EM for the Data General Eclipse S/130

Supervisor: Dr. R.E.M. Cooper.

Robert Joe
October 1985
# Table of Contents

1. Introduction .............................................................................................................. 1

2. The EM machine ....................................................................................................... 4
   2.1 Registers .............................................................................................................. 4
   2.2 Memory .............................................................................................................. 5
   2.3 Data address space ............................................................................................ 6
   2.4 The stack and the procedure calling mechanism ............................................. 6

3. The Eclipse S/130 ...................................................................................................... 9
   3.1 Addressing modes ............................................................................................. 9
      3.1.1 Absolute addressing mode ........................................................................ 9
      3.1.2 P.C. relative addressing mode ................................................................... 9
      3.1.3 Accumulator relative addressing mode ..................................................... 10
      3.1.4 Direct and indirect addressing ..................................................................... 10

4. Code Generation ..................................................................................................... 12
   4.1 The table driven code generator ....................................................................... 12
      4.1.1 Creation of the code generator .................................................................. 13
      4.1.2 How the code generator works .................................................................... 13
         4.1.2.1 Simulating the runtime stack .............................................................. 13
         4.1.2.2 Keeping track of register contents ...................................................... 14
         4.1.2.3 Keeping track of condition codes ....................................................... 15
   4.2 Alternatives for implementing a back end ......................................................... 16
   4.3 Description of the machine dependent driving table ......................................... 18
      4.3.1 Constant definitions .................................................................................. 18
      4.3.2 Register definitions .................................................................................... 18
      4.3.3 Token definitions ....................................................................................... 20
      4.3.4 Token expressions ..................................................................................... 22
      4.3.5 Code rules .................................................................................................. 22
4.4 Quality of code ................................................................. 27

5. Mapping the EM machine onto the Eclipse ................................................. 30
5.1 Pointer and word sizes ........................................................................... 30
5.2 Byte addressability .............................................................................. 30
5.3 Design of the runtime stack .................................................................. 31
   5.3.1 Alternatives for the runtime stack .............................................. 31
5.3.2 Using the hardware stack ............................................................... 33
   5.3.2.1 The procedure calling mechanism ................................... 34
   5.3.2.2 Local variables and parameters ....................................... 35
5.3.3 Using the software stack ................................................................. 40
   5.3.3.1 The procedure calling mechanism ................................... 40
   5.3.3.2 Local variables and parameters ....................................... 41

6. Runtime Library ........................................................................................... 42
6.1 Implementation of runtime routines .................................................... 42
6.2 Trap handling ......................................................................................... 43
6.3 System calls ......................................................................................... 44

7. Summary and Conclusions ........................................................................ 45

8. References ................................................................................................. 49

9. Appendices ............................................................................................... 50
   Appendix A - Back end table and machine dependent tables.
   Appendix B - Runtime routines.
   Appendix C - Default trap handler.
   Appendix D - System call routines.
1. Introduction

The aim of this project was to develop a back end for EM with the Data General Eclipse S/130 as the target machine, and consequently to bring up the Pascal-VU compiler running under DG UNIX.

A commonly used method for implementing \( N \) languages on \( M \) different machines has involved writing or modifying a compiler for each language on each of the \( M \) machines. This means that \( N \times M \) programs must be written in order to implement a compiler for each of \( N \) languages on \( M \) different machines. Writing good compilers is both a time consuming and expensive task. The most common attempt to solve this \( N \times M \) problem is to write \( N \) front ends, each of which translates one source language to a common intermediate language, and \( M \) backends each of which translates programs from the intermediate language to the assembly language or machine code of the target machine. Using this approach only \( N+M \) programs have to be written to implement \( N \) languages on \( M \) machines instead of \( N \times M \) programs. Examples of this approach in solving the \( N \times M \) problem include the BCPL to OCODE compiler and the Pascal-P compiler. A compiler building system called the Amsterdam Compiler Kit has been developed at vrije universiteit in Amsterdam. "The Amsterdam Compiler Kit is an integrated collection of programs designed to simplify the task of producing portable (cross) compilers and interpreters" [1]. The idea behind the Amsterdam Compiler Kit is to solve the \( N \times M \) problem using an intermediate code as described above. The intermediate code used is called EM code. A compiler produced using the Amsterdam Compiler Kit consists of up to seven phases as illustrated in figure 1.1.
A secondary aim of the project is to draw some conclusion based on the experienced gained, as to whether table driven code generation is a worthwhile thing.

The back end was developed on a VAX 11/750 running ULTRIX-32 (the ULTRIX-32 system is based on 4.2BSD). The version of the Amsterdam Compiler Kit available was tailored specifically for PDP 11 machines running UNIX version 7. Thus to a lesser extent the project also entailed making changes to parts of the kit in order to get the required parts running on the VAX.

The second section of this report presents a look at the EM machine while section three gives an overview of the addressing modes available on the Eclipse. Code generation is introduced in section four and a discussion is presented on the table driven code generator from the Amsterdam Compiler Kit and the back end table for the Eclipse. Section five deals with the problem of mapping the architecture of the EM machine onto the architecture Eclipse. Section six gives a brief discussion on the implementation of the runtime library for the Pascal-VU compiler. Finally, section seven presents a summary, and draws some conclusions based on the work done on the project, and presents suggestions for the future work required in order to complete the porting of the Pascal-VU compiler to the Eclipse.
Figure 1.1 Possible phases of a compiler under the Amsterdam Compiler Kit
2. The EM machine

The EM machine is a stack based hypothetical machine. The assembly language of the EM machine is referred to as EM code. EM code was designed to allow translation to a wide range of existing machines and EM machine code was designed to allow interpretation on a wide range of machines. The architecture of the EM machine was designed to ease the task of code generation for high level languages such as Pascal, C, Ada. The built in object types include signed and unsigned integers, floating point numbers, pointers and sets of bits.

2.1 Registers

The EM machine has no general purpose registers, any intermediate results and expression evaluations are performed using the stack. However there are a few internal registers with specific functions as follows:

**PC** - Program Counter. Points to the next instruction to be executed.

**LB** - Local Base pointer. Points to the first word above the local variables belonging to the currently active procedure.

**AB** - Argument Base pointer. Points to the first parameter of the currently active procedure.

**SP** - Stack Pointer. Points to the top of the stack ie the word on the stack with the lowest address.
HP - Heap Pointer. Points to the top of the heap.

2.2 Memory

The EM machine has separate instruction and data spaces and is byte addressable. Furthermore memory in the EM machine is divided into fragments which must be word aligned. A fragment consists of consecutive machine words and has a base address and a size. The reason for having memory divided into fragments is to ease implementation of EM on a wide range of existing machines which have varying memory structures. Objects in the EM machine must obey the following rules and constraints:

a) The size of an object must be a multiple of the word size or a divisor of the word size.

b) If an object whose size is a divisor of the word size is pushed onto the stack, a word is pushed onto the stack and the object occupies the least significant part of the word pushed. The rest of the word is set to zero. Popping an object from the stack whose size is a divisor of the word size removes a word from the stack, stores the least significant byte(s) of this word in memory and discards the rest of the word.

c) The address of a multiple byte object is the **lowest** address of all bytes it occupies.
d) For objects smaller than the word size, the address must be a multiple of the object size. For all other objects the address must be a multiple of the word size.

2.3 Data address space

The data address space is divided into three parts called the global data area, local data area and the heap data area. The size of the global data area is fixed per program and is addressed absolutely in the machine language. The heap data area grows upwards from low EM addresses to high EM addresses. The heap is initially empty and the initial value of the heap pointer HP marks the low end of the heap. The local data area is described in section 2.4.

2.4 The stack and the procedure calling mechanism

The stack on the EM machine grows from high EM addresses to low EM addresses. The stack is used for expression evaluation and for storing the activation records of currently active procedures. Actual parameters are pushed onto the stack before a procedure call. The procedure call instructions fill the return status block, and allocate space for local variables, implementation dependent information, and dynamic local generators. On return from a procedure call, any return value is copied to the function return area, the activation record created by the call is deallocated, and the status of the calling procedure is restored. The calling procedure is responsible for removing the parameters from the stack. The layout of each activation record is shown in figure 2.1.
parameters belonging to the procedure whose activation record is stored below

current activation record

- actual parameter n-1
- actual parameter 0
- return status block
- local variable 0
- local variable m-1
- implementation dependent
- dynamic local generators

Figure 2.1 Activation record on the EM machine
The parameters of a procedure belong to the activation record of the calling procedure. Before a procedure is invoked, the actual parameters are pushed onto the stack of the calling procedure. Parameters are referenced relative to \texttt{AB} with positive offsets, i.e., with offsets of zero and greater. Some EM instructions assume the parameter at offset zero contains the \texttt{LB} of the statically enclosing procedure, i.e., the static link. Thus if a procedure contains code to access the static link, the calling procedure must ensure the static link is passed as a parameter with offset zero from \texttt{AB}.

The \textbf{return status block} is implementation dependent and contains information such as the return address of the calling procedure. The size of this block must be \textbf{fixed} for any given implementation because some EM instructions must be able to obtain the base address of the procedure parameters and local variables. The \textbf{local variable} space is reserved by the \texttt{called} procedure. The space allocated must be a multiple of the wordsize. Local variables are referenced relative to \texttt{LB} with negative offsets. The \textbf{implementation dependent} block is used to save any implementation dependent data which couldn't otherwise be saved in the return status block. This block could, for example, be used to save a variable number of registers. The space for \textbf{dynamic local generators} is used to store local objects whose size is unknown at compile time. For example, it could be used to store flexible arrays in Algol68.
3. The Eclipse S/130

The Eclipse S/130 is a general purpose 16 bit machine with four 16 bit registers (referred to as r0, r1, r2, and r3). This section presents an overview of the addressing modes available on the Eclipse as referenced by other sections of this report. Other details of the Eclipse are explained in the sections which refer to them.

3.1 Addressing modes

Word addressing in the Eclipse S/130 can be done using absolute addressing, program counter relative addressing and accumulator relative addressing. Direct or indirect addressing can be used in any of these modes.

3.1.1 Absolute addressing mode

In absolute addressing mode, the effective address is simply the displacement specified in the instruction. For example the instruction

\[
\text{lda r0,4}
\]

loads the contents of memory location 4 into register r0.

3.1.2 P.C. relative addressing mode

In P.C. relative addressing mode, the effective address is found by adding the program counter to the displacement specified in the
instruction. For example the instruction

\texttt{lda r0,4,1}

loads the \texttt{contents} of the fourth word following this instruction.

\textbf{3.1.3 Accumulator relative addressing mode}

In accumulator relative addressing mode, the effective address is found by adding the contents of the accumulator specified in the instruction to the displacement specified in the instruction. Only registers r2 and r3 may be used as an index register in accumulator relative addressing. For example, the instruction

\texttt{lda r0,10,r2}

loads a word from some memory location into register r0. The address of the memory location is found by adding 10 to the contents of register r2.

\textbf{3.1.4 Direct and indirect addressing}

Direct addressing uses the intermediate address without modification. Indirect addressing uses the intermediate address as a pointer to the next address. Indirect addressing is specified in an instruction by preceding the displacement with an \texttt{@} sign. For example the instruction

\texttt{lda r0,@4}
loads the contents of the memory location pointed to by the contents of memory location 4.
4. Code Generation

The traditional approach to code generation has been to provide a set of routines where each routine is responsible for producing target machine instructions for different kinds of statements in some high level language. Code generation this way entails tasks such as register management, storage allocation for intermediate results, optimisation, deciding what machine instructions to generate, and deciding the order in which computations should be done. A lot of work has been done in the past for producing tools for automatically generating lexical analysers and parsers. Examples of this include LEX, a tool for automatically generating a lexical analyser, given a list of regular expressions describing a language, and YACC, a tool for automatically generating a parser given a grammar for some language. Similar work has been done in recent years on the code generation phase of compilation in order to produce tools for automatically generating code generators, given a description of some target machine. Such a tool has been developed as part of the Amsterdam Compiler Kit and forms the basis of the material presented in the remainder of this section.

4.1 The table driven code generator

Part of the Amsterdam compiler kit is a code generator generator and some machine independent C code which is used as a basis for creating code generators for various target machines.
4.1.1 Creation of the code generator

To create a code generator which generates assembly language for a target machine, given EM code as input, a machine dependent driving table and some machine dependent C code must be provided for each target machine. The code generator generator reads the driving table and creates two files `tables.h` and `tables.c`. These are then combined with the machine dependent C code and some machine independent C code to create a code generator for the target machine.

4.1.2 How the code generator works

Although a simple code generator could simply translate each EM instruction into the corresponding assembler instruction(s) of the target machine, the code generator produced from the Amsterdam Compiler Kit attempts to produce good code by simulating the runtime stack of the program being compiled, keeping track of the contents of registers and maintaining the status of condition codes.

4.1.2.1 Simulating the runtime stack

During code generation the simulated stack mirrors what the stack of a hardware EM machine would do at runtime. The idea behind simulating the runtime stack is to delay the emission of code as long as possible. This is used as a means to avoid generating any unnecessary code. As we shall see later, the machine dependent driving table specifies what is to be placed on the simulated stack. Thus, how well the code generator simulates the runtime stack is mainly dependent on how well the machine
dependent driving table has been written.

The code generator simulates the runtime stack of the program being compiled by maintaining a fakestack of tokens. A token on the fakestack represents a value and normally corresponds to an addressing mode or register of the target machine. For example, a token corresponding to a register represents the value in this register and a token corresponding to an addressing mode represents the value of a word in memory which can be accessed using that addressing mode. As sequences of EM instructions are read in and processed, assembler code for the target machine is produced and/or tokens are removed from or placed on the fakestack. One way to understand how the code generator simulates the runtime stack is to look at the actions of a hardware EM machine if it were to execute the same sequence of EM instructions the code generator takes as input. If the EM machine pushes a value onto the stack, then this would correspond to the code generator pushing a token representing this value onto the fakestack. If the EM machine takes the top two values off the stack, executes an add (add) instruction using these two values and places the result on the stack, then this would correspond to the code generator generating code to add the two values represented by the two tokens on top of the fakestack and replacing these two tokens with a token representing the result of the add operation.

4.1.2.2 Keeping track of register contents

The idea behind keeping track of the contents of registers is to avoid generation of redundant loads and stores. The code generator keeps track of the contents of registers by associating registers with tokens. Thus a
register has a value represented by a *token* associated with that register. The back end writer is responsible for associating registers with tokens and for disassociating registers from tokens whenever code is generated which *changes* the contents of a register.

### 4.1.2.3 Keeping track of condition codes

Many machines have a *test* instruction which take a single operand and set the condition codes according to the value of the operand. A typical use of such an instruction is to follow it by a branch instruction which branches to some label according to the status of the condition codes set by the *test* instruction. For example, consider the following sequence of instructions of the MC68000 microprocessor:

```
tst d0
bgt label1
```

The first instruction compares register $d0$ with zero and sets the condition codes according to the results of the test. The second instruction branches to *label1* if register $d0$ was greater than zero. Some instructions on the MC68000 microprocessor set the condition codes according to the result of the operation performed by that instruction. In this situation there is no need to issue a *test* instruction following such an instruction. The idea behind keeping track of the status of condition codes is to prevent such redundant *test* instructions from being generated. The instruction set of the Eclipse has instructions which perform some operation and then skip the next sequential word if the result of that instruction obeys some specified condition, i.e., instructions exist which
combine a comparison and branch operation into a single instruction. The back end table for the Eclipse contains rules which generate such instructions, and thus there is no need to maintain the status of condition codes in the back end table.

4.2 Alternatives for implementing a back end

Two alternatives were available to choose from in order to implement a back end for EM. The first alternative was to write the back end in some high level language such as C and the second alternative was to use the table driven code generator available under the Amsterdam Compiler Kit. Using the first alternative provides the following advantages:

a) The initial overhead of understanding how the code generator works and learning the syntax and semantics of the back end tables (machine dependent tables) used to drive the code generator would no longer be applicable. This is desirable for a project of limited time.

b) There would no longer be a need to modify parts of the kit related to code generation in order to get the required parts of the kit running on the VAX.

c) Writing in a high level language would provide far greater flexibility than was available within the machine dependent tables.

Thus it can be seen that writing the back end in some high level language is certainly a feasible alternative. However the merits of using the table driven code generator to implement the back end were more attractive.
than those for implementing the back end in some high level language, thus the back end was implemented using the table driven code generator for the following reasons:

a) Many existing EM back ends for different machines have been implemented using the table driven code generator and there are similarities between back ends for different machines. This provides two advantages. First, studying currently existing back end tables is an aid in understanding the syntax and semantics of a back end table. Secondly, maintenance of the new back end by those other than the implementor is made considerably easier since it is likely that those who are to maintain it are already familiar with the syntax and semantics of back end tables and thus there is little overhead required in order to become familiar with the workings of the new back end table compared to the overhead required in becoming familiar with a back end written in some high level language.

b) The code generator is more reliable because it is easier to check that the description given by the back end table results in generation of correct code than it is to check the implicit description embodied in the logic of a code generator program.

c) Implementation of code generation tasks for improving the quality of code generated, such as keeping track of register contents, and simulating the runtime stack of the program being compiled are no longer required since these methods are built in to any code generators created. Instead, only a good understanding of why and how these methods are used is required.
4.3 Description of the machine dependent driving table

The machine dependent driving table contains a description of the target machine and rules specifying which actions the code generator should take given certain sequences of EM Instructions. This section discusses each part of the driving table for the Eclipse. Sufficient details about the syntax and semantics of the driving table are provided in order to understand the discussion on the Eclipse table. Any examples or references to parts of the driving table refer to the driving table of the Eclipse unless otherwise stated. A listing of the driving table can be found in Appendix A. Specific details not covered in this section can be found in [2].

4.3.1 Constant definitions

This section assigns values to three constants. EM_WSIZE is assigned the value 2 since each word on the Eclipse occupies 16 bits (2 bytes = 16 bits). EM_PSIZE is assigned the value 2 since pointers on the Eclipse are 16 bits wide. EM_BSIZE is assigned the value 4 since this is the number of bytes between AB and LB. The value of EM_BSIZE is added to the offset of instructions dealing with parameters during code generation so that the resulting code references both parameters and local variables via the local base pointer.

4.3.2 Register definitions

This part of the table describes the registers available on the Eclipse and assigns properties to specific registers. For convenience, the register
definitions for the Eclipse have been reproduced below.

\[
\begin{align*}
ac0 &= ("r0", 2), ac, aczero. \\
ac1 &= ("r1", 2), ac, acone. \\
ac2 &= ("r2", 2), ac, addrac. \\
lb &= ("r3", 2), localbase.
\end{align*}
\]

The identifier before the '=' sign is a **register identifier** and is the name used to refer to a register throughout the back end table. Any *token* residing on the *fakestack* represents a value. This value could be the contents of a register and thus there must be some means of identifying a *token* on the *fakestack* which represents the contents of a register. The *register identifier* is used for this purpose. The string is the name of the register recognised by the assembler. Following the string is the size of the register in bytes. The identifiers following the closing parentheses are properties the register being defined has. The reason for associating registers with specific properties is that during code generation, registers must be allocated for use; this is done by specifying the register properties themselves. As can be seen from above, the Eclipse has four registers (accumulators), each one being two bytes long. Registers zero, one, and two have the property *ac* which is used to indicate these registers are available for general purpose use, for example they can be used for performing addition, subtraction etc. Registers zero, one, and two also have their own unique property associated with them ie properties *aczero*, *acone*, and *addrac* respectively. The reason for having these additional properties is that some instructions on the Eclipse work with specific registers as *implied operands*. For example the *muls* instruction *always* multiplies the signed contents of register one.
by the signed contents of register two and adds the signed contents of register zero to the result of the multiplication. Thus it is sometimes required to allocate a specific register and this is done by using the unique register property the required register has. The property `addrac` associated with register two is also used to indicate that this register can be used as an address register. In Eclipse terms this means register two is available for use in **accumulator relative addressing mode**.

Register three is reserved for use as the local base pointer (frame pointer). The register property `localbase` is used to indicate this.

### 4.3.3 Token definitions

It was mentioned in section 4.1.2.1 that the code generator simulates the runtime stack of the program being compiled by maintaining a `fakestack` of `tokens`. This part of the table describes all `tokens` that can reside on the `fakestack` during code generation. An example of a token definition for the Eclipse is:

```
acrelative = (REGISTER accum; INT addr;) 2 cost = (0,0) "%[addr],%[accum]"
```

The Identifier before the `=` sign is the name of the token and is used throughout the table to identify `tokens` of that type residing on the `fakestack`. The text between `[` and `]` are attributes of the token and are similar to fields of a record in Pascal. Attributes are of type REGISTER, INT, or STRING. Following this is the size of the token in bytes. Since a `token` on the `fakestack` represents a value, the size is simply the size of the value represented. Since `tokens` normally correspond to addressing modes or registers of the target machine, and `tokens` (which represent
values) may be output during code generation, the cost and output format of the *token* may be specified following the size of the token. The cost of a *token* is given as a pair of numbers. The first number is the number of bytes occupied by the corresponding addressing mode, and the second number is the cost in execution time using that addressing mode. The costs in the Eclipse table have been specified because execution time of instructions on the Eclipse vary depending on the addressing mode of the instructions operand(s). No information was available concerning the cost in machine cycles, different addressing modes had on various instructions. Instead the cost in machine cycles of instructions with a particular addressing mode $a$ and the *additional* cost of instructions using a different addressing mode $b$ was available. Since this was the case the execution time part of the costs of type $a$ tokens were made zero and the execution time part of the costs of type $b$ tokens were made equal to the *additional* machine cycles required if an instruction were to use an addressing mode associated with that particular token. The size part of the costs were made zero for all tokens because the length of instructions on the Eclipse are only dependent on the instructions themselves and not on the addressing mode of an instructions operands. The *tokens* in the Eclipse table and the addressing modes they correspond to are shown below in figure 4.1. An example of an instruction using this addressing mode (the output format of the *token*) is also given.
Token expressions are similar to sets in Pascal and are used to associate a group (set) of tokens with an identifier. This allows the identifier to be used in the code rules and saves specifying the set of tokens each time. The Pascal set operators + (union), - (difference) and * (intersection) may be used.

### 4.3.5 Code rules

This is the largest part of the driving table and contains rules specifying which actions the code generator should take given certain sequences of EM instructions. The syntax of a code rule is:
The code generator scans the code rules looking for a rule which matches the input with the EM pattern and the fakestack with the stack pattern. If there is more than one matching code rule then the cost part of the code rule is examined and the code rule with the cheapest cost is selected and the following operations take place according to the remaining parts of the code rule:

a) The code specified in the code part is output.

b) Any tokens specified by the stack replacement part are pushed onto the fakestack to reflect the result of the operation.

c) Any EM instructions specified by the EM replacement part are pushed back onto the input.

The code rules in the Eclipse table are too numerous to be discussed in detail here. However the following example will help clarify how the code rules specify the actions the code generator should take given certain sequences of EM instructions. Consider the following sequence of EM instructions which adds two to the word in memory location 4:

```
loc 2 ; load the constant 2 onto the stack
lo 4 ; load the word at memory location 4 onto the stack
adi 2 ; pop top 2 elements, add them and leave result on stack
ste 4 ; store result of add into memory location 4
```

and the following code rules selected from the Eclipse table:
loc $1 >= 1 && $1 <= 4 | | | (immediaten, $1) | |
loe | | | (absolute, $1) | |
ad $1 == 2 | acscr immediaten | "adi %[2],%[1]"
este | any | remove (absolute)
remove (absolute)
moves (%[1], (absolute, $1)) | |

| any | allocate (%[1], ac = %[1]) | %[a] | |

Figure 4.2 illustrates the states of the fakestack stack, after code has been generated for each instruction. A description of each state (stage) is as follows:

**Step 1**: The fakestack is initially empty.

**Step 2**: The loc 2 instruction matches the first code rule since the parameter 2 is in the range 1 to 4. The stack pattern specified in this code rule is empty, thus it matches any fakestack configuration. The token (immediaten, 2) (which represents the constant 2) is placed on top of the fakestack to reflect the result of the operation.

**Step 3**: The loe 4 instruction matches the second code rule since the parameter 4 matches anything (there is no constraint on the value of the parameter for this code rule). Again the stack pattern specified in the code rule is empty, thus it matches any
<table>
<thead>
<tr>
<th>Fakestack</th>
<th>Input remaining</th>
<th>Code generated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>loc 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>loe 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>adi 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ste 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>loe 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>adi 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ste 4</td>
<td></td>
</tr>
<tr>
<td>(immediate, 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(absolute, 4)</td>
<td>adi 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ste 4</td>
</tr>
<tr>
<td>(immediate, 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ac0)</td>
<td>adi 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ste 4</td>
</tr>
<tr>
<td>(immediate, 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ac0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ste 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ac0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lda r0,4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lda r0,4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>adi 2,r0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lda r0,4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>adi 2,r0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sta r0,4</td>
</tr>
</tbody>
</table>
fakestack configuration. The token \{absolute, 4\} (which represents the word in memory location 4) is placed on top of the fakestack to reflect the result of the operation.

**Step 4:** The \textit{add 2} instruction matches the EM pattern of the third code rule. However the stack pattern of the code rule does not match the current fakestack configuration, and thus the fakestack must somehow be changed or coerced into a configuration which matches this code rule. In order to achieve this, coercions are provided in the back end table for changing the configuration of the fakestack to some other configuration. Coercions are simply code rules without any EM pattern part. In this case the coercion following all the code rules above is used. The stack pattern of this coercion matches any fakestack configuration (since \textit{any} matches anything) and the code part of this coercion results in code begin generated which loads some register (in this case register r0 has been selected by the code generator) with the value represented by the \textit{token} on top of the fakestack. In this case the value represented by the \textit{token} is the contents of memory location 4 and thus the instruction \textit{ldar0,4} is generated. The EM replacement part of the coercion tells the code generator to push a \textit{token} representing register r0 (which now contains the contents of memory location 4) onto the fakestack. The effect of all this was to coerce the \textit{token} \{absolute, 4\} into the \textit{token} \{ac0\}.

**Step 5:** After the change in the fakestack configuration done in step 4, the third code rule now matches the \textit{add 2} instruction. The code
part of this code rule generates an \texttt{adi} instruction, using the output format of the top two \textit{tokens} originally on top of the \textit{fakestack} to give the instruction \texttt{adi 2,r0}. The EM replacement part of this code rule tells the code generator to push the register \textit{token} representing the contents of register \texttt{r0} onto the \textit{fakestack}. This register \textit{token} represents the result of adding the two values represented by the top two \textit{tokens} originally on top of the \textit{fakestack}.

\textbf{Step 6} : The \texttt{ste 4} instruction matches the fourth code rule and the code part of this code rule generates an instruction to store the contents of register \texttt{r0} (since there is a register \textit{token} on the \textit{fakestack} representing the contents of register \texttt{r0}) into memory location \texttt{4}. Thus the instruction \texttt{sta r0,4} is generated.

\section*{4.4 Quality of code}

The code generator produced for the Eclipse produces good code by simulating the runtime stack of the program being compiled and by maintaining the contents of registers. However the instruction set of the Eclipse and the restrictiveness of the back end table does not lend itself nicely to producing good code from some EM instructions. For example consider the \texttt{blt} instruction which branches to some label if the second word from the top of the stack is less than the first word on top of the stack. The code rule for the \texttt{blt} instruction is as follows:

\begin{verbatim}
blt | ac ac STACK | "sgt %[1],%[2]"
"jmp .+3"
\end{verbatim}
This code rule may produce the following code:

```
sgt r0, r1
jmp .+3
 ejmp label1
```

The first instruction compares registers r0 and r1 and skips over the next sequential word (the second instruction in this case) if the contents of register r0 is greater than the contents of register r1. The second instruction jumps over the third instruction. The third instruction jumps to label label1. An ejmp instruction rather than a jmp instruction is required if the displacement is not in the range -128 to +127. Ideally one would like to have had the following code produced:

```
a) sge rl, r0
   jmp label1
```

or

```
b) sgt r0, rl
   jmp .+3
   ejmp label1
```

depending on the displacement from the current location to that of label1.

However, the restrictiveness of the back end table does not allow a code rule to find out what the displacement (relative to the last instruction generated) of a certain label is and thus only the code sequence (b) can be generated if correct code is to be ensured. The instruction set of the
Eclipse is *unusual* when compared to the instruction set of other 16 bit machines and microprocessors. For example, most machines have instructions which branch to some label according to the status of specified condition codes set by some previous instruction. In comparison, the Eclipse has instructions which skip the next sequential *word* if the result of that instruction obeys a specified condition. Since EM code was designed to allow translation to a wide range of existing machines and the Eclipse has an *unusual* instruction set, the code produced from translating some EM instructions is not good. The example above with code sequence (b) clearly illustrates this point.
5. Mapping the EM machine onto the Eclipse

The EM architecture is designed to be implemented on many machines, however several problems arise when the architecture of the EM machine is mapped onto the architecture of the Eclipse. This section discusses these problems and proposes solutions on how the architecture of the EM machine can be mapped onto the architecture of the Eclipse.

5.1 Pointer and word sizes

EM is a family of intermediate languages where each member of the family has the same instruction set but differ in their pointer and word sizes. Thus each EM implementation has the same instruction set but differ in their memory alignment restrictions and the implicit size assumed in some of the instructions. Since pointers on the Eclipse are 16 bits wide and a machine word on the Eclipse occupies 16 bits, the implementation of EM on the Eclipse has two byte pointers and words.

5.2 Byte addressability

Although the Eclipse has instructions to access different bytes in memory, it is not byte oriented because each byte does not have its own address. This was a problem because EM was designed for machines whose memory consists of 8 bit bytes each with its own address (the EM machine itself is byte oriented). The only way around this problem was to restrict object sizes such that objects occupying one byte were not permitted in the Eclipse implementation. No restrictions had to be placed on object sizes which occupied an odd number of bytes greater than one because
these object sizes are not allowed due to rule (a) in section 2.2. No restrictions had to be placed on instructions which pushed and popped single bytes onto and from the stack because of rule (b) specified in section 2.2.

5.3 Design of the runtime stack

5.3.1 Alternatives for the runtime stack

The stack on the EM machine grows downwards in memory whereas the hardware stack on the Eclipse grows upwards in memory. Two alternatives exist for implementing the runtime stack on the Eclipse. The first alternative is a software stack which grows downwards in memory such as used by the C compiler running DG UNIX and the second alternative is to use the hardware stack on the Eclipse. The following points must be taken into consideration in deciding whether to use a software stack or a hardware stack:

a) The stack on the EM machine grows downwards in memory, so if a software stack were used, there would be a simple and direct mapping of the addresses of objects stored on the stack of the EM machine to those stored on the software stack of the Eclipse. If a hardware stack were used then there would no longer be a simple and direct mapping of the addresses of objects stored on the stack of the EM machine to those stored on the hardware stack of the Eclipse and certain complications would arise. These are discussed later on in the report.

b) The C compiler and library routines running under DG UNIX use a
software stack which grows downwards in memory. Thus if a software stack is to be used it would be a simple matter of ensuring the parameter passing mechanism and the method used to return function results matches up in order to interface to existing library routines. This, for example could be used to allow programs compiled under the Pascal-VU compiler to call standard C library routines via external procedures or functions. If a hardware stack were used then the parameter passing mechanism would be completely different and it would be difficult, if not impossible to interface to existing library routines.

c) It is more difficult to use a software stack compared with using a hardware stack since there are no longer explicit instructions for popping and pushing values onto the stack, saving and restoring the local base and return address on procedure entry and exit, and allocating and deallocating space for local variables on procedure entry and exit.

d) Front ends for EM have their own library routines written in either C or EM code. Thus there is no real need to interface to existing library routines.

For reasons (c) and (d) above, a hardware stack was desirable and this was keenly pursued. Solutions to the problems of using the hardware stack were found and thus the back end was written using the hardware stack implementation. Unfortunately when the back end was near completion and the implementation of library routines for the Pascal-VU compiler were considered, it was found that the heap management routines assumed that
the heap grew upwards in memory which meant that the software stack had to grow downwards in memory in order to make the best available use of free memory. Thus a choice had to be made as to whether the heap management routines were to be rewritten in order to handle a heap which grows downwards in memory, or whether the back end was to be modified in order to handle a software stack. The advantage of modifying the heap management routines was that the back end would not have to be modified in order to handle a software stack. However, the advantages of modifying the back end to handle a software stack are:

a) Once a suitable software stack implementation was found, modifying the back end to handle the software stack would be reasonably straightforward.

b) It is likely that library routines for different front ends assume that the heap grows upwards in memory. Thus once the back end has been modified to handle a software stack, implementing new front ends would not require modification to their library routines in order to handle a heap which grows downwards in memory.

Thus a hardware stack was initially used, but for reasons described above, this was abandoned and a software stack was eventually used. The remainder of this section discusses the EM implementation using the hardware stack and the EM implementation using the software stack.

5.3.2 Using the hardware stack

The use of the hardware stack allows the subroutine call (jsr) and
return (ret) instructions to be used. These instructions reserve certain
registers and memory locations for use as the stack pointer and local base
pointer. The local base pointer (frame pointer) is stored in register r3
and in memory location 418. The stack pointer is stored in memory
location 408. The stack limit is stored in memory location 428, this is
also used as the heap pointer. There is no need for an argument pointer
because the code generator automatically adds the size of the return
status block to the offset of instructions dealing with parameters so that
both parameters and local variables are referenced via the local base
pointer.

5.3.2.1 The procedure calling mechanism

The structure of an activation record using the hardware stack is
shown in figure 5.1. Parameters are pushed onto the stack and deallocated
from the stack by the calling procedure. The information in the return
status block is restored on exit from the procedure using the ret
instruction. Space for local variables and any dynamic local generators is
allocated by the called procedure using the save instruction. The
implementation dependent part of the activation record shown in figure
2.1 is nonexistent because all the necessary information is saved in the
return status block.
5.3.2.2 Local variables and Parameters

The stack of the EM machine and the stack of the Eclipse grow in opposite directions. Thus the offset from the localbase used to access parameters and local variables from the stack of the EM machine is not the same offset used to access the same parameters and local variables from the stack of the Eclipse. Thus some way must be found to map the address of parameters and local variables on the stack of the EM machine to the stack of the Eclipse. There are two cases to consider here, one for objects which occupy one word on the stack and one for objects which occupy multiple words on the stack.
First, consider the case where objects on the stack occupy one word. Consider a procedure call, where the called procedure has three parameters and two local variables. Each parameter and local variable occupy one word (two bytes). The stack of the EM machine would look like this:

<table>
<thead>
<tr>
<th>Offset from LB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+4</td>
<td>Parameter 2</td>
</tr>
<tr>
<td>+2</td>
<td>Parameter 1</td>
</tr>
<tr>
<td>+0</td>
<td>Parameter 0</td>
</tr>
<tr>
<td>-2</td>
<td>Local 1</td>
</tr>
<tr>
<td>-4</td>
<td>Local 2</td>
</tr>
</tbody>
</table>

![Figure 5.2 Stack of the EM machine with one word parameters and local variables.](image)

From figure 5.2 we can see that the parameters and local variables are pushed onto the stack in the order parameter 2, parameter 1, parameter 0, local 1, local 2. If these parameters and local variables were to be pushed onto the stack of the Eclipse in the same order, then the stack would look like this:
It can be seen from figure 5.2 and figure 5.3 that the offset of a local variable or parameter on the EM machine can be mapped onto the Eclipse by negating the offset and then dividing by two. For example parameter 1 has the address LB+2 on the EM machine. This is mapped onto the Eclipse by negating the offset 2, and dividing by two. Doing this gives us LB-1 as the address of parameter 1 on the Eclipse, which can be seen to be correct by looking at figure 5.3. Second, consider the case where objects on the stack occupy more than one word. From section 2.2 we know that the address of a multiple word object in the EM machine is the lowest address of all words it occupies, thus multiple word objects pushed onto the stack of the EM machine must be pushed onto the stack starting at the word with the highest address. To find a suitable mapping, consider the following procedure call where the called procedure has one parameter and one local variable each of which is an array of ten elements with lower bound one and upper bound ten. Each element of the array occupies one word (two bytes).
Figure 5.4 Storage of local arrays on the Em machine and Eclipse
Figure 5.4 shows what the stack of the EM machine would look like and what the stack of the Eclipse would look like if the elements of both arrays were pushed onto the stack in the same order they were pushed onto the stack of the EM machine. By looking at figure 5.4, two points can be noted. First, negating the offset of an array on the stack of the EM machine and dividing by two gives the correct offset for the same array on the stack of the Eclipse. Second, the \(n\)th element of an array on the stack of the Eclipse has a higher address than the \(n+1\)th element of the same array. Thus if the individual words of objects which occupy multiple words are pushed onto the stack of the Eclipse in the same order as they are pushed onto the stack of the EM machine, then negating the offset for an object on the stack of the EM machine and then dividing by two gives the correct offset for the same object on the stack of the Eclipse. Using this mapping creates two potential problems. First, the address of a multiple word object on the stack of the Eclipse is the highest address of all words it occupies, and second, an element of an array stored on the stack of the Eclipse has a higher address than an element of the same array if the subscript is smaller. Since the current implementation of EM on the Eclipse does not support multiple word primitives the first potential problem is of no concern. The second potential problem does affect the current EM implementation because it is possible to have local arrays (either declared locally or passed as value parameters). The solution to the second problem is to subtract the offset of an element of an array from its base address, rather than add it, if the array is local to a procedure.
5.3.3 Using the software stack

The use of a software stack allows some freedom in selecting specific registers and memory locations to be used as the stack pointer and local base pointer. Register r3 and memory location 41B are used as the local base pointer. Memory locations 30B to 37B on the Eclipse are auto-decrementing locations. Whenever these memory locations are addressed indirectly, the contents of the addressed location are decremented by one, and the decremented value is used to continue the addressing chain. Thus, an auto-decrementing location is ideally suited for use as a stack pointer since only one instruction is required to push some value onto the stack. Auto-decrementing location (30B) has been chosen for use as the stack pointer. Memory location 40B is used to store the heap pointer. The value of the heap pointer is also the address of the last usable location on the stack.

5.3.3.1 The procedure calling mechanism

The structure of an activation record using the software stack is shown in figure 5.5. The calling procedure is responsible for pushing any parameters and the value of the local base pointer onto the stack. The local base pointer is saved on the stack before a procedure call because register r3 (which is used to store the local base pointer) is loaded with the return address when a subroutine call instruction (jsr) is executed. The return address is saved by the called procedure. The information in the return status block is restored on exit from the procedure. Space for local variables and any dynamic local generators is allocated by the
called procedure by subtracting the require amount from the stack pointer. The implementation dependent part of the activation record shown in figure 2.1 is nonexistent because all the necessary information is saved in the return status block.

![Activation record using the software stack.](image)

**Figure 5.5** Activation record using the software stack.

### 5.3.3.2 Local variables and parameters

Since the software stack grows in the same direction as the stack of the EM machine, there is a simple mapping of the address of parameters and local variables from the stack of the EM machine to the stack of the Eclipse. This mapping simply involves taking the offsets of the parameters and local variables and dividing them by two. The division by two is necessary because the EM machine is byte oriented and the Eclipse is not.
6. Runtime Library

The code generated from the Eclipse code generator contains calls to several runtime routines. For example, there are calls to routines which handle the `inn`, `set`, and `cms` EM instructions which are used for manipulating `sets` such as those available in Pascal. A listing of these runtime routines can be found in Appendix B, and will not be discussed any further here.

The Pascal-VU compiler generates EM code which contain calls to runtime routines responsible for argument control, file handling, string handling, trap handling, arithmetic routines, heap management, array packing/unpacking, and debugging.

6.1 Implementation of runtime routines

The Pascal-VU compiler has most of its runtime routines written in either C or EM code. For runtime routines written in EM code the obvious method of implementation was to compile them into Eclipse assembly language using the Eclipse code generator, thus this was done. For runtime routines written in C, the choice was not so obvious and two methods of implementation were possible. The first method involved compiling the runtime routines using the C compiler running under DG UNIX. This was by far the easiest method, because once the routines were compiled, the resulting object files were available for use by linking them to the object files of the compiled program. However, this method introduces an additional overhead because the standard library (/lib/libc.a) and runtime startoff (/lib/crt0.o) routines required for programs compiled under the
UNIX C compiler must be included, and for this reason this method was not used for implementing the runtime routines. The second method (the method chosen) involved compiling the runtime routines using the C to EM compiler, and then compiling the resulting EM code using the Eclipse code generator.

6.2 Trap handling

All errors discovered by the Pascal-VU runtime system cause an EM trp instruction to be executed. The trp instruction generates a trap and expects the trap number on top of the stack. The Pascal-VU runtime system allows a user to define their own trap handling routines written in Pascal. In order to do this, the runtime system provides the two routines trp and encaps. The trp routine and the encaps routine are declared in a Pascal program as follows:

```
procedure trap (trapno: integer); extern;

procedure encaps (procedure p;
                 procedure q(trapno: integer)); extern;
```

The encaps routine encapsulates the execution of p with the trap handler q. This means that during the execution of p, any traps which occur, result in q being called with trap number passed as a parameter. To prevent self recursive traps from occurring, eg integer overflow in the integer overflow handling procedure, the execution of any encapsulated routine such as q causes the previous trap handler to be restored. If the user has no previous trap handler specified then the default trap handler is
invoked which produces an error message according to the value of the trap number and terminates program execution. A listing of the default trap handler can be found in Appendix C.

6.3 System calls

Many of the runtime routines make use of the UNIX version 7 system calls. Thus routines had to be written to accept the required parameters and make the appropriate system call. Implementation of these routines was relatively straightforward since there were already similar routines written for the Eclipse which provided a framework from which to work from. System calls were implemented in the assembler routines using the \texttt{sci} mnemonic. Using the \texttt{sci} mnemonic, the system call number and the parameters may be specified in one of two ways. The first method is to follow the \texttt{sci} mnemonic by the system call number, which is then followed by any parameters to the system call. This method is only useful if there are no parameters to the system call or the parameters are the same for each call made. The second method puts a zero following the \texttt{sci} mnemonic. This indicates an indirect call, where the system call number and parameters are specified in the data segment. The address of the data is specified following the zero. This method is useful if the parameters vary for different calls because it is possible to write into the data segment in order to change the parameters. A listing of the routines for system calls can be found in Appendix D.
7. Summary and Conclusions

The unsuitable architecture of the Eclipse and the use of an implementation of the Amsterdam Compiler Kit tailored specifically for PDP 11 machines running UNIX version 7 has made much of this project an uphill battle. Not all of the goals mentioned in the introduction have been totally satisfied. However, in as far as the project has come, the following things have been achieved:

a) A back end table and the required machine dependent C routines have been written for the Eclipse. This opens up other languages which can now be easily implemented on the Eclipse.

b) The implementation of runtime routines required by code generated by the Eclipse code generator.

c) Successful modifications to parts of the Amsterdam Compiler Kit in order to get them running on the VAX.

d) An appreciation of the methods of table driven code generation.

e) The beginnings of the implementation of the runtime library for the Pascal-VU compiler. More specifically much of the implementation dependent routines such as system calls, and runtime error messages have been implemented.

One of the goals mentioned in the introduction, once the back end was written, was to bring up the Pascal-VU compiler running under DG UNIX.
Unfortunately this process is still in its completion stages and has not been achieved. More specifically the following things remain to be done:

a) A few of the system call routines remain to be implemented, this is relatively straightforward.

b) The runtime routines must be compiled into Eclipse assembly language using the Eclipse code generator.

c) Code must be generated from the Pascal-VU compiler currently running on the VAX and fed through the Eclipse code generator. The resulting code must then be linked with the required runtime routines and executed for testing purposes.

d) The EM coded version of the Pascal-VU compiler must be compiled using the Eclipse code generator and finally transferred across to the Eclipse.

After working with the Amsterdam Compiler Kit for several months now, one must ask the question, is EM a good thing? Clearly from the point of view of the $N^p$ problem discussed in the introduction EM is a good thing because it solves the $N^p$ problem. However from the practical point of view one must consider the advantages and disadvantages which the table driven code generator from the Amsterdam Compiler Kit provides. The advantages of table driven code generation have already been mentioned and for convenience have been summarised as follows:
a) The back end becomes easier to maintain by someone other than the implementor, if that person is already familiar with the syntax and semantics of back end tables.

b) The code generator is more reliable.

c) Some of the code generation tasks for improving the quality of code generated are provided automatically.

The disadvantages of working with the table driven code generator are:

a) There is a large overhead required in fully understanding how the table driven code generator works.

b) It lacks the flexibility sometimes required in order to produce good code under difficult situations.

c) Logical errors present in the back end table causes the code generator to issue an error message which specified an assertion (boolean expression) had failed in the machine independent parts of the code generator. It is difficult trying to relate these back to what the code generator was attempting to do according to the semantics of the back end table.

The merits of using the table driven code generator are very strong, however these are set back by the initial overheads, lack of flexibility, and lack of meaningful runtime error messages. It is difficult to draw some conclusion on whether table code generation is a good thing or not.
However it should be said that the authors opinion is that the merits of using table driven code generation make table driven code generation worthwhile, and as research in this field is still active, there is room for improvement such that the only overhead incurred will be that of understanding how the table driven code generator works. Once this has been achieved, then table driven code generation would certainly have to be the choice over more traditional methods of code generation.

Regardless of the goals achieved in this project and of the merits of table driven code generation, it is hoped that this report will serve to be of some assistance to those wishing to implement back ends for EM. In particular it should provide a good reference for those wishing to implement back ends for target machines which do not have an ideal architecture.
8. References


9. Appendices

The appendices can be found in the accompanying folder.