hexadecimal by preceding the number with one of the characters "B", "O" or "H".

To define symbols for our 2- bit argument RR, one might type:

```
     NO. OF VALID SYMBOLS: 4

          SYMBOL:  ACC         CORRESPONDING INTEGER: Ø
          SYMBOL:  RØ          CORRESPONDING INTEGER: 1
          SYMBOL:  R1          CORRESPONDING INTEGER: 2
          SYMBOL:  PC          CORRESPONDING INTEGER: 3
     4 SYMBOLS DEFINED.
```

When one uses the instruction, one can now specify argument R, using its symbols in place of integer codes.

Separators

Two more items of information are required before all necessary information has been gathered for an argument field. Firstly, the user must indicate which character will precede the argument as a separator in the assembly language format of the instruction. When the instruction is used, the mnemonic will define the fields which have fixed values. The variable fields are described by arguments separated from the mnemonic and from each other by spaces or other specified separators. Spaces may be used as separators without an explicit declaration.

Argument Order

Secondly, the user must indicate what the numeric order of the argument will be in the assembly language format. The arguments after the instruction mnemonic must appear in a specific order so that correct validation is possible. Although the argument fields are defined in the order in which they appear in the word format definition, this is not necessarily their assembly language order. They must eventually occupy consecutive positions, however, (e.g. three arguments must occupy positions 1, 2 and 3 not positions 1, 3, and 5). In response to the prompt for an argument position, the user may type "*" to list the instruction definition as it stands (probably incomplete).

The assembly format of our sample instruction

```
LDA  R,A
```

demands the following positional and preceding separator definitions:

```
ARGUMENT POSITION FOR FIELD R: 1
ARGUMENT POSITION FOR FIELD A: 2

SEPERATOR TO PRECEDE FIELD R: " "
SEPERATOR TO PRECEDE FIELD A: ","
```

Related Formats

To save the user having to select a word format for each instruction, he is able to define several instructions having the same format by replying "Y" (yes) to the question "Another instruction definition with the same word format ?". In fact, an affirmative reply to the subsequent question "Any other instruction definition ?" will allow the user to remain in the instruction definition mode and define another instruction with a different format.

7.2.3 Editing Word Definitions (Option D on main menu)

It should be borne in mind that each instruction definition includes a reference to the word format description to which it conforms. For this reason, any alteration of a word format descriptor implies that the definitions of instructions which refer to the word format will also be altered. A word format description may be deleted (in which case all instructions which refer to it are also deleted), or altered. Alterations only allow changing the textual description of the word or the width of a field. The names of fields and the number of fields in the word cannot be altered since this would have too wide an effect on the referencing instructions. To perform such operations, delete the word format and redefine it and all instructions referring to it.

The machine definition program only allows the user to enter word edit mode if at least one word definition already exists. Having selected this mode from the main menu, the user is required to provide the index number of the word format description which he would like to edit. The index numbers of all defined word formats can be listed by responding with the character "*".

The program will display the chosen word format for the user, to verify that it is in fact the information he wishes to alter.

To choose the type of modification he wishes to make, the user must select an option from the sub-menu which is displayed. One of the following four responses is required:

D. - Delete the word definition.

If any instructions refer to the word format to be deleted, these will be displayed on the screen with a warning that they too will be deleted. The user is given the opportunity to abort the deletion operation. Any word formats and instructions which are deleted will be removed from the definition tables, and any other entries with higher index numbers than the deleted entries will be moved down so that no gaps are left in the tables. **Once word format and instruction definitions have been deleted, they CANNOT be retrieved.**

A. - Alter the word description.

This selection requires to user to type a new word format description which will overwrite the current description.

F. - Alter the width (number of bits) of a field.

To effect this modification, the user is required to supply the name (alphabetic character) of the field. The program will respond with the current width (in bits) of the field and will request the new width to be typed. To leave the field unchanged, the current width may be typed, or the edit may be abandoned. The field width must be in the range 1..LARGEFIELD (LARGEFIELD = 32 on this implementation). Before the new field width is recorded, all instructions which refer to the word format being altered as well as the new word format itself are listed on the screen in their modified state. The user may decide to revert to the original field width if he wishes, or confirm that the alteration should be permanently recorded. During the modification process, the ranges of integer arguments in the instruction definitions which refer to the modified word format, will automatically be altered so as best to fit the new field width. If the new field width exceeds the maximum integer length for the implementation, the field will become an unrestricted argument. Conversely, if a currently unrestricted argument field is narrowed in width to the extent that it no longer exceeds the maximum integer length, it will be set up as a positive integer range. If the modified field is associated with the mnemonic in an instruction, then if it is reduced in width, the most significant bits will be pruned. Alternatively, if its width is increased, leading zeros are added.

X. - Exit word edit mode

This selection exits word edit mode and returns to the main menu. The word format being modified will retain the definition it had just prior to leaving the edit mode.

After each edit operation, the current state of the word format undergoing modification will be displayed.

7.2.4 Editing Instruction Definitions (Option E on main menu)

An instruction description may be deleted, or may have its mnemonic or the values of its mnemonic fields altered, or the legal range for its variable fields altered.

It is not possible to alter

(a) a variable field to a mnemonic field or vice versa,

(b) the size of a field (this involves altering the word format),

(c) the order in which variable arguments are used, or

(d) symbolic names for a field (this might affect other instructions whose fields refer to the same list of symbolic names).

To make any of the above changes, one has to delete the instruction and redefine it.

The machine definition program only allows the user to enter instruction edit mode if at least one instruction has already been defined. Having selected this mode in the main menu, the user is required to provide the index number of the instruction definition which he would like to edit. The index numbers of all instructions can be listed by responding with the character "*".

The program will display the chosen instruction for the user, to verify that it is in fact the information he wishes to alter.

To choose the type of modification he wishes to make, the user must select an option from the sub-menu which is displayed. One of the following five responses is required:

D. - Delete the instruction definition.

After making sure that the deletion request was intentional, the program removes the instruction from the instruction definition table. Any entries with higher index numbers than the deleted instruction will be moved down so that no gaps are left in the table. **Once an instruction definition has been deleted, it CANNOT be retrieved.**

A. - Alter the mnemonic.

This selection requires to user to type a new mnemonic for the instruction. All the rules for mnemonics in section 7.2.2 apply to the editor as well.

M. - Alter the value of a mnemonic field.

If this option is selected, the program will step through all the fields which are associated with the instruction mnemonic, giving the user an opportunity to alter the bit pattern if he wishes to. The bit pattern is expected to be typed in binary, but, as usual, may be typed in one of the other common bases preceded by the appropriate base character.

V. - Alter the legal range of an argument field.

This selection steps through all the argument fields for the instruction, allowing the user to alter the argument range. Only two range options are available. If the width of the field exceeds the maximum integer width for the implementation, an unrestricted field is forced upon the user. Otherwise, the user is required to enter the lower and upper bounds for an integer range. The default base for these values is decimal but other bases may be used if the values are preceded by the appropriate base character.

X. - Exit instruction edit mode

This selection exits instruction edit mode and returns to the main menu. The instruction being modified will retain the definition it had just prior to leaving the edit mode.

After each edit operation, the current state of the instruction description undergoing modification will be displayed.

7.2.5 Reading Definitions From a File (Option R on main menu)

The machine definition tables are usually written to a disc file at the end of the machine definition program. During a subsequent run, the user might want to read back these definitions to modify them in some way. When doing so, it must be borne in mind that any definitions currently in the tables will be overwritten by those read in from the disc file.

When this option is selected in the main menu, the user is required to confirm his selection (in case the request was accidentally evoked) and to supply the name of the file from which the tables are to be read (On the current implementation, such files are given the default extension ".MAC").

7.2.6 Listing Definitions To the Screen (Option L on menu)

If the user requests a definition listing from the main menu, a sub-menu is displayed, from which he must select one of the following three options:

1. List all word format definitions on the screen.
2. List instruction definitions in full detail on the screen. This selection requires the user to type the index number of a word format definition. All instructions which conform to the named word format will be listed. If a zero (Ø) is typed in response to this request, all instructions will be listed.

3. All word format definitions and all instruction descriptions will be listed if option 3 is selected.

7.2.7 <u>Exiting the Program</u> (Option X on main menu)

On exiting the program from the main menu, the user is prompted to save the definition tables if he has made any new definitions. The program is rather persistent with this particular prompt should the user decide not to save his definitions, since this could be a regrettable decision.

A user wishing to save his definitions, must provide the name of the file which will be created for this purpose. (On the current implementation, such files are given the default extension ".MAC" if no extension is supplied by the user. The file name may optionally be preceded by a device code, e.g. A: or B:). If the file already exists, the program will request the user's permission to overwrite it.

The message "Cannot create file - (filename)" generally indicates that there is insufficient free space on the disc.

7.3 ERROR CONDITIONS

Below is listed the set of error messages which a user of the Machine Definition program is likely to encounter. "ACTION" describes the steps the user should take to overcome the error condition.

1 FIELD DESCRIPTORS MUST BE ALPHABETIC "A" -> "Z"
Action: Retype the field description using only alphabetic capital letters and separating spaces.

2 REPETITION OF A FIELD DESCRIPTOR
Action: The character representing the current field has been used to represent a previous field. Retype the field description using unique characters.

3 WORD DEFINITION TABLE FULL
Action: There is no more space in the word definition table. Save your current definitions and exit the program. You will have to recompile the program (see Appendix A1.6), increasing the maximum number of word definitions (WORDMAX in the const declaration of all modules).

4 L1 # L2 FIELD DEFINITION DOES NOT MATCH WORD LENGTH
Action: L1 is sum of the number of bits in all fields. L2 is the defined length of the word form. Re-define the word length and field information making them agree in length.

5 INTEGER VALUE REQUIRED
Action: A non-numeric character has been typed in an integer input field. Re-type the correct integer value.

6 INSTRUCTION DEFINITION TABLE FULL
Action: There is no more space in the instruction definition table. Save your current definitions and exit the program. You will have to recompile the program (see Appendix A1.6), increasing the maximum number of instruction definitions (INSETMAX in the const declaration of all modules).

7 value > length BITS - VALUE TOO LARGE FOR FIELD
Action: "value" is too large to be stored in a field of "length" bits. Retype the value.

8 NUMERIC VALUE REQUIRED
Action: Type a numeric value in response to the request for input.

9 c FIELD DOES NOT EXIST
Action: c is the character typed. No field exists with this name. Retype the field name.

10 MNEMONIC ALREADY DEFINES ANOTHER INSTRUCTION
Action: Select a different mnemonic name and retype.

11 INVALID NON-PRINTABLE CHARACTER IN MNEMONIC
Action: Retype the mnemonic name using only printable characters.

12 TOO MANY FIELDS ASSOCIATED WITH MNEMONIC
Action: Combine adjacent fields into one field if possible, or save your current definitions and exit the program. You will have to recompile the program (see Appendix A1.6), increasing the maximum number of fields which can be associated with the mnemonic (MNEMFMAX in the const declaration of all modules).

13 c FIELD ALREADY DEFINED
Action: c is the character typed. This field can only have its value altered by the instruction edit mode. Select another field name or abandon the definition.

14 BASE n ILLEGAL DIGIT FOR GIVEN BASE
Action: Retype the numeric input using digits for base n.

15 n NO WORD DEFINITION WITH THIS INDEX
Action: n is an invalid word format index. Retype the word format index, or type * to list all defined word formats.

16 BAD SELECTION
Action: Select a valid option - retype your selection.

17 FIELD EXCEEDS MAXINT - UNRESTRICTED FIELD ASSUMED
Action: This is a warning message. No user action is required. The program effects its own correction.

18 FIELD SMALL ENOUGH TO ASSUME SELECTION 1
Action: This is a warning message. No user action is required. The program effects its own correction.

19 n INVALID INSTRUCTION INDEX
Action: n is an invalid instruction index. Retype the instruction index, or type * to list all instructions.

20 lower >= upper UPPER BOUND DOES NOT EXCEED LOWER BOUND
Action: Retype the bounds, making sure that the upper bound exceeds the lower bound.

21 otherwidth <> thiswidth WIDTH DOES NOT MATCH WIDTH OF CURRENT FIELD
Action: Specify an alternative instruction index or define the current argument field explicitly.

22 c NO SUCH FIELD IN CHOSEN INSTRUCTION
Action: c is the character name typed. Select another field name or abandon.

23 symbol SYMBOLIC NAME ALREADY USED
Action: Make up an alternative symbolic name.

24 position NOT A VALID ARGUMENT POSITION
Action: Retype the numeric argument position.

25 position ARGUMENT POSITIONS ARE NOT CONSECUTIVE
Action: Respecify the argument fields, ensuring that they are consecutive.

26 position POSITION ALREADY USED
Action: An argument field has already been defined which uses this position. Choose an alternative position or abandon and redefine the instruction.

27 NO WORDS DEFINED YET
Action: The desired operation will only be allowed once word formats have been defined.

28 NO INSTRUCTIONS DEFINED YET
Action: The desired operation will only be allowed once instructions have been defined.

29 SYMBOL DEFINITION TABLE FULL
Action: There is no more space in the symbol definition table. Save your current definitions and exit the program. You will have to recompile the program (see Appendix A1.6), increasing the maximum number of unique argument fields referenced using associated mnemonics (VARSMAX in the const declaration of all modules).

33 newwidth ILLEGAL FIELD WIDTH
Action: Retype the new field width.

8. USING THE CODE-SKELETON DEFINITION PROGRAM

The current program has been implemented in Pascal MT+ on a CP/M based system (North Star Advantage), but could be easily altered to run on other Pascal systems, with only minor modifications needed to the code-skeleton definition program and to the following description of how to use it.

8.1 CURRENT IMPLEMENTATION REQUIREMENTS

Before the Code-Skeleton definition program can be used, the Machine Definition program described in chapter 3 must be used to set up a file (a ".MAC" file on this implementation) of instruction formats.

The following six files are required to run the current CP/M based (North Star Advantage) version of the code-skeleton definition program.

CODDEF.COM	-	Executable Root Program
CODDEF.001	}	
CODDEF.002	}	
CODDEF.003	}	Overlays for the program
CODDEF.004	}	
CODDEF.005	}	

Also useful, but not mandatory, are the following two message files which improve the friendliness of the program.

CODDEF.HLP	-	Help message file
CODDEF.MES	-	Error message file

To run the program, type

```
CODDEF <RET>
```

and answer the questions or make selections from menus which follow. The relocatable code skeletons will be saved on a user named file.

During the course of the program, two temporary work files are created. They are named CODDEF.TMP and CODDEF1.TMP. These names should not be used for any other files.

The disc space required to store the root program and overlays is 4Øk, with an additional 8k required if the message files are included. The machine definition and resultant skeleton file require typically 1Øk of storage while at least 6k of free space should be left on the disc for use by the temporary files.

8.2 DRIVING THE PROGRAM

Provided the help message file (CODDEF.HLP) is present, the user may request help at any time by responding to an invitation for input with

```
? <RET>.
```

Help relevant to the current operation will be supplied. In some cases, additional help or an example will be offered.

If the character "?" is a valid part of a mnemonic, separator or argument field in the assembly language for the target machine, then the user should choose another character as his help request character. This can be done when invited to do so at the start of the program.

The sequence <ESC> <RET> can be used to abandon any operation. Execution returns to the main menu or to the appropriate sub-menu. Once an operation has been abandoned, the memory-resident code-skeleton is returned to its state prior to commencement of the operation.

At the start of a session, the user is requested to supply the names of two disc files. The first is the machine definition file, set up by the process outlined in chapter 7. If this file cannot be found, the program terminates. The second file is the one on which code-skeletons are stored. If this file does not exist, it will be created. If it does exist, it will be modified.

The main menu is displayed automatically. The user selects one of eight options by typing a representative character.

```
E - Edit/Define a Code-Skeleton
D - Delete a Code-Skeleton
L - List a Code-Skeleton
S - Edit/Define a Run-time Subroutine
R - Delete a Run-time Subroutine
P - Print a Run-time Subroutine
B - Alter the Output Base
X - Exit
```

After each major operation, the program returns to the main menu.

For the options discussed in paragraphs 8.2.1, 8.2.3 and 8.2.5 below, the user is required to supply the index number of the intermediate code whose code-skeleton is to be operated on. This number is used by the program to associate a particular code-skeleton with an intermediate code. The allocation of numbers is at the discretion of the user (within an implementation dependent range, currently set at Ø..77) and is made by the user. It is usually convenient to use the function value of an intermediate code as its index number.

8.2.1 <u>Editing or Defining a Code-Skeleton</u> (Option E on menu)

All code skeletons are initially empty. However, if a previous run has already established a code-skeleton file, and this file was opened at the start of the session, some of the code skeletons might already be defined.

On entering the definition/edit mode, if a code skeleton exists for the requested intermediate-code, it will be recalled for editing. This will be indicated by the message "EDITING SKELETON". Otherwise, the message "DEFINING NEW SKELETON" invites the user to begin inserting a new code skeleton. The rules for editing are the same in both cases (remember that a code skeleton may be empty).

The edit mode may be described as the combination of an assembler and a line-by-line editor. In describing the body of target code which produces the effect of an intermediate language code, it is convenient to borrow some terminology from the medical world. Hence we speak of the code sequence as a "code-skeleton". We shall take the analogy further and refer to an individual instruction within the code-skeleton as a "rib". The "current" rib/instruction is that instruction within the code-skeleton upon which an edit command will operate.

When edit mode is entered, the current rib is an imaginary one at the top of the skeleton. For example, a three rib skeleton for an 8Ø8Ø target machine might be:

```
   top:                     <-- CURRENT
 rib 1:     LXI    D,Ø
 rib 2:     PUSH   D
 rib 3:     CALL   IWRTE
```

Thus a request to print the current instruction will produce the message "top of skeleton". To print out rib 1, one must first establish this rib as the current one.

```
  top:
rib 1:      LXI    D,Ø    <-- CURRENT
rib 2:      PUSH   D
rib 3:      CALL   IWRTE
```

Of course, if the skeleton is empty, then it consists only of the imaginary top rib. In this case, any attempt to move down the skeleton will produce the message "bottom of skeleton".

Since this part of the program operates interactively, there is no prompting from the computer. However, the help request character will be recognised and serviced. All edit commands are control sequences (the <CTRL> key is pressed along with another key), and the program will attempt to interpret any normal character sequence as an instruction.

<CTRL> **X** will exit from the definition/edit mode to the main menu, saving the current state of the code skeleton. This is perhaps the most important sequence to remember as it is the only means of returning to the main menu.

If one wishes to define a new code skeleton, one simply begins typing the legal machine mnemonics, one on a line. The program will validate them and report any errors. The most recent rib typed becomes the current.

To edit a code skeleton, one makes use of the following control sequences: (Most of these operations end by displaying the new current rib. Thus the current instruction is the one most recently displayed on the screen).

<CTRL> T	move to **Top** of code skeleton.
<CTRL> N	**Next** instruction becomes the current, i.e. advance one instruction.
<CTRL> B	**Back** up one instruction, i.e. the previous instruction becomes the current.
<CTRL> P	**Print** the current instruction.
<CTRL> L	**List** the entire code skeleton. (This does not alter the pointer to current rib).
<CTRL> R	**Remove** the current instruction from the code skeleton. (The next instruction becomes the current, unless the last instruction is being deleted, in which case the previous becomes the current). **Once an instruction has been removed, it CANNOT be retrieved.**

To **insert** a new instruction , simply position the current rib pointer at the instruction **before** the position to be occupied by the new instruction. Then type the new instruction. INSTRUCTIONS ARE INSERTED AFTER THE CURRENT INSTRUCTION. The newly inserted instruction will become the current one.

To insert an instruction at the top of the code skeleton, move (<CTRL> T) to the top of the skeleton and type the new instruction.

To alter a rib, it must be deleted, and the replacement line inserted.

An editing session might take the following course:

```
<CTRL> T      top:                       <-- CURRENT
            rib 1:     LXI    D,Ø
            rib 2:     PUSH   D
            rib 3:     CALL   IWRTE

<CTRL> N      top:
            rib 1:     LXI    D,Ø       <-- CURRENT
            rib 2:     PUSH   D
            rib 3:     CALL   IWRTE

<CTRL> N      top:
            rib 1:     LXI    D,Ø
            rib 2:     PUSH   D         <-- CURRENT
            rib 3:     CALL   IWRTE

XCHG          top:
            rib 1:     LXI    D,Ø
            rib 2:     PUSH   D
            rib 3:     XCHG             <-- CURRENT
            rib 4:     CALL   IWRTE

<CTRL> B      top:
            rib 1:     LXI    D,Ø
            rib 2:     PUSH   D         <-- CURRENT
            rib 3:     XCHG
            rib 4:     CALL   IWRTE

<CTRL> R      top:
            rib 1:     LXI    D,Ø
            rib 2:     XCHG             <-- CURRENT
            rib 3:     CALL   IWRTE

<CTRL> X    (Return to the main menu,  retaining the
            most recent version of the code-skeleton)
```

If, while inserting a new rib, the user types the abandon sequence (<ESC><RET>), the skeleton returns to its condition before the insertion. As a new instruction is entered, it is checked field by field. As soon as an error is detected, evasive action is taken before the user is allowed to continue. For example, if the instruction mnemonic is not recognised, the instruction is immediately abandoned. Once the instruction mnemonic has been identified, the program is able to offer assistance when further errors are detected. The user is offered information about the correct format for the instruction (order of arguments, valid separators, etc.) and legal values for argument fields. Thus as long as the machine definition program has been run correctly, it is impossible for the code-skeletons to contain assembly/compilation errors.

Normally, numeric argument fields are entered using decimal values. If the user prefers, the field value may be prefixed:

(1) with a "B", and entered as a binary number (e.g. B1ØØØ11),

(2) with an "H", and entered as an hexadecimal number (e.g. H23)

or (3) with an "O", and entered as an octal number (e.g. O43).

A check is made to ensure that the value will fit into the defined legal range for the field. Negative values are converted according to 2's complement rules, and must be input using a minus sign.

All integer fields will be listed in the most recently defined default output base (see section 8.2.7).

Argument fields which require a symbolic name (for example register fields) must be typed using one of the defined symbols. An illegal symbol will prompt the program to offer a list of legal symbolic names for the particular argument.

Address fields should be substituted with a compile-time address name, optionally followed by an offset in the range -128..+127. For example,

```
JMP PC+8
```

will jump to the address of the current instruction (its code generation time address) plus 8 words (i.e. minimum addressable memory units). As for numeric fields, an address offset is usually entered using a decimal value. However, prefixing the number with "B", "O" or "H" enables binary, octal or hexadecimal to be used.

Table 8.2.1 gives a list of reserved compile-time names which may be used in a numeric field. In addition to these names, the user may also use the names of his own run-time subroutines (see section 8.2.2) in an address field.

Note the difference between the names X, XN and XA in table 8.2.1. As an example, suppose the intermediate code LOAD X has the code-skeleton LDI X. Then LOAD 1Ø would be translated as LDI 1Ø. However, if the skeleton was LDI XN, then LOAD 1Ø would be translated as LDI -1Ø. For the names X and XN, the argument value itself is used. However, suppose the intermediate code BRANCH X has the code-skeleton JMP XA. BRANCH 1Ø is translated into a jump to the code-skeleton in the target program which represents the intermediate code whose address in the intermediate program is 1Ø. So a branch to code-skeleton 1Ø might be translated as JMP 276.

TABLE 8.2.1

RESERVED COMPILE-TIME VALUES

Name	Compile-Time Value
PC	address of current instruction.
X	value of argument 1 of the current ILC (intermediate language code).
Y	value of argument 2 of the current ILC.
XN	negative of arg 1 value of the current ILC
YN	negative of arg 2 value of the current ILC.
XA	arg 1 of the current ILC used as an address. The equivalent target code address will be substituted.
YA	arg 2 as an address as for XA.
WRTLN	address of RTS (run-time subroutine) to write out the current output buffer.
IWRTE	address of RTS for integer output.
FPWRTE	address of RTS for floating-point output.
STRWRT	address of RTS for string output.
IREAD	address of RTS for integer input.
FPREAD	address of RTS for floating-point input.
FLOAT	address of RTS for integer to floating-point conversion.
FIX	address of RTS for floating-point to integer conversion.
IADD	address of RTS for integer addition.
ISUB	address of RTS for integer subtraction.
IMUL	address of RTS for integer multiplication.
IDIV	address of RTS for integer division.
FPADD	address of RTS for floating-point addition.
FPSUB	address of RTS for floating-point subtraction

(table 8.2.1 continued...)

FPMUL	address of RTS for floating-point multiplication.
FPDIV	address of RTS for floating-point division.
ERROR	address of RTS for run-time error reporting.
CHKSUB	address of RTS for run-time range checking.

8.2.2 Editing or Defining a Run-Time Subroutine (Option S)

A run-time subroutine is a small piece of code which is common to several code-skeletons. It can be referenced often by using its name in the address field of a referencing instruction. The actual address will be filled in at code-generation time.

All run-time subroutines are included only once in the run-time object program. Thus run-time subroutines can contribute towards more economical use of memory.

The user is able to define and edit run-time subroutines in exactly the same way as code-skeletons. Edit commands are identical to those described in section 8.2.1. From a definition point of view, the run-time subroutine differs from the normal code-skeleton only in the way it is referenced.

Whereas normal code-skeletons are referred to by their index numbers, a run-time subroutine is given a name. On entering run-time subroutine edit mode, the user must supply the name of the run-time subroutine he wants to define or modify. This can be any name of his choice, except those already used as compile-time address names (i.e. the names of Table 8.2.1). However, names beginning with "D", "B", "O", "H" or a numeric digit should be avoided as they might be confused by the program with numeric input. If the run-time subroutine does not yet exist, it will be created. A response of * <RET> to the prompt for the name of the run-time subroutine, will produce a list of names of all user-declared subroutines.

For example, run-time subroutine SAVE might be useful in an 8080 implementation for saving all registers on the stack before some operation which overwrites them.

```
SAVE:     XTHL          ; save H,L
          PUSH   D      ; save D,E
          PUSH   B      ; save B,C
          PUSH   PSW    ; save A,PSW
          PCHL          ; return
```

The run-time subroutine is called from a code-skeleton in the normal way for the assembly language:

```
        CALL    SAVE      ; save all registers
        MVI     A,5       ; instructions which
         .                ; overwrite registers
         .
         .
```

If the user gives one of his run-time subroutines the name "START", its code will be given special status as the initialisation code for the program. A routine with this name (if one exists) is always placed at the start of the executable code. This feature can be used by the code generator implementor to make initial assignments, or to restart the program by branching to address "START".

No high level mechanism exists for passing parameters to and from user defined run-time subroutines. Data communication is via the registers or the stack at the implementor's discretion.

8.2.3 Deleting a Code-Skeleton (Option D on main menu)

After making sure that the deletion request is intentional, the program removes the code-skeleton. This means that the skeleton will revert to an "undefined" state. **Once a code-skeleton has been deleted, it CANNOT be retrieved.**

8.2.4 Deleting a Run-Time Subroutine (Option R on main menu)

The user is required to supply the name of the run-time subroutine in the manner described in section 8.2.2. Remember that * <RET> can be used to list the names of all user-defined subroutines. After making sure that the deletion request is intentional, the program deletes the subroutine's code-skeleton and removes its name from the list of user defined subroutines. **Once a run-time subroutine has been deleted, it CANNOT be retrieved.**

8.2.5 Listing a Code-Skeleton to the Screen (Option L)

The user merely supplies the index number of the code-skeleton to be listed, in assembly format, on the screen. If the skeleton has not yet been defined, or has been deleted, the message "empty skeleton" is displayed.

8.2.6 Listing a Run-Time Subroutine to the Screen (Option P)

Once again, the user is required to supply the name of the run-time subroutine in the manner described in section 8.2.2. The subroutine is listed as for a code-skeleton.

8.2.7 Altering the Default Integer Output Base (Option B)

By selecting option B in the main menu, the user is allowed to set the numeric base (binary, octal, decimal, hexadecimal or other) in which all integer values will be listed. The base remains valid for the duration of the program unless it is reset. At the start of the program, the default base is set to 1Ø, i.e. decimal.

Before prompting the user to supply the new output base, the program lists the current output base value. This might be useful if the user is not sure what base he is currently using. The CP/M implementation allows integer fields to be printed out in any numeric base in the range 2 to 16; e.g. binary, octal, decimal, hexadecimal, or anything in between. To alter the base, type the decimal numeric value of the desired base, e.g. 2 for binary. Negative numbers are output preceded by a minus sign in all number systems except binary, in which case negative numbers are output as a 2's complement number. For positional systems with a base greater than 1Ø, the alphabetic characters A, B, C, ... are used to represent the digits 1Ø, 11, 12, ...

For example, An ICL 19ØØ code sequence to report run-time error 24, is listed in various bases as:

```
base 10:        LDN     3  24
                BRN        ERROR

base 2:         LDN     3  000000011000
                BRN        ERROR

base 8:         LDN     3  30
                BRN        ERROR

base 13:        LDN     3  1B
                BRN        ERROR

base 16:        LDN     3  18
                BRN        ERROR
```

(The register field in this example has been defined as the symbolic name "3" representing the integer 3).

8.2.8 Exiting the Program (Option X on main menu)

On leaving the program from the main menu, all code-skeletons and user-defined run-time subroutines are consolidated on the code-skeleton file named by the user at the start of the program. During this process, the program makes use of a number of temporary work files. Since this disc file reorganization usually takes a few minutes, the program writes out the character "." periodically to reassure the user that it has not crashed.

8.2.9 A General Note on the Current Implementation

For rather esoteric programming reasons (outlined in Appendix B1.2.2 for the more diligent reader), the current CP/M implementation of the Code-Skeleton Definition Program saves new versions of code-skeletons on an inaccessible temporary file as soon as a subsequent skeleton is called up. So as to alleviate this inconvenience, the user is advised to make all necessary modifications and listings of a skeleton before calling up another. Should one need to revive an altered skeleton later, it is necessary to exit (to allow disc reorganization to take place) and begin a new session.

An earlier version of the Code-Skeleton Definition Program is available which does not have this disadvantage, but which is restricted in other ways.

8.3 ERROR CONDITIONS

Below is listed the set of error messages which a user of the Code-Skeleton Definition program is likely to encounter. "Action" describes the steps the user should take to overcome the error condition.

1 opcode - NOT A VALID INSTRUCTION MNEMONIC
Action: Retype the instruction if a typing error caused the problem, or return to the machine definition program and add the desired instruction to the system.

2 ILLEGAL OPERAND VALUE
Action: Retype the instruction, or request help on valid operand values.

3 UNRESTRICTED FIELD MAY ONLY BE A COMPILE-TIME ADDRESS
Action: A field of this size may only be an address. Retype the instruction using the name of a compile-time address.

4 symbol - NOT A LEGAL COMPILE-TIME ADDRESS NAME
Action: Retype the instruction using a valid reserved or user-defined address name.

5 INTEGER VALUE REQUIRED
Action: A non-numeric character has been typed in an integer input field. Re-type the correct integer value.

6 INVALID INTERMEDIATE CODE NUMBER
Action: Retype your selection within the legal range (Ø..77 on the current implementation).

7 value > length BITS - VALUE TOO LARGE FOR FIELD
Action: "value" is too large to be stored in a field of "length" bits. Retype the value.

8 NUMERIC VALUE REQUIRED
Action: Type a numeric value in response to the request for input.

9 NAME ALREADY USED FOR A RESERVED COMPILE-TIME ADDRESS
Action: Select a unique run-time subroutine name.

10 RUN-TIME SUBROUTINE TABLE FULL
Action: There is no more space in the user-defined run-time subroutine table. Save your current definitions and exit the program. You will have to recompile the program (see Appendix B1.4), increasing the maximum number of defined run-time subroutines and the maximum number of skeletons (RTSMAX and MAXNOSKEL in the const declaration of all modules).

11 INVALID NON-PRINTABLE CHARACTER IN MNEMONIC
Action: Retype the mnemonic name using only printable characters.

12 INVALID NUMERIC BASE (RANGE = 2..16)
Action: The program reverts to the original base. The user may retype the numeric base within the valid range if he wishes.

14 BASE n ILLEGAL DIGIT FOR GIVEN BASE
Action: Retype the numeric input using digits for base n.

In later versions of the Code-Skeleton Definition Program, the following messages may also be encountered and are self-explanatory.

CODE-SKELETON n HAS ALREADY BEEN ALTERED IN THIS SESSION AND WRITTEN OUT TO THE TEMPORARY FILE. YOU WILL NOT BE ABLE TO ACCESS IT AGAIN IN THE CURRENT SESSION. EXIT THE PROGRAM AND START A NEW SESSION.

RUN-TIME SUBROUTINE name HAS ALREADY BEEN ALTERED IN THIS SESSION AND WRITTEN OUT TO THE TEMPORARY FILE. YOU WILL NOT BE ABLE TO REFERENCE IT AGAIN UNTIL THE NEXT SESSION.

9. USING THE CODE GENERATION PROGRAM

9.1 CURRENT IMPLEMENTATION REQUIREMENTS

Before the Code Generation program can be used, the following programs must have been run: (a) The Machine Definition program described in chapter 7 which sets up a file of instruction formats, (b) the Code-Skeleton definition program of chapter 8 which sets up the code-skeleton file and (c) a suitable parser which outputs the intermediate language code version of the source program.

The following two files, which can be stored in 32k of disc space, are required to run the current CP/M based (North Star Advantage) version of the general code generation program.

CODGEN.COM	-	Executable Root Program.
CODGEN.DRV	-	Parameter table.

To run the program, type

```
CODGEN <ret>
```

9.2 THE USER'S POINT OF VIEW

The end user of the code generator will be prompted to supply the names of two files: (1) the intermediate language code input file and (2) the target machine code output file. During code generation, the program writes out the character "." periodically.

9.3 IMPLEMENTOR'S NOTE

A general code generator obviously requires more information than the names of the source and target files. However, it would be most inconvenient for the user to have to supply this information at each compilation. The code generator therefore uses a parameter file, called CODDEF.DRV, which must be set up by the implementor to furnish details about the run-time environment. The format of this file is shown in table 9.3.1. Since the format is straightforward, the file may be set up using an editor. However, should the implementor feel unsure about setting up the file, a program does exist (called DRVTAB) which prompts the implementor for each parameter and performs appropriate validation. An example of a parameter file is given in section 3.4.

Apart from setting up this file, the compiler writer merely has to ensure that the parser outputs the intermediate codes in the desired format (all are integer codes followed by two arguments, which have zero values if not used), and that the loader (see chapter 4) is able to accept the format of the target code output file (Table 9.3.2).

The code generator uses the function values of the intermediate codes to index the code-skeleton file. If this is not the relationship which the implementor used when defining his code skeletons, he is responsible for writing an intermediate stage to convert the intermediate codes to index values.

Since the Code Generator is the most likely program in the system to need modification during implementation of a new compiler (for example, if the implementor wants to use a different type of intermediate code), the reader is referred to Appendix C1 for implementation details of the program.

TABLE 9.3.1

FORMAT OF THE CODE GENERATOR PARAMETER FILE

NAME -	name of the Machine Definition File
NAME -	name of the Code-Skeleton File
BITS -	size in bit of smallest addressable unit of memory
BASE -	Output base for object code file - this is expressed in "number of bits per digit" to allow bases which are powers of 2; e.g. 1 for binary, 3 for octal, 4 for hexadecimal.
SIZE -	The size of the address field in words, i.e. the number of words of memory required to address the whole of memory.
PC -	Start address of application programs in the run-time environment.
RUN-TIME - ADDRESSES	Now follow the addresses of all standard run-time subroutines which are included in the load-time package. The addresses must be given in the order indicated below, followed by any further subroutines which the compiler writer might have implemented.
	WRTLN, IWRTE, FPWRTE, STRWRT, IREAD, FPREAD, FLOAT, FIX, IADD, ISUB, IMUL, IDIV, FPADD, FPSUB, FPMUL, FPDIV, ERROR, CHKSUB ...

TABLE 9.3.2

FORMAT OF THE TARGET CODE OUTPUT FILE

```
string of program code.....#
backpatch table.....#
start address #
```

All output to the target code file is in the form of strings of digits in the base implied by BASE in the parameter file. For bases greater than 1Ø, the letters "A", "B", "C", ... are used to represent the digits 1Ø, 11, 12, ... as for hexadecimal. At the end of each section of information, a terminating "#" character is output. For example, if BASE=4 then the hexadecimal output for the 8-bit codes 27, 254 will be 1BFE#. The program code is absolute (i.e. generated for loading into a particular memory address), and forward references are filled in by the loader after consulting the backpatch table. The backpatch table consists of the string of the following sequences: address of word, value to be logically "OR-ed" with the word. This is repeated for as many words as need to be patched. The address is expressed using the number of bits given by SIZE * BITS in the parameter file. Finally, the start address is the address to which the loader must jump, on completion of the backpatching process, to begin execution of the application program.

REFERENCES

[1] E. MARSHALL and S. HAMPLE, "Children's Letters to God", COLLINS/FOUNTAIN, 1977.

[2] M.V. ZELKOWITZ and W.G. BAIL, "Optimization of Structured Programs", SOFTWARE - PRACTICE AND EXPERIENCE, Vol. 4, No. 1, 1974.

[3] G.A. KILDALL, "A Unified Approach to Global Program Optimization", PROC. ACM SYMP: Principles of Programming Languages, 1973.

[4] D.E. KNUTH, "A History of Writing Compilers", COMPUTERS AND AUTOMATION, Vol. 11, No. 12, Dec. 1962.

[5] J.A. FELDMAN and D. GRIES, "Translator Writing Systems", COMM. ACM, Vol. 11, No. 2, Feb. 1968.

[6] A.V. AHO, "Translator Writing Systems: Where Do They Now Stand", COMPUTER, Vol. 13, No. 8, Aug. 1980.

[7] A.V. AHO and J.D. ULLMAN, "Principles of Compiler Design", ADDISON-WESLEY, 1977.

[8] D. GRIES, "Compiler Construction for Digital Computers", WILEY, 1971.

[9] E.L. BAUER and J. EICKEL, "Compiler Construction - An Advanced Course", SPRINGER-VERLAG, 1974.

[10] R. BORNAT, "Understanding and Writing Compilers", MACMILLAN, 1979.

[11] R.E. BERRY, "Programming Language Translation", WILEY, 1981.

[12] D.W. BARRON, "Pascal - The Language and its Implementation", WILEY, 1981.

[13] D.E. KNUTH, " The Art of Computer Programming, Vol 1", ADDISON-WESLEY, 1968.

[14] S.C. JOHNSON, "A Portable Compiler: Theory & Practice", PROC. ACM SYMP: Principles of Programming Languages, 1978.

[15] S.L. GRAHAM, "Table-Driven Code Generation", COMPUTER, Vol. 13, No. 8, Aug. 1980.

[16] B.W. LEVERETT, G.G. CATTELL, O. HOBBS, M.NEWCOMER, H. REINER, R. SCHATZ and A. WULF, "An Overview of the Production-Quality Compiler-Compiler Project", COMPUTER, Vol. 13, No. 8, Aug. 1980.

[17] P. KORNERUP, B.B. KRISTENSEN and O.L. MADSEN, "Interpretation and Code Generation Based on Intermediate Languages", SOFTWARE - PRACTICE AND EXPERIENCE, Vol. 1Ø, No. 8, Aug. 198Ø.

[18] O. LECARME and M.C. PEYROLLE-THOMAS, "Self-Compiling Compilers: An Appraisal ot their Implementation and Portability", SOFTWARE - PRACTICE AND EXPERIENCE, Vol.8, No. 2, March 1978.

[19] R.G.G. CATTELL, J.M. NEWCOMER and B.W. LEVERETT, "Code Generation in a Machine-Independent Compiler", PROC. SIGPLAN SYMP: Compiler Construction, Aug. 1979.

[2Ø] M. GANAPATHI and C.N. FISHER, "Automated Compiler Code Generation and Reusable Machine-Dependent Optimization - A Revised Bibliography", SIGPLAN NOTICES, Vol. 18, No. 4, April 1983.

[21] S.L. GRAHAM, R.R. HENRY and R.A. SCHULMAN, "An Experiment in Table Driven Code Generation", SIGPLAN NOTICES, Vol. 17, No. 6, June 1982.

[22] W.A. BARRETT and J.D. COUCH, "Compiler Construction: Theory and Practice", SRA, 1979.

[23] P.D. TERRY, "Notes on CLANG, A Compiler for Teaching Concurrent Programming", RHODES UNIVERSITY Internal Publication, 1983.

[24] W.J. WELLER, A.V. SHATZEL and H.Y. NICE, "Practical Microcomputer Programming - The Intel 8Ø8Ø", 1979.

[25] R. ASHLEY and J.N. FERNANDEZ, "Introduction to 8Ø8Ø/8Ø85 Assembly Language Programming", WILEY, 1981.

[26] A.R. MILLER, "8Ø8Ø/Z8Ø Assembly Language: Techniques for Improved Programming", WILEY, 1981.

[27] J.W. COFFRON, "Practical Hardware Details for 8Ø8Ø, 8Ø85, Z8Ø & 68ØØ Microprocessor Systems", PRENTICE-HALL, 1981.

[28] W. CHRISTENSEN, "8Ø8Ø Programming Tutorial Articles", LIFELINES, Sep. 1981 to Nov 1982.

[29] J.G. LIPHAM, "Relocating 8Ø8Ø System Software", BYTE, Jan 198Ø.

[3Ø] K. BARBIER, "CP/M Assembly Language Programming", PRENTICE-HALL, 1983.

[31] RMS-11 MACRO-11 Reference Manual, Digital Equipment Corporation, 1979.

[32] COMPASS Version 3 Reference Manual, Control Data Corporation, 1982.

[33] PLAN Reference Manuals (30 A,B,C), ICL Publications, 1972.

[34] A.S. TANENBAUM, "Structured Computer Organization", PRENTICE-HALL, 1976.

[35] PASCAL /MT+ User's Guide, Release 5, LIFEBOAT ASSOCIATES, 1981.

[36] T.S. FRANK, "Introduction to the PDP-11 and its Assembly Language", PRENTICE-HALL, 1983.

[37] S.C. JOHNSON, "Language Development Tools on the UNIX System", COMPUTER, Vol. 13, No. 8, Aug 1980.

[38] N. WIRTH, "Pascal-S: A Subset and its Implementation", Edited D.W. BARRON [12].

[39] P.D. TERRY, "Pascal-S+, A System for Beginners", RHODES UNIVERSITY Internal Publication, 1981.

[40] M.H. WILLIAMS, "The Programming Language BPL", COMPUTER JOURNAL, Vol. 25, No. 3, Aug 1982.

[41] D.A. GEWIRTZ, "Two More C Compilers", MICROSYSTEMS, Nov/Dec 1982.

[42] R.A. PHRANER, "Nine C Compilers for the IBM PC", BYTE, Aug 1983.

A P P E N D I C E S

Due to their bulk, the appendices are not included in this document. A summary of their contents appears below. Should the reader wish to obtain a copy of the appendices or copies of the programs which are listed in the appendices, they may do so by contacting the Computer Science Department, Rhodes University. In addition to the standard routines of appendix 4 for the 8080 processor, equivalent run-time subroutines have been written for code generators implemented on the ICL 1900 GEORGE system and the PDP-11 RSX-11M operating system.

APPENDIX A - The Machine Definition Program

This appendix contains:

A1. Implementation details of the Machine Definition Program.

A2. A Pascal source listing of the Machine Definition Program.

A3. A listing of the Help Message File.

A4. A listing of the Error Message File.

A5. An Example Run.

APPENDIX B - The Code-Skeleton Definition Program

This appendix contains:

B1. Implementation details of the Code-Skeleton Definition Program.

B2. A Pascal source listing of the Code-Skeleton Definition Program.

B3. A listing of the Help Message File.

B4. A listing of the Error Message File.

B5. An Example Run.

APPENDIX C - The General Code Generation Program

This appendix contains:

C1. Implementation details of the General Code Generation Program.

C2. A Pascal source listing of the General Code Generation Program.

APPENDIX D - 8080 Monitor Routines and Absolute Loader

This appendix contains an 8080 Assembler source listing of the standard run-time subroutines and an absolute loader suitable for use with the general code generator.

APPENDIX E - The Clang Compiler

This appendix contains a Pascal source listing of the simple CLANG Parser and Interpreter [23].