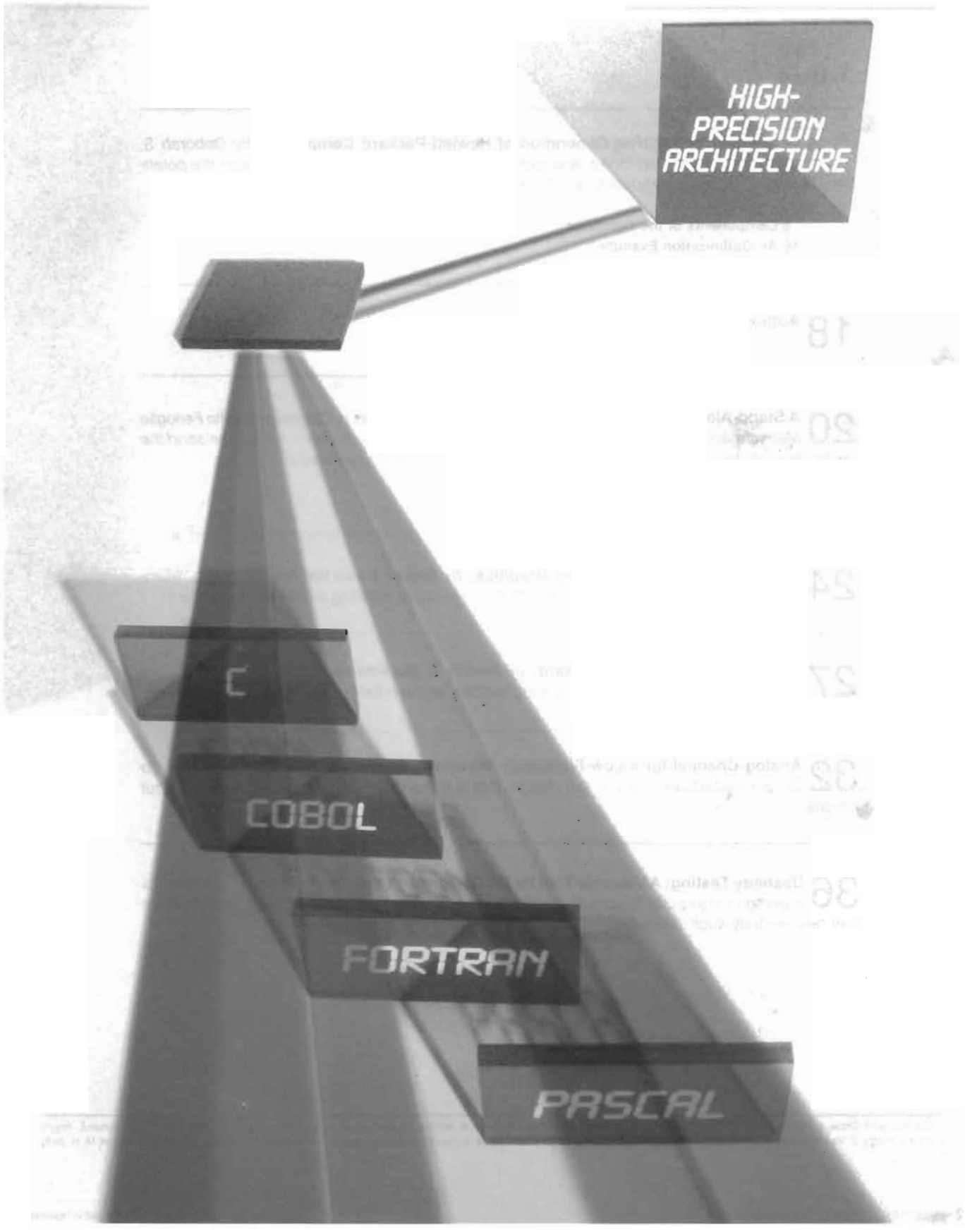
HEWLETT·PACKARD
JOURNAL

JANUARY 1986

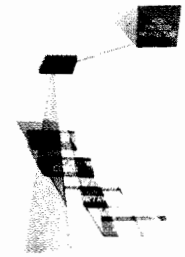


HEWLETT-PACKARD JOURNAL

January 1986 Volume 37 • Number 1

Articles

In this Issue



Hewlett-Packard's next-generation computers are now under development in the program code-named Spectrum, and are scheduled to be introduced in 1986. In our August 1985 issue, Joel Birnbaum and Bill Worley discussed the philosophy and the aims of the new computers and HP's architecture, which has been variously described as reduced-complexity, reduced instruction set computer (RISC), or high-precision. Besides providing higher performance than existing HP computers, an important objective for the new architecture is to support efficient high-level language development of systems and applications software. Compatibility with existing software is another important objective. The design of high-level language compilers is extremely important to the new computers, and in fact, the architecture was developed jointly by both hardware and software engineers. In the article on page 4, three HP compiler designers describe the new compiler system. At introduction, there will be Fortran, Pascal, COBOL, and C compilers, with others to become available later. An optional component of the compiler system called the optimizer tailors the object code to realize the full potential of the architectural features and make programs run faster on the new machines. As much as possible, the compiler system is designed to remain unchanged for different operating systems, an invaluable characteristic for application program development. In the article, the authors debunk several myths about RISCs, showing that RISCs don't need an architected procedure call, don't cause significant code expansion because of the simpler instructions, can readily perform integer multiplication, and can indeed support commercial languages such as COBOL. They also describe millicode, HP's implementation of complex functions using the simple instructions packaged into subroutines. Millicode acts like microcode in more traditional designs, but is common to all machines of the family rather than specific to each.

The article on page 20 introduces the HP 7090A Measurement Plotting System and the articles on pages 24, 27, and 32 expand upon various aspects of its design. The HP 7090A is an X-Y recorder, a digital plotter, a low-frequency waveform recorder, and a data acquisition system all in one package. Although all of these instruments have been available separately before, for some measurement applications where graphics output is desired there are advantages to having them all together. The analog-to-digital converter and memory of the waveform recorder extend the bandwidth of the X-Y recorder well beyond the limits of the mechanism (3 kHz instead of a few hertz). The signal conditioning and A-to-D conversion processes are described in the article on page 32. The servo design (page 24) is multipurpose—the HP 7090A can take analog inputs directly or can plot vectors received as digital data. A special measurement graphics software package (page 27) is designed to help scientists and engineers extend the stand-alone HP 7090A's capabilities without having to write their own software.

No matter how good you think your design is, it will confound some users and cause them to circumvent your best efforts to make it friendly. Knowing this, HP's Personal Computer Division has been conducting usability tests of new PC designs. Volunteers who resemble the expected users are given a series of tasks to perform. The session is videotaped and the product's designers are invited to observe. The article on page 36 reports on the sometimes humorous and always valuable results.

-R.P. Dolan

What's Ahead

The February issue will present the design stories of three new HP instrument offerings. The cover subject will be the HP 5350A, HP 5351A, and HP 5352A Microwave Frequency Counters, which use gallium arsenide hybrid technology to measure frequencies up to 40 GHz. Also featured will be the HP 8757A Scalar Network Analyzer, a transmission and reflection measurement system for the microwave engineer, and the HP 3457A Multimeter, a seven-function, 3½-to-6½-digit systems digital voltmeter.

The HP Journal encourages technical discussion of the topics presented in recent articles and will publish letters expected to be of interest to our readers. Letters must be brief and are subject to editing. Letters should be addressed to: Editor, Hewlett-Packard Journal, 3000 Hanover Street, Palo Alto, CA 94304, U.S.A.

Compilers for the New Generation of Hewlett-Packard Computers

Compilers are particularly important for the reduced-complexity, high-precision architecture of the new machines. They make it possible to realize the full potential of the new architecture.

by Deborah S. Coutant, Carol L. Hammond, and Jon W. Kelley

WITH THE ADVENT of any new architecture, compilers must be developed to provide high-level language interfaces to the new machine. Compilers are particularly important to the reduced-complexity, high-precision architecture currently being developed at Hewlett-Packard in the program that has been code-named Spectrum. The Spectrum program is implementing an architecture that is similar in philosophy to the class of architectures called RISCs (reduced instruction set computers).[1] The importance of compilers to the Spectrum program was recognized at its inception. From the early stages of the new architecture's development, software design engineers were involved in its specification.

The design process began with a set of objectives for the new architecture.[2] These included the following:

- It must support high-level language development of systems and applications software.
- It must be scalable across technologies and implementations.
- It must provide compatibility with previous systems.

These objectives were addressed with an architectural design that goes beyond RISC. The new architecture has the following features:

- There are many simple instructions, each of which executes in a single cycle.
- There are 32 high-speed general-purpose registers.
- There are separate data and instruction caches, which are exposed and can be managed explicitly by the operating system kernel.
- The pipeline has been made visible to allow the software to use cycles normally lost following branch and load instructions.
- Performance can be tuned to specific applications by adding specialized processors that interface with the central processor at the general-register, cache, or main memory levels.

The compiling system developed for this high-precision architecture* enables high-level language programs to use these features. This paper describes the compiling system design and shows how it addresses the specific requirements of the new architecture. First, the impact of high-level language issues on the early architectural design decisions is described. Next, the low-level structure of the compiling system is explained, with particular emphasis on areas that have received special attention for this architecture: program analysis, code generation, and optimization. The paper closes with a discussion of RISC-related issues and how they have been addressed in this compiling system.

*The term "high-precision architecture" is used because the instruction set for the new architecture was chosen on the basis of execution frequency as determined by extensive measurements across a variety of workloads.

Designing an Architecture for High-Level Languages

The design of the new architecture was undertaken by a team made up of design engineers specializing in hardware, computer architecture, operating systems, performance analysis, and compilers. It began with studies of computational behavior, leading to an initial design that provided efficient execution of frequently used instructions, and addressed the trade-offs involved in achieving additional functionality. The architectural design was scrutinized by software engineers as it was being developed, and their feedback helped to ensure that compilers and operating systems would be able to make effective use of the proposed features.

A primary objective in specifying the instruction set was to achieve a uniform execution time for all instructions. All instructions other than loads and branches were to be realizable in a single cycle. No instruction would be included that required a significantly longer cycle or significant additional hardware complexity. Restricting all instructions by these constraints simplifies the control of execution. In conventional microcoded architectures, many instructions pay an overhead because of the complexity of control required to execute the microcode. In reduced-complexity computers, no instruction pays a penalty for a more complicated operation. Functionality that is not available in a single-cycle instruction is achieved through multiple-instruction sequences or, optionally, with an additional processor.

As the hardware designers began their work on an early implementation of the new architecture, they were able to discover which instructions were costly to implement, required additional complexity not required by other instructions, or required long execution paths, which would increase the cycle time of the machine. These instructions were either removed, if the need for them was not great, or replaced with simpler instructions that provided the needed functionality. As the hardware engineers provided feedback about which instructions were too costly to in-

clude, the software engineers investigated alternate ways of achieving the same functionality.

For example, a proposed instruction that provided hardware support for a 2-bit Booth multiplication was not included because the additional performance it provided was not justified by its cost. Architecture and compiler engineers worked together to propose an alternative to this instruction. Similarly, several instructions that could be used directly to generate Boolean conditions were deleted when they were discovered to require a significantly longer cycle time. The same functionality was available with a more general two-instruction sequence, enabling all other operations to be executed faster.

The philosophy of reduced-complexity computers includes the notion that the frequent operations should be fast, possibly at the expense of less frequent operations. However, the cost of an infrequent operation should not be so great as to counterbalance the efficient execution of the simple operations. Each proposed change to the architectural specification was analyzed by the entire group to assess its impact on both software and hardware implementations. Hardware engineers analyzed the instruction set to ensure that no single instruction or set of instructions was causing performance and/or cost penalties for the entire architecture, and software engineers worked to ensure that all required functionality would be provided within performance goals. Compiler writers helped to define conditions for arithmetic, logical, and extract/deposit instructions, and to specify where carry/borrow bits would be used in arithmetic instructions.

As an example of such interaction, compiler writers helped to tune a conditional branch nullification scheme to provide for the most efficient execution of the most common branches. Branches are implemented such that an instruction immediately following the branch can be executed before the branch takes effect.[1] This allows the program to avoid losing a cycle if useful work is possible at that point. For conditional branches, the compiler may or may not be able to schedule an instruction in this slot that can be executed in both the taken-branch and non-taken-branch cases. For these branches, a nullification scheme was devised which allows an instruction to be executed only in the case of a taken branch for backward branches, and only in the case of a non-taken branch for forward branches. This scheme was chosen to enable all available cycles to be used in the most common cases. Backward conditional branches are most often used in a loop, and such branches will most often be taken, branching backwards a number of times before falling through at the end of the iteration. Thus, a nullification scheme that allows this extra cycle to be used in the taken-branch case causes this cycle to be used most often. Conversely, for forward branches, the nullification scheme was tuned to the non-taken-branch case. Fig. 1 shows the code generated for a simple code sequence, illustrating the conditional branch nullification scheme.

Very early in the development of the architectural specification, work was begun on a simulator for the new computer architecture and a prototype C compiler. Before the design was frozen, feedback was available about the ease with which high-level language constructs could be translated to the new instruction set. The early existence of a prototype compiler and simulator allowed operating system designers to begin their development early, and enabled them to provide better early feedback about their needs, from the architecture as well as the compiler.

```
(a)
L1  LDW          4(sp),r1        ; First instruction of loop
    :
    :
    COMIBT,>=,N  10,r2,L1+4      ; Branch to L1+4 if 10 >= r2
    LDW          4(sp),r1        ; Copy of first loop instruction,
                                 ; executed before branch takes effect

(b)
    COMIBF,=,N   0,r1,L1         ; Branch if r1 is not equal to 0
    ADDI         4,r2,r2         ; First instruction of then clause
    :
    :
L1  :
    :
```

Fig. 1. *An illustration of the conditional branch nullification scheme. (a) The conditional branch at the end of a loop will often be followed by a copy of the first instruction of the loop. This instruction will only be executed if the branch is taken. (b) The forward conditional branch implementing an if statement will often be followed by the first instruction of the then clause, allowing use of this cycle without rearrangement of code. This instruction will only be executed if the branch is not taken.*

At the same time, work was begun on optimization techniques for the new architecture. Segments of compiled code were hand-analyzed to uncover opportunities for optimization. These hand-optimized programs were used as a guideline for implementation and to provide a performance goal. Soon after the first prototype compiler was developed, a prototype register allocator and instruction scheduler were also implemented, providing valuable data for the optimizer and compiler designers.

Compiling to a Reduced Instruction Set

Compiling for a reduced-complexity computer is simplified in some aspects. With a limited set of instructions from which to choose, code generation can be straightforward. However, optimization is necessary to realize the full advantage of the architectural features. The new HP compiling system is designed to allow multiple languages to be implemented with language-specific compiler front ends. An optimization phase, common to all of the languages, provides efficient register use and pipeline scheduling, and eliminates unnecessary computations. With the elimination of complex instructions found in many architectures, the responsibility for generating the proper sequence of instructions for high-level language constructs falls to the compiler. Using the primitive instructions, the compiler can construct precisely the sequence required for the application.

For this class of computer, the software architecture plays a strong role in the performance of compiled code. There is no procedure call instruction, so the procedure calling sequence is tuned to handle simple cases, such as *leaf routines* (procedures that do not call any other procedures), without fixed expense, while still allowing the complexities of nested and recursive procedures. The saving of registers at procedure call and procedure entry is depen-

(continued on page 7)

Components of the Optimizer

The optimizer is composed of two types of components, those that perform data flow and control flow analysis, and those that perform optimizations. The information provided by the analysis components is shared by the optimization components, and is used to determine when instructions can be deleted, moved, rearranged, or modified.

For each procedure, the control flow analysis identifies basic blocks (sequences of code that have no internal branching). These are combined into intervals, which form a hierarchy of control structures. Basic blocks are at the bottom of this hierarchy, and entire procedures are at the top. Loops and if-then constructs are examples of the intermediate structures.

Data flow information is collected for each interval. It is expressed in terms of resource numbers and sequence numbers. Each register, memory location, and intermediate expression has a unique resource number, and each use or definition of a resource has a unique sequence number. Three types of data flow information are calculated:

- Reaching definitions: for each resource, the set of definitions that could reach the top of the interval by some path.
- Exposed uses: for each resource, the set of uses that could be reached by a definition at the bottom of the interval.
- UNDEF set: the set of resources that are not available at the top of the interval. A resource is available if it is defined along all paths reaching the interval, and none of its operands are later redefined along that path.

From this information, a fourth data structure is built:

- Web: a set of sequence numbers having the property that for each use in the set, all definitions that might reach it are also in the set. Likewise, for each definition in the set, all uses it might reach are also in the set. For each resource there may be one or many webs.

Loop Optimizations

Frequently the majority of execution time in a program is spent executing instructions contained in loops. Consequently, loop-based optimizations can potentially improve execution time significantly. The following discussion describes components that perform loop optimizations.

Loop Invariant Code Motion. Computations within a loop that yield the same result for every iteration are called loop invariant computations. These computations can potentially be moved outside the loop, where they are executed less frequently.

An instruction inside the loop is invariant if it meets either of two conditions: either the reaching definitions for all its operands are outside the loop, or its operands are defined by instructions that have already themselves been identified as loop invariant. In addition, there must not be a conflicting definition of the instruction's target inside the loop. If the instruction is executed conditionally inside the loop, it can be moved out only if there are no exposed uses of the target at the loop exit.

An example is a computation involving variables that are not modified in the loop. Another is the computation of an array's base address.

Strength Reduction and Induction Variables. Strength reduction replaces multiplication operations inside a loop with iterative addition operations. Since there is no hardware instruction for integer multiplication in the architecture, converting sequences of shifts and adds to a single instruction is a performance improvement. Induction variables are variables that are defined inside the loop in terms of a simple function of the loop counter.

Once the induction variables have been determined, those that are appropriate for this optimization are selected. Any multiplications involved in the computation of these induction variables are replaced with a COPY from a temporary. This temporary holds the initial value of the function, and is initialized preceding the loop. It is updated at the point of all the reaching definitions of the induction variable with an appropriate addition instruction. Finally, the induction variable itself is eliminated if possible.

This optimization is frequently applied to the computation of array indices inside a loop, when the index is a function of the loop counter.

Common Subexpression Elimination

Common subexpression elimination is the removal of redundant computations and the reuse of the one result. A redundant computation can be deleted when its target is not in the UNDEF set for the basic block it is contained in, and all the reaching definitions of the target are the same instruction. Since the optimizer runs at the machine level, redundant loads of the same variable in addition to redundant arithmetic computations can be removed.

Store-Copy Optimization

It is possible to promote certain memory resources to registers for the scope of their definitions and uses. Only resources that satisfy aliasing restrictions can be transformed this way. If the transformation can be performed, stores are converted to copies and the loads are eliminated. This optimization is very useful for a machine that has a large number of registers, since it maximizes the use of registers and minimizes the use of memory.

For each memory resource there may be multiple webs. Each memory web is an independent candidate for promotion to a register.

Unused Definition Elimination

Definitions of memory and register resources that are never used are removed. These definitions are identified during the building of webs.

Local Constant Propagation

Constant propagation involves the folding and substitution of constant computations throughout a basic block. If the result of a computation is a constant, the instruction is deleted, and the resultant constant is used as an immediate operand in subsequent instructions that reference the original result. Also, if the operands of a conditional branch are constant, the branch can be changed to an unconditional branch or deleted.

Coloring Register Allocation

Many components introduce additional uses of registers or prolong the use of existing registers over larger portions of the procedure. Near-optimal use of the available registers becomes crucial after these optimizations have been made.

Global register allocation based on a method of graph coloring is performed. The register resources are partitioned into groups of disjoint definitions and uses called register webs. Then, using the exposed uses information, interferences between webs are computed. An interference occurs when two webs must be assigned different machine registers. Registers that are copies of each other are assigned to the same register and the copies are eliminated. The webs are sorted based on the number of interfer-

ences each contains. Then register assignment is done using this ordering. When the register allocator runs out of registers, it frees a register by saving another one to memory temporarily. A heuristic algorithm is used to choose which register to save. For example, registers used heavily within a loop will not be saved to free a register.

Peephole Optimizations

The peephole optimizer uses a dictionary of equivalent instruction patterns to simplify instruction sequences. Some of the patterns identify simplifications to addressing mode changes, bit manipulations, and data type conversions.

Branch Optimizations

The branch optimizer component traverses the instructions, transforming branch instruction sequences into more efficient instruction sequences. It converts branches over single instructions to instructions with conditional nullification. A branch whose target is the next instruction is deleted. Branch chains involving both unconditional and conditional branches are combined into shorter sequences wherever possible. For example, a conditional branch to an unconditional branch is changed to a single conditional branch.

Dead Code Elimination

Dead code is code that cannot be reached at program execution, since no branch to it or fall-through exists. This code is deleted.

Scheduler

The instruction scheduler reorders the instructions within a basic block, minimizing load/store and floating-point interlocks. It also schedules the instructions following branches.

Suneel Jain
Development Engineer
Information Technology Group

(continued from page 5)

dent on the register use of the individual procedure. A special calling convention has been adopted to allow some complex operations to be implemented in low-level routines known as *millicode*, which incur little overhead for saving registers and status.

Compiling to a reduced instruction set can be simplified because the compiler need not make complicated choices among a number of instructions that have similar effects. In the new architecture, all arithmetic, logical, or conditional instructions are register-based. All memory access is done through explicit loads and stores. Thus the compiler need not choose among instructions with a multitude of addressing modes. The compiler's task is further simplified by the fact that the instruction set has been constructed in a very symmetrical manner. All instructions are the same length, and there are a limited number of instruction formats. In addition to simplifying the task of code generation, this makes the task of optimization easier as well. The optimizer need not handle transformations between instructions that have widely varying formats and addressing modes. The symmetry of the instruction set makes the tasks of replacing or deleting one or more instructions much easier.

Of course, the reduced instruction set computer, though simplifying some aspects of the compilation, requires more of the compilers in other areas. Having a large number of registers places the burden on the compilers to generate code that can use these registers efficiently. Other aspects of this new architecture also require the compilers to be more intelligent about code generation. For example, the instruction pipeline has become more exposed and, as mentioned earlier, the instruction following a branch may be executed before the branch takes effect. The compiler therefore needs to schedule such instructions effectively. In addition, loads from memory, which also require more than a single cycle, will interlock with the following instruction if the target register is used immediately. The compiler can increase execution speed by scheduling instructions to avoid these interlocks. The optimizer can also improve the effectiveness of a floating-point coprocessor by eliminating unnecessary coprocessor memory accesses and by reordering the floating-point instructions.

In addition to such optimizations, which are designed to exploit specific architectural features, conventional optimizations such as common subexpression elimination, loop invariant code motion, induction variable elaboration, and local constant propagation were also implemented.[3] These have a major impact on the performance of any computer. Such optimizations reduce the frequency of loads, stores, and multiplies, and allow the processor to be used with greater efficiency. However, the favorable cost/performance of the new HP architecture can be realized even without optimization.

The Compiler System

All of the compilers for the new architecture share a common overall design structure. This allows easy integration of common functional components including a symbolic debugger, a code generator, an optimizer, and a linker. This integration was achieved through detailed planning, which involved the participation of engineers across many language products. Of the new compilers, the Fortran/77, Pascal, and COBOL compilers will appear very familiar to some of our customers, since they were developed from existing products available on the HP 3000 family of computers. All of these compilers conform to HP standard specifications for their respective languages, and thus will provide smooth migration from the HP 1000, HP 3000, and HP 9000 product lines. The C compiler is a new product, and as mentioned earlier, was the compiler used to prototype the instruction set from its earliest design phase. The C compiler conforms to recognized industry standard language specifications. Other compilers under development will be integrated into this compiler system.

To achieve successful integration of compilers into a homogeneous compiling system it was necessary to define distinct processing phases and their exact interfaces in terms of data and control transfer. Each compiler begins execution through the *front end*. This includes the lexical, syntactic, and semantic analysis prescribed by each lan-

guage standard. The front ends generate intermediate codes from the source program, and pass these codes to the code generators. The intermediate codes are at a higher level than the machine code generated by a later phase, and allow a certain degree of machine abstraction within the front ends.

Two distinct code generators are used. They provide varying degrees of independence from the front ends. Each interfaces to the front ends through an intermediate code. One of these code generation techniques has already been used in two compiler products for the HP 3000. Fig. 2 shows the overall design of the compilers. Each phase of the compilation process is pictured as it relates to the other phases. The front ends are also responsible for generating data to be used later in the compilation process. For example, the front end generates data concerning source statements and the types, scopes and locations of procedure/function and variable names for later use by the symbolic debugger. In addition, the front end is responsible for the collection of data to be used by the optimizer.

These compilers can be supported by multiple operating systems. The object file format is compatible across operating systems.

Code Generation

The code generators emit machine code into a data structure called SLLIC (Spectrum low-level intermediate code). SLLIC also contains information regarding branches and their targets, and thus provides the foundation for the building of a control flow graph by the optimizer. The SLLIC data structure contains the machine instructions and the specifications for the run-time environment, including the program data space, the literal pool, and data initialization. SLLIC also holds the symbolic debug information generated by the front end, is the medium for later optimization, and is used to create the object file.

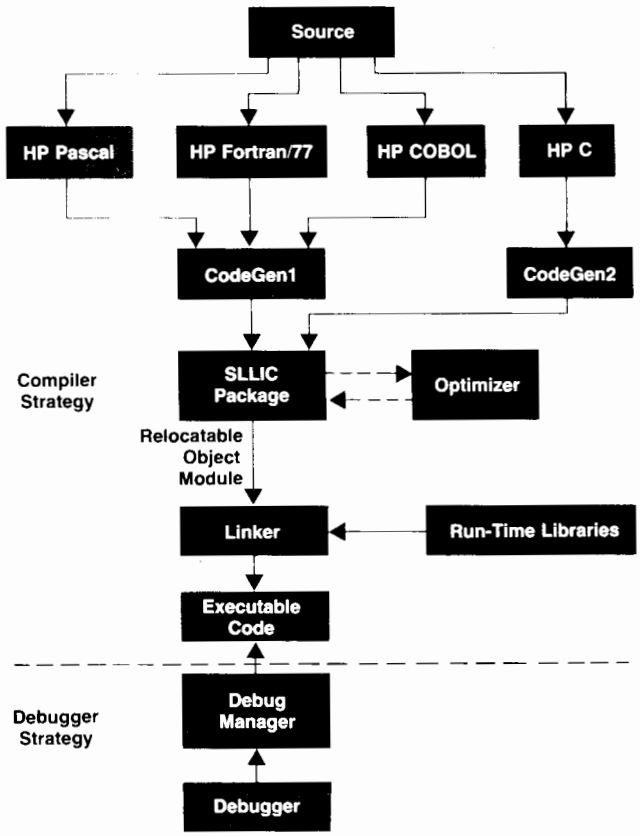


Fig. 2. *The compiler system for HP's new generation of high-precision-architecture computers.*

The reduced instruction set places some extra burden on the code generators when emitting code for high-level language constructs such as byte moves, decimal operations, and procedure calls. Since the instruction set contains no complex instructions to aid in the implementation of these constructs, the code generators are forced to use combinations of the simpler instructions to achieve the same functionality. However, even in complex instruction set architectures, complex case analysis is usually required to use the complex instructions correctly. Since there is little redundancy in the reduced instruction set, most often no choice of alternative instruction sequences exists. The optimizer is the best place for these code sequences to be streamlined, and because of this the overall compiler design is driven by optimization considerations. In particular, the optimizer places restrictions upon the code generators.

The first class of such restrictions involves the presentation of branch instructions. The optimizer requires that all branches initially be followed by a NOP (no operation) instruction. This restriction allows the optimizer to schedule instructions easily to minimize interlocks caused by data and register access. These NOPs are subsequently replaced with useful instructions, or eliminated.

The second class of restrictions concerns register use. Register allocation is performed within the optimizer. Rather than use the actual machine registers, the code generators use symbolic registers chosen from an infinite register set. These symbolic registers are mapped to the set of actual machine registers by the register allocator. Although register *allocation* is the traditional name for such an activity, register *assignment* is more accurate in this context. The code generators are also required to associate every syntactically equivalent expression in each procedure with a unique symbolic register number. The symbolic register number is used by the optimizer to associate each expression with a value number (each run-time *value* has a unique number). Value numbering the symbolic registers aids in the detection of common subexpressions within the optimizer. For example, every time the local variable i is loaded it is loaded into the same symbolic register, and every time the same two symbolic registers are added together the result is placed into a symbolic register dedicated to hold that value.

Although the optimizer performs transformations at the machine instruction level, there are occasions where it could benefit from the existence of slightly modified and/or additional instructions. *Pseudoinstructions* are instructions that map to one or more machine instructions and are only valid within the SLLIC data structure as a software convention recognized between the code generators and the optimizer. For example, the NOP instruction mentioned above is actually a pseudoinstruction. No such instruction exists on the machine, although there are many instruction/operand combinations whose net effect would be null. The

NOP pseudoinstruction saves the optimizer from having to recognize all those sequences. Another group of pseudoinstructions has been defined to allow the optimizer to view all the actual machine instructions in the same canonical form, without being restricted by the register use prescribed by the instructions. For example, some instructions use the same register as both a source and a target. This makes optimization very difficult for that instruction. The solution involves the definition of a set of pseudoinstructions, each of which maps to a two-instruction sequence, first to copy the source register to a new symbolic register, and then to perform the operation on that new register. The copy instruction will usually be eliminated by a later phase of the optimizer.

Another class of perhaps more important pseudoinstructions involves the encapsulation of common operations that are traditionally supported directly by hardware, but in a reduced instruction set are only supported through the generation of code sequences. Examples include multiplication, division, and remainder. Rather than have each code generator contain the logic to emit some correct sequence of instructions to perform multiplication, a set of pseudoinstructions has been defined that makes it appear as if a high-level multiplication instruction exists in the architecture. Each of the pseudoinstructions is defined in terms of one register target and either two register operands or one register operand and one immediate. The use of these pseudoinstructions also aids the optimizer in the detection of common subexpressions, loop invariants, and induction variables by reducing the complexity of the code sequences the optimizer must recognize.

Control flow restrictions are also placed on generated code. A *basic block* is defined as a straight-line sequence of code that contains no transfer of control out of or into its midst. If the code generator wishes to set the carry/borrow bit in the status register, it must use that result within the same basic block. Otherwise, the optimizer cannot guarantee its validity. Also, all argument registers for a procedure/function call must be loaded in the same basic block that contains the procedure call. This restriction helps the register allocator by limiting the instances where hard-coded (actual) machine registers can be *live* (active) across basic block boundaries.

Optimization

After the SLLIC data structure has been generated by the code generator, a call is made to the optimizer so that it can begin its processing. The optimizer performs intraprocedural local and global optimizations, and can be turned on and off on a procedure-by-procedure basis by the programmer through the use of compiler options and directives specific to each compiler. Three levels of optimization are supported and can also be selected at the procedural level.

Optimization is implemented at the machine instruction level for two reasons. First, since the throughput of the processor is most affected by the requests made of the memory unit and cache, optimizations that reduce the number of requests made, and optimizations that rearrange these requests to suit the memory unit best, are of the most value. It is only at the machine level that all memory accesses become exposed, and are available candidates for such optimizations. Second, the machine level is the common denominator for all the compilers, and will continue to be for future compilers for the architecture. This allows the implementation of one optimizer for the entire family of compilers. In addition to very machine specific optimizations, a number of theoretically machine independent optimizations (for example, loop optimizations) are also included. These also benefit from their low-level implementation, since all potential candidates are exposed. For example, performing loop optimizations at the machine level allows the optimizer to move constants outside the loop, since the machine has many registers to hold them. In summary, no optimization has been adversely affected by this strategy; instead, there have been only benefits.

Level 0 optimization is intended to be used during program development. It is difficult to support symbolic debugging in the presence of all optimizations, since many optimizations reorder or delete instruction sequences. Nonsymbolic debugging is available for fully optimized programs, but users will still find it easier to debug nonoptimized code since the relationship between the source and object code is clearer. No code transformations are made at level 0 that would preclude the use of a symbolic debugger. In particular, level 0 optimizations include some copy and NOP elimination, and limited branch scheduling. In addition, the components that physically exist as part of the optimizer, but are required to produce an executable program, are invoked. These include register allocation and branch fixing (replacing short branches with long branches where necessary).

After program correctness has been demonstrated using only level 0 optimizations, the programmer can use the more extensive optimization levels. There are two additional levels of optimization, either of which results in code reordering. The level any particular optimization component falls into is dependent upon the type of information it requires to perform correct program transformations. The calculation of data flow information gives the optimizer information regarding all the resources in the program. These resources include general registers, dedicated and status registers, and memory locations (variables). The information gleaned includes where each resource is defined and used within the procedure, and is critical for some optimization algorithms. Level 1 optimizations require no data flow information, therefore adding only a few additional optimizations over level 0. Invoking the optimizer at level 2 will cause all optimizations to be performed. This requires data flow information to be calculated.

Level 1 optimization introduces three new optimizations: peephole and branch optimizations and full instruction scheduling. Peephole optimizations are performed by pattern matching short instruction sequences in the code to corresponding templates in the peephole optimizer. An example of a transformation is seen in the C source expression

```
if (flag & 0x8)
```

which tests to see that the fourth bit from the right is set in the integer flag. The unoptimized code is

```
LDO       8(0),19      ; load immediate 8 into r19
AND       31,19,20     ; intersect r31 (flag) with r19 into r20
COMIBT,=  0,20,label   ; compare result against 0 and branch
```

Peephole optimization replaces these three instructions with the one instruction

```
BB,>=     31,28,label  ; branch on bit
```

which will branch if bit 28 (numbered left to right from 0) in r31 (the register containing flag) is equal to 0.

Level 1 optimization also includes a branch optimizer whose task is to eliminate unnecessary branches and some unreachable code. Among other tasks, it replaces branch chains with a single branch, and changes conditional branches whose targets are unconditional branches to a single conditional branch.

The limited instruction scheduling algorithm of level 0 is replaced with a much more thorough component in level 1. Level 0 scheduling is restricted to replacing or removing the NOPs following branches where possible, since code sequence ordering must be preserved for the symbolic debugger. In addition to this, level 1 instructions are scheduled with the goal of minimizing memory interlocks. The following typify the types of transformations made:

- Separate a load from the instruction that uses the loaded register
- Separate store and load instruction sequences
- Separate floating-point instructions from each other to improve throughput of the floating-point unit.

Instruction scheduling is accomplished by first constructing a dependency graph that details data dependencies between instructions. Targeted instructions are separated by data independent instructions discovered in the graph.

The same register allocator is used in level 0 and level 1 optimization. It makes one backwards pass over each procedure to determine where the registers are defined and used and whether or not they are live across a call. It uses this information as a basis for replacing the symbolic registers with actual machine registers. Some copy elimination is also performed by this allocator.

Level 2 optimizations include all level 1 optimizations as well as local constant propagation, local peephole transformations, local redundant definition elimination, common subexpression and redundant load/store elimination, loop invariant code motion, induction variable elaboration and strength reduction, and another register allocator. The register allocator used in level 2 is partially based on graph coloring technology.[4] Fully optimized code contains many more live registers than partially optimized or nonoptimized code. This register allocator handles many live registers better than the register allocator of levels 0 and 1. It has access to the data flow information calculated for the symbolic registers and information regarding the frequency of execution for each basic block.

Control Flow and Data Flow Analysis

All of the optimizations introduced in level 2 require data flow information. In addition, a certain amount of control flow information is required to do loop-based optimizations. Data flow analysis provides information to the optimizer about the pattern of definition and use of each resource. For each basic block in the program, data flow information indicates what definitions may reach the block (reaching definitions) and what later uses may be affected by local definitions (exposed uses). Control flow information in the optimizer is contained in the basic block and interval structures. *Basic block analysis* identifies blocks of code that have no internal branching. *Interval analysis* identifies patterns of control flow such as if-then-else and loop constructs.[5] Intervals simplify data flow calculations, identify loops for the loop-based optimizations, and enable partial update of data flow information.

In the optimizer, control flow analysis and data flow analysis are performed in concert. First, basic blocks are identified. Second, local data flow information is calculated for each basic block. Third, interval analysis exposes the structure of the program. Finally, using the interval structure as a basis for its calculation rules, global data flow analysis calculates the reaching definitions and exposed uses.

Basic block analysis of the SLLIC data structure results in a graph structure where each basic block identifies a sequence of instructions, along with the predecessor and successor basic blocks. The interval structure is built on top of this, with the smallest interval being a basic block. Intervals other than basic blocks contain subintervals which may themselves be any type of interval. Interval types include basic block, sequential block (the subintervals follow each other in sequential order), if-then, if-then-else, self loop, while loop, repeat loop, and switch (case statement). When no such interval is recognized, a set of subintervals may be contained in either a proper *interval*

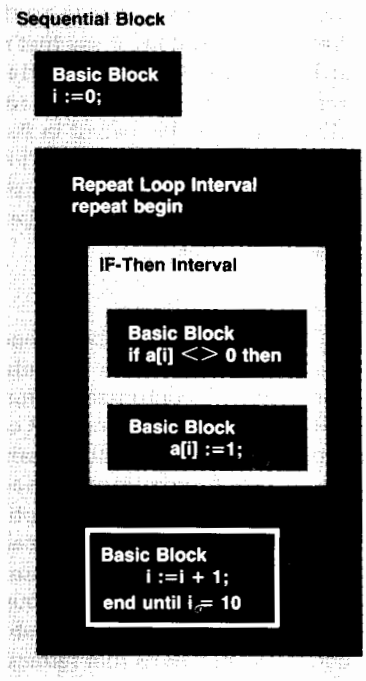


Fig. 3. *This figure illustrates the interval structure of a simple sequence of Pascal code. The nested boxes represent the interval hierarchy.*

(if the control flow is well-behaved) or an improper *interval* (if it contains multiple-entry cycles or targets of unknown branches). An entire procedure will be represented by a single interval with multiple descendants. Fig. 3 shows the interval structure for a simple Pascal program.

Calculation of data flow information begins with an analysis of what resources are used and defined by each basic block. Each use or definition of a resource is identified by a unique sequence number. Associated with each sequence number is information regarding what resource is being referenced, and whether it is a use or a definition. Each SLLIC instruction entry contains sequence numbers for all of the resources defined or used by that instruction. The local data flow analysis determines what local uses are exposed at the top of the basic block (i.e., there is a use of a resource with no preceding definition in that block) and what local definitions will reach the end of the block (i.e., they define a resource that is not redefined later in the block). The local data flow analysis makes a forward and backward pass through the instructions in a basic block to determine this information.

Local data flow information is propagated out from the basic blocks to the outermost interval. Then, information about reaching definitions and exposed uses is propagated inward to the basic block level. For known interval types, this involves a straightforward calculation for each subinterval. For proper intervals, this calculation must be performed twice for each subinterval, and for improper intervals, the number of passes is limited by the number of subintervals.

As each component of the optimizer makes transformations to the SLLIC graph, the data flow information becomes inaccurate. Two strategies are employed to bring this information up-to-date: *patching* of the existing data flow information and partial recalculation. For all optimizations except induction variable elimination, the data flow information can be patched by using information about the nature of the transformation to determine exactly how the data flow information must be changed. All transformations take place within the loop interval in induction variable elimination. The update of data flow information within the loop is performed by recalculating the local data flow information where a change has been made, and then by propagating that change out to the loop interval. The effect of induction variable elimination on intervals external to the loop is limited, and this update is performed by patching the data flow information for these intervals.

Aliasing

The concept of resources has already been presented in the earlier discussion of data flow analysis. The optimizer provides a component called the resource manager for use throughout the compiler phases. The resource manager is responsible for the maintenance of information regarding the numbers and types of resources within each procedure. For example, when the code generator needs a new symbolic register, it asks the resource manager for one. The front ends also allocate resources corresponding to memory locations for every variable in each procedure. The resources allocated by the resource manager are called resource numbers. The role of the resource manager is especially important in this family of compilers. It provides a way for the front end, which deals with memory resources in terms of programmer variable names, and the optimizer, which deals with memory resources in terms of actual memory locations, to communicate the relationship between the two.

The most basic use of the resource numbers obtained through the resource manager is the identification of unique programmer variables. The SLLIC instructions are decorated with information that associates resource numbers with each operand. This allows the optimizer to recognize uses of the same variable without having to compare addresses. The necessity for communication between the front ends and the optimizer is demonstrated by the following simplified example of C source code:

```
proc() {
  int i, j, k, *p;
   .
   .
   .
i = j + k;
*p = 1;
i = j + k;
   .
   .
   .
}
```

At first glance it might seem that the second calculation of j + k is redundant, and in fact it is a common subexpression that need only be calculated once. However, if the pointer p has been set previously to point to either j or k, then the statement *p = 1 might change the value of either j or k. If p has been assigned to point to j, then we say that *p and j are *aliased* to each other. Every front end includes a component called a *gatherer*[6] whose responsibility it is to collect information concerning the ways in which memory resources in each procedure relate to each other. This information is cast in terms of resource numbers, and is collected in a similar manner by each front end. Each gatherer applies a set of language specific alias rules to the source. A later component of the optimizer called the aliaser reorganizes this information in terms more suitable for use by the local data flow component of the optimizer.

Each gatherer had to solve aliasing problems specific to its particular target language. For example, the Pascal gatherer was able to use Pascal's strong typing to aid in building sets of resources that a pointer of some particular type can point to. Since C does not have strong typing, the C gatherer could make no such assumptions. The COBOL compiler had to solve the aliasing problems that are introduced with the REDEFINE statement, which can make data items look like arrays. Fig. 4 shows the structure of the new compilers from an aliasing perspective. It details data and control dependencies. Once the aliasing data has been incorporated into the data flow information, every component in the optimizer has access to the information, and incorrect program transformations are prevented.

The aliaser also finishes the calculation of the aliasing

relationships by calculating the transitive closure* on the aliasing information collected by the gatherers. The need for this calculation is seen in the following skeleton Pascal example:

```
procedure p;
 begin
 p : ^integer;
 q : ^integer;
   .
   .
   .
 p := q;
   .
   .
   .
 q := p;
   .
   .
   .
 end;
```

The aliasing information concerning q must be transferred to p, and vice versa, because of the effects of the two assignment statments shown. The aliaser is an optimizer component used by all the front ends, and requires no language specific data. Another type of memory aliasing occurs when two or more programmer variables can overlap with one another in memory. This happens within C unions and Fortran equivalence statements. Each gatherer must also deal with this issue, as well as collecting information concerning the side effects of procedure and function calls and the use of arrays.

*Transitive closure: For a given resource, the set of resources that can be shown to be aliased to the given resource by any sequence of aliasing relationships.

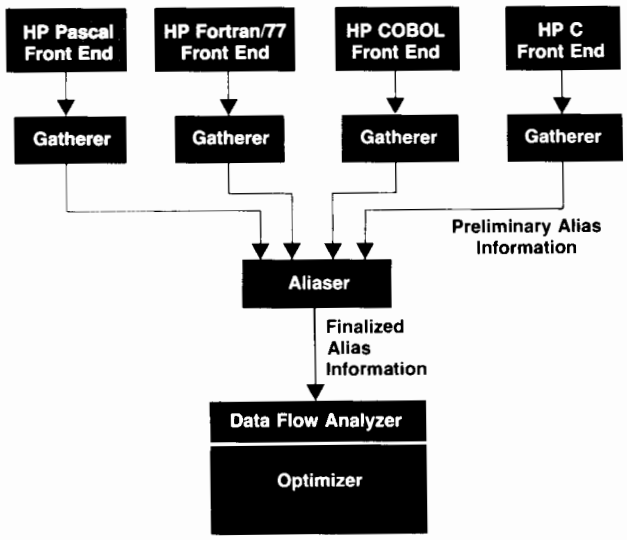


Fig. 4. *Scheme for the collection of alias information.*

The SLLIC Package

The SLLIC data structure is allocated, maintained, and manipulated by a collection of routines called the *SLLIC package*. Each code generator is required to use these routines. The SLLIC package produces an object file from the SLLIC graph it is presented with, which is either optimized or unoptimized. During implementation it was relatively easy to experiment with the design of the object file, since its creation is only implemented in one place. The object file is designed to be transportable between multiple operating systems running on the same architecture.

The SLLIC graph also contains the symbolic debug information produced by the front end. This information is placed into the object file by the SLLIC package. The last step in the compilation process is the link phase. The linker is designed to support multiple operating systems. As much as possible, our goal has been for the new compilers to remain unchanged across operating systems, an invaluable characteristic for application development.

Addressing RISC Myths

The new compiling system provides a language development system that is consistent across languages. However, each language presents unique requirements to this system. Mapping high-level language constructs to a reduced-complexity computer requires the development of new implementation strategies. Procedure calls, multiplication, and other complex operations often implemented in microcode or supported in the hardware can be addressed with code sequences tuned to the specific need. The following discussion is presented in terms of several misconceptions, or myths, that have appeared in speculative discussions concerning code generation for reduced-complexity architectures. Each myth is followed by a description of the approach adopted for the new HP compilers.

Myth: An architected procedure call instruction is necessary for efficient procedure calls.

Modern programming technique encourages programmers to write small, well-structured procedures rather than large monolithic routines. This tends to increase the frequency of procedure calls, thus making procedure call efficiency crucial to overall system performance.

Many machines, like the HP 3000, provide instructions to perform most of the steps that make up a procedure call. The new HP high-precision architecture does not. The mechanism of a procedure call is not architected, but instead is accomplished by a software convention using the simple hardwired instructions. This provides more flexibility in procedure calls and ultimately a more efficient call mechanism.

Procedure calls are more than just a branch and return in the flow of control. The procedure call mechanism must also provide for the passing of parameters, the saving of the caller's environment, and the establishment of an environment for the called procedure. The procedure return mechanism must provide for the restoration of the calling procedure's environment and the saving of return values.

The new HP machines are register-based machines, but

by convention a stack is provided for data storage. The most straightforward approach to procedure calls on these machines assumes that the calling procedure acquires the responsibility for preserving its state. This approach employs the following steps:

- Save all registers whose contents must be preserved across the procedure call. This prevents the called procedure, which will also use and modify registers, from affecting the calling procedure's state. On return, those register values are restored.
- Evaluate parameters in order and push them onto the stack. This makes them available to the called procedure which, by convention, knows how to access them.
- Push a *frame marker*. This is a fixed-size area containing several pieces of information. Among these is the *static link*, which provides information needed by the called procedure to address the local variables and parameters of the calling procedure. The return address of the calling procedure is also found in the stack marker.
- Branch to the entry point of the called procedure.

To return from the call, the called procedure extracts the return address from the stack marker and branches to it. The calling procedure then removes the parameters from the stack and restores all saved registers before program flow continues.

This simple model correctly implements the steps needed to execute a procedure call, but is relatively expensive. The model forces the caller to assume all responsibility for preserving its state. This is a safe approach, but causes too many register saves to occur. To optimize the program's execution, the compiler makes extensive use of registers to hold local variables and temporary values. These registers must be saved at a procedure call and restored at the return. The model also has a high overhead incurred by the loading and storing of parameters and linkage information. The ultimate goal of the procedure call convention is to reduce the cost of a call by reducing memory accesses.

The new compilers minimize this problem by introducing a procedure call convention that includes a register partition. The registers are partitioned into *caller-saves* (the calling procedure is responsible for saving and restoring them), *callee-saves* (the called procedure must save them at entry and restore them at exit), and *linkage* registers. Thirteen of the 32 registers are in the caller-saves partition and 16 are in the callee-saves partition. This spreads the responsibility for saving registers between the calling and called procedures and leaves some registers available for linkage.

The register allocator avoids unnecessary register saves by using caller-saves registers for values that need not be preserved. Values that must be saved are placed into registers from the callee-saves partition. At procedure entry, only those callee-saves registers used in the procedure are saved. This minimizes the number of loads and stores of registers during the course of a call. The partition of registers is not inflexible; if more registers are needed from a particular partition than are available, registers can be borrowed from the other partition. The penalty for using these additional registers is that they must be saved and restored, but this overhead is incurred only when many registers are needed, not for all calls.

In the simple model, all parameters are passed by being placed on the stack. This is expensive because memory references are made to push each parameter and as a consequence the stack size is constantly altered. The new compilers allocate a permanent parameter area large enough to hold the parameters for all calls performed by the procedure. They also minimize memory references when storing parameters by using a combination of registers and memory to pass parameters. Four registers from the callee-saves partition are used to pass user parameters; each holds a single 32-bit value or half of a 64-bit value. Since procedures frequently have few parameters, the four registers are usually enough to contain them all. This removes the necessity of storing parameter values in the parameter area before the call. If more than four 32-bit parameters are passed, the additional ones are stored in the preallocated parameter area. If a parameter is larger than 64 bits, its address is passed and the called procedure copies it to a temporary area.

Additional savings on stores and loads occur when the called procedure is a leaf routine. As mentioned previously, the optimizer attempts to maximize the use of registers to hold variable values. When a procedure is a leaf, the register allocator uses the caller-saves registers for this purpose, thus eliminating register saves for both the calling and called procedures. It is never necessary to store the return address or parameter registers of a leaf routine since they will not be modified by subsequent calls.

Leaf routines do not need to build a stack frame, since they make no procedure calls. Also, if the allocator succeeds in representing all local variables as registers, it is not necessary to build the local variable area at entry to the leaf procedure.

The convention prescribes other uses of registers to eliminate other loads and stores at procedure calls. The return address is always stored in a particular register, as is the static link if it is needed.

To summarize, the procedure call convention used in the new HP computers streamlines the overhead of procedure calls by minimizing the number of memory references. Maximal use of registers is made to limit the number of memory accesses needed to handle parameters and linkage. Similarly, the convention minimizes the need to store values contained in registers and does not interfere with attempts at optimization.

Myth: The simple instructions available in RISC result in significant code expansion.

Many applications, especially commercial applications, assume the existence of complex high-level instructions typically implemented by the system architecture in microcode or hardware. Detractors of RISC argue that significant code expansion is unavoidable since the architecture lacks these instructions. Early results do not substantiate this argument.[7,8] The new HP architecture does not provide complex instructions because of their impact on overall system performance and cost, but their functionality is available through other means.

As described in an earlier article,[2] the new HP machines

do not have a microcoded architecture and all of the instructions are implemented in hardware. The instructions on microcoded machines are implemented in two ways. At the basic level, instructions are realized in hardware. More complex instructions are then produced by writing subroutines of these hardware instructions. Collectively, these constitute the microcode of the machine. Which instructions are in hardware and which are in microcode are determined by the performance and cost goals for the system. Since HP's reduced instruction set is implemented solely at the hardware level, subroutines of instructions are equivalent to the microcode in conventional architectures.

To provide the functionality of the complex instructions usually found in the architecture of conventional machines, the design team developed the alternative concept of *millicode instructions* or routines. Millicode is HP's implementation of complex instructions using the simple hardware instructions packaged into subroutines. Millicode serves the same purpose as traditional microcode, but is common across all machines of the family rather than specific to each.

The advantages of implementing functionality as millicode are many. Microcoded machines may contain hidden performance penalties on all instructions to support multiple levels of instruction implementation. This is not the case for millicode. From an architectural viewpoint, millicode is just a collection of subroutines indistinguishable from other subroutines. A millicode instruction is executed by calling the appropriate millicode subroutine. Thus, the expense of executing a millicode instruction is only present when the instruction is used. The addition of millicode instructions has no hardware cost and hence no direct influence on system cost. It is relatively easy and inexpensive to upgrade or modify millicode in the field, and it can continue to be improved, extended, and tuned over time.

Unlike most microcode, millicode can be written in the same high-level languages as other applications, reducing development costs yet still allowing for optimization of the resultant code. Severely performance-critical millicode can still be assembly level coded in instances where the performance gain over compiled code is justified. The size of millicode instructions and the number of such instructions are not constrained by considerations of the size of available control store. Millicode resides in the system as subroutines in normally managed memory, either in virtual memory where it can be paged into and out of the system as needed, or in resident memory as performance considerations dictate. A consequence of not being bound by restrictive space considerations is that compiler writers are free to create many more specialized instructions in millicode than would be possible in a microcoded architecture, and thus are able to create more optimal solutions for specific situations.

Most fixed instruction sets contain complex instructions that are overly general. This is necessary since it is costly to architect many variations of an instruction. Examples of this are the MVB (move bytes) and MVW (move words) instructions on the HP 3000. They are capable of moving any number of items from any arbitrary source location to any target location. Yet, the compiler's code generators frequently have more information available about the operands of these instructions that could be used to advantage if other instructions were available. The code generators frequently know whether the operands overlap, whether the operands are aligned favorably, and the number of items to be moved. On microcoded machines, this information is lost after code generation and must be recreated by the microcode during each execution of the instruction. On the new HP computers, the code generators can apply such information to select a specialized millicode instruction that will produce a faster run-time execution of the operation than would be possible for a generalized routine.

Access to millicode instructions is through a mechanism similar to a procedure call. However, additional restrictions placed on the implementation of millicode routines prevent the introduction of any barriers to optimization. Millicode routines must be leaf routines and must have no effect on any registers or memory locations other than the operands and a few scratch registers. Since millicode calls are represented in SLLIC as pseudoinstructions, the optimizer can readily distinguish millicode calls from procedure calls. Millicode calls also use different linkage registers from procedure calls, so there is no necessity of preserving the procedure's linkage registers before invoking millicode instructions.

The only disadvantage of the millicode approach over microcode is that the initiation of a millicode instruction involves an overhead of at least two instructions. Even so, it is important to realize that for most applications, millicode instructions are infrequently needed, and their overhead is incurred only when they are used. The high-precision architecture provides the frequently needed instructions directly in hardware.

Myth: RISC machines must implement integer multiplication as successive additions.

Integer multiplication is frequently an architected instruction. The new architecture has no such instruction but provides others that support an effective implementation of multiplication. It also provides for inclusion of a high-speed hardware multiplier in a special function unit.[2]

Our measurements reveal that most multiplication operations generated by user programs involve multiplications by small constants. Many of these occurrences are explicitly in the source code, but many more are introduced by the compiler for address and array reference evaluation. The new compilers have available a trio of instructions that perform shift and add functions in a single cycle. These instructions, SH1ADD (shift left once and add), SH2ADD (shift left twice and add) and SH3ADD (shift left three times and add) can be combined in sequences to perform multiplication by constants in very few instructions. Multiplications by most constants with absolute values less than 1040 can be accomplished in fewer than five cycles. Negatively signed constants require an additional instruction to apply the sign to the result. Multiplication by all constants that are exact powers of 2 can be performed with a single shift instruction unless overflow conditions are to be detected. Additionally, multiplications by 4 or 2 for indexed address-

ing can be avoided entirely. The LDWX (load word indexed) and LDHX (load half-word indexed) instructions optionally perform unit indexing, which combines multiplication of the index value with the address computation in the hardware.

The following examples illustrate multiplication by various small constants.

```
Source code:
        4*k
Assembly code:
        SH2ADD   8,0,9          ; shift r8 (k) left 2 places,
                                  add to r0 (zero) into r9
Source code:
        -163*k
Assembly code:
        SH3ADD   8,8,1          ; shift r8 (k) left 3 places, add
                                  to itself into r1
        SH3ADD   1,1,1          ; shift r1 left 3 places, add to
                                  itself into r1
        SH1ADD   1,8,1          ; shift r1 left 1 place, add to
                                  k into r1
        SUB      0,1,1          ; subtract result from 0 to
                                  negate; back into r1
Source code:
        A(k)
Assembly code:
        LDO      -404(30),9     ; load array base address
                                  into r9
        LDW      -56(0,30),7    ; load unit index value into r7
        LDWX,S   7(0,9),5       ; multiply index by 4 and
                                  load element into r5
```

When neither operand is constant or if the constant is such that the in-line code sequence would be too large, integer multiplication is accomplished with a millicode instruction. The multiply millicode instruction operates under the premise that even when the operands are unknown at compile time, one of them is still likely to be a small value. Application of this to the multiplication algorithm yields an average multiplication time of 20 cycles,which is comparable to an iterative hardware implementation.

Myth: RISC machines cannot support commercial applications languages.

A popular myth about RISC architectures is that they cannot effectively support languages like COBOL. This belief is based on the premise that RISC architectures cannot provide hardware support for the constructs and data types of COBOL-like languages while maintaining the one-instruction-one-cycle advantages of RISC. As a consequence, some feel that the code expansion resulting from performing COBOL operations using only the simple architected instructions would be prohibitive. The significance of this is often overstated. Instruction traces of COBOL programs measured on the HP 3000 indicate that the frequency of decimal arithmetic instructions is very low. This is because much of the COBOL program's execution time is spent in the operating system and other subsystems.

COBOL does place demands on machine architects and compiler designers that are different from those of languages like C, Fortran, and Pascal. The data items provided in the latter languages are represented in binary and hence are native to the host machine. COBOL data types also include packed and unpacked decimal, which are not commonly native and must be supported in ways other than directly in hardware.

The usual solution on conventional machines is to provide a commercial instruction set in microcode. These additional instructions include those that perform COBOL field (variable) moves, arithmetic for packed decimal values, alignment, and conversions between the various arithmetic types.

In the new HP machines, millicode instructions are used to provide the functionality of a microcoded commercial instruction set. This allows the encapsulation of COBOL operations while removing the possibility of runaway code expansion. Many COBOL millicode instructions are available to do each class of operation. The compiler expends considerable effort to select the optimal millicode operation based on compile-time information about the operation and its operands. For example, to generate code to perform a COBOL field move, the compiler may consider the operand's relative and absolute field sizes and whether blank or zero padding is needed before selecting the appropriate millicode instruction.

Hardware instructions that assist in the performance of some COBOL operations are architected. These instructions execute in one cycle but perform operations that would otherwise require several instructions. They are emitted by the compiler in in-line code where appropriate and are also used to implement some of the millicode instructions. For example, the DCOR (decimal correct) and UADDCM (unit add complement) instructions allow packed decimal addition to be performed using the binary ADD instruction. UADDCM prepares an operand for addition and the DCOR restores the result to packed decimal form after the addition. For example:

```
r1 and r2 contain packed decimal operands
r3 contains the constant X'99999999'

UADDCM 1,3,31   ; pre-bias operand into r31
ADD    2,31,31  ; perform binary add
DCOR   31,31    ; correct result
```

Millicode instructions support arithmetic for both packed and unpacked decimal data. This is a departure from the HP 3000, since on that machine unpacked arithmetic is performed by first converting the operand to packed format, performing the arithmetic operation on the packed data, and then converting the result back to unpacked representation. Operations occur frequently enough on unpacked data to justify the implementation of unpacked arithmetic routines. The additional cost to implement them is minimal and avoids the overhead of converting operands between the two types. An example of the code to perform an unpacked decimal add is:

r1 and r2 contain unpacked decimal operands
r3 contains the constant X'96969696'
r4 contains the constant X'0f0f0f0f'
r5 contains the constant X'30303030'

```
ADD     3,1,31      ; pre-bias operand into r31
ADD     31,2,31     ; binary add into r31
DCOR    31,31       ; correct result
AND     4,31,31     ; mask result
OR      5,31,31     ; restore sum to unpacked decimal
```

In summary, COBOL is supported with a blend of hardware assist instructions and millicode instructions. The compiled code is compact and meets the run-time execution performance goals.

Conclusions

The Spectrum program began as a joint effort of hardware and software engineers. This early communication allowed high-level language issues to be addressed in the architectural design.

The new HP compiling system was designed with a reduced-complexity machine in mind. Register allocation, instruction scheduling, and traditional optimizations allow compiled programs to make efficient use of registers and low-level instructions.

Early measurements have shown that this compiler technology has been successful in exploiting the capabilities of the new architecture. The run-time performance of compiled code consistently meets performance objectives. Compiled code sizes for high-level languages implemented

An Optimization Example

This example illustrates the code generated for the following C program for both the unoptimized and the optimized case.

```
test ( )
{
int i, j;
int a1[25], a2[25], r[25][25];

for (i = 0; i < 25; i + +) {
    for (j = 0; j < 25; j + +) {
        r [i] [j] = a1 [i] * a2 [j];
    }
  }
}
```

In the example code that follows, the following mnemonics are used:

rp	return pointer, containing the address to which control should be returned upon completion of the procedure
arg0	first parameter register
arg1	second parameter register
sp	stack pointer, pointing to the top of the current frame
mret0	millicode return register
mrp	millicode return pointer.

The value of register zero (r0) is always zero.

The following is a brief description of the instructions used:

LDO	immed(r1),r2	r2←r1 + immed.
LDW	immed(r1),r2	r2←*(r1 + immed)
LDWX,S	r1(r2),r3	r3←*(4*r1 + r2)
STW	r1,immed(r2)	*(r2 + immed)←r1
STWS	r1,immed(r2)	*(r2 + immed)←r1
STWM	r1,immed(r2)	*(r2 + immed)←r1 AND r2←r2 + immed
COMB,<=	r1,r2,label	if r1 <= r2, branch to label
BL	label,r1	branch to label, and put return address into r1 (for procedure call)
BV	0(r1)	branch to address in r1 (for procedure return)
ADD	r1,r2,r3	r3←r1 + r2
SH1ADD	r1,r2,r3	r3←2*r1 + r2
SH2ADD	r1,r2,r3	r3←4*r1 + r2
SH3ADD	r1,r2,r3	r3←8*r1 + r2
COPY	r1,r2	r2←r1
NOP		no effect

In the following step-by-step discussion, the unoptimized code on the left is printed in black, and the optimized code on the right is printed in color. The code appears in its entirety, and can be read from the top down in each column.

Save callee-saves registers and increment stack pointer. Unoptimized case uses no register that needs to be live across a call.

LDO	2760(sp),sp	STW	2, -20(0,sp)
		STWM	3,2768(0,sp)
		STW	4, -2764(0,sp)

Assign zero to i. In the optimized case, i resides in register 19.

STW	0, -52(0,sp)	COPY	0,19

Compare i to 25. This test is eliminated in the optimized case since the value of i is known.

LDW	-52(0,sp),1
LDO	25(0),31
COMB,<=,N	31,1,L2

In the optimized version, a number of expressions have been moved out of the loop:

{maximum value of j}	LDO	25(0),20
{address of a1}	LDO	-156(sp),22
{address of a2}	LDO	-256(sp),24
{address of r}	LDO	-2756(sp),28
{initial value of 100*i}	LDO	0(0),4
{maximum value of 100*i}	LDO	2500(0),2

Initialize j to zero, and compare j to 25. This test has also been eliminated in the optimized version, since the value of j is known. Note that j now resides in register 21.

L3			
STW	0, -56(0,sp)	COPY	0,21
LDW	-56(0,sp),19		

in this low-level instruction set are comparable to those for more conventional architectures. Use of millicode instructions helped achieve this result. Complex high-level language operations such as procedure calls, multiplication, and COBOL constructs have been implemented efficiently with the low-level instructions provided by the high-precision architecture. A later paper will present performance measurements.

Acknowledgments

The ideas and results presented in this paper are the culmination of the work of many talented engineers involved with the Spectrum compiler program. We would like to acknowledge the individuals who made significant technical contributions to the work presented in this paper in the following areas: early compiler development and optimizer investigation at HP Laboratories, optimizer development, aliasing design and implementation in the compiler front ends, code generator design and implementation, procedure call convention design, and object module specification.

Megan Adams
Robert Ballance
Bruce Blinn
William Buzbee
Don Cameron
Peter Canning
Paul Chan
Cary Coutant
Eric Eidt
Phil Gibbons
Adiel Gorel
Richard Holman
Mike Huey
Audrey Ishizaki
Suneel Jain
Mark Scott Johnson


```
LDO          25(0),20
COMB,<=,N    20,19,L1
```

In the optimized version, the load of a1[i] is moved out of the inner loop, since the value of i is constant in the inner loop.

```
                                    LDWX,S    19(0,22),23
```

Register 28 contains the address of r, and register 4 contains the value 100*i, which is the offset of the ith row of array r. This is constant over the inner loop, and has been moved out.

```
                                    ADD       28,4,3
L6
```

The loop begins with the load of a1[i] into the first parameter register. This value has already been loaded in the optimized version, and need only be copied.

```
LDO          -156(sp),21
LDW          -52(0,sp),22
LDWX,S       22(0,21),arg0          COPY      23,arg0
```

The value of a2[j] is loaded into the second parameter register, and the multiply millicode instruction is called. In the optimized case, the address of a2[0] and the value of j are both already in registers.

```
LDO          -256(sp),1
LDW          -56(0,sp),19
BL           mull,mrp               BL        mull,mrp
LDWX,S       19(0,1),arg1           LDWX,S    21(0,24),arg1
```

Store the result into r[i][j]. The three SHxADD instructions calculate 100*i. Note that most of the following is loop invariant, and has been moved out of the loop in the optimized case.

```
LDO          -2756(sp),19 {address of r}
LDW          -52(0,sp),20 {value of i}
SH1ADD       20,20,21     {r21 ← 3*i}
SH3ADD       21,20,22     {r22 ← 25*i}
SH2ADD       22,0,1       {r1 ← 100*i}
ADD          19,1,31      {address of r + 100*i}
LDW          -56(0,sp),19 {value of j}
SH2ADD       19,31,20     {add j*4 to address}   SH2ADD   21,3,31
STWS         mret0,0(0,20) {store}               STWS     mret0,0(0,31)
```

Increment j.

```
LDW          -56(0,sp),21                       LDO       1(21),21
LDO          1(21),22
STW          22,-56(0,sp)
```

Compare j to the value 25 (already in register 20 in the optimized version). The position after the conditional branch contains no useful instruction in the unoptimized case. In the optimized version, the first instruction of the loop has been copied to this position, and the target adjusted to the following instruction. Because the branch has the nullification flag set (,N), the following instruction will not be executed when the branch is not taken.

```
LDW          -56(0,sp),1
LDO          25(0),31
COMBF,<=     31,1,L6                COMBF,<=,N  20,21,L6+4
NOP                                 LDWX,S      21(0,24),25
L1
```

Increment i, and test for the end of the loop. In the optimized version, induction variable elaboration has removed the 100*i multiplication, and added a new induction variable to contain that value. This value, in register 4, is now tested against a maximum value of 2500, contained in register 2. This branch has been scheduled like the previous branch.

```
LDW          -52(0,sp),19
LDO          1(19),20               LDO       1(19),19
STW          20,-52(0,sp)           LDO       100(4),4
LDW          -52(0,sp),21
LDO          25(0),22
COMBF,<=     22,21,L3               COMBF,<=,N 2,4,L3-4
NOP                                 COPY      0,21
L2
```

Finally, the registers are restored, and control is returned to the calling procedure.

```
                                    LDW       -2788(0,sp),2
                                    LDW       -2764(0,sp),4
BV           0(rp)                  BV        0(rp)
LDO          -2760(sp),sp           LDWM      -2768(0,sp),3
```

Steven Kusmer
Tom Lee
Steve Lilker
Daniel Magenheimer
Tom McNeal
Sue Meloy
Terrence Miller
Angela Morgan
Steve Muchnick
Karl Pettis
David Rickel
Michelle Ruscetta
Steven Saunders
Carolyn Sims
Ron Smith
Kevin Wallace
Alexand Wu

We feel privileged to have the opportunity to present their work. We would like to extend special thanks to Bill Buzbee for his help in providing code examples, and to Suneel Jain for providing the description of the optimization components.

References

1. D.A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, Vol. 28, no. 1, January 1985, pp. 8-21.
2. J.S. Birnbaum and W.S. Worley, Jr., "Beyond RISC: High-Precision Architecture," *Hewlett-Packard Journal*, Vol. 36, no. 8, August 1985, pp. 4-10.
3. A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.
4. G.J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *Proceedings of the SIGPLAN Symposium on Compiler Construction*, June 1982, pp. 98-105.
5. M. Sharir, "Structural Analysis: A New Approach To Flow Analysis in Optimizing Compilers," *Computer Languages*, Vol. 5, Pergamon Press Ltd., 1980.
6. D.S. Coutant, "Retargetable High-Level Alias Analysis," *Conference Record of the 13th ACM Symposium on Principles of Programming Languages*, January 1986.
7. J.A. Otto, "Predicting Potential COBOL Performance on Low-Level Machine Architectures," *SIGPLAN Notices*, Vol. 20, no. 10, October 1985, pp. 72-78.
8. G. Radin, "The 801 Computer," *Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 39-47.

Authors

January 1986

Jon W. Kelley

With HP since 1975, Jon Kelley has worked on BASIC and RPG compilers for the HP 300 Business Computer and on a prototype optimizer. He has also contributed to the development of code generators for HP 3000 Computers and for the Spectrum program. Jon graduated in 1974 from the University of California at Berkeley with a BA degree in computer science. He lives in Sunnyvale, California and lists fly-fishing, hunting, and flying as outside interests.

Deborah S. Coutant

Debbie Coutant earned a BA degree in psychology from the University of Arizona in 1977 and an MS degree in computer science from the University of Arizona in 1981. After joining HP's Information Networks Division in 1981, she worked on Pascal for HP 3000 Computers and later investigated compiler optimization techniques and contributed to the development of code generators and optimizers for the Spectrum program. She is the author of a paper on retargetable alias analysis and is a member of the ACM and SIGPLAN. Born in Bethpage, New York, Debbie lives in San Jose, California. She's married and enjoys playing the French horn in community orchestras. Her other outside interests include racquetball and camping.

Carol L. Hammond

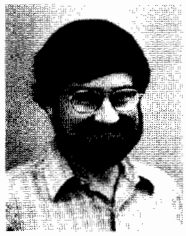
With HP since 1982, Carol Hammond manages an optimizer project in the computer language laboratory of HP's Information Technology Group. In earlier assignments at HP Laboratories she wrote architecture verification programs and worked on a compiler project. She is a member of ACM and SIGPLAN. Carol was born in Long Branch, New Jersey and studied physics at the University of California at Davis (BS 1977). She worked as a professional musician for four years before resuming her studies at the University of California at Berkeley, completing work for an MS degree in computer science in 1983. She lives in San Jose, California and still enjoys singing and playing the piano.

20 Measurement Plotting System

Thomas H. Daniels

With HP since 1963, Tom Daniels was project manager for the HP 7090A Measurement Plotting System and earlier was project manager for the HP 9872A Plotter. He coauthored an HP Journal article on the HP 9872A. Tom was born in Los Angeles, California and received a BSEE degree from Northrop University in 1963. He's now a resident of Escondido, California, is married, and has two children. His outside interests include woodworking and restoring old Chevrolet Vegas.

John Fenoglio



An R&D manager at HP's San Diego Division, John Fenoglio has been with the company since 1972. He was section manager for the HP 7090A Measurement Plotting System and has held various engineering, marketing, and management positions for analog and digital products. He was born in Texas and holds a BSEE degree from California State Polytechnic University (1972) and an MSEE degree from San Diego State University (1974). He is also the coauthor of an HP Journal article on the HP 7225A Plotter. John is a resident of San Diego, California and an adventure-seeker. He is a skier and scuba diver and has flown in exotic locations, from the polar ice caps to South American jungles. He also restores sports cars and is an amateur radio operator. He has bounced signals off the moon using a 40-foot dish antenna.

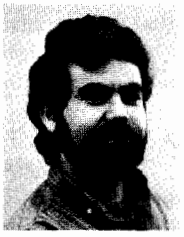
24 Measurement Graphics

Steven T. Van Voorhis

Steve Van Voorhis holds a BA degree in biology and psychology from the University of California at San Diego (1973) and an MA degree in electrical engineering from San Diego State University (1980). With HP since 1981, he's a project leader in the design graphics section of HP's San Diego Division and was a design engineer on the HP 7090A Measurement Plotting System. He was also a research assistant and design engineer at the University of California at San Diego and is the coauthor of three papers on human visual and auditory responses. A native of California, he was born in Los Angeles and now lives in Solana Beach. He and his wife and two sons enjoy spending time at the beach. He is also a soccer player and is remodeling his house.

27 Software

Emil Maghakian



With HP since 1979, Emil Maghakian has worked on HP 7310 Printer firmware, on ink-jet printer technology, and on software for various products, including the HP 7090A Measurement Plotting System. He was born in Tehran, Iran and studied computer science at Virginia Polytechnic Institute and State University, receiving his BS degree in 1976 and MS degree in 1978. He was an instructor at Hollins College before coming to HP. His professional interests include computer graphics and man-machine interface problems. A resident of Escondido, California, Emil is married and has two children. He's active in various Armenian organizations in San Diego and is interested in public speaking. He also enjoys soccer and aerobics.

Francis E. Bockman

A computer software specialist with HP's San Diego Division, Frank Bockman has investigated a computer-aided work system and charting modules and has contributed to the development of measurement graphics software for the HP 7090A Measurement Plotting System. He was born in San Diego, California, came to HP as a student intern in 1980, and completed his BA degree in computer science from the University of California at San Diego in 1982. His professional interests include computer graphics, vector-to-raster conversion, and image rendering. Frank lives in Escondido, California, is married, and has a daughter. He is active in his church and enjoys soccer, racquetball, woodworking, gardening, and wine tasting.

32 Analog Channel

Jorge Sanchez

With HP since 1980, Jorge Sanchez attended San Diego State University, completing work for a BSEE degree in 1977 and an MSEE degree in 1981. His work on the HP 7090A Measurement Plotter includes designing the analog channel and the calibration algorithms for the analog channel as well as contributing to electromagnetic compatibility design. He is now developing new products as an R&D project manager. His previous professional experience was with National Semiconductor Corporation and with NCR Corporation. Jorge was born in Tecate, Mexico and currently lives in San Diego, California with his wife and two children. He is an avid sports fan and enjoys running, swimming, playing the piano, and gardening.

36 Usability Testing

Daniel B. Harrington

Dan Harrington holds degrees in physics from Albion College (BA 1950) and the University of Michigan (MS 1951). With HP since 1972, he has held a variety of management and engineering positions. He has been a product manager and an applications engineer and is now a publications engineer. He also worked at another company on the development and marketing of a mass spectrometer. He is named coinventor for a patent on a mass spectrometer and inventor for three other patents on various topics. Born in Detroit, Michigan, Dan lives in Corvallis, Oregon, is married, and has three children. He is president of the local Kiwanis Club and likes photography, music, camping, and travel.