WRITABLE INSTRUCTION SET, STACK ORIENTED COMPUTERS: The WISC Concept

Philip Koopman Jr.
WISC Technologies, Inc.
Box 429 Route 2
La Honda, CA 94020

ABSTRACT

Conventional computers are optimized for executing programs made up of streams of serial instructions. Conversely, modern programing practices stress the importance of non-sequential control flow and small procedures. The result of this hardware/software mismatch in today's general purpose computers is a costly, sub-optimal, self-perpetuating compromise.

The solution to this problem is to change the paradigm for the computing environment. The two central concepts required in this new paradigm are efficient procedure calls and a user-modifiable instruction set. Hardware that is fundamentally based on the concept of modularity will lead to changes in computer languages that will better support efficient software development. Software that is able to customize the hardware to meet critical application-specific processing requirements will be able to attempt more difficult tasks on less expensive hardware.

Writable Instruction Set/Stack Oriented Computers (WISC computers) exploit the synergism between multiple hardware stacks and writable microcode memory to yield improved performance for general purpose computing over conventional processors. Specific strengths of a WISC computer are simple hardware, high throughput, zero-cost procedure calls and a machine language to microcode interface.

WISC Technologies' CPU/32 is a 32-bit commercial processor that implements the WISC philosophy.

INTRODUCTION

People buy computers to solve problems. People measure the success of computers by how much was saved by using a computer to solve their problems.

What is the expense of using a computer to solve a problem? Computers cost users not only money for hardware and software, but also resources for training, labor, and waiting for solutions (both during development and during use). In the early days, the cost of solving problems with computers was predominated by hardware costs. Miraculously, hardware costs have plunged even while capabilities have grown by leaps and bounds. As a result, the problems that

computers are solving (and the programs that solve them) have grown much more complex. This has lead to the dramatic shift in recent years of spending more time and money on computer software than on hardware.

Since expensive, complex software now dominates the cost of providing computer solutions to problems, much effort is going into changing the way software is written. These efforts often end up placing more demands upon hardware ("hardware is cheap"). Unfortunately, it never seems that hardware speed increases can quite keep up with added software demands ("software expands to fill all available computer resources"). Consequently, much research is being conducted on ways of making processors run programs more efficiently for any given hardware fabrication technology.

The premise of this paper is that there are two fundamental problems with current general-purpose software/hardware environments: a lack of efficient hardware support for procedure calls, and an inability to tailor hardware to applications based on software requirements. The WISC architecture described in this paper provides efficient hardware support for procedure calls by using a combination of two hardware stacks and a dedicated address field in the instruction format. The WISC architecture also supports cost-effective modification and expansion of instruction sets by providing writable microcode memory with a simple format.

This paper first describes some of the historical roots for the problems with conventional hardware/software environments, then describes the concepts, implementation, and implications of the WISC approach to providing a more unified hardware/software environment. Although much of this discussion is applicable to all computing environments, the scope of this paper is limited to general-purpose processing on single-processor computers.

THE HARDWARE/SOFTWARE EVOLUTION CYCLE

In order to see how the hardware environment can be poorly matched to the needs of the software environment, consider the historical pattern of steps in the hardware/software evolution cycle since the days of the first computers:

1) Profile existing software. How does a designer determine what instructions should be included in a new computer? Since the first use of most hardware is to run existing programs, the most scientific way to design an instruction set is to measure instruction execution frequencies on computers already in use. Such measurements usually reveal a preponderance of register manipulation

instructions and simple memory loads and stores.

2) Design a computer that efficiently executes existing software. When the new machine is built, it will use faster hardware and a larger memory to execute more complex (and memory-hungry) versions of existing programs faster. Compilers for existing languages will be modified to take advantage of the new hardware resources, and perhaps some new features will be tacked onto the local dialect of the language to make use of added hardware capabilities.

3) Write compilers that make new programs look like existing software. When a new language or a new dialect is developed, the compiler writer is interested in both improving the software environment and in generating efficient code. To accomplish these often divergent goals, compiler writers use optimization techniques to transform the source code into a program that will execute as efficiently as possible on available hardware. Since the hardware designed in steps 1 and 2 is optimized for certain types of operations, the output of these compilers will tend to use these same types of operations wherever possible.

Some of the most common optimizations that compiler writers use include unrolling loops into in-line code (figure 1a) and expanding the lowest level procedures as macros within calling routines (figure 1b). These two optimizations are important in our discussion, because they both tend to require increased program memory usage in exchange for increased execution speed. This is based on the almost universal assumption that hardware is most efficient at executing in-line code.

4) Write new applications using the new compilers (which produces more machine code optimized for existing hardware). When it comes time for new application programs to be written, programmers can be counted on to exploit all the strengths (and quirks) of the newly available compilers and hardware.

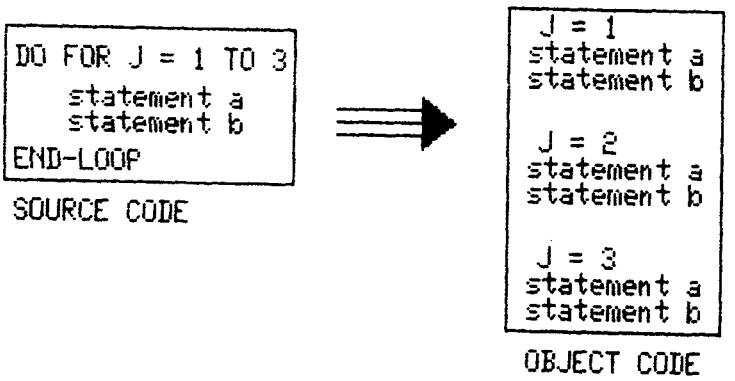

Figure 1a. Unrolling Loops.

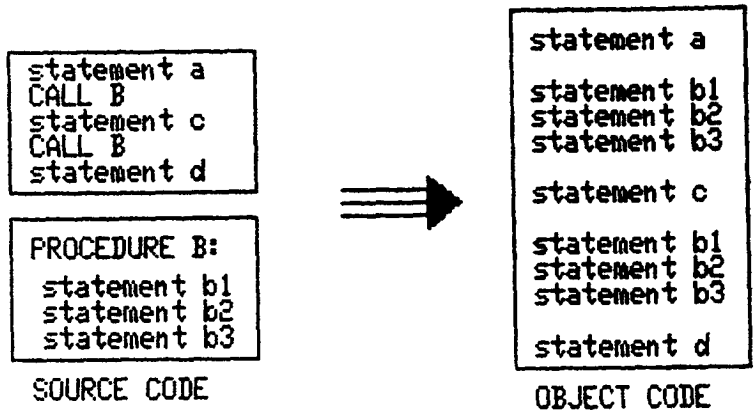

Figure 1b. Expanding procedures in-line.

Despite the insulating effects of high level languages between programmers and machines, programmers are uncomfortably aware of any software features that reduce performance. When programs perform poorly because they are not suitable for automatic compiler optimization, the user is compelled to re-write programs to avoid inefficient structures or buy a more powerful (and more expensive) machine. This tends to further skew usage statistics, since new machines are perceived to be more expensive than clever but shabby software techniques.

5) Go to step (1) above, and get yet another computer that is even better at running existing programs.

This development cycle clearly favors the propagation of initial biases in computer design to successive generations of machines. Could it be that years of pursuing this cycle has resulted in instruction sets that still favor the operations present in the early machines? Is this filtering process the real mechanism that lead to the concept of RISC architectures?

HARDWARE EVOLUTION

Having examined the process by which we ended up with today's computing environment problems, let us take a look at some of the evolutionary steps computer hardware architecture has taken along the way.

The history of computers has been a story of providing faster hardware with increased capacity in smaller packages with lower prices. The primary emphasis has been on reducing the cost of computing by reducing the cost to purchase and operate hardware. Measurements that indicate the cost effectiveness of hardware include the cost per megabyte of program memory and the cost per millions of instructions executed per second. From the point of view of

the purchaser, hardware becomes more of a bargain every year (or month, or even day).

There have been two central problems to be overcome in increasing hardware performance: arithmetic computation speed and memory access speed.

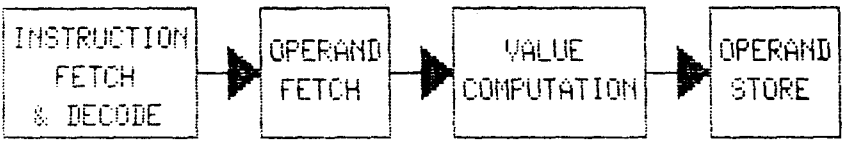

Figure 2a. Pipelining.

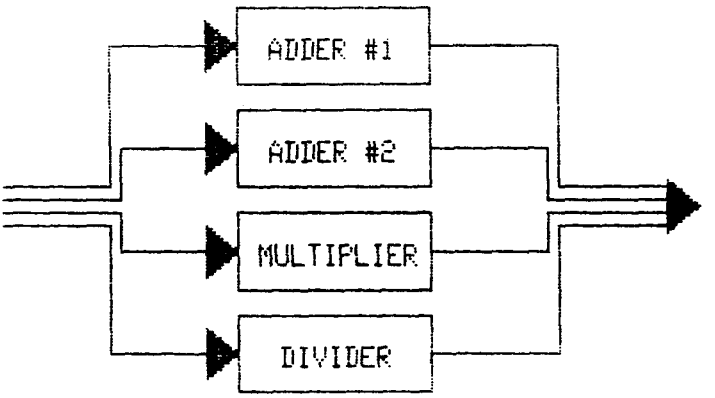

Figure 2b. Parallelism.

Arithmetic computation speed was a major problem in early computers. Originally, the arithmetic computation speed limitation was overcome by using pipelining (figure 2a) and parallelism within the system (figure 2b). For example, separate portions of a processor could concentrate on fetching instructions, fetching operands, computing values, and storing results (pipelining). Additionally, individual hardware adders, multipliers, and dividers could work simultaneously on data within the computation section of the processor (parallelism). Recently, the increasing speed and complexity of VLSI circuitry (and especially the availability of inexpensive, fast floating point arithmetic chips) have greatly reduced arithmetic computation speed as a problem in general purpose programing.

As the time to perform arithmetic operations has been reduced, main memory access speed has emerged as the leading speed bottleneck. Historically, there have always been two kinds of memory available to computer designers: small high-speed memory, and slow bulk memory. Today, the trend continues. Affordable high capacity memory chips leap by factors of four in size every few years with modest increases in speed. Fast static memory increases moderately

in size, but increases dramatically in speed.

As CPU speeds have outstripped bulk memory speeds, memory bandwidth limitations have become more severe. There are two ways to solve this problem: speed up average memory access time, and increase the amount of work done per memory access. Cache memory decreases average memory access time at the cost of added complexity by using the small, high speed memory devices to retain copies of instructions and/or data that are likely to be needed by the CPU. Caching schemes usually rely on the concept of locality: programs tend to execute instructions in sequence, and tend to access data in clumps.

Other techniques to speed memory access include interleaving banks of memory and pre-fetching opcodes beyond the current operation being executed. Both methods tend to increase speed for sequentially executing programs at the cost of added hardware complexity. Separate data and program memories can also increase available memory bandwidth, but are beyond the scope of this paper.

The second method of reducing the effects of a memory access bottleneck is the technique of increasing the average amount of work done by each opcode fetched from memory. This has lead to the development of what is now called the Complex Instruction Set Computer (CISC) machine. CISC machines are based on the concept of reducing the semantic gap between high level language source code and its corresponding machine code. The theory is that if a high level language specifies a complex operation such as a character string move, it should be able to communicate this operation with a single machine instruction and consume only one memory cycle for opcode fetching. A simple, non-CISC machine would have to synthesize a complex operation from a sequence of simple instructions (consuming multiple memory cycles for opcodes), resulting in a semantic gap between the intent of the high level language and the way the intent

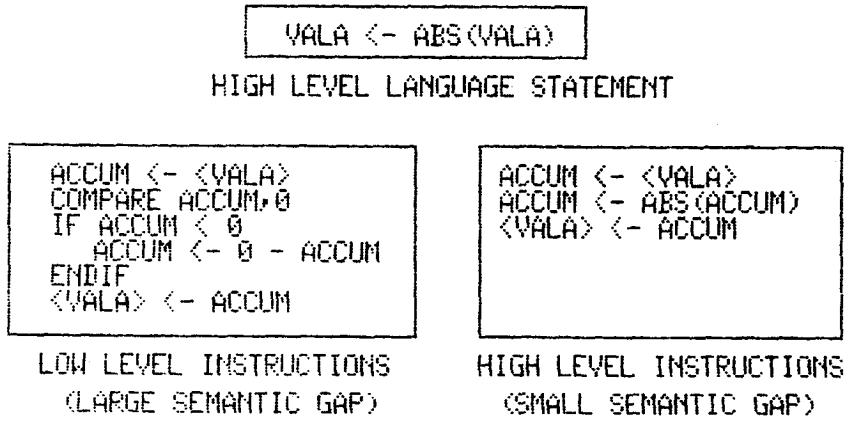

Figure 3. Semantic Gap.

must be communicated to the machine (figure 3). Some other examples of complex instructions supported in modern CISC architectures include frame based procedure parameter passing, array address calculation, and linked list pointer maintenance.

As instruction sets have become more complex, hard-wired computers that decode and execute instructions by using only logic gates have become too complex to be cost effective for most applications. Consequently, the use of microcoded machines has come to dominate the computer industry.

Microcoded computers execute several fast low-level instructions (called micro-instructions) to interpret and execute each machine instruction. Since each machine instruction may invoke a sequence of one or more micro-instructions, microcoded designs allow straightforward implementation of the complex instructions of a CISC machine. As the instruction set grows in size and complexity, microcoded designs simply increase the size of the ROM or RAM for storing micro-programs. Since microcoded designs store the mechanism for decoding and executing instructions in memory instead of as a network of logic gates, many design errors may be corrected simply by changing the microcode of the machine. This provides a significant savings in development time and cost over making changes to logic gates in a hard-wired computer design.

Since adding instructions is relatively inexpensive in microcoded CISC machines, these machines usually attempt to reduce the size of the semantic gap by providing an abundance of complicated instructions designed to directly implement high level language functions. Unfortunately, as the semantic gap is reduced in this manner, CISC machines run into a different problem: semantic mismatch.

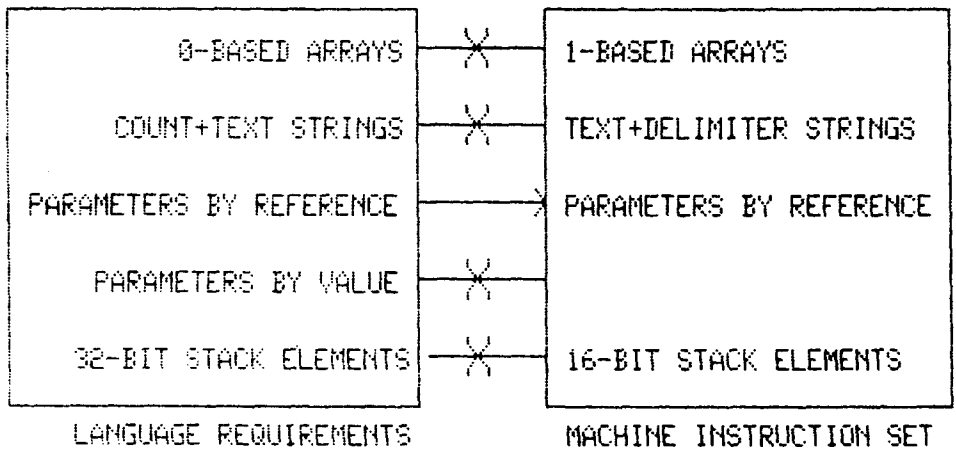

Figure 4. Semantic mismatch.

Semantic mismatch take places when a complex machine instruction doesn't exactly match the requirements of the high level language being used (figure 4). Semantic mismatch usually occurs because real-life CISC machines have a single instruction set that must meet the requirements of many diverse programing languages and application programs. This means that the instruction set is, of necessity, a compromise.

Examples of how languages differ in their requirements include: zero-based versus one-based array addressing, procedure stack frame parameter organization, linked list pointer organization, and string count and delimiter organization. In addition, new complicated instructions are often not smart enough to efficiently handle special degenerate (but frequent) cases such as parameterless procedure calls. As a result, compilers often ignore many of the very complex instructions added (at considerable effort) to new machines. Most compiled programs tend to use simple to moderately complex instructions.

The result of using the above approaches to increasing hardware power has been that most machines are well adapted to executing sequential programs of medium level complexity instructions.

SOFTWARE EVOLUTION

In early computers, hardware cost so much and was so scarce that any amount of programing effort was justifiable just to get an answer. As hardware has become less expensive, programs have become more complex, and software has grown tremendously in complexity and cost. Today, software is by far the most expensive part of any complex computer-based solution to a problem.

Most programing is now done in high level languages. There are two broad classes of high level languages in use: special purpose languages and general purpose languages.

Special purpose languages such as LISP, Prolog, and Smalltalk are based on computation models that stress unconventional approaches to problem solving. They typically do not address the issue of computational efficiency on general purpose computers. These languages tend to trade computational efficiency for flexibility and freedom of expression for specific tasks. Since these languages are often developed in research environments with ready access to powerful computers, computational efficiency is not a primary consideration.

While special purpose languages are important for their application areas, the very same features that make them powerful as a programing tool are the very things that make them perform poorly on limited resource conventional

computers. Some of the special features are dynamic memory management (especially garbage collection), run-time operand binding, and inter-procedure communication protocols. Today's trend is to either provide language-specific hardware, or more powerful but more expensive than average hardware to run programs written in these languages.

Most application programs are written in general purpose languages such as FORTRAN, BASIC, COBOL, Pascal, C, and Ada. The early high level programing languages such as FORTRAN were direct extensions of the philosophy of the machines they were run on: sequential Von Neumann machines with registers. Consequently, these languages and their general usage have developed to emphasize long sequences of assignment statements with only occasional conditional branches and procedure calls.

In recent years, however, the complexion of software has begun to change. The currently accepted best practice in software design centers around structured programing using modular designs. On a large scale, the use of modules is essential for partitioning tasks among programmers. On a smaller scale, procedures control complexity by limiting the amount of information that a programmer must deal with at any given time.

Procedures (often called subroutines) started out in early computers as a memory-saving device used at the cost of reduced execution speed. In modern programing languages, the importance of using procedures for software productivity is taken for granted; memory savings are an almost incidental advantage.

Modern languages such as Modula-2, Pascal, and Ada are designed specifically to promote modular design. The one hardware innovation that has resulted from the increasing popularity of these structured languages has been a register used as a stack pointer into main memory. With the exception of this stack pointer and a few complex instructions (which are not always usable by compilers), hardware has remained basically unchanged. Because of this, the machine code output of optimizing compilers for modern languages still tends to look a lot like output from earlier, non-structured languages.

Herein lies the problem. Conventional computers are still optimized for executing programs made up of streams of serial instructions. Execution traces for most programs show that procedure calls make up a rather small proportion of all instructions. Conversely, modern programing practices stress the importance of non-sequential control flow and small procedures. The clash between these two realities leads to a sub-optimal, and therefore costly, hardware/software environment on today's general purpose computers.

This does not mean that programs have failed to become

more organized and maintainable using structured languages, but rather that efficiency considerations and the use of hardware that encourages writing sequential programs has prevented modular languages from achieving all that they might. Although the current philosophy is to break programs up into very small procedures, most programs still contain fewer, larger, and more complicated procedures than they should.

How many functions should a typical procedure have? In Psychology of Communication: Seven Essays, George Miller gives strong evidence that the number seven (plus or minus two) applies to many aspects of thinking. The way the human mind copes with complicated information is by chunking groups of similar objects into fewer, more abstract objects. In a computer program, this means that each procedure should contain approximately seven fundamental operations (such as assignment statements or procedure calls) in order to be easily grasped. If a procedure contains more than seven distinct operations, it should be broken apart by chunking related portions into subordinate procedures to reduce the complexity of each portion of the program. In another part of the book, George Miller shows that the human mind can only grasp two or three levels of nesting of ideas within a single context. This strongly suggests that deeply nested loops and conditional structures should be arranged as nested procedure calls, not as convoluted indented structures within a procedure.

The only question now is, why don't most programmers follow these guidelines?

The most obvious reason that programmers avoid small, deeply nested procedures is the cost in speed of execution. Subroutine parameter setup and the actual procedure calling instructions can swamp the execution time of a program if used too frequently. All but the most sophisticated optimizing compiler can not help if procedures are deeply nested, and even those optimizations are limited. As a result, efficient programs tend to have a relatively shallow depth of procedure nesting.

Another reason that procedures are not used more is that they are difficult to program. Often times the effort to write the pro-forma code required to define a procedure makes the definition of a small procedure too burdensome. When this awkwardness is added to the considerable documentation and project management obstacles associated with creating a new procedure in a big project, it is no wonder that average procedure sizes of one or two pages are considered appropriate.

There is deeper cause why procedures are difficult to create in modern programing languages, and why they are used less frequently than the reader of a book on structured programing might expect: conventional programing languages

and the people who use them are steeped in the traditions of batch processing. Batch processing gives little reward in testability or convenience for working with small procedures. Truly interactive processing (which does not mean doing batch-oriented edit-compile-link-execute-crash-debug cycles from a terminal) is only available in a few environments, and is not taught to any large extent in universities.

As a result of all these factors, today's programing languages provide some moderately useful capabilities for efficient modular programing. Today's hardware and programing environments unnecessarily restrict the usage of modularity, and therefore unnecessarily increase the cost of providing computer-based solutions to problems.

UNIFICATION OF SOFTWARE AND HARDWARE

Developments in the conventional programing environment may be reaching a dead end. Recent uniprocessor hardware innovations tend to focus on either special purpose processing for symbol manipulation or distilling conventional machine instruction sets with yet another pass through the analysis-implementation-programing cycle discussed earlier.

The premise of this paper is that there is still considerably more mileage to be gained from uniprocessors by breaking out of the past cycles and looking at the hardware/software problem as a whole. The answer lies not with a new hardware architecture that mirrors current software, nor in changing software to suit current hardware. The answer lies in a redefinition of how we think about hardware and software. In this manner, we can aspire to achieve a unified hardware/software computing environment.

The first step in defining a unified general purpose computing environment is to take to heart the philosophy of breaking a problem up into smaller sub-problems. Instead of building a computer that supports procedure calls as special operations, what if we design a computer to expect subroutine calls as its primary mode of operation?

Implementing this idea results in a machine that is unlike conventional processors in a very fundamental way: it is designed for non-sequential program execution. It becomes a "tree processing machine". Programs are no longer lists of sequential instructions with occasional branches and procedure calls (figure 5). Programs are now organized as a tree structure, with each instruction containing operations and/or pointers to lower level nodes in the tree (figure 6). In such a machine, the very notion of a program "counter" becomes obsolete.

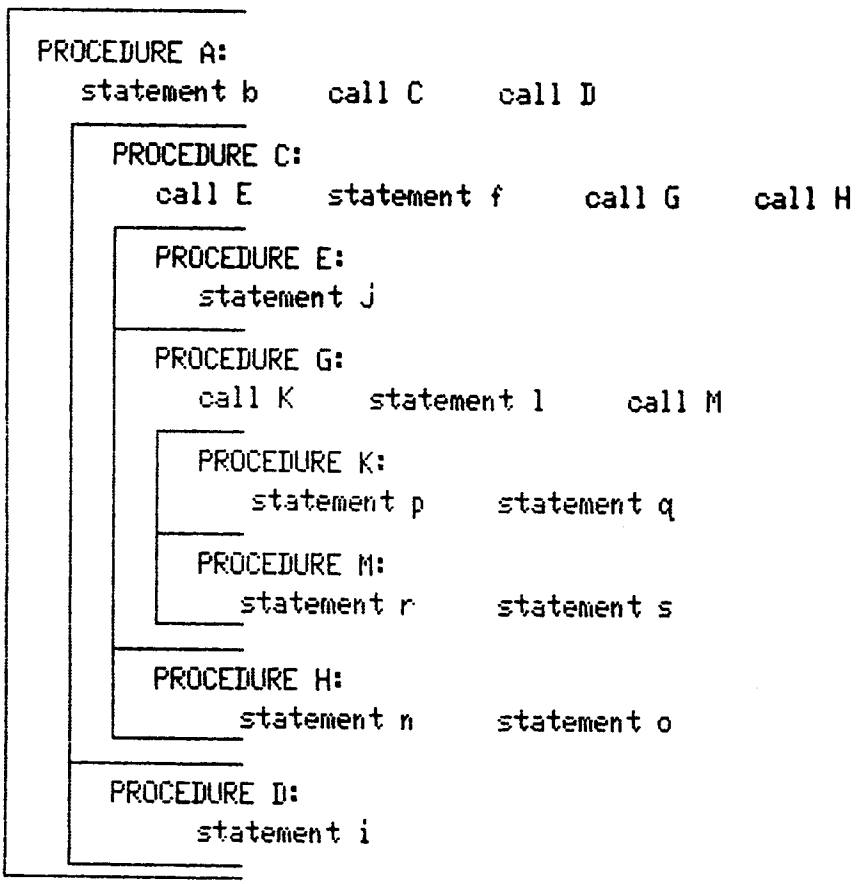

Figure 5. A typical sequential program

If this machine could actually process procedure calls simultaneously with other operations, modularity in programs would not be penalized. Such a machine would encourage better software design, and could fundamentally alter the way programmers think about programs.

Now that we have the concept of hardware that is efficient at implementing software procedures, how can we change the software to better match the hardware? The answer to this question lies in the concept of a modifiable microcoded instruction set.

As discussed previously, reducing the semantic gap of a processor can increase processing speed by reducing memory bandwidth requirements. The only pitfall is that if a pre-defined instruction set does not closely match the requirements of a language or application program, semantic mismatch negates the usefulness of many complicated instructions. Since general purpose machines are expected to perform well on a wide variety of problems in many different languages, the answer is to change the instruction set as required to suit each application program. This is most easily done with a writable microcode memory (often called writable control store).

With writable microcode memory, the user can modify the instruction set of the machine to fit each application

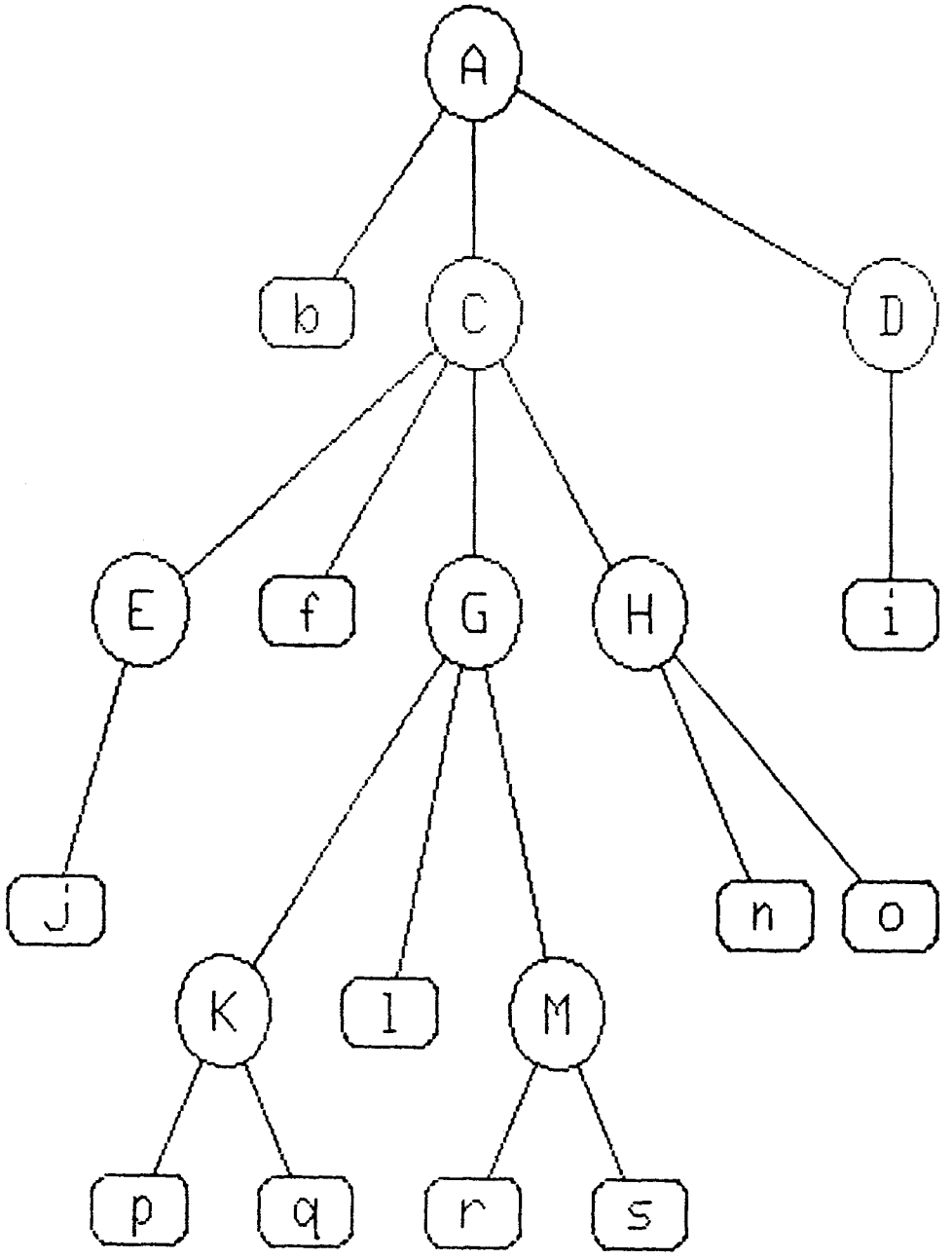

Figure 6. A typical program tree

program or programing language support environment. Applications can be initially written using a simple, generic instruction set. Then new instructions can be added to eliminate performance bottlenecks in heavily used code sequences.

The combination of tree-processing hardware with software that can modify the machine's instruction set for best efficiency can produce unexpected benefits in both hardware and software performance. The next section discusses an architectural approach to implementing such a machine, and the benefits that may be derived.

THE WISC APPROACH

The Writable Instruction Set Computer (WISC) approach to computer design provides a computer that efficiently supports the integrated hardware/software development environment just discussed. A WISC machine has high-speed procedure processing capability along with the capability to redefine the instruction set. WISC machines implement these goals by using multiple hardware stacks for operand and procedure return address storage, and writable microcode memory for storing the instruction set definitions. WISC machines also have a fixed instruction format for simplicity and speed of operation, and strive to meet the criterion of usefully employing all available memory cycles.

Once the decision is made to use a hardware stack in a design, an interesting and somewhat unexpected cascade of benefits is realized. These benefits lead to the architectural features of WISC machines.

The WISC machine discussed in this paper uses two hardware stacks: one for data parameters and one for return parameters. The first benefit of using these hardware stacks is that the overhead cost normally associated with procedure calls is greatly reduced. During a procedure call, the hardware return stack eliminates the need to save a return address to main memory. Additionally, the hardware data stack eliminates the need to save registers and data values to memory and/or fetch procedure input parameters from memory within a procedure.

Now, however, the unexpected benefits begin to accrue. A pure stack machine has no need for parameters with opcodes (except for memory addresses.) Since all operations are relative to the current position of the stack pointer, each opcode can be a simple parameterless field of five to ten bits. This greatly simplifies instruction decoding logic since implicit operands eliminate the need for explicit addressing modes, register specifications, etc. In a microcoded machine, this means that the opcode can directly access a microcode word with no decoding logic. All this

makes the hardware simpler, faster, and less expensive to develop and manufacture.

Since intermediate operands are kept on the hardware data stack, each microcoded instruction need take only one memory reference cycle (with loads and stores taking two memory cycles). Since microcoded primitives can be kept simple enough to execute within a single memory access cycle, there is no need for a complex pipeline to perform decoding, operand-fetching, execution, and result storage. A simple overlapped instruction fetch/decode and instruction execution strategy is quite ample to use all available memory bandwidth.

As an added bonus of using a stack-oriented instruction set, procedure calls may be made at zero cost in execution time for most cases. Since a stack-oriented opcode need only take roughly one-quarter of a 32-bit instruction word, the remaining instruction word bits are available to use as a procedure branching address (figure 7). If an overlapped fetch/decode and execution strategy is used, procedure calls, procedure returns, and unconditional branches may be processed in parallel with operation decoding.

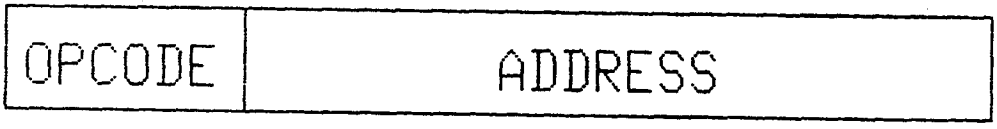

Figure 7. Generic WISC instruction format.

Now add the power of a changeable microcoded instruction set to the hardware stack machine just described. Since a fixed instruction format stack machine is free from complex opcode format interpretation and other complications, the hardware design is simple. And, simple hardware means simple microcode.

One problem with the few writable instruction sets available on current machines is that the microcode is just too hard to write. Microcode formats of 48 to 128 bits are very common. That's a lot of complexity for a programmer to handle! In fact, such complex microcode formats make expectations of customizing instruction sets for applications unrealistic. As will be shown later, a single-format 32-bit micro-instruction format is more than sufficient for a WISC machine.

Since a WISC architecture can be designed with a simple microcode format, moderately sophisticated users (such as compiler writers) can customize the architecture to meet their needs. Use of writable microcode memory leads to an increase in semantic content (and therefore a reduction of the semantic gap) for each instruction, and therefore more work done per memory access. It also eliminates the problem

of semantic mismatch, since the instruction set can be modified to suit the quirks of any application or language-support environment.

There is yet another benefit to the WISC approach. The combination of hardware stacks with writable microcode memory results in the blurring of the boundaries between high level programs, machine code, and microcode.

Consider the conventional processor. High level structured programs are converted from groups of procedures with stack-oriented local variables to machine code. A considerable change in the look and feel of the program takes place as high level language operations are transformed into groups of primitive operations. While a complex machine instruction set may support such stack operations as frame pushes and pops, and even fetch a variable given a frame pointer and an offset, the paradigm switches from variables and frames in high level languages to registers and memory pointers in machine code.

The means of passing information between many high level language procedures is the stack. The way of passing information between conventional machine language instructions is through registers or discrete memory locations. The fundamental mechanisms are completely different. If an instruction could be added to microcode memory to replace a procedure, it would result in re-writing the calling code to format the operands in registers instead of in a stack frame.

Now consider a WISC machine. WISC machines accomplish efficient procedure calling in part by the use of a data stack to pass information from calling programs to procedures. WISC instruction formats are greatly simplified by using this same data stack for holding operands. This means that a procedure can be transparently replaced with a microcoded primitive by simply replacing the procedure call with an opcode. There is no impact to any other aspect of the source code. This not only simplifies the substitution of microcoded primitives for high level source code fragments, but can actually lead to a view of microcode memory as a cache memory for frequently used operations.

In practice, this view of microcode memory as a cache memory allows the developer to selectively optimize the hardware for each application. This could be done by pencil and paper analysis of the program, or by using execution profiling software to create a histogram of execution frequencies for each section of code. The most heavily executed procedures can then be partly or wholly transferred from high level code to microcode, resulting in a significant speed increase. In the case of providing run-time support for the output of a compiler, the microcoded instruction set can be tailored to exactly implement the types of operations supported by the language. In either of

these cases, the microcode becomes a sort of cache memory for storing the operations that need to be executed most frequently.

This view of microcode memory as a sort of instruction cache is the final link of a chain that transforms a WISC machine to something beyond a conventional processor; it makes the WISC machine into a tree processing machine that merges all levels of processing into a unified hardware/software environment. Instead of representing programs as sequences of in-line instructions that are occasionally interrupted by procedure calls, the WISC processor views programs as an orderly nested series of procedure calls, with the final level of procedure call being a call to microcode memory.

Now that WISC machines are viewed as tree processors, several changes in programming take place. If a suitable microcoded instruction set is used, compiled object code can closely correspond to the original source code, resulting in simpler and more efficient compilers and debugging tools. There is no mismatch between the high level language source code and the actual machine code executed at run time.

Additionally, procedures are not viewed by the programmer as a collection of in-line code fragments, but rather as tree structure. The branches of this tree structure represent the control flow structure of the program (procedure calls, returns, and jumps). The leaves of the tree are represent procedure calls into microcode (figure 6 above).

From the above features we can see that a WISC machine uses simple, and therefore fast hardware to execute high semantic content instructions that closely reflect the structure of the software. Programmers are not penalized for organizing programs into small, understandable procedures. This results in compact tree-oriented program structures which are composed of hierarchically arranged solutions to sub-problems. Thus programs can be simultaneously optimized for small memory space, fast execution speed, and low development cost. This allows the hardware/software environment to deliver cost-effective solutions to the user's problems.

DESIGN OF A 32-BIT WISC MACHINE

In order to reify the conceptual design just presented, it is necessary to define the high level design of a WISC machine. For the purposes of this paper, the design of a 32-bit WISC machine called the CPU/32 will be discussed in detail.

It turns out that after a WISC machine is specified as having hardware stacks and a writable instruction set, the

single most important part of the design is the instruction format. The key to high-speed processing with zero-cost procedures is to use a fixed length instruction format that contains both an opcode and a procedure address.

The CPU/32 uses a 9-bit opcode (figure 8). These 9 bits can form the page address for a page of microcode memory, eliminating virtually all instruction decoding logic. This allows for up to 512 opcodes in the machine.

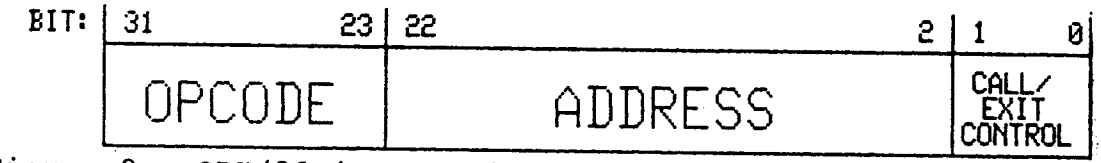

Figure 8. CPU/32 instruction format.

The remaining 23 bits of the 32 bit instruction format are dedicated to address and control information. If all instructions are aligned on byte boundaries that are evenly divisible by 4, then the high 21 bits of the remaining 23 bits in the instruction can address an instruction word in memory (with the low order 2 address bits masked to 0). The lowest order 2 bits of each instruction can then be used as a branching mode selection: procedure call, procedure return, or unconditional jump. These 23 bits can be used to execute an unconditional jump, procedure call, or (ignoring the address field) procedure return in parallel with opcode execution. The CPU/32 can process procedure calls for free!

As additional embellishments, this instruction format allows tail-end recursion elimination by substituting an unconditional branch for a procedure call as the last instruction of a procedure, and facilitates conditional branching and looping by having the branch destination address readily available.

The CPU/32's block diagram is shown in figure 9. The CPU/32's resources include a data stack, an ALU with a data register (Data Hi) and a transparent latch, an auxiliary (Data Lo) register that can connect with the Data Hi register for 64-bit shifting, a return stack with a bi-directional data path to the memory addresser for procedure call address manipulation, a memory addresser, program memory, and microcoded controller. All of the resources are connected to a central data bus, with access to I/O services through an appropriate host interface. All data paths and registers in the CPU/32 are 32-bits wide.

There are several interesting aspects to the CPU/32. One feature that is not always found on hardware-based stack designs is that the Data Hi register above the ALU can hold the top data stack element. This allows the use of a single-ported data stack RAM. Another is that the stack pointers can be loaded with values from the data bus. This

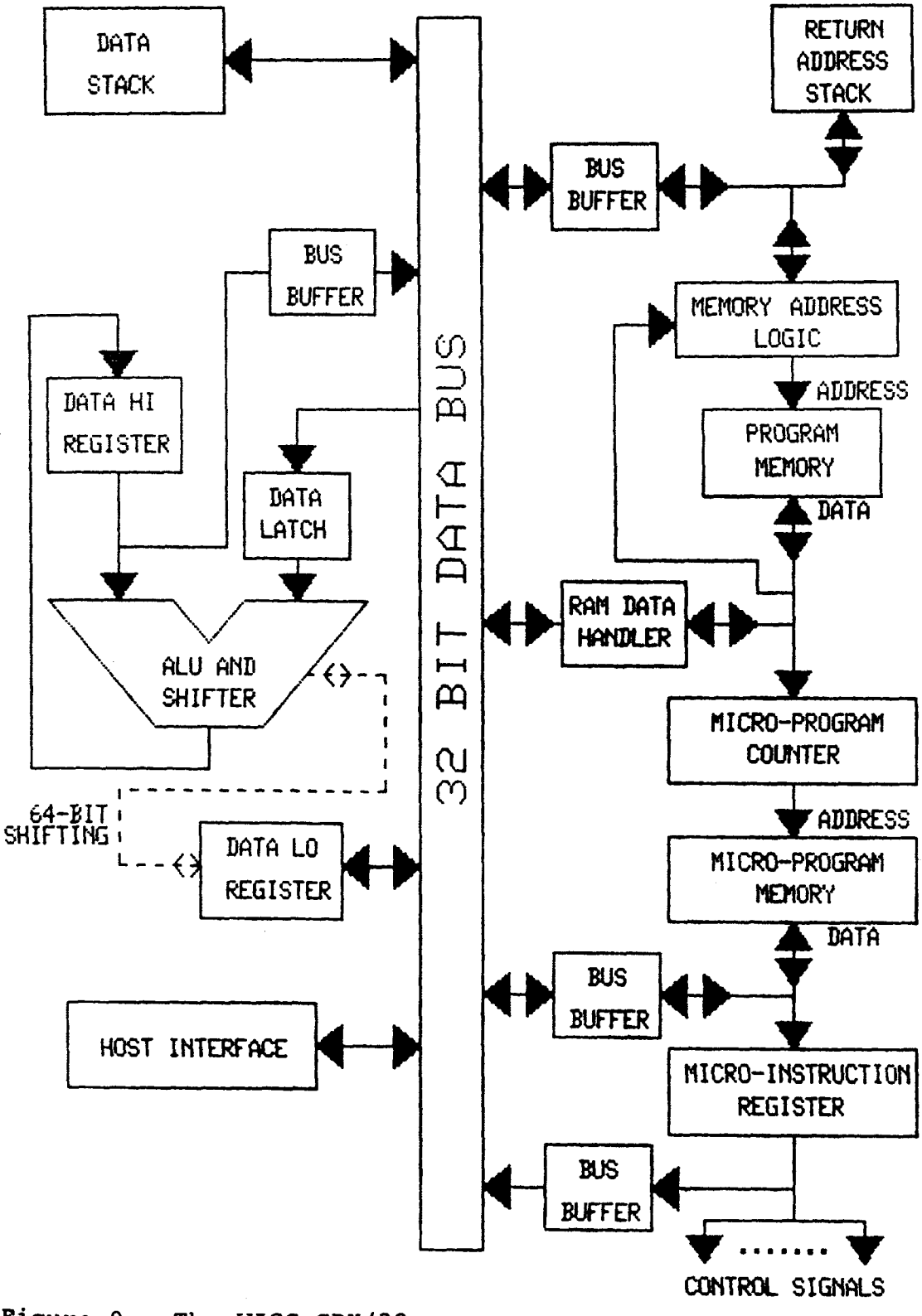

Figure 9. The WISC CPU/32.

makes accessing deeply buried stack elements relatively easy by eliminating the need for repetitive stack pushing and popping.

The use of a transparent latch on the ALU inputs allows connecting any data bus resource to one side of the ALU within one clock cycle, but also allows the latch to retain an intermediate value without disturbing the contents of the Data Hi register. This capability results in a savings of a clock cycle any time the top of stack element in Data Hi needs to be swapped with a cell in the data stack RAM.

The CPU/32 has no program counter. Each instruction contains the address of the next instruction. The only exception to this is when procedure returns are being processed, in which case the return stack value is passed directly through the memory address logic to access the next sequential instruction in the calling program.

While there is no program counter, there is an incrementer within the program memory logic that is used to add a one word displacement to procedure call addresses before they are saved on the stack. This incrementing is required in order to generate correct return addresses. The incrementer is also useful in block memory moves.

The micro-instruction register forms a one-stage micro-instruction pipeline that eliminates wasted time which would otherwise result from waiting for micro-program memory access. The only drawbacks to this design are that a two micro-cycle minimum is imposed on all op-codes, and delayed micro-instruction branches must be used for condition code testing. However, the small, high speed memory used to implement the micro-program memory and data stack memory allows for two micro-code cycles within each memory cycle time, essentially eliminating the impact of these drawbacks on system performance.

The micro-instruction format is shown in figure 10. Each micro-instruction uses 30 of the available 32 bits.

The entire instruction decoding path, from the return address stack all the way through to the micro-instruction register, is totally independent of the data bus. This allows ALU and data stack operations to proceed while simultaneously fetching and decoding instructions. This structure allows nearly 100% of the memory bandwidth to be used productively.

In the CPU/32, each instruction is fetched and decoded during a two micro-cycle period, waits in the micro-instruction pipeline for one clock cycle, then executes in two or more additional microcycles. The average instruction execution rate is just under one instruction per two micro-cycles.

BITS	USAGE
0-3	Bus source select
4-7	Bus destination select
8-9	Data stack pointer control
10-11	Return stack pointer control
12-13	ALU multiplexer shift control
14-15	unused
16-19	ALU function select
20	ALU mode select
21	ALU carry-in & shift-in
22-23	Data Lo register shift control
24-26	Microcode conditional branch select
27-28	Microcode next address generation
29	Increment microcode page register
30	Fetch & decode next macro-instruction
31	Memory address increment control

Figure 10. CPU/32 micro-instruction format.

An interesting software implication of the opcode format and system design is that opcodes interspersed with procedure calls must be compacted into single instructions in order to get zero-cost procedure calls. The procedure call in each instruction takes effect after the opcode has been completed. The only times that procedure calls are not zero-cost are in deeply nested procedures where there are no opcodes before the first procedure call in each successive level. Subroutine returns are zero-cost if the last instruction in a procedure is an opcode reference.

A possible compiler optimization that can easily increase efficiency is the substitution of an unconditional branch for a procedure call if the last primitive within a procedure is itself a procedure call (this is often called tail-end recursion elimination). Another possible optimization is a "bubbling-up" of the first opcode of a procedure to a calling program when the calling program would otherwise be forced to execute a null op-code in a series of consecutive procedure calls.

The system software for the CPU/32 obviously plays an important part in the establishment of a productive computing environment. While languages such as C are very well suited to the WISC architecture, eventually a new language will evolve to exploit the new capabilities of tree-oriented processors. Such a language would likely have: small, easily defined procedures; interactive development, compilation, and testing at the procedure level; easy access to a microcode assembler; extensibility of both data and compiler control structures; a high level infix syntax; a library of commonly needed functions; and support for module archiving and reuse.

THE WISC TECHNOLOGIES CPU/32

Now that the design for the CPU/32 has be presented, the question is, can such a machine actually be built? The answer is, of course, yes. WISC Technologies' CPU/32 is a commercial system that implements all of the philosophy and architectural features discussed in this paper.

Additional CPU/32 implementation features not previously discussed are a DMA memory transfer capability with the host computer, hardware and software interrupt support, and support for byte-oriented memory access.

CONCLUSION

WISC Technologies' CPU/32 is an implementation of a new way of thinking about computing environments: tree-organized program structures that emphasize modular programing for general-purpose computing. Preliminary use of WISC machines indicates that performance is equal to or better than other high-performance general purpose uniprocessors over broader classes of problems than might be expected. In particular, expert system programs with their tree-traversal emphasis are particularly well suited to WISC-type architectures.

If the past patterns of hardware and software evolution can be broken, we might yet see quantum leaps in programmer productivity. I think that WISC computers are more than just another novel architecture. I think that they are a new way of looking at the bottom line of computing: getting problems solved.

SOURCES CONSULTED

A. Agrawala and R. Rauscher, Foundations of Microprogramming: Architecture, Software, and Applications, Academic Press, New York NY, 1976.

M. Andrews, Principles of Firmware Engineering in Microprogram Control, Computer Science Press, Potomac MD, 1980.

R. Blake, "Exploring a Stack Architecture", Computer, May 1977, pp. 30-39.

D. Bulman, "Stack Computers: An Introduction", Computer, May 1977, pp. 18-28.

R. Colwell et al., "Computers, Complexity, and Controversy", Computer, May 1977, pp. 30-39.

M. Flynn, "Directions and Issues in Architecture and Language", Computer, October 1980, pp. 5-22.

F. Hill and G. Peterson, Digital Systems: Hardware Organization and Design, (2nd ed.), John Wiley & sons, 1978.

M. Katevenis, Reduced Instruction Set Computer Architectures for VLSI, MIT Press, Cambridge MA, 1985.

P. Koopman Jr., "Microcoded Versus Hard-wired Control", Byte, January 1987, pp. 235-242.

P. Koopman Jr. and G. Haydon, "MVP Microcoded CPU/16 - Architecture", The Journal of Forth Applications and Research, Volume 4, Number 2, 1986, pp. 277-280.

P. Koopman Jr., "The WISC Concept", Byte, April 1987, pp. 187-217.

P. Lewis et al., Compiler Design Theory, Addison-Wesley, Reading MA, 1978.

G. Miller, Psychology of Communication: Seven Essays, Basic Books, New York NY, 1967.

V. Milutinovic, Tutorial on Advanced Microprocessors and High-level Language Computer Architecture, IEEE Computer Society Press, Washington DC, 1986.

G. Myers, Advances in Computer Architecture, John Wiley & Sons, New York, 1982, pp. 212-214.

J. Park, "Toward the Development of a Real-Time Expert System", The Journal of Forth Applications and Research, Volume 4, Number 2, 1986, pp. 133-154.

D. Patterson and C. Sequin, "A VLSI RISC", Computer, September 1982, pp. 8-21.

S. Przybylski et al., Organization and VLSI Implementation of MIPS, Stanford University Technical Report Number 84-259, April 1984.

P. Schulthess, "Reduced High-Level-Language Instruction Set", IEEE Micro, June 1984, pp. 55-67.

A. Tanenbaum, "Implications of Structured Programming for Machine Architecture", Communications of the ACM, Vol. 21 No. 3, March 1978, pp. 237-246.

J. Tremblay and P. Sorenson, The Theory and Practice of Compiler Writing, McGraw-Hill, New York NY, 1985.

W. Wulf, "Compilers and Computer Architecture", Computer, July 1981, pp 41-47.