Evaluation of an Interpreted Architecture for Pascal on a Personal Computer

Chad Leland Mitchell

Technical Report No. 83-253

December 1983

The work described herein was supported by IBM under contract 40-81 using facilities supported by NASA under contract NAGW419.

Evaluation of an Interpreted Architecture for Pascal on a Personal Computer

by

Chad Leland Mitchell

Technical Report No. 83-253

December 1983

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

Abstract

This report describes the design and implementation of an interpreter on a personal computer. The architecture interpreted was specially designed for the execution of Pascal and belongs to the class of architectures known as Direct Correspondence Architectures. The evaluation of the interpreter provides information about the suitability of the host for this architecture and identifies features of the architecture which are not adequately supported by the host.

Key Words and Phrases: Pascal, Personal Computers, Interpreters, Execution Architectures, Direct Correspondence Architectures (DCAs)

Table of Contents

List of Figures

List of Tables

1. Introduction

ADEPT (A Directly Executed Pascal Translation) is an architecture designed for the execution of programs written in Pascal and belongs to a class of architectures referred to as Direct Correspondence Architectures (DCAs). It is described in [7]. Scott Wakefield implemented a compiler which translates a subset of Pascal to ADEPT and an interpreter for ADEPT. His compiler runs on a DEC 1020 running the TOPS-20 operating system and it is written in Pascal. His interpreter runs on the Emmy machine which is a special purpose machine designed for the emulation of other machines [5].

The ADEPT interpreter described in this report was implemented on an IBM Personal Computer. It executes the same architecture as the interpreter on the Emmy machine but is much more amenable to alterations in the architecture or in statistics gathering requirements. In addition, it has a user interface which optionally displays memory activity and some built in statistics gathering functions.

Throughout the design and implementation both host and image architectures were evaluated. These evaluations provide insights about the appropriateness of the IBM Personal Computer as a host for DCA architectures and allows identification of some features of ADEPT which should be modified or supported by hardware extensions to support reasonable execution speed of this architecture on the IBM Personal Computer.

2. The Structure of the Architecture

ADEPT instructions are not aligned on word boundaries although branch targets and procedure headers are. Instead, as many instruction syllables as possible are packed into the 32-bit words used for instruction memory. Most types of syllables have the special value of 0 recognized as meaning that the actual syllable is in the next instruction word and this is the way that the instruction counter is incremented as instructions are executed.

Each instruction in ADEPT starts with a format syllable. This specifies the implicit operands, the number of explicit operands, and the type of operator (if any). Following the format syllable are 0 or more operand identifiers as specified by the format. Then comes the operand (if any) which may specify more implicit operands and that more explicit operands follow.

While other syllables have fixed lengths specified by the architecture, operand identifiers vary in length from procedure to procedure. Within a given procedure, operand identifiers are the minimum length necessary to encode all of the unique explicit operands which must be referenced in that procedure. These identifiers index into a local contour or stack frame which contains one location for each unique explicit operand referenced[1]. The stack area above the current stack frame is used for the implicit operands which are managed by formats which imply popping from and/or pushing to the top of stack.

Each operand referenced has a tag which indicates its type and its indirection. At procedure invocation, all operands in the stack frame (also known as a contour) have their flag set to invalid. Then, as they are written, their tag is set appropriately. Parameter, uplevel-reference, and constant operands may be read when the tag is still invalid which causes a reference to the procedure code pre-header to fetch the appropriate value or indirection indicator. Indirection is followed by the interpreter to find the home location for variable storage and retrieval.

Most formats simply indicate operand sources and destination for binary and unary operations. Others indicate the source for the index operand for array operations. Still others indicate the source for the pointer operand for pointer operations. One format indicates a call to a standard

---

[1] Operands of set types require two locations and operands of array types require extra locations to contain the lengths of substructures unless those substructures have length = 1

procedure or function (such as COS or WRITE). In that case the next syllable indicates the call being made and it is followed by operand identifiers as appropriate for that call. A few indicate unconditional flow control such as Goto, Procedure Call, Procedure Return, and Function Return. Conditional flow control is expressed by binary and unary operators such as IF x>y GOTO. Also included are operators which control FOR loops. Branch addresses are expressed as signed program counter relative offsets.

The procedure call format has an operand whose contour location contains the address of the procedure. When a procedure is called a new linkage chapter and contour area are added to the top of the contour stack. The number of parameters is indicated in the procedure header. Parameter identifiers are contained in instruction words which immediately follow the procedure call. The indexed parameter locations are copied from the caller stack frame to the callee stack frame. Contour locations are then allocated for other operands referenced in the new procedure and these locations have their flag set to invalid as previously mentioned. Procedure return and function return pop a stack frame and restore values including such things as identifier length and location of the procedure pre-header. Function return also puts a value into a contour location or leaves it on the top of the stack as indicated by a 1 bit flag which starts the word at the return location.

The compiler specifies 24-bits addresses of 32-bit words for loading, but the top 8 bits are always 00000000. The architecture assumes 32-bit words addressed by up to 32-bits (really 28 because of tags). The contour stack is assumed to reside in a very high speed memory. Instruction memory can be separated from what we will call array memory. The former is never written to during execution. The latter contains arrays, records, and perhaps contour stack overflow.

3. The Structure of the Interpreter

One of the requirements of the project was that the main body of the interpreter be written in the version of Pascal available on the IBM Personal Computer (which will be referred to from here on as PC Pascal). This allows easier modification of the interpreter than if it were all written in assembly language. Changes in the architectural specification can be more easily reflected in the interpreter. It is also easier to add complex data gathering functionality to the interpreter for measuring different aspects of the architecture.

Early in the design it was decided to break the interpreter into three modules using the interface facilities provided by the implementation of Pascal available for the target machine. The original three modules were ADEPTI.PAS which contained the main program, EXECUT.PAS which contained the procedures to actually execute instructions, and MEMORY.PAS which abstracted the architectural notions of memory onto the actual hardware. The functionality of EXECUT.PAS was later divided into two modules, the second being DOOPER.PAS which performs the operations specified in the instructions and also performs explicit operand fetching.

MEMORY.PAS and DOOPER.PAS both have associated assembly language modules MEM.ASM and DOO.ASM. Each PASCAL module except ADEPTI.PAS has an associated interface file (such as MEMORY.INT) which defines the constants, types, variables, procedures, and functions which it provides to the other modules. There is a strict hierarchy of the modules. If we define "x >> y" as meaning that x depends on y then we can say that ADEPT1 >> EXECUT >> DOOPER >> MEMORY.

3.1 MEMORY

The first module written (and the last finished after many revisions) was MEMORY.PAS. The ADEPT compiler assumes a memory of at least 64K 32-bit words or 256K bytes. The first IBM Personal Computer had 128K of RAM and the one used toward the end of the project had 256K. Since the operating system and the interpreter would need some of that space, it was immediately obvious that some kind of memory mapping would be necessary. Furthermore, PC Pascal limits its memory space for data to one 8088 segment (64K bytes) and individual variables such as arrays are limited in length to 32K bytes.

Memory exports the definition of an ADEPT word as

```
AdeptWord =
  RECORD
    Hi,Lo: WORD
  END;
```

The type WORD is predefined in PC Pascal as unsigned 16-bit INTEGER. Two of them form an unsigned 32-bit word which we then use as a word in the ADEPT architecture. Note that bytes are swapped in each WORD, but WORDs are in order within AdeptWords.

The central procedures provided by MEMORY are MemoryWrite and MemoryRead which are defined as

```
PROCEDURE MemoryWrite(Addr: AdeptWord; AValue: Adeptword);
PROCEDURE MemoryRead(Addr: AdeptWord; VAR AValue: Adeptword);
```

They read memory words from and write memory words to the image architecture memory. Note that the addresses are also AdeptWords. The high order WORD of the address is assumed to be 0 and ignored in the current implementation.

MEMORY originally implemented the three kinds of memory as three separate arrays of AdeptWord and with knowledge about the compiler could tell which category applied to a given address. To assist in the debugging of the interpreter, these procedures were expanded to display interactively each memory access and a new procedure was added which would produce a dump of the image memory. Later, a faster memory system was implemented. A procedure defined as

```
PROCEDURE MemoryLoad(Addr: AdeptWord; AValue: Adeptword);
```

was added which would relocate the addresses generated by the compiler so that all of the benchmarks would fit in 16K 32-bit words. It also looks at the values and recognizes address constants transforming them into the smaller address space. MemoryRead and MemoryWrite were moved to MEM.ASM and access the image memory by shifting the low order WORD of the address left by two bits converting it from an AdeptWord address to a Byte address and then the address is used as an index into a separate 64K byte segment which is reserved for the image memory. These new routines do not distinguish between the three kinds of memory. Rather, if memory activity is being counted or displayed, a single flag is checked which will cause Pascal display and statistics gathering routines to be called. Pascal routines are also called if errors are detected in Assembler procedures so that the user interface can be made uniform. The resulting memory system runs fast for normal cases and the overhead for statistics gathering and error handling is only paid if it is used.

The procedures SetUp and Cleanup are called from the main program before interpretation begins and after it ends. Setup allows the user to specify the degree of statistics gathering desired and initializes the screen if memory activity is to be displayed. Cleanup prints any counts or statistics and allows the user to request a dump of image memory. ErrorExit, which is called if any errors are detected, prints an appropriate error message on the screen and in the statistics file, produces a memory dump to the statistics file, and terminates the program after closing the files.

MEMORY is thus the core of the interpreter in another way in that it contains all of the functions concerned with the user interface. Within the statistics gathering procedures, if memory activity is being displayed, procedure CheckWait is called after displaying the activity. This procedure will check for single-step mode, a user set breakpoint being accessed, or a pause request and will only return when execution should proceed.

The following five procedures put a tag as appropriate into the value word and then jump to MemoryWrite returning from there and going on to the statistics and display procedure as selected there:

```
PROCEDURE IntStore(Address,AValue: AdeptWord);
PROCEDURE EquStore(Address,AValue: AdeptWord);
PROCEDURE RealStore(Address: AdeptWord; AValue: REAL);
PROCEDURE FloatWrite(Addr: AdeptWord; AValue: Adeptword);
PROCEDURE TruncWrite(Addr: AdeptWord; AValue: Adeptword);
```

The second parameter in each case comes in without a tag present. FloatWrite also converts the value as a 32-bit integer to a REAL and TruncWrite converts it from a REAL to a 32-bit integer. The adept interpreter uses the PC Pascal real number package, but tags present a problem solved by having RealStore and FloatWrite rearrange the word so that the low order 4 bits of the mantissa are the ones overwritten by the tag. The function

```
FUNCTION GetReal(AValue: AdeptWord): REAL;
```

returns the REAL with the bits returned to their proper position and the low order 4 bits of the mantissa set to 1000 to round out the effects of the loss of precision.

MEMORY also contains the abstraction of the instruction register and instruction counter. Procedure SetNewIR sets a new instruction counter and reloads the instruction register. Procedure SetNextIR increments the instruction counter and reloads the instruction register. In most machines, it would be necessary to post-increment the instruction counter so that branching would

work correctly. The execution cycle in ADEPT is based on fetching the next format rather than fetching the next instruction word so that pre-incrementing will work just as well. The Emmy implementation uses post-incrementing, but this implementation uses pre-incrementing to demonstrate that it will work correctly.

Although it could be accomplished by a few Pascal instructions, incrementing and decrementing of AdeptWords is required throughout the interpreter. Assembler routines AdeptInc and AdeptDec have been provided to meet this need. Two other very common operations are pushing an AdeptWord into memory relative to a pointer and popping something back out again so AdeptPush and AdeptPop are also provided. They access the image memory segment directly and are the only routines other than MemoryRead and MemoryWrite which do so.

3.2 ADEPT1

The ADEPT1 module contains the main program which is defined as

```
PROGRAM AdeptInterpreter(INPUT,OUTPUT,
                         AdeptInput,AdeptOutput,
                         CodeFile,StatFile);
```

The files INPUT and OUTPUT are standard and are used for communication from the keyboard and to the screen. AdeptInput and AdeptOutput are the files which will be associated with standard input and standard output for the interpreted program. The CodeFile contains the object code to be executed. The StatFile will contain all statistics, traces, and dumps generated by an execution.

The main program reads the CodeFile and calls MemoryLoad on each word to load it into the image memory segment. The CodeFile is currently the ASCII file generated by the compiler, but it would be straightforward to load directly from binary images of chapters of the memory. The main program then calls SetUp. After that it calls CallProcedure to simulate a procedure call to the main routine of the program being interpreted. It then calls ExecuteInstructions to begin the execution of the instructions in that main routine. When that call returns, the main program calls Cleanup and then terminates.

3.3 EXECUT

The main loop of execution is within the procedure ExecuteInstructions which is called only once for a given execution. It loops until a termination flag is set, fetching formats and executing them as appropriate. For most formats it sets up the operands and destination and then calls BinaryOp or UnaryOp in DOOPER. For a standard call format it simply calls ExecuteStandardCall in' DOOPER. For operations on structured types, it sets up one or two operands as appropriate and then calls on one of:

```
PROCEDURE DoArray(X: AdeptWord);
PROCEDURE DoPtr(X: AdeptWord);
PROCEDURE DoStruc(X,Y: AdeptWord);
```

These fetch a structured type operation and based on that may fetch additional operands. They then perform the indicated operation which will be an index and move of some kind.

Procedure calls are handled by

```
PROCEDURE CallProcedure(HeaderAddress: AdeptWord);
```

This procedure performs the necessary call operations as required by the architecture. It handles linkage, parameter passing, and the initialization of the invalid contour locations. It also sets up an new instruction counter and instruction register. It then returns to the caller. It is called once by the main program to set up the main routine of the interpreted program. It is then called anytime that ExecuteInstructions encounters a procedure call format. When it returns to ExecuteInstructions the latter continues with its loop executing the instructions of the newly called procedure. It might seem that ExecuteInstructions and CallProcedure should be mutually recursive. That is, it may seem that a call to CallProcedure should set things up and then call ExecuteInstructions. However, since all necessary state is kept in the image machine contour stack, this would be needless recursion and would be a poor model of what a hardware implementation of this architecture would probably do.

The procedure ReturnCommon undoes the linkage and restores the state so that ExecuteInstructions can then continue with the execution of the caller. When procedure return formats are encountered then ReturnCommon is called. Function returns formats are handled by first evaluating the return value, then calling ReturnCommon, and then putting the return value in the proper place.

The locations for all implicit operands are managed by the procedures within EXECUT, but locations for and values of explicit operands are obtained by calls to procedures in DOOPER.

3.4 DOOPER

The module DOOPER contains routines which perform two very distinct functions and it could have easily been divided into two very distinct modules. DOOPER.PAS includes the procedures ExecuteStandardCall, DoGoto, BinaryOp, and UnaryOp which fetch operation syllables and perform the indicated operations. DoGoto fetches the jump address and sets a new instruction counter and instruction register, These are supported by assembler routines in DOO.ASM which perform the necessary 32-bit operations. These are AdeptAdd, AdeptSub, AdeptMul, AdeptDiv, AdeptMod, AdeptComp, and InSet.

The other functionality implemented in DOOPER involves the extraction of syllables from the instruction register and the interpretation of operand identifiers and contour tags. For example the function defined as

```
FUNCTION ExtractTag(VAR AValue: AdeptWord): TagType;
```

returns as its value the tag of AdeptWord and modifies the parameter by sign extending it into the full 32-bits. There are 10 functions whose names begin with "Fetch" such as FetchFormat. These fetch and return a syllable of the specified type. Those which should cause the instruction counter to be incremented on a 0 value do so by calling SetNextIR in MEMORY. There are also four "Get" functions which extract parts of a procedure header and CallByRef which extracts bits from the header one at a time which indicate if a given parameter is called by reference or by value. All of these field extraction functions are in DOO.ASM because they involve operations not efficiently implementable in PC Pascal. Several of them are only used by procedures in EXECUT. They are included in DOO.ASM since they are similar to those needed within DOOPER.PAS and also to avoid the need for an additional 'assembler file for them.

Operand identifiers are interpreted by the procedure defined as

```
PROCEDURE FindValue(ValueWanted,ArrayWanted: BOOLEAN:
                    IDWord: AdeptWord;VAR Address,AValue: AdeptWord;
                    VAR Tag: TagType);
```

This procedure receives the contents of the indexed contour location in IDWord and some flags about its interpretation in the first two parameters. It returns its home address, its value and its

home tag value. It follows indirection to non-locals and fills in invalid locations with constants. It is the most complex and difficult to follow part of the entire interpreter and it is the only place where (tail) recursion is used. It is used in following the pointer chain from home-invalid situations through more home-invalid situations. Most of the time its multitude of parameters and returned values are not needed. Thus, in an attempt to speed up bottlenecks, six new routines were added:

```
PROCEDURE GetAdrVal(VAR Address,AValue: AdeptWord);
PROCEDURE GetAdrWTag(VAR Address: AdeptWord; VAR Tag: TagType);
PROCEDURE GetAdrOnly(VAR Address: AdeptWord);
PROCEDURE GetValOnly(VAR AValue: AdeptWord);
PROCEDURE GetAdrDesc(VAR Address: AdeptWord);
FUNCTION GetElementSize:AdeptWord;
```

These have only the parameters they need and in the case of valid local variables and value parameters they handle the request and return. Otherwise, they call FindValue with the appropriate additional parameters.

4. Evaluation

It was originally thought that there would be two chapters entitled "An Evaluation of the Host" and "An Evaluation of the Interpreter". It did not prove appropriate to evaluate the interpreter in that manner. The majority of the interesting problems solved by the interpreter are those which were caused by the host,

Module		Size (Bytes)
MEMORY	INT	3150
EXECUT	INT	203
DOOPE R	INT	2784
MEMORY	PAS	20589
DOOPER	PAS	25230
EXECUT	PAS	25540
ADEPT1	PAS	3045
MEM	ASM	18983
DOO	ASM	21611
MEMORY	OBJ	16007
DOOPER	OBJ	11856
EXECUT	OBJ	15127
ADEPTI	OBJ	2054
MEM	OBJ	1719
DOO	OBJ	1716
ADEPTI	EXE	131328

Table 4-l: Sizes of ADEPT Interpreter Files

Before discussing specific aspects of the interpreter we can make a few general remarks. Referencing Table 4-l we note that five of the six source files are very nearly the same size showing that the problem was eventually broken up into fairly equal pieces. This holds true for the size of the compiled Pascal code, but the object file sizes show that most of the size of the interpreter is in the handling of all the different cases by the Pascal routines and that while there are many 'assembler routines and a large amount of assembler source, that most of the routines are in fact small and very specific. This is a reflection of the strong attempt to put everything in Pascal except where performance would be significantly degraded or where Pascal simply did not have the capability to implement a function in a reasonable way (such as some of the bit extraction).

In understanding the meaning of the size of the executable (EXE) file we should note that its size includes the wholly empty 64K segment for the image memory and the Pascal runtime library. Less than 48K is actually interpreter code and data.

4.1 32-bit Operations

ADEPT is a 32-bit architecture. It could have been modified to use 16-bit integers for all arithmetic and to use only the low order 16-bits of addresses in all address arithmetic. This would have made at least two of the benchmarks work differently. It would not have been true to the architectural specification.

Although the memory read and write routines insist that addresses to the 32-bit AdeptWords have 0s in the upper 16-bits, no other part of the system assumes so and all other calculations are done with the full 32-bits. The 32-bit operations are provided by a collection of Assembler routines. The most heavily used are the AdeptDec and AdeptInc routines which increment and decrement 32-bit quantities including the instruction counter and stack pointer. They are efficiently implemented and their cost is about that of the procedure call and return plus twice the cost of a 16-bit increment. AdeptAdd and AdeptSub are similar in that regard and all are dominated by the procedure call and return overhead.

The 32-bit increment would be necessary even if the architecture did not call for it since the 16-bit integers provided by PC Pascal are not large enough to contain the statistics which are gathered by the interpreter.

The 32-bit multiply was implemented satisfactorily. It has the procedure call overhead and some additional calculation, but it performs the minimum number of 16-bit multiplications which are necessary for the calculation. Thus, when the upper WORD of (the absolute values of) both operands are 0 it only performs one 16-bit multiply. In this case it is at least within a factor of two in speed relative to the 16-bit multiplies performed by programs compiled in PC Pascal. If one operand has significance in both WORDs then two 16-bit multiplies are performed. If both operands have significance in both WORDs then the result would take more than 32-bits to represent and an error is detected.

The 32-bit divide and mod routines have not been satisfactorily implemented. While it would be possible to perform the division necessary one bit at a time, that would be far too slow for benchmark comparisons since it would make the division totally dominate the execution time for programs which in a normal environment would not be dominated by division. The current implementation of these two routines special cases division by 1 and division by 2 returning the

dividend or the dividend shifted right respectively. Otherwise it uses the 16-bit divide provided by the 8088 which divides a 32-bit dividend by a 16-bit divisor producing a 16-bit quotient and a 16-bit remainder. This routine requires that either the divisor is 1 or 2 or that the divisor, quotient, and remainder all fit in 16-bits. If this condition is not met, a divide-by-O interrupt is generated by the hardware which the PC Pascal runtime system may or may not deal with correctly. The machine may simply branch and hang. The only solutions to this would be to detect the condition before performing the operation or to temporarily replace the divide-by-O interrupt vector with the address of a local handler while performing the divide.

One result of this deficiency relative to divide is that the Kalman benchmark routines which read and wrote reals would not work. They were replaced with calls on the standard functions ReadReal and WriteReal. Another result is that routines which read and write 32-bit integers could not be written since they would perform division by 10 on 32-bit quantities whose quotient would not fit in 16 bits.

One other 32-bit function satisfactorily implemented for convenience was AdeptComp which compares two AdeptWords as 32-bit integers returning -1,0, or 1 to represent the possible results of the comparison.

4.2 Bit Packed Instructions

Perhaps the area in which the host was most poorly suited to this architecture was the extraction of the various bit packed instruction syllables from the 32-bit AdeptWords. On an ideal host for this architecture, such extraction would take one cycle through a dedicated barrel-shifter and could be pipelined with the execution steps implied by the syllables. This was not the case for the interpreter on the IBM Personal computer.

The 8088 has a full set of 8-bit and 16-bit shift and rotate instructions. They can shift or rorate l-bit at a time or a shift amount can be loaded into CL and they can be used to shift or rotate by that amount. A shift or rotate of a register by l-bit takes 2 cycles. A shift or rotate of a register by an amount in CL takes 8 cycles plus 4 times the amount in CL. Thus when shifting by a known amount, it is always faster to have a series of l-bit shifts than loading CL and shifting by the amount in it.

To implement a 32-bit extraction one must perform two 16-bit extractions in parallel, one bit at a time. This would mean repeating the following three instructions N times to extract an N-bit syllable. In this case we have the 32-bit instruction register in BX and DX and the resulting syllable in AX.

```
FETCHI: SHL DX,1      ;Do a one bit left shift of AX,BX,DX
        RCL BX,1
        RCL AX,1
```

Although many alternatives were tried, it was found that enclosing these three statements in a loop controlled by the current identifier length was the fastest somewhat conventional way to extract the identifier syllable. The loop consists of the three preceeding instructions followed by

```
LOOP FETCH1   ;Repeat it IDLEN times
```

The loop instruction takes 17 cycles if we stay in the loop and 5 if we drop through. Thus, one unconventional way to speed things up would be to have a long sequence of the three shift instructions and as each context is entered, modify the code following the correct number of shifts to be a jump to the end of the routine. This would speed the identifier extraction up considerably.

The loop is preceeded by code which loads BX and DX and clears AX. It is followed by code which stores BX and DX back as the new instruction register and it returns the identifier in AX as is the convention for PC Pascal. This code including the call and the return takes an additional 162 cycles. Thus the extraction of an N bit identifier syllable takes 23N + 150 cycles whereas a 16-bit register to register add takes 2 cycles and a memory to register add with index and displacement takes 22 cycles.

The other syllable extraction routines are faster since the fact that we know exactly how many times to execute the three shift instructions means they do not need the loop. Instead the instructions are just repeated in line. With some suggestions from Scott Wakefield, some of these routines were enhanced to do right shifts of 8 bytes by moving bytes around and then shifting from there as needed. An example would be FetchStandardCall in Figure 4-1 which shifts right by 8 bits using MOV and then left by 3 bits using the inverse of the three previously mentioned shift instructions.

Extraction of other fields from 32-bit quantities such as the fields in procedure headers is also expensive, but the only other field that is really extracted frequently is the tag field in countour entries. The function ExtractTag which removes the tag from an AdeptWord and returns it takes 170 cycles to execute.

```
SUBTTL FUNCTION FetchStandardCall: Bits6
;       (* FetchBinaryOp,FetchArrayOp also *)
        PAGE
;Fetch a Standard Call Number from the Current Adept Word
        PUBLIC FETCHSTANDARDCALL
FETCHSTANDARDCALL PROC FAR
        PUBLIC FETCHBINARYOP
FETCHBINARYOP LABEL FAR
        PUBLIC FETCHARRAYOP
FETCHARRAYOP LABEL FAR
;Nothing is saved because there are no parameters or locals
;AX,BX,DX are changed
FETCHC1: MOV BX,CUR ;Get High bits
        CMP BH,0      ;Will it be zero requiring a memory read?,
        JE  FETCHC2   ;YES: Branch
        MOV DX,CUR+2 ;Get Low Bits
        MOV AL,BH     ;Rotate 8 bits
        MOV BH,BL     ;Rotate 8 bits
        MOV BL,DH     ;Rotate 8 bits
        MOV DH,DL     ;Rotate 8 bits
        SUB DL,DL     ;Clear lowest bits
        SHR AL,1      ;Do a one bit right shift of AX,BX,DX
        RCR BX,1
        RCR DX,1
        SHR AL.1      ;Do a one bit right shift of AX,BX,DX
        RCR BX,1
        RCR DX,1
        SHR AL.1      ;Do a one bit right shift of AX,BX,DX
        RCR BX,1
        RCR DX,1
        MOV CUR,BX    ;Save new current AdeptWord
        MOV CUR+2,DX
        RET           ;Result is in AX
FETCHC2: CALL SETNEXTIR ;Get the next instruction word
        JMP FETCHC1   ;Get the StandardCall from it
FETCHSTANDARDCALL ENDP
```

Figure 4-1: Assembler Procedure FetchStandardCall

4.3 User Interface

The interpreter has a visual display of ADEPT memory accesses and can trace and count such accesses. An Adept address breakpoint can be set or the execution can be single stepped across Adept memory accesses. Detected errors produce a file dump of Adept memory. The dump is also optional after normal termination. The display option can be selected just before execution begins. The user can also request a trace of the accesses to a specific kind of memory. If none of these are selected then the user may still optionally count memory accesses.

These features were used for debugging the interpreter. They also provide facilities for

understanding and measuring the ADEPT architecture. They do not provide a meaningful user interface for users of the Pascal programs being interpreted. In the capacity for which they were designed they are very useful.

Early in the project it was realized that good debugging tools would be necessary to be able to get the interpreter running correctly in a reasonable amount of time. The original user interface included the trace options and most of the display option. If display was selected, one could run or single step. Within the statistics gathering procedures, if memory activity is being displayed then the procedure DisplayAdeptWord is called to display the word and address used. This is an assembler routine which dumps an AdeptWord to the screen in hexadecimal using the low-level I/O routines in ROM. Pascal I/O was originally used, but using the low level routines sped up the execution of the entire interpreter by a factor of about four when running with display selected.

Interpretation still runs a factor of about 50 slower when the display option is selected. From this fact we learn that when a system is displaying a detailed information about execution activity, the execution speed will be dominated by the display overhead and the remaining features of the architecutre will have little impact on execution speed.

As longer programs were being executed in the testing of the interpreter, the need for the additional features of the user interface became apparent. If neither display nor trace were selected then accesses were not counted either. In some cases we wanted the counts, but did not want to wait 80 hours for a benchmark to run with display selected. We added an option to just count accesses, but neither trace nor display. The execution which would have take about 80 hours with display selected takes less than three hours with only count selected.

The counts and dumps are also used to compare runs on this interprerter with counts and stack depths calcualted using the interpreter running on the Emmy machine.

The PC Pascal I/O routines were used for most of the reading from AdeptInput and writing to AdeptOutput. This limits such operations which involve integers to 16-bit quantities. The one exception, writing of integers, uses the procedure

```
PROCEDURE AdeptWrite(F: TEXT; AValue: AdeptWord);
```

which was designed so that the 32-bit counts could be written to the statistics file. This procedure

uses the PC Pascal WORD write routine for integers from 0 to 64K and the PC Pascal INTEGER write routine for integers from -32K to -1. In all other cases it writes the 32-bit quantity in hexadecimal. This is not really an acceptable long term solution, but as explained previously it follows directly from the unacceptable solution to the 32-bit divide problem.

4.4 PC Pascal

The PC Pascal compiler suffers from some of the defects common to most compilers. For example, it occasionally overreacts to minor errors by guessing corrections which in turn generate hundreds of errors. Some of its guesses at corrections could be much better without adding any complexity to the compiler. The compiler also runs slowly, especially with floppy disk drives. The link& is also very slow which means that some of the advantages of separate compilation are lost.

Its extensions to Pascal occasionally leave something to be desired. For example, the module interface requires that all of the constant names in an enumerated type be exported explicitly. It is also manditory that the dependance graph between interfaces be a tree since interfaces are referenced by textual inclusion. Perhaps the biggest problem with the modularity extensions is that if an error is detected in an interface the message often describes the type of error without specifying where it occured or which names are involved. For example, an error message might say that something was listed as exported but never declared and the error message would not indicate which of the fifty names in the export list had that problem.

Some of its problems are more significant. If it is terminated with Ctrl-Break while executing (such as after the user recognizes that the pages and pages of error messages appearing all came from a single syntax error) then it leaves files hanging and neither closes nor deletes them. This means that floppies fill up quickly with unusable sectors. A disk must be reorganized with CHKDSK to reclain these lost sectors. Worse yet, this same problem also occurs if the compiler is called with an incorrect file name (referring to a nonexistant file) or if the second pass is called after the first pass fails. This is just one of the ways that it does not work well with the operating system. It works especially poorly with DOS 2.0 not knowing about path names etc. It also does not know about standard input and output indirection and while the latter works, the former does not since it refuses to believe that it hit end-of-file on standard input.

One problem which consumed many days on this project was the poor specification of how the

```
TITLE ASSEMBLER PROCEDURES FOR MEMORY MODULE (Mem.ASM)
;
SUBTTL  ADEPTMEM  SEGMENT
        PAGE
ADEPTMEM SEGMENT 'CODE'  ;Appears like a code segment to the linker
        PUBLIC ADEPTMEMORY
ADEPTMEMORY DD 16383 DUP(0); Adept Memory
ADEPTMEM ENDS
SUBTTL  DATA  SEGMENT
        PAGE
DGROUP  GROUP  DATA
DATA    SEGMENT PUBLIC 'DATA'
        EXTRN DESTCS:BYTE
        EXTRN DOCOUNT:BYTE
NEGATERESULT  DB  ?       ;This is a flag used by FloatWrite,TruncWrite
DATA    ENDS
SUBTTL  START  OF  CODE  SEGMENT
;External Procedures Referenced
        EXTRN  ADEPTOVERFLOW:FAR
        EXTRN  TRUNCOVERFLOW:FAR
        EXTRN  INVALIDMEMORYREAD:FAR
        EXTRN  INVALIDMEMORYWRITE:FAR
        EXTRN  MEMORYWRITEDISPLAY:FAR
        EXTRN  MEMORYREADDISPLAY:FAR
        EXTRN  INVALIDADEPTPOP:FAR
        EXTRN  INVALIDADEPTPUSH:FAR
        EXTRN  ADEPTPOPDISPLAY:FAR
        EXTRN  ADEPTPUSHDISPLAY:FAR

MEMASM  SEGMENT  PARA  'CODE'
        ASSUME  CS:MEMASM,DS:DGROUP
;
ENDADEPTMEM DD 0 ;Since the linker/assembler prevent exactly 64K in a
;                  segment, here are the last four bytes of the 64K
;                  Adept Memory segment (just to keep it from
;                  overwriting the first four bytes of DUMPWORD)
;
```

Figure 4-2: First Part of MEM.ASM

compiler interfaces with the linker. Figuring out how to generate assembler routines which are called by Pascal routines and call Pascal routines was straightforward; however, the need to share global variables between assembler and Pascal routines created a problem. Putting in references to the DATA segment was not enough because while the linker maps would show them as sharing a segment, the runtime system would rearrange things so that they no longer did share the same segment. The selection of the correct segment options so that the linker would set things up so that the runtime system would map the shared data segments onto each other was a matter of extensive trial and error. In Figure 4-2 we see the start of MEM.ASM which shows how the ADEPT memory segment is set up and how we interface with the Pascal globals and procedures referenced. The

GROUP and SEGMENT pseudo operations for DATA are both necessary to have the external data references work right.

4.5 Speed

Once the main body of the interpreter was completed, its execution speed was measured and the measurements repeated after stages of modification and enhancement. At first a small routine called Dummy.Pas was used which executed a small loop 30000 times. Later, the benchmarks used by Scott Wakefield to test the interpreter on the Emmy machine were used. They are described in [7]. In Table 4-2 we see a summary of these measurements along with descriptions of the major changes in the interpreter which might have contributed to a difference in execution speed.

We note (as was previously mentioned) that changing to direct console I/O sped up the execution with display selected by a factor of about 4 and that the display option still runs about 50 times slower than without it. We also note that speeding up various parts of the interpreter produced small improvements, but that there did not seem to be a specific bottleneck in non-display execution. The interpreter seems to be close to its optimal speed with the current constraints.

The four benchmarks used in later testing were also compiled in PC Pascal. Start and Finish routines which display start and finish times were included in all three so that times could be compared excluding load time. Sort and FFT use a random number generator which while generating numbers in the range 0 to 32K, has partial results involving the Seed which need 32-bit arithmetic. To keep the algorithms identical, the Seed in each case was changed to be of type AdeptWord and the Adept Multiply, Divide, and MOD routines were linked in and used only in the random number calculations. This means that the benchmarks can be used to compare execution speeds between code compiled in PC Pascal and in ADEPT.

Table 4-3 summarizes the results of the execution of the benchmarks compiled by the PC Pascal compiler. The tests were performed with and without the PC Pascal DEBUG option on. With it on, bounds checking and various other checks are performed such as making sure that the stack area does not overflow on procedure call. Since ADEPT does not perform the value bounds checking, but does check bounds on memory accesses and also does some checking of types in array referencing and some checking at procedure invocation it seems fairer to have the DEBUG option on for these comparisons, but ADEPT as implemented is really somewhere between the two.

All times are in seconds.
Numbers marked by * were measured for 1/100th
normal execution and then estimated.

Date	Routine (or Change)	Execution Times with Options: No Count	Count/No Disp	Display
23-Sep-83	Dummy	283		56600*
23-Sep-83	(ExtractTag & IntStore Moved to MEM.ASM)			
23-Sep-83	Dummy	264		50300*
24-Sep-83	(Simplify Display Writing, speed up SetNextIR)			
24-Sep-83	Dummy	230	248	38700*
24-Sep-83	(Change to direct console I/O for Display)			
24-Sep-83	Dummy			8792(9100*)
26-Sep-83	(Combine memories into a single Pascal array, make counts 32-bits)			
26-Sep-83	Dummy	221	268	9100*
27-Sep-83	(Move to IBM-XT and DOS 2.0)			
27-Sep-83	Dummy	222		9000*
27-Sep-83	(Add overflow check in IntStore, speed up GetValue)			
27-Sep-83	Dummy	213		9600*
30-Sep-83	(Improvements to fetching of syllables)			
30-Sep-83	Dummy	208		
30-Sep-83	Sortp	1176	1442	53900"
30-Sep-83	FFT	289	360	
06-Oct-83	(Memory Read/Write moved to MEM.ASM, 64K ADEPT memory segment)			
06-Oct-83	Sortp	1146		
06-Oct-83	FFT	287	390	
06-Oct-83	Puzzle	6181		
08-Oct-83	(Speed up AdeptDec, Only Pass tags when necessary)			
08-Oct-83	FFT	281	383	
08-Oct-83	(Shorten Int/Equ/Real Stores, Split ASM routines into two modules)			
08-Oct-83	FFT	279	382	
08-Oct-83	(Float, Trunc, Pop, & Push moved to MEM.ASM)			
08-Oct-83	Sortp	1112		
08-Oct-83	FFT	262	376	
10-Oct-83	Puzzle	5984	8438	
21-Oct-83	Kalman	139		

Table 4-2: Times for ADEPT Executions

Date	Routine	Time	Adept Ratio	Debug?
01-Oct-83	Sortp	92	12.1	ON
05-Oct-83	FFT	25	10.5	ON
05-Oct-83	Puzzle	421	14.2	ON
21-Oct-83	Kalman	21	6.6	ON
12-Nov-83	Sortp	12	92.7	OFF
12-Nov-83	FFT	12	21.8	OFF
12-Nov-83	Puzzle	48	124.7	OFF
12-Nov-83	Kalman	15	9.3	OFF

Table 4-3: Times for PC Pascal

There seem to be four main reasons why the adept interpreter runs slowly on this machine. The

first is that there is more than an order of magnitude difference in access speed between operands in registers and operands in memory. The second is that nothing is kept in data registers across procedure boundaries except unstructured return values from functions. This is necessary since the interpreter is partially written in Pascal, but putting it all in assembler would not help here. There are too few registers to keep values in some of them across procedure calls anyway. This is a real deficiency of the 8086/8088 architecture. These first two reasons amplify each other's effect on execution speed. They would make almost any interpreter very slow on this host.

The third reason is that there is a very high branch penalty and especially a high cost for procedure calls. This causes special problems in the interpretation of an architecture which has so many orthogonal instruction syllables. We cannot have one case statement which branches on the cross product of all possible syllable combinations. Not only does one syllable tell us how to fetch the next ones, but the cross product would be too large anyway. The result is that we must have different procedures for each type of syllable which extracts that syllable and handles the cases of its values. Thus, the execution of an instruction takes a great many cycles, most of which are branch penalty cycles.

The fourth reason is that the design of ADEPT made the strong assumption that the contour stack would be kept in high speed memory. While memory traffic to other kinds of memory is reduced in DCAs by putting locals in the contour, the total of memory traffic including that to the contour is significantly increased. The IBM Personal Computer does not provide a higher speed memory in which the contour stack can reside and thus this essential assumption of DCAs is violated.

5. Future Work

There are four different kinds of future work which are possible. First, one might continue the comparison of the ADEPT interpreter with other options by comparing the benchmarks to similar routines coded in an interpreted language such as BASIC on the IBM Personal Computer. As further test points, they may also be compared with the same algorithms compiled to P-Code and to Modula-II.

An alternative would be to explore the contribution to execution time of the need for 32-bit arithmetic by creating a version of the interpreter which would only support 16-bit arithmetic operations redefining the AdeptWord as containing a INTEGER in the low order 16-bits and putting only the tag in the upper WORD. Neither of these first two options is being pursued at the moment, but they may be taken up at a later date.

The third option, being worked on by a research group at the IBM Research Center at Yorktown Heights, is the creation of a revised definition of ADEPT which would take into account recent research in this area including the results of this project. The new architecture or architectures might also have some hardware support to get around some of the deficiencies detected in the host.

The final option, currently being pursued by the author of this paper, is the use of this interpreter as a tool for measuring in great detail various aspects of the ADEPT architecture and of DCA architectures in general.

Appendix A. ADEPTI Users' Manual

The ADEPT interpreter expects four parameters all of which are files. The first two are respectively input and output files for the program under emulation. If one or both are not needed, they must be specified anyway. Furthermore, a RESET of the AdeptInput file and a REWRITE of the AdeptOutput file are performed even if the files are unused. The third file is the compiled program to be executed. The annotated format is fed in through this file. The fourth file is a statistics file which is produced by the execution. It minimally contains the start and stop time of the execution and may contain much more.

Once the ADEPT interpreter has read in the program and has done the necessary address conversions, and before the execution begins, the interpreter asks a series of questions to which Y or N (or y or n) are expected as answers. The first three questions concern the ability of the interpreter to produce an address trace in the statistics file. Traces for the three different kinds of addresses are specified separately so that only one or two of the types may be traced. Selecting these options can produce very large statistics files.

The next question concerns display mode. If display is selected, a visual display of memory activity is provided as explained later. If display or any of the traces are selected, memory accesses are counted as well since the overhead of counting these is little on top of the display or trace overhead. Thus, execution would commence after the fourth question. If no traces are kept and memory accesses are not displayed then counting can add significant overhead (up to 50%) and a fifth question permits disabling of the counting function.

If the display option is selected, then all memory accesses and instruction fetches will be displayed on the display. The user will have several options. The program can be single stepped, where a step constitutes a single memory access. It can be allowed to run and then paused or switched back to single stepping as desired. A single breakpoint address can be set. On accessing that address, the system will switch to single step. The system can also be told to run fast. While this option is active, the interpreter executes without displaying memory activity. It does still check for user input requesting a change of mode. Although this "F" option will not run as fast as if the display option were not selected at all, it runs much faster than if the accesses were being displayed. All of these display options are intended to help in the debugging of the interpreter and of compilers and related tools rather than in the debugging of the programs being interpreted.

References

[1] *Macro Assembler*
International Business Machines Corporation, 1981.

[2] *Pascal Compiler by Microsoft*
International Business Machines Corporation, 1981.

[3] *Guide to Operations*
International Business Machines Corporation, 1982.

[4] *Disk Operating System by Microsoft*
International Business Machines Corporation, 1983.

[5] Neuhauser, C. J.
Emmy System Processor: Principles of Operation.
Technical Note 114, Computer Systems Laboratory, Stanford University, Stanford, California 94305, May, 1977.

[6] Scanlon, L. J.
IBM PC Assembly Language: A Guide for Programmers.
Robert J. Brady Co., 1983.

[7] Wakefield, Scott.
Studies in Execution Architectures.
PhD thesis, Stanford University, December, 1982.