Reducing execution parameters through correspondence in computer architecture

by Scott P. Wakefield
Michael J. Flynn

The purpose of this study is to develop and extend techniques to provide architectural correspondence between high-level language objects and hardware resources so as to minimize execution time parameters (memory traffic, program size, etc.). A resulting computer instruction set called *Adept* has been emulated, and a compiler has been developed with it as the target language. Although the study was restricted to Pascal, the resulting data are generally applicable to the execution time environment of any procedure-based language. Data indicate that significant bandwidth reductions are possible compared to System/370, VAX, P-code, etc. Specifically, the average improvement ratios realized were instruction bandwidth reduction: 3.46; data read reduction (in bytes): 5.42; data write reduction (in bytes): 14.72.

©**Copyright** 1987 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

1. Introduction

The purpose of this paper is to provide data on the behavior of an idealized computer architecture. The architecture itself, called *Adept*, is in correspondence with the abstract machine defined by the constructs of a procedural language such as Pascal. (The name Adept is derived from "a directly executed Pascal translation.")

As used here, the architecture of a computer is simply its instruction set. The computer consists of its instruction set, its storage, over which the instructions are defined, and an interpretive mechanism (the *host*), which causes the state transitions specified by the instruction set to be carried out.

The question of what makes a good instruction set involves many factors, including but not limited to the following:

- Implementation time. The time to design a host for an instruction set is often critical. Simpler instruction sets require less design effort.
- Technology. If the computer is to be realized on one chip, the number of devices available is crucial. Small and simple instruction sets can be supported by fewer devices better than instruction sets with larger vocabularies.
- Compatibility. Compatibility with an existing instruction set may dominate all other considerations due to market factors.

- Code density. The efficient use of memory can be an important design attribute.

Customized architectures for particular languages or applications can be developed with superior attributes [1].

Two different starting points might be taken to understand and evaluate an instruction set:

1. A top-down view, where one derives the architecture in terms of the primitives of the language or application.
2. A bottom-up approach, where one derives the instruction set based upon technological and implementation considerations.

In the first approach, one refines the architecture based upon implementation and technological considerations—for example, deleting infrequently used primitives and speeding up others [2, 3]. In the second approach, one produces a good host design and uses software, either compiler technology or interpreters, to match the host with the environment. The more one knows about the environment, the better one can support it in hardware; thus floating-point operations might be used for scientific applications or register windows in the RISC machine [4] to support the C language environment. In the final analysis, both of these viewpoints must be considered.

The purpose of this and our earlier, related work has been to study the relationship between source language and architecture. Implementation considerations being equal, an architecture that minimizes the space–time product of the use of memory for a program will be the best. If we view the processor simply as an instruction set and its implementation, then from the memory's point of view, it is better to have

1. Small program size.
2. A minimum number of instructions to execute a program (dynamic instruction count).
3. A minimum amount of memory traffic (bandwidth).

We attempt to minimize the product of the memory space occupied by a program and the program execution time. The execution time is dependent on factors other than those listed above, especially the implementation (the execution time per instruction required by the processor), which is only partially determined by the dynamic execution count and the memory traffic requirements. The dynamic instruction count is a primary measure of execution time. Instruction set implementations which provide close to one cycle per instruction are possible using familiar test programs [5], though not all architectures are equally efficient in their use of a given processor technology. We are not primarily interested in measuring this implementation efficiency, but rather in understanding the correspondence or representational efficiency of the architecture with respect to the source program. Data characterizing the behavior of such architectures can be valuable in making later trade-offs in implementation.

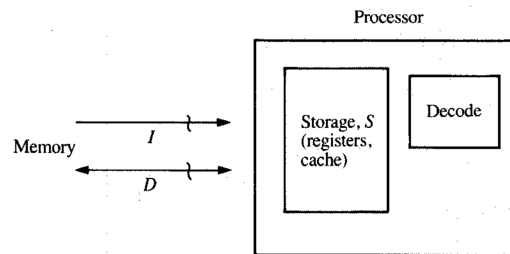


Figure 1

Basic trade-off among memory bandwidth (I and D), internal processor storage (S), and processor decode requirements.

In our earlier work [6–8] we developed several techniques for instruction set design to enable concise mapping of high-level language statements into executable form. In the initial study we selected Fortran as the high-level language because of its relative simplicity. The resulting architecture, *DELtran* [9], showed significant improvement in both conciseness (static measures of space requirements) and execution parameters, such as dynamic instruction counts and memory referencing activity. This earlier work was limited in several important respects. The source language itself was static, and several interesting techniques for object referencing were left unevaluated. Secondly, the variety of source programs evaluated was limited since full compilation facilities were not developed—much of the evaluation of sample programs relied on hand coding. In the present work a full compilation facility is available to translate Pascal source programs into Adept instructions. The evaluation of Adept is also materially aided by the availability of our Emulation Laboratory with emulators for a variety of alternate instruction sets, such as IBM System/370, PDP-11, HP 1000, and P-code. While the precise details of the design decisions of Adept [10] were determined by Pascal, the approach used can be applied to a variety of procedure-based languages, and the data presented are generally applicable.

2. The DCA approach to instruction set design

In order to achieve architectural effectiveness as measured by the basic parameters of program size, instruction count, and memory bandwidth, one can vary the complexity of encoding instructions and fields within instructions, as well as making more extensive use of processor storage (registers). The trade-off is among the following parameters (see **Figure 1**):

1. Overall execution effectiveness vs. instruction decode complexity.
2. Memory traffic to fetch instructions (instruction bandwidth).
3. Memory traffic to fetch and store data (data bandwidth).

The first issue seems simplest, but is actually related to the other two. The fewer instructions executed, the better, but minimizing the number of instructions may increase decoding complexity by requiring the decoder to interpret multiple formats. Alternatively, we can retain minimal instruction count *and* simple instruction format decode (e.g., a three-address format), but at the expense of instruction bandwidth (larger instructions). The basic issue in architecture is trading off the various parameters to achieve optimal cost performance. If, for example, a processor executes data transformations slowly (relative to memory bandwidth), it is of little interest to minimize instruction or data bandwidth or even overhead instructions (load, store, etc.); execution time is dominated by ALU operation.

In encoding identifiers there is a trade-off between coding efficiency (the space required to represent an object or operand name) and decoding complexity. Specifically, there are three levels of decoding complexity:

1. Bit-Dependent Code (BDC). The size of the encoded object depends on the contents of the object. A Huffman code is an example of a BDC.
2. Bit-Variable Code (BVC). The size of the object is known (as in a *block code*) to the decoder, but its starting point with respect to word boundaries is variable (i.e., does not always start at the same bit within a word).
3. Fixed-Block Codes (FBC). The size *and* starting point are known to the decoder.

Pure FBCs are used in only the simplest computer architectures. (Not even all RISCs use FBCs. Instructions in Stanford's MIPS computer, for example, are encoded in 12, 20, or 32 bits, depending on the instruction class, the number of operands, and the size of the immediate field [11].) Most common is the use of fixed-size identifiers with a limited number of starting points. The use of BDC or Huffman-type codes provides the maximum representational efficiency, but also maximum decoder complexity. Moreover, the key to making BDCs work is ensuring that the most often used messages are the shortest. Thus, in order to achieve optimal spatial efficiency, the frequency distribution of objects must be determined before encoding can be performed.

While there have been many approaches to language-oriented instruction set design, the *direct correspondence architecture*[1] (DCA) approach is distinguished in at least two ways from these earlier efforts:

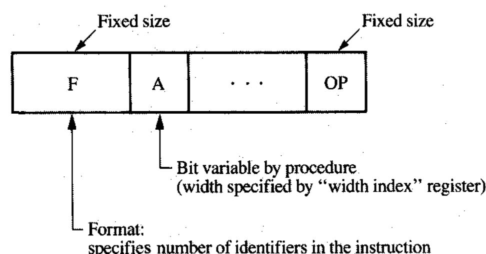


Figure 2

The use-ordered DCA instruction.

1. It uses a robust set of formats to eliminate overhead instructions.
2. It uses the contour model (originally developed by J. D. Johnson [12]) as an implementation vehicle for object accessing.

In Adept, we have attempted to minimize the basic parameter values by making assumptions about the trade-offs available among the various parameters illustrated in Figure 1. Specifically, we make the following assumptions in Adept:

1. That the execution of semantic operations (as distinguished from overhead operations) is fast. Thus, execution time will be dominated by the number of overhead instructions and by memory referencing activity. Minimizing overhead instructions will proportionately improve overall execution time.
2. To eliminate overhead instructions and minimize the total dynamic instruction count, a robust set of formats is used in Adept, somewhat increasing the complexity of instruction decode.
3. Instruction bandwidth is minimized by using bit-variable encoding of fields or objects within an instruction. This is a strategic middle ground between fixed block encoding (no variability between starting point and size) and bit-dependent encoding. BDC codes require more complex decoding than FBC codes.
4. Data bandwidth is minimized through judicious use of processor storage. Assignment of objects to contour storage (an idealized register set) has been done on the basis of expected usage by the program as defined by the

[1] Some previous references to DCA methodology called it the *directly executed language* (DEL) approach, but we found many readers mistakenly associating the name with the investigations of Yaohan Chu and others involving run-time source code parsing.

semantics associated with that object in the high-level language. For example, local scalars are assigned to contour storage, whereas local structures (arrays, etc.) are retained in main memory. No *a priori* limit is placed on the size of contour storage in Adept.

5. Opcode and object distributional data are not known *a priori*—in fact, it is these data that are to be collected. Bit-variable codes are spatially most efficient in the absence of frequency data; this forms the basis for DCA encoding. Clearly, the choice between bit-variable and fixed-block encoding is a technology/implementation decision.

A DCA instruction (**Figure 2**) consists of a format syllable, from zero to three object identifiers, and an operation syllable. The format and operation syllables are fixed in size, in most cases five bits each. The number of object identifiers is determined by the format syllable, and varies between zero and three syllables. The width of an object identifier syllable is determined by the number of unique objects in a particular scope of definition, as determined by the high-level language. This width is $\lceil \log_2 \rceil$ of the number of unique objects in a particular scope, creating a bit-variable code (BVC). For example, in Pascal, a procedure with 21 unique objects would use five bits to encode syllables to represent operands.

• *Formats*

It is well known that statements in high-level languages can be represented as trees or, more accurately, as directed acyclic graphs (DAGs). In a DAG, an interior node consists of a high-level language operation. Instruction sets that use a single format cannot concisely represent all possible relationships described by a node. The classic case of

$a + b \rightarrow c$

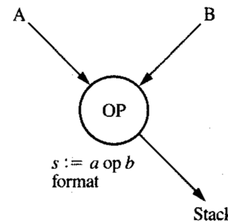
is represented in a stack architecture as

push a
push b
add
pop c

In a simple register-oriented architecture, the *pushes* are replaced by register loads and the *pop* is replaced by a *store*. In the case of a three-address architecture, the above can be captured concisely in one instruction; however, for the evaluation of complex expressions or statements with redundant identifiers, such as $a + a \rightarrow b$, significant redundancy is introduced. In the DCA approach, each DAG node is associated with a single instruction.

A stack is used for communicating results between internal nodes of a statement. Thus, the statement

$a \times b + c \rightarrow d$



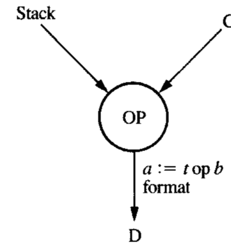


Figure 3

A two-instruction representation of the statement A × B + C → D.

illustrated in **Figure 3** is represented as two instructions:

F19 $a\ b\ \times$

F12 $c\ d\ +$

F19 is the instruction format in **Table 1** which takes two

Table 1 Transformationally complete formats. (a, b, and c are any explicit operand identifiers, t is the top of the evaluation stack, s is the next element in the stack, and u is the previous element.)

Format	*Transformation*	*Number of explicit operand identifiers*
F1	$a := b$	2
F2	$a :=$ op(a)	1
F3	$a :=$ op(b)	2
F4	$a :=$ op(t)	1
F5	$s :=$ op(a)	1
F6	$t :=$ op(t)	0
F7	$a := a$ op a	1
F8	$a := b$ op a	2
F9	$a := t$ op a	1
F10	$a := a$ op b	2
F11	$a := b$ op b	2
F12	$a := t$ op b	2
F13	$a := b$ op c	3
F14	$a := a$ op t	1
F15	$a := b$ op t	2
F16	$a := u$ op t	1
F17	$s := a$ op a	1
F18	$t := t$ op a	1
F19	$s := a$ op b	2
F20	$t := a$ op t	1
F21	$u := u$ op t	0

explicit identifiers, a and b, and pushes the result onto a stack, labeled s (the "$s := a$ op b" format). The push is not accomplished by a separate instruction, but rather implied in the format itself. Similarly, F12 takes the stack (performing a pop) and c as source operands, and assigns the result to the second explicit identifier, in this case d (the "$a := t$ op b" format). Thus, in compilation, code generation consists of selecting the appropriate format which binds variables to each operation (DAG node). The format implicitly specifies the number of variable identifiers required by the operation. Naturally, many variations are possible, depending on the level of redundancy that can be tolerated in the resulting instruction set. As few as four formats provide an interesting basis for an instruction set reasonably free of redundancy [7]. In the DCA experiments to date, we have used transformationally complete format sets—sets without redundancy (Table 1). In Adept, 35 formats are used, including branches and array index operations as formats.

• *Using contours for operand identification*

In addition to representing the node of the parse tree, there must be a method for specifying the operands that are associated with the node. Essentially there are two methods for specifying operands. One is to embed the needed information within the instructions, and the other is to gather all such information into descriptor tables to which the instructions can refer. Most architectures use both of these methods. For example, if a displacement address requires 12 or fewer bits, it is placed in an immediate field of an instruction for the IBM System/370. If it is larger than 12 bits, it is placed in a full 32-bit word and is loaded into a register when it is needed during execution.

The scope of definition for the names of operands in a high-level language program can be described as a contour [12]. In such a description a procedure or function is declared within a level of scoping and can be represented as a region of higher altitude within a plane. An entire program with procedures and functions declared within the main program and within each other can be represented as a kind of contour map. By the scoping rules of a block-structured language such as Pascal, statements in the body of any one procedure or function can refer only to operands at the local level and at levels that can be reached from the local level by descending. Descriptions of operands can be collected into tables. There can be a different table for each procedure and function and one for the main program. If these tables contain an entry for every operand accessed from each block, the tables completely define the contour model for the entire program. The table of descriptors for each block is called the *contour* for that block [9].

If descriptors are collected into a contour, then to make reference to an operand through its descriptor, an instruction needs merely an index into the contour; each identifier field requires $\lceil \log_2 n \rceil$ bits, where n is the number of descriptors in the contour, although fixed-block encodings may also be used.

The decision as to what range of statements a contour should service is not clear-cut. At one extreme, a single contour could contain the descriptors for all the operands in an entire program. This would result in the smallest total amount of memory devoted to storing operand descriptors, since there would be no duplication of descriptors in multiple contours. But then, every reference in the instructions to descriptors in the single contour would have to specify one out of all the descriptors for the program, so the identifier field would require more bits of memory for this scheme than for any other. At the other extreme, a different contour could be associated with each statement, containing descriptors only for operands to which that statement refers. While the identifier fields under this scheme are likely to be small, the large number of contours use a great deal of memory and require the processor to switch between contours frequently during execution.

A simple compromise was suggested by Johnson [12], in which a different contour is associated with each procedure and function and with the main program. If a contour contains descriptors only for operands to which its corresponding statements actually refer (the technique chosen for Adept), switching between contours during execution becomes part of the semantics of call and return instructions. What most distinguishes this scheme from others, though, is that it takes advantage of the locality of the scopes of operands. Structured programming practices call for procedures that encapsulate various phases of the work to be done. If a program is well written, then variables that are used in only one phase are declared local to a procedure, and descriptors for them need appear in only one contour. Similarly, some global variables may be heavily used in some phases and not at all in others. These need only appear in contours for procedures that use them. The use of such contour tables reduces the total memory size without drastically increasing the amount of switching between contours during execution.

The goal of contour design is to minimize both instruction and data memory traffic without requiring compiler-managed register-assignment optimization. In the processor the register set is replaced with the *contour register set*, which holds the collection of all currently encapsulated single contours. The contour register set is similar to multiple-windowed register sets, but the size of a single contour is completely variable. Its starting point is defined by a pointer (environmental pointer, or EP in Johnson's nomenclature [12]), which is defined on procedure entry.

Keeping individual contours small keeps identifier fields short and minimizes required instruction bandwidth. But more importantly, if contours are small, it becomes economical to store them in fast memory devoted to that purpose. Contour register sets then have the advantages of

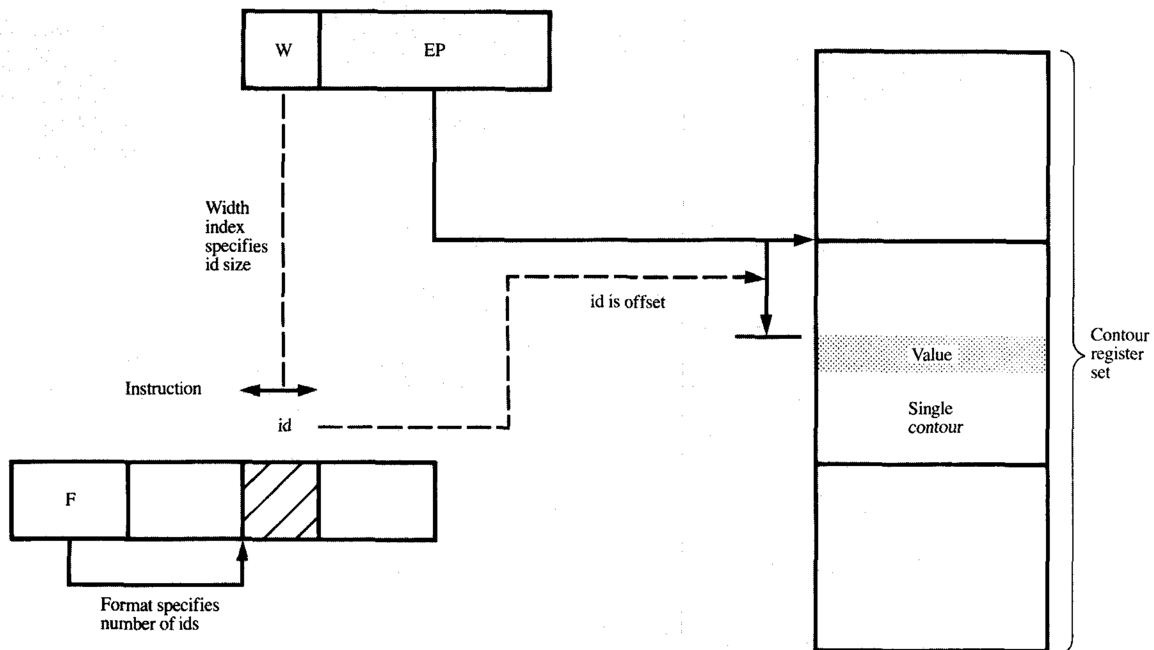


Figure 4

Operand accessing in a DCA.

registers and cache memories without their overhead. Like registers, they have small operand specification fields compared to main memory addresses; but unlike registers, there is an entry for every operand that needs to be referenced, so there is no register allocation to be done. Like a cache, a sufficiently large contour contains the working set of simple variables, but achieves that effect without the hardware support that a cache needs in the form of its partial or full associative store for addresses. Instead, the concentration of variables in the contour is done at compile time, and the contour can be kept in a simple, random-access memory. (This is the basis for a related study by Alpert [13].)

The contour for a given block, say, for a procedure, may be different each time the procedure is called, especially if the high-level language allows recursion. The same procedure may be called more than once before it returns from the first time it is called, with different parameters each time and with different values for its local variables. Also, variables that are neither local nor global but declared at an intermediate level may have different values for different instantiations. Not only may the contour for the procedure be different on multiple calls, but the contours for earlier calls may be needed after subsequent calls on the same procedure. For these reasons, storage space for new contours cannot be allocated statically, and there must be a dynamic control mechanism in the processor. Contours can be allocated directly on a stack, or they can be allocated statically for the first instantiation of each procedure, with a stack onto which all but the most recent instantiation are pushed when recursion is detected at run time. Allocating contours directly on a stack has two advantages: It takes only as much memory as is needed, and it returns quickly from calls.

A width index must be specified for each environment, determining the size of identifier fields in that particular scope. This is shown in **Figure 4.**

At this point, it is important to review what the contour is and is not. The contour register set is the conceptually fast data memory for an abstract language machine. It is unbounded *a priori*, as is the size of the identifiers. Since the contour set is more limited ("expensive") than memory, only

data assigned to it are both within a current scope of definition *and* can be characterized (by type, attribute, or use) as being "frequently" used; other data will remain in "main memory." Different assumptions as to frequency of data referencing or relative cost of contour storage will lead to different implementation strategies of the contour set, but a major goal is to minimize data memory traffic.

The contour model is not an implementation of fast memory *per se*, but rather provides the basis for an implementation. Several implementations can be viewed as following the contour model:

1. Register sets—especially register sets with designated purposes, such as local variables or globals.
2. The RISC register windows. Multiple overlapping register sets provide access to local variables and call-by-value parameters without save/restore overhead. The scheme is reportedly quite effective in supporting the C language [4], which does not allow referencing of variables in an enclosing lexical level—a service not provided by the registers and required, for example, by Pascal.
3. The Ditzel–McLellan stack buffer [14]. Even closer to the contour model is the (1K-word) buffer for a C-based stack frame. In this implementation the entire stack frame (including structured variables, for example) is buffered.
4. The contour buffer. Developed by Alpert [13] and based in part on data contained in this work, this provides a "contour" buffer of 256 words—sufficient to contain stack frame contour references without structured variables, as measured by our test programs.

• *Characteristics of classes of objects*

The contour model can be extended to cover the identification of objects, including operands, labels, operators, etc. The method that is most suitable for object identification depends on the object's characteristics. Objects found in high-level languages include operators, constants, labels, variables, and procedures and functions. Procedures and functions have the same characteristics as constants by run time, so the discussion for constants below applies to them as well.

Operators

The size of the set of operators in a program tends to be fixed, for two reasons. First, for most programming languages, the programmer cannot declare new operators beyond those already defined in the language. Thus, no matter how large a program grows, the number of distinct operators is bounded by the number in the language. Second, the number of operators in the language is not large. For many programs the usage of operators extends to within a binary order of magnitude of the total set, suggesting the fixed encoding adopted for Adept.

Constants

Descriptors for constants can be placed in a table in a manner indistinguishable from that for variables, except that for Pascal, variables do not have initial values as do constants. Alternatively, constants can be inserted into the instruction stream [15].

In an architectural model which has unbounded contour storage, it is clear that bandwidth will be minimized by assigning constants to this storage, an approach followed in Adept. In practical implementations, the encoding of constants is determined by the relative costs of instruction bandwidth and contour storage (however such storage is implemented).

Labels

Explicit labels and other unlabeled destinations of branch instructions are similar to constants in that their values are known at compile time, making it possible for the compiler to place them either in a table or in the instruction stream. Labels differ from constants in one crucial way, however: Many instructions may refer to one constant, but with only three infrequent exceptions, a given label will be found as the destination of only one branch instruction.

Explicit labels in Pascal are completely under programmer control, so anything is possible, but to keep programs structured their use should be limited. Most destinations appear only once, whether they come from explicit labels or structured flow-of-control statements, so for Adept they are placed in the instruction stream rather than a table.

Variables

One technique for identifying variables is to allocate space in the contour for the values of all the variables. This is virtually the same, however, as allocating space for them in main memory. Ditzel and McLellan [14] have proposed a stack frame cache related to this technique.

A second approach is to allocate space for the values of the variables in main memory and to place the addresses of the variables in the contour. Now the contour is much smaller than for the previous case, because the addresses of arrays are much smaller than their values. The DELtran processor uses this kind of contour [9].

A third approach is to consider the amount of memory taken up by the address of a variable and compare it to the amount taken up by its value. If the value is the same size or smaller than the address, then overall memory requirements are reduced by directly placing the value in the contour and allocating no main memory space at all for that variable. (This is the approach used in Adept.) This modification uses the least contour memory by a small amount. A more important effect is that references to simple variables require only one access to memory instead of two, and access is to the faster contour. Between 74.7 percent and 78 percent of

all dynamic operand references [4] are either to constants or to simple variables.

3. The Adept architecture

• *Implementation details*

As an experiment to evaluate the characteristics of an architecture based on the concepts of the previous section, the Adept architecture was developed for the high-level programming language Pascal [16].

The implementation was done on the Stanford Emmy, a 32-bit microprogrammable processor designed specifically for emulation research [17]. Emmy has a fast storage of 4096 32-bit words accessible from the register set in one cycle. This fast storage is used in Adept to contain both the emulator and the run-time contours.

A new contour is created by the call instruction. Every procedure block (every procedure and function and the main body of the program) begins execution after a call instruction has been executed. The contour created for a block is used to identify all objects referred to in that block. At the beginning of a block's code, following information about the number of parameters and the way they are to be passed, is a 12-bit field giving the size of the block's contour. The call instruction creates descriptors for the parameters as it passes them and inserts them as the first entries in the contour. The information needed to create parameter descriptors is contained in the contours already on the stack. Every descriptor contains a tag. After passing the parameters, the call instruction sets the tag fields in the rest of the contour to a value called "invalid." During subsequent execution, the first time one of these descriptors is referenced, the rest of the descriptor is prepared and the tag takes on some value other than invalid. Information needed to create these descriptors is contained in a pointer table, located in memory before the block's executable code. During compilation, descriptors are added to the table in the order that their corresponding objects are encountered in the statements of the block. Then, the pointers are placed in memory in reverse order in an attempt to maximize locality; objects mentioned early in the block will have descriptors close to the first executable code. The relationship between contour and memory is shown in **Figure 5.**

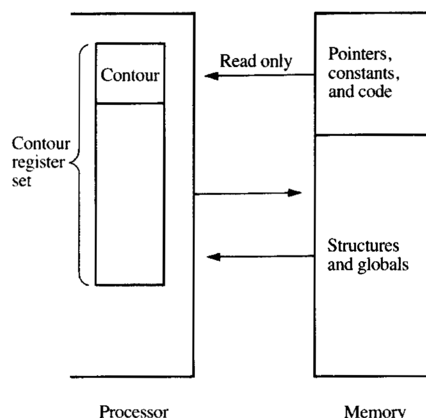


Figure 5

Storage allocation in the Adept processor emulator.

Local scalars Parameters	Require no memory reference
Constants	Require initial reference only
Others: Globals Arrays	Require initial reference for pointer, and subsequent memory access for value on each reference

Figure 6

Single-contour memory traffic.

In the contour (**Figure 6**), local, simple variables consist of a tag and the value of the variable. Local, simple variables are always assigned before they are used, requiring no information in the pointer memory. Attempts to use such variables before their assignment will cause a processor error signal. The compiler assigns identifiers so that local, simple variables have identifiers smaller than those assigned to other objects. This allows the pointer memory to contain initialization information for a contiguous block of descriptors without any unused entries for local, simple variables. The same word in the header of a procedure that contains the size of the contour also contains a 12-bit field giving the identifier of its first object in the pointer memory.

The descriptors for constants are simply copied from the pointer memory the first time they are used and thereafter are indistinguishable from local, simple variables. This means that languages which allow the programmer to declare an initial value for simple variables can have them translated as if they were Pascal-style constants. The first time they are accessed they have an initial value, but they can be subsequently modified.

Object	Pointer/constant memory descriptor		
Simple constant	Tag	Value	
Nonlocal variable (single or array)	Tag	Levels back	id
Local array variable	Tag	l	Relative array base
Global array variable (or string constant)	Tag	g	Absolute array base

Figure 7

Descriptors as they appear in the pointer/constant memory (l and g indicate local or global, determined by a single bit).

Object	Contour	
Simple constant or local, simple variable	Tag	Value
Nonlocal, simple variable	Tag	Address of value
Array variable	Tag	Absolute array base

Figure 8

Descriptors that can appear in the contour.

Nonlocal, simple variables can occur as call-by-reference parameters in which the actual parameter is a simple variable. In this case, the call instruction uses the descriptor for the actual parameter to create a new descriptor with a tag of "nonlocal." If the descriptor for the actual parameter has a tag of "nonlocal," the processor simply copies the entire descriptor. Thus, there is always only one level of indirection for nonlocal, simple variables.

Figure 7 summarizes the kinds of descriptors that can appear in the pointer memory. Recall that there are no entries for local, simple variables, because they are created when they are first assigned. Nor are there any entries for formal parameters; they are created by the call instruction. **Figure 8** lists the three kinds of descriptors that can appear in the contour. Pointer memory can be characterized as using lexical-level addressing, but the contour is very different. After an object is first referenced, the contour either contains its value or points directly to its value; all address chains are gone.

• *Test programs used*

The first two programs that we used to test the Adept architecture are more numeric in nature than the others. The first, FFT, has short but nested loops. The second, Kalman, may be the most representative of scientific programs actually in use. The third test program, Puzzle, solves a bin-packing problem. The fourth is an implementation of the quicksort algorithm, Sort, whose transformation of data consists of simply moving them around. The last program, Walk, is an example of a simple recursive-descent compiler that also traverses the parse tree it creates. The programs were carefully selected to provide a reasonably diverse mix of source program material. Details are found in Wakefield [10].

• *Architectures used*

First, all five test programs were cross-compiled to produce Adept code. The Adept compiler is a recursive-descent compiler consisting of 9500 lines of Pascal. Unlike some other compilers, it does not rely on a preliminary pass over the entire source program in which the program is translated into an intermediate form. The Adept compiler constructs a code tree to represent each procedure block as it is read from the input source file, then passes over the tree a second time generating Adept code.

The Adept processor emulator consists of 2569 32-bit words of microcode, of which 181 are devoted to carrying our floating-point operations and 584 are for input and output. Since the microstore is shared by the emulator and the contour store, 1527 words are available for the contour store. The emulator was instrumented to produce the counts presented below. Recall that all variables in the implementation are 32 bits wide.

Emulator size gives a rough estimate of instruction set complexity. For comparison (all numbers excluding I/O and floating-point), Adept has 1904 words of microcode, System 360 has 2100, and DELtran [7] has 800.

Second, the test programs were compiled on a Hewlett-Packard HP 1000 F-series computer [18] and executed on a simulator for that machine written by John D. Johnson. Compiler switches were set for real numbers to be 32 bits wide, integers to be 32 bits, and subranges to be 16 bits if possible and 32 bits otherwise.

Third, the test programs were compiled for the IBM System/370 by using the Pascal/VS compiler [19] with optimization. The load modules were run on the Stanford Emmy emulating an IBM System/370 [20].

Table 2 Adept format frequency for the Kalman program. The total count is 77 035 instructions.

Percent	*Cumulative*	*Class name*
37.702	37.702	array$x$
14.573	52.276	$a := a$ op b
9.150	61.426	$u := u$ op t
9.114	70.540	$s := a$ op b
6.608	77.149	$a := a$ op t
4.767	81.917	$s :=$ op(a)
2.928	84.845	$a := b$
2.726	87.571	callstdproc
2.287	89.859	$a := b$ op c
2.232	92.091	$t := t$ op a
1.835	93.927	$t :=$ op(t)
1.809	95.737	goto
1.718	97.455	$a := u$ op t
0.554	98.009	call
0.498	98.508	$a := t$ op b
0.345	98.853	$t := a$ op t
0.336	99.189	$a :=$ op(b)
0.293	99.483	procreturn
0.262	99.745	funcreturn
0.140	99.885	$a := b$ op t
0.050	99.936	$a :=$ op(t)
0.028	99.964	array$t$
0.024	99.989	$a :=$ op(a)
0.010	100.000	$a := b$ op a

Fourth, the test set was translated by the Pascal "P" Compiler into P-code, which is the object code for a hypothetical stack computer [21]. Emmy was programmed to interpret the resulting P-code and monitor the execution of the programs [22].

Fifth, one of the programs in the test set, Kalman, was compiled into code for the DEC VAX 11/780 architecture by the Berkeley Pascal UNIX compiler.

4. Results and observations

Recall that the purpose of this study is not to define an implementation of an architecture, but rather to provide a basis for making implementation trade-offs. The Adept experiment, as reported here, has already been elaborated on in several ways by subsequent research. Among the more interesting results are the following.

• *Code size vs. decoding complexity*

While the issue of decoding complexity vs. code size is resolved primarily by technological and implementation considerations, some useful data can be provided by an architectural study. The relative infrequency of a number of the format types included in the transformationally complete format set for a sample program (**Table 2**) indicates that improved code density can be achieved by better format set selection. Other work examines some subsetting possibilities [7]. Moreover, the use of a robust format set together with variable-bit coded identifiers might strike the reader as overkill on code density. Our colleague C. Mitchell [23] has

Table 3 Relative dynamic code size of various format sets.

Adept (as reported)	1.0
System/370	3.30
Optimized Adept	0.947
Explicit temporary formats	1.047
A three-address format only	1.177

Table 4 Dynamic instruction counts: Effect of bit-variable coding.

	Adept	*Explicit temporary*	*Three-operand*
BVC	1.00	1.05	1.18
4/8/12/...	1.18	1.28	1.42
6/12/18/...	1.13	1.22	1.41
8/16/24/...	1.43	1.58	1.81
12 only	1.78	2.06	2.33
16 only	2.12	2.68	3.003
System/370	3.30	—	—

run a number of different block-encoded format and identifier combinations on a similar test program set.

Table 3 gives the relative dynamic size (based on the Adept reported here) of executable code for our test program set, for IBM System/370, and various modifications to the Adept format set—all, however, based on bit-variable coding of identifiers. Optimized Adept is simply Adept with a five-bit rather than a six-bit format syllable (because the number of format combinations barely exceeds 32, it is relatively straightforward to compress the format syllable to the five-bit target). For the explicit temporary format set, the evaluation stack is eliminated and replaced with explicitly named temporaries. This would reduce the number of format combinations to eight formats. Such an arrangement would increase the code space by only ten percent over the optimized Adept. Finally, if only a single evaluation format were used—the three-address format—code size would increase by only 23 percent over optimized Adept so long as bit-variable encoding of identifiers was retained. Indeed, robust formats together with bit-variable-encoded identifiers appear to be duplicative techniques unless code density is a singular consideration in the design of the architecture.

Now consider the use of block codes as compared to bit-variable codes with a variety of format sets.

Table 4 shows the Adept format set, the explicit temporary format set, and the single three-address format used with a variety of different identifier encoding techniques. Again, bit-variable encoding is used in Adept, as reported here. The 4/8 encoding represents the code size when all identifiers are encoded on four-bit boundaries. Similarly, 6/12 and 8/16 represent incremental identifier container sizes. Distinction among identifiers is achieved by environment—once one enters a procedure, the container size is fixed for that procedure, just as in the bit-variable-

encoded Adept. Finally, fixed 12-bit and 16-bit identifiers were also tested. The test programs used required no more than 12 bits of data address (data storage requirements never exceeded 2^{12} words). The reader will note an apparent anomaly in the test programs in that the 6/12 encoding is actually better than the 4/8. The majority of contour sizes were between 17 and 64, resulting in identifier scopes which could be captured in six bits but not in four. A transformationally complete format set, even the single three-address format, provides a significant advantage by eliminating overhead instructions. The single three-address format, even with a 16-bit fixed identifier, is still competitive with load/store architectures, such as System/370. As a corollary, designers of load/store register set-type architectures must pay careful attention to instruction bandwidth requirements.

Some variability is quite valuable in specifying operand identifiers. In particular, using block encoding based upon four or six bits still retains a significant spatial benefit.

It is important to recall that for all of these cases no escape-type codes are required; size is not content-dependent, but rather is known on procedure entry. The value of the identifier's size can be stored in a register and the fields broken apart as soon as the format syllable is decoded, except in the case where the format syllable itself is trivial—the three-address format case—where fields are predetermined.

- *Data referencing strategies*

Some measures that are important in the design of processor storage are shown in **Tables 5** and **6.** The contour should be at least large enough to hold the largest contour encountered in the program to be run. If the contour is to be large enough to hold the entire stack of contours throughout execution, then the maximum size of the stack is also of interest (**Table 7**). In another study associated with the Adept experimental architecture, Alpert [13] has extended both the scope and resolution of the data on the dynamic behavior of contours in Pascal. Using larger test programs,[2] he reports a mean contour size of 26.1 entries, with composition given in **Table 8.**

If local branch labels were included in the contour, rather than explicitly included in the instruction stream (as in Adept), one would add about 9.6 entries to each contour.

The cited data represent mean values with relatively large variance. As Alpert [13] notes, "most procedures refer to a relatively few objects, so their contours are small, but some procedures refer to many objects, particularly initialization procedures."

[2] The method used in this study differed from that used in the evaluation of Adept in that the existence of a viable architecture was assumed, and an existing compiler was modified to emit instructions to increment counters. This eased the burden of debugging all phases of compilation and execution on an experimental architecture, and made the consideration of much larger programs practical.

Table 5 Single contour size.

Single contour composition	*Average*	*Maximum*	*Minimum*
Number of local scalars/ parameters (no memory reference required)	6.73	14.60	1.93
Number of all other data types	9.29	18.80	3.41
Size total	16.02	33.40	5.34

Table 6 Instructions executed.

Number of instructions executed per procedure block	*Average*	*Maximum*	*Minimum*
	538.23	2115.41	12.35

Table 7 Maximum size of contour register set.

Size of contour register set	*Average*	*Maximum*	*Minimum*
Total contour memory words required during execution	481.2	1475	50
Size of largest contour in words	109	194	17

Table 8 Single contour composition.

Contour make-up by data type:		
a) Entries not requiring memory access		
	Local scalars	73%
	Parameters	27%
b) Entries requiring memory access		
	Constants	58%
	Globals	27%
	All others	15%

In Adept the existence of a large contour store is assumed. The optimization emphasis is on instruction and data bandwidth in exchange for this larger storage. Varying the trade-off among instruction and data bandwidth and contour storage gives rise to a multiplicity of lexically based storage strategies, some of which are mentioned earlier.

Note that constants can easily be placed in the instruction stream as literals (as in the PDP-11), or treated as data references, or treated as contour entries (as in Adept). Most constants are short; their values can frequently be encoded within the opcode or an opcode byte. A particularly interesting DCA format-based technique to reduce the number of contour entries for constants has been reported in detail elsewhere [24]. This technique uses about 10 additional formats to specify right or left operands as scalar constants whose value is contained in the designated

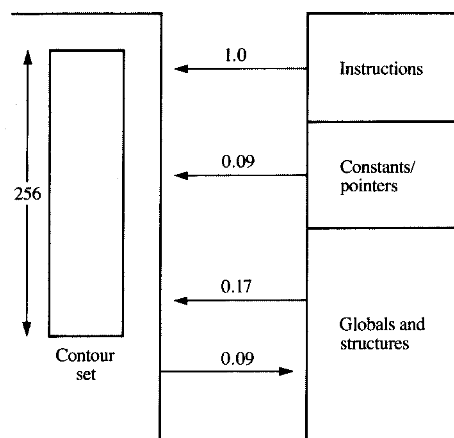


Figure 9

Adept memory traffic per instruction (256 contour entries provide a hit rate over 90%).

identifier. (Of course, the value of the constant must be less than or equal to the container allocated to the identifier.) Constants of value larger than the identifier are treated as variables. This eliminates most constant entries for contours at the expense of about one format bit per instruction.

As with constants, globals can also be distinguished within the instruction stream. Probably the easiest way to make this distinction is to tag each identifier within the instruction as being global or nonglobal. This would add an average of about two bits per instruction, increasing the overall program size and (to a first approximation) the instruction bandwidth by about 12 percent. The net effect would be to reduce contour size to about half that shown in Table 5.

Local structures (arrays, etc.) occur in only a small fraction of contours—less than 20 percent. They have a significant variance, however, and for this reason are probably best not allocated to the contour storage, consistent with the practice described in Adept.

The RISC register window [4] can be viewed as an approximation to the contour memory. The RISC register window consists of a register-addressing space of 32 registers, of which 16 are designated for global use and 16 are allocated for local use (six are caller/callee-overlapped for passing parameters). Four sets have been implemented and additional sets have been suggested. The fixed register allocation is a problem in that the high variance of contour size will create additional data traffic (either for overflows or in its inability to capture required local data in cases of large contours). There is an obvious associated problem of access to nonlocal, nonglobal variables, as required by Pascal (but not by C). However, our frequency data suggest that such references are rare enough to justify special compiler handling of them, i.e., retaining of all such potential referands in memory. However, note that detection of aliased variables is a serious problem for a Pascal compiler.

The Ditzel–McLellan “C” cache stack [14] is even closer to the contour model. Labels and constants are encoded in the instruction stream and globals are handled within the instruction by a global bit. A notable distinction is the allocation of local structures to the buffer. As mentioned before, it is probably a better idea to retain the local structures in memory because of the high variance of contour storage requirements.

Alpert has suggested a contour buffer [13] which more or less directly implements the contour as described in our experiment, i.e., assigning constants, globals, etc., to the contour. Such a buffer of 256 entries will fault on less than 5 percent of the contours entered. If constants are removed, a buffer of 256 entries will fault on about 1.5 percent of the contours entered.

Alpert has shown that his contour buffer with 256 entries will fault approximately as often as a similar contour buffer with 128 entries, but with the constants excluded and reflected in the instruction stream, or with 64 entries and constants and global traffic excluded. The contour buffer provides significant reduction in the instruction bandwidth required over other approaches, but does so at the expense of a larger number of buffer entries. The RISC register window suffers both in terms of instruction bandwidth and data storage requirements due to its rigid allocation policy. Of course, the same policy ensures relatively straightforward implementation.

Figure 9 contrasts the trade-off between bandwidth and processor storage. In **Figure 10,** small constants are placed in the id field of the instruction; large constants are treated as variables. Scalar globals are flagged by a bit in the id fields and are directly accessed in memory without the need for a pointer in contour storage.

Both of these schemes retain a consistent memory image at all times. Further reductions in bandwidth are possible by, for example, copying global values into processor registers. This, of course, means that multiple processors will not see the same global processor state. The effect of consistency requirements on bandwidth is still under study.

As additional storage becomes available within a processor, a reasonable strategy for its use would be to reduce the instruction and data bandwidth requirements. Our data indicate a broad spectrum of possible trade-offs, up to about 256 to 512 words of contour-type storage. At that point one has basically captured all scalar data and minimized program representation requirements up to the point of using data-dependent encoding or frequency-encoded representations. Handling structured data, whether

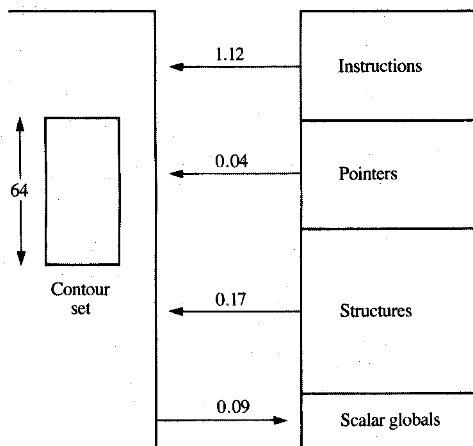


Figure 10

Adept memory traffic modified for constants stored in instructions and separate global reference (now 64 entries provide a hit rate over 90%).

global, local, or of other lexical scope, remains a question for future work, especially where implementations will allow more than 512 words of fast storage. While a data cache for such references is always a possibility, the cache structures themselves are not particularly good vehicles for implementing access to array-type structures because of limited array element locality.

• *Contrasting RISC and Adept*

Largely through related studies at Stanford [25, 26] we can comment on a topical issue, the so-called RISC approach to architecture, as contrasted with Adept. Initially they appear as opposite approaches to the architectural problem: a highly encoded Adept approach vs. a simple encoded RISC approach. Actually, the extent of the differences depends at least partially on the definition one chooses for the RISC approach.

What is a RISC? The RISC approach has been defined in various ways [27]. Most RISC definitions include the following features:

- A load/store architecture.
- Fixed instruction size (usually 32 bits).
- Single instruction format.
- One-cycle instruction execution.
- A relatively small opcode vocabulary and few addressing modes.

This allows or requires additional features such as

- Hard-wired control.
- Fast cycle times.
- Pipelined execution.
- More extensive compiler effort to optimize code.

A large register set, with or without register windows, is a feature of many RISC implementations. If one includes such a set, especially with register windows, as a part of a RISC architecture, then at least as far as the trade-off between data bandwidth required of memory and on-chip storage is concerned, RISC is similar to Adept. Adept attempts to minimize data traffic through the use of contours and contour buffers by increasing the on-chip storage (Figure 1). A large register set, especially with extensive windowed features, is an approximation to a contour buffer, and while the encoding and object assignment differ, the net effect is rather similar as measured in percentage reduction in memory traffic to support the data stream per unit data storage added to a chip.

A more fundamental difference between the RISC architectures and Adept is the trade-off between instruction bandwidth required to support program execution and decoder complexity. RISC favors a very simple decoder which is achieved by use of the load/store architecture with fixed instruction size and format. Adept increases decoder complexity for the use of bit-variable encodings and a more robust format set. To see the net effect of all this, we refer to the work of our colleague C. Mitchell [25]. In order to look only at the effects of object encoding and formats, Mitchell uses a standard set of functional operations in the instruction set for both a fixed 32-bit load/store instruction set (designated *FIX32*) and Adept. Thus, both opcode vocabularies are the same as the vocabulary of Pascal for his test programs, except for the load and store and other memory management instructions defined by the RISC architecture and Adept. Using the same compiler front end and generating Adept and FIX32 code, he analyzed the effect of instruction encoding on instruction bandwidth required from memory (**Figure 11**). FIX32 has a single 16-element register set, while FIX32w has multiple, windowed register sets of 128 total registers [4]. Overall, because of its more efficient encoding, Adept requires about a third the number of instructions to execute a program as the FIX32 (RISC) approach. Perhaps it is more striking to see the effect of instruction bandwidth required as cache storage is added to both RISC and Adept. The figure presents data for an instruction cache only, 16-byte line, 2-way set associative. The more concisely encoded the instruction stream, the more rapidly the working set is contained by the cache. Thus, it takes four times the cache size for a poorly encoded instruction stream (FIX32) to achieve the same memory bandwidth as is achievable with a concisely encoded

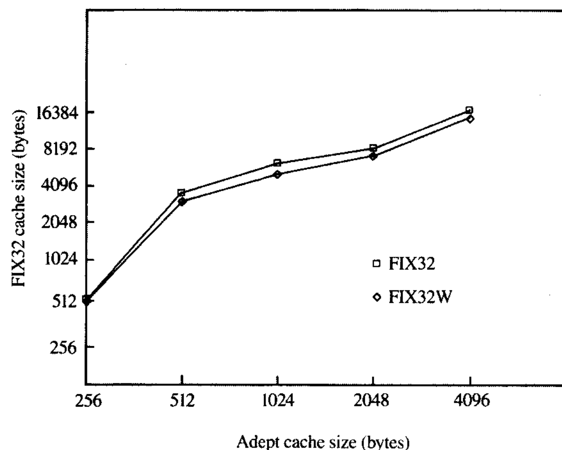


Figure 11

To achieve constant memory traffic, required FIX32 cache sizes for various Adept cache sizes (instruction cache only).

instruction stream. Adept increases the instruction decoder complexity to achieve low instruction bandwidth, or is able to realize a fixed instruction traffic by using a much smaller cache than a less highly encoded instruction stream. Whether the increased decoder complexity is worth the savings in instruction bandwidth or not is determined by implementation considerations. For on-chip implementations and large instruction cache sizes, the additional decoder complexity offers a promising alternative, as the cache area can easily dominate on-chip area considerations. Of course, a complex instruction decoder may influence cycle time considerations. However, cycle time is determined by the slowest of a number of processor actions:

- Instruction decode.
- Cache access time.
- Register access time.
- ALU time.

Published data indicate that, at least for RISC processors, the cycle time is determined primarily by register access time or ALU time and not by instruction decode.

RISC processors with small register sets and without on-board cache represent a trade-off wherein on-chip decode and storage are minimized at the expense of increased memory bandwidth. Adept takes the other approach, maximizing on-chip storage and instruction encoding to achieve minimum instruction and data bandwidth required. By proper instruction encoding and data storage allocation, Adept allows for the optimized utility of on-chip cache. The RISC approach with large register sets appears anomalous, in that it emphasizes the reduction of data bandwidth through on-chip storage yet does not make a similar accommodation for instruction bandwidth.

5. Conclusions

Procedural programming languages imply an abstract machine for execution. An instruction set in correspondence with this language can be derived. Using a robust set of formats, and concise encoding of objects, redundancy in this instruction set representation can be minimized. The contour model of data referencing describes a minimum of data traffic required from a large register set to maintain a consistent global memory. The resulting instruction set architecture provides a useful lower bound on memory traffic as defined by the program. Trade-offs between memory traffic and processor storage and decoding capabilities define possible design alternatives.

Acknowledgments

We are grateful to our colleagues Chad Mitchell and Don Alpert, whose later analysis of Adept relative to other architectural issues significantly enhanced this work. Bob Centers, Dick Conn, Warren Cory, John Hennessy, Jerry Huck, and John Johnson also worked closely with us and contributed algorithms, microcode, and helpful criticism. Susan Gere helped with the figures. This research was supported in part by the Army Research Office–Durham under Contract No. DAAG29-82-K-0109, and by Fairchild Camera and Instrument Corporation.

References and note

1. Y. Patt, W. Hwa, and M. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," *Proceedings, MICRO-18*, ACM, December 1985, pp. 103–108.
2. R. I. Winner and E. M. Carter, "Towards Type-Oriented Dynamic Vertical Migration," *Proceedings, MICRO-16*, ACM, December 1983, pp. 128–139.
3. V. Milutinovic, D. Roberts, and K. Hwang, "Mapping HLL Constructs into Microcode for Improved Execution Speed," *Proceedings, MICRO-17*, ACM, December 1984, pp. 2–11.
4. David A. Patterson and Carlo H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *SIGarch Newsletter* **9**, No. 3, 443–457 (May 1981). (This issue is the Conference Proceedings of the 8th Annual Symposium on Computer Architecture, sponsored by the IEEE Computer Society and the Association for Computing Machinery.)
5. Subrata Dasgupta, "A Model of Clocked Micro-Architectures for Firmware Engineering," *Proceedings, MICRO-17*, ACM, December 1984, pp. 298–308.
6. Michael J. Flynn, "The Interpretive Interface: Resources and Program Representation in Computer Organization," *Proceedings of the Symposium on High Speed Computer and Algorithm Organization*, University of Illinois, Urbana, April 1977.
7. Michael J. Flynn and Lee W. Hoevel, "Execution Architecture: The DELtran Experiment," *IEEE Trans. Computers* **C-32**, No. 2, 156–175 (February 1983).
8. Lee W. Hoevel, "DELtran Principles of Operation: A Directly Executed Language for FORTRAN-II," *Technical Note CSL-*

TN-77-108, Computer Systems Laboratory, Stanford University, Stanford, CA, March 1977.
9. Lee W. Hoevel, "Directly Executed Languages," Ph.D. thesis, The Johns Hopkins University, Baltimore, MD, April 1978.
10. Scott Wakefield, "Studies in Execution Architectures," Ph.D. thesis, Stanford University, Stanford, CA, January 1983 (*Computer Systems Lab Technical Report 83-237*).
11. T. Gross and J. Gill, "A Short Guide to MIPS Assembly Instructions," *Technical Note CSL-83-236*, Computer Systems Laboratory, Stanford University, Stanford, CA, 1983.
12. John D. Johnson, "The Contour Model of Block Structured Processes," *Sigplan Notices* **6**, 55–82 (February 1971).
13. Donald Alpert, "Memory Hierarchies for Directly Executed Language Microprocessors," Ph.D. thesis, Stanford University, Stanford, CA, June 1984.
14. David R. Ditzel and H. R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 48–56.
15. Michael J. Flynn and Lee W. Hoevel, "Measures of Ideal Execution Architectures," *IBM J. Res. Develop.* **28**, No. 4, 356–369 (July 1984).
16. Niklaus Wirth, "The Programming Language Pascal," *Acta Informat.* **1**, 35–63 (1971).
17. Charles Neuhauser, "Emmy System Processor—Principles of Operation," *Technical Note CSL-TN-77-114*, Computer Systems Laboratory, Stanford University, Stanford, CA, May 1977.
18. *Pascal/1000 Reference Manual*, Hewlett-Packard Company, Data Systems Division, Cupertino, CA, first edition, 1981.
19. *Pascal/VS Programmer's Guide*, Order No. SH20-6162; available through IBM branch offices.
20. Walter A. Wallach, "EMMY/360 Functional Characteristics," *Technical Report CSL-TR-76-114*, Computer Systems Laboratory, Stanford University, Stanford, CA, June 1976.
21. K. V. Nori, U. Ammann, K. Jensen, and H. H. Nageli, "The Pascal ⟨P⟩ Compiler: Implementation Notes," *Technical Report*, Eidgenössische Technische Hochschule Zurich, Instituts für Informatik, Zurich, Switzerland, July 1976.
22. Donald Alpert, "A Pascal P-Code Interpreter for the Stanford Emmy," *Technical Report 164*, Computer Systems Laboratory, Stanford University, Stanford, CA, September 1979.
23. Chad L. Mitchell, "The Instruction Bandwidth of Direct Correspondence Architectures," *Technical Report CSL-TR-84-267*, Computer Systems Laboratory, Stanford University, Stanford, CA, December 1984.
24. M. J. Flynn, J. D. Johnson, and S. P. Wakefield, "On Instruction Sets and Their Formats," *IEEE Trans. Computers* **C-34**, No. 3, 242–254 (March 1985).
25. Chad L. Mitchell, "Processor Architecture and Cache Performance," *Technical Report CSL-TR-86-296*, Computer Systems Laboratory, Stanford University, Stanford, CA, June 1986.
26. M. J. Flynn, C. Mitchell, and J. Mulder, "And Now a Case for More Complex Instruction Sets," accepted for publication in *IEEE Computer*.
27. R. P. Colwell, C. Y. Hitchcock III, E. D. Jensen, H. M. Brinkley Sprunt, and C. P. Kollar, "Computers, Complexity and Controversy," *Computer* **18**, No. 9, 8–19 (September 1985).

Received November 11, 1986; accepted for publication April 2, 1987

Scott P. Wakefield *IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Wakefield received the B.S. in electrical engineering–computer science from the University of Colorado, Boulder, and the M.S. and Ph.D. degrees from Stanford University, California, in 1975, 1976, and 1983, respectively. In 1974 while at Stanford he reviewed studies of neural networks from the perspective of a computer designer. Since 1983 he has been with the IBM Thomas J. Watson Research Center, where he has worked on direct correspondence architectures and is currently working on the Research Parallel Processor Prototype, RP3. In 1987, he received an IBM Research Division Award for attaining the ability to simulate the RP3 Processor Memory Element.

Michael J. Flynn *Stanford University, Computer Systems Laboratory, Electrical Engineering Department, Stanford, California 94305.* Professor Flynn received his B.S. in electrical engineering from Manhattan College in 1955, his M.S. from Syracuse University in 1960, and his Ph.D. from Purdue University in 1961. He joined IBM in 1955 and worked for ten years in the areas of computer organization and design. He was design manager of prototype versions of the IBM 7090 and 7094/11, and later was design manager for the System/360 Model 91 central processing unit. Professor Flynn was a faculty member of Northwestern University from 1966 to 1970 and of The Johns Hopkins University from 1970 to 1974. In 1973–74 he was on leave from Johns Hopkins to serve as Vice President of Palyn Associates, Inc.—a computer design firm in San Jose, California, where he is now a senior consultant. Since 1975 he has been Professor of Electrical Engineering at Stanford University, and was Director of the Computer Systems Laboratory from 1977 to 1983. Dr. Flynn has served on the IEEE Computer Society's Board of Governors and as Associate Editor of the *IEEE Transactions on Computers.* He was founding chairman of both the ACM Special Interest Group on Computer Architecture and the IEEE Computer Society's Technical Committee on Computer Architecture.