The WISC Concept

A proposal for a writable instruction set computer

Phil Koopman

THE TRADITIONAL COMPLEX instruction set computer architecture with its large, complicated instruction set has become the mainstay of the microprocessor industry. Recently, however, proponents of the reduced instruction set computer architecture have made the controversial claim that RISC architectures can execute programs more quickly than CISC machines. Before you decide which side of the line you're on, I'd like to present an alternative computer architecture that combines elements of both RISC and CISC philosophies to produce an interesting, streamlined, flexible, and potentially fast machine.

My proposed architecture is called WISC, for writable instruction set computer. My purpose is not to show that either the RISC or CISC approach is somehow wrong, but rather to introduce an alternative that blends RISC and CISC concepts into a simple but powerful architecture.

First, I want to look at the key ideas from the RISC and CISC concepts. Then I can select the best ideas for the proposed WISC architecture. Finally, I will combine these ideas to define the WISC architecture and consider an overview design for a generic WISC machine.

Key RISC Concepts

RISC systems are based on the concept of optimizing the few instructions that are used the most and eliminating infrequently used instructions to reduce hardware complexity and increase hardware speed. I will look at the key RISC concepts, examine their strong or weak points, and pick the ones that are most desirable for an alternative architecture.

First, RISC machines must execute all instructions in a single memory cycle. Some authors have referred to this as single-clock-cycle operation, but the real resource limitation is the amount of time required to reference program memory. The idea here is that if a CPU can execute instructions as quickly as they are fetched from memory, maximum system throughput speed will result. Clearly, using as much of the memory bandwidth as is available is a desirable goal for WISC.

RISC machines must use hard-wired control. The intent of using hard-wired control is to allow for fast single-memory-cycle operation of op codes and (when combined with a very small instruction set) reduce the amount of silicon area required for implementation on a single chip.

But it is not clear whether hard-wired control is an absolute requirement. Since a designer can make a small amount of microcode memory extremely fast in relation to large amounts of program memory (while achieving a reasonable cost/performance trade-off), there is no reason why a microcoded processor cannot achieve single-memory-reference-cycle operation for most operations.

As for the chip-area argument, microcoded designs can have fewer gates than hard-wired designs (exclusive of the actual microcode memory). If I wish, I can use the extra silicon area available in a streamlined WISC single-chip implementation for microcode memory.

Next, RISC machines use relatively few instructions and addressing modes. This concept is a side effect of the need to keep things simple in a hard-wired, single-cycle processor. If a chip can support additional instructions without reducing the clock-cycle speed for basic instructions—as is often the case with microcoded CPUs but usually not with hard-wired CPUs—no real incentive exists to limit the number or types of instructions. Instructions with fancy indirect-address modes or multiple-memory-cycle operation should be supported if the net result is a speed-up of the entire system for an important application program or language run-time environment. So a WISC design should not unnecessarily restrict the number and variety of possible instructions.

RISC processors use a load/store design, which allows "load from memory" and "store to memory" as the only memory-reference instructions. This tends to reduce clock-cycle times by shortening delays in the memory-to-CPU data path and simplifying control logic. It also simplifies restarting after a virtual memory page fault. However, if virtual memory is not being used (as is the case in the vast majority of personal computers today) or if a memory reference can be combined with another operation for a net savings

continued

By day Phil Koopman is a U.S. Navy submariner and engineering duty officer; by night he designs computer hardware, software, and microcode. He can be reached at 20 Cattail Lane, North Kingstown, RI 02852.

No evidence exists that a fast computer requires an architecture with a difficult assembly language.

in time, then no reason exists for restricting the system to a load/store design. Thus, WISC computers should not be limited to a load/store design.

RISC machines use a fixed instruction format. Fixed instruction formats allow simpler decoding of instructions and reduced hard-wired logic. They also minimize the number of microcoded instructions that are wasted on shifting and interpreting op codes and operands.

Making all instructions the same size (e.g., a 16-bit format aligned on even-byte boundaries on a 16-bit machine) makes a lot of sense for simple, fast hardware design. You can argue that compressing variable-length instructions into the smallest space possible speeds program execution by reducing the number of memory accesses. But the trade-offs in unpacking these compressed instructions and formatting them properly for execution might eat up much of the savings with more complex hardware and extra instruction fetching when refilling a prefetch pipeline after a branch. Most people seem willing to increase memory space somewhat for faster program execution speeds. So WISC should use a fixed instruction format.

Finally, RISC machines trade off more sophisticated compiler technology for less complex hardware. This argument is based on the assumption that all programming is done in high-level languages that shield the user from the machine. No doubt sophisticated compiler technology can improve the speed of a high-level language program. It remains to be seen whether this speed increase can surpass the capability of an experienced assembly language programmer to handcraft the few lines of code that might break the speed bottleneck for a complex application program. Inasmuch as no evidence exists that a fast computer requires an architecture with a difficult assembly language, WISC should not have features that demand the use of a sophisticated compiler, although it could benefit from such a compiler.

A Major RISC Problem

For all its good, the RISC design has an Achilles' heel. The low semantic content of each instruction requires a high memory bandwidth, resulting in a sharp memory price/performance trade-off.

Consider the common operation of decrementing the value at a memory location. In a RISC machine this would be accomplished by a load, decrement register, and store using five memory cycles: three for instructions and two for memory data references. An efficient CISC or WISC architecture might support a single decrement instruction that uses only three memory cycles: one for the instruction and two for memory data references. If many commonly required high-level language functions are not supported in a RISC machine, memory access for instructions can create a bottleneck.

Another example is the absolute value operation applied to a value already resident in a CPU register or hardware data stack. In any processor without this function as a built-in primitive, absolute value determination consists of a sign comparison, a conditional branch, and a subtraction (or two's complement). This is a total of three instructions and a possible conditional branch that upsets any instruction pipelining that might exist. If the absolute value function is included in the instruction set, execution requires only one memory reference.

Now you might be thinking, "What about a memory cache? Doesn't that solve the memory bottleneck problem?" But a cache is only a partial solution. First a cache speeds up memory references only on the second and subsequent accesses to a memory location. Thus, the effectiveness of a cache is reduced by compiler optimizations such as unrolling loops. Second, a cache introduces additional system cost and complexity and results in extra delay when encountering a cache "miss" that requires fetching an instruction from memory. Finally, a cache design is often based on the concept of "locality" of programs. This contradicts the current software doctrine of breaking up programs into smaller and smaller procedures and functions for modularity and reusability—or forces greater memory usage by compiling functions and subroutines as in-line code, which further reduces cache effectiveness.

Simply put, it is better to have no memory bottleneck problem than to have a limited memory bandwidth with a cache. Therefore, WISC should be designed to minimize the number of memory references needed to accomplish each function in a high-level program.

To avoid the RISC memory bottleneck problem and achieve high performance, I can borrow some concepts from CISC machines. A CISC machine's CPU has an extensive and complex instruction set that attempts to support high-level language control and data structures directly. All of today's widely used 16-bit microprocessors are CISC designs.

Borrowing from CISC

Two common CISC traits that might be useful in a WISC design are a minimal semantic gap and the inclusion of as many high-level language-oriented instructions as possible.

The driving force behind the complexity of a CISC machine is the desire to speed up common high-level language operations such as character-string manipulation, pointer maintenance, looping, and array handling. By reducing the so-called semantic gap between the high-level language statements used in a program and the machine-code instructions available on the CISC machine, programs should require fewer memory references, take up less space, and run faster. To handle the very complex instructions that can be used, designers of CISC machines often use microcoded implementations. Likewise, to provide complex instructions while minimizing hardware complexity, WISC should employ a microcoded design.

An unfortunate side effect of complex and comprehensive instruction formats can be an excessive amount of decoding logic or multiple microcycles just to decode an instruction before any real work is done. But this side effect can be reduced by the adoption of a simple fixed instruction format for WISC instructions. Using a fixed instruction format eliminates complex manipulation of instructions to extract the meaning of an op code and its operands, thus reducing hardware requirements and speeding up the processor.

Powerful high-level language-oriented instructions, such as decrementing a memory-location value or string manipulations, can speed up programs significantly by reducing the number of instruction fetches from program memory. The only pitfall is that such instructions must be well suited to high-level languages, or compilers ignore them in favor of synthesizing primitive instruction sequences that do the job exactly. Examples of problem areas include zero-based versus one-based arrays and loop counters, subroutine calling, parameter passing, and list/record data-structure manipulation.

The answer to the semantic mismatch caused by high-level language instructions that don't quite meet high-level language requirements is to customize the processor's instruction set for each language environment. This customization

continued

would be accomplished in WISC with a writable microprogram memory, sometimes called a writable control store, that employs high-speed RAM to store microcode. Such an arrangement would let the processor's microcoded instruction set be changed as the operating system requires.

Therefore, a WISC goal should be to execute all instructions in a single memory-reference cycle and use 100 percent of available memory bandwidth, except where a microcoded complex instruction clearly results in performance superior to multiple simple instructions for a particular application or high-level language run-time environment. Of course, instructions involving memory operand access will be longer than a single memory cycle, but they will nonetheless tend to keep the memory productively engaged at all times.

Using Stacks

The WISC architecture should use one final feature to synergistically work with other design aspects to increase speed and decrease complexity of the system: hardware-implemented push-down last-in/first-out stacks.

The stack concept has proved its value in computers and modern-language implementations that use stacks for implementing subroutine return-address storage or parameter passing. However, these stacks are generally realized as an address register that points to main memory, with perhaps the top few elements of the stack located in special registers. I propose using completely independent high-speed memories to implement two stacks for the WISC architecture. One stack would be primarily for subroutine return-address storage and the other for data storage.

The advantage of a hardware return-address stack is that subroutine calls and returns can be processed at a high speed, with the return address transferred to or from the return stack in parallel with decoding the next instruction. A hardware data stack lets subroutine parameters be passed to subroutines without main-memory accesses in addition to providing for a large amount of scratch work space for storing temporary results. In fact, the underlying structure of modern languages such as Modula-2 seems to presume the existence of a stack of some sort.

In addition to reducing subroutine-call overhead, use of a data stack simplifies (and quickens) the machine's operation by eliminating the need for operand decoding. Since a stack machine implicitly addresses certain elements on the stack relative to the current stack pointer position, the CPU does not suffer any delays while source and destination registers are selected from a large register bank. Furthermore, the instruction bits freed by not needing fields for selecting registers allows the use of a narrow word size (16 bits or less), packing multiple op codes into each program word, or using constants or other values in the same word as an op code, all while maintaining a simple instruction format.

In-line literal values are required in a stack machine only for providing values for variable initialization, arithmetic constants, or branching addresses. These values can either be incorporated into unused instruction bits or placed into a memory cell after the instruction requiring the value. One interesting approach that some stack-oriented processors use is to have two instruction types: one for operations (consisting of an op code with no parameters) and one for subroutine branches (consisting of only an address with a flag indicating an implied op code of a call).

So the WISC design should include

continued

hardware stacks. The use of hardware stacks will reduce subroutine-call overhead and the complexity and delay associated with operand decoding, since all operands are implicit.

A Generic WISC Computer

Having described the attributes of a WISC computer, I would like to present a generic architecture for WISC implementation. Figure 1 shows a block diagram of one possible format for a WISC computer.

The resources of this generic WISC computer are a data stack, an ALU with a small number of registers (perhaps only one), a return stack with a bidirectional data path to the program counter for subroutine-call address manipulation, a program memory, and a microcoded controller. All the resources are connected to a central data bus, with access to I/O services through an appropriate interface.

The WISC machine in figure 1 has several interesting aspects. One feature not always found on hardware-based stack designs is that the registers above the ALU can hold the top one or two data-stack elements. These registers allow the use of a single-ported data-stack RAM.

The entire instruction decoding path, from the return-address stack all the way through to the microinstruction register, is completely independent of the data bus. This independence allows for ALU and data-stack operations on data while instructions are fetched and decoded simultaneously. This structure allows use of nearly 100 percent of the memory bandwidth. An added benefit is that there is no need to implement an instruction prefetch unit; no time is lost flushing an instruction queue when a branch is encountered. In fact, implementing a delayed branch similar to the ones used by some RISC machines can eliminate almost all idle or wasted memory cycles.

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

A design approach for instruction decoding that could greatly simplify the CPU hardware would be to use, for example, an 8-bit op code that directly addresses a word in the microcode memory. This would directly address the first microprogram instruction of a page of microprogram memory; one page of microprogram memory would be allocated to each op code. This would allow complete flexibility in instruction set assignment while using very little instruction decoding logic.

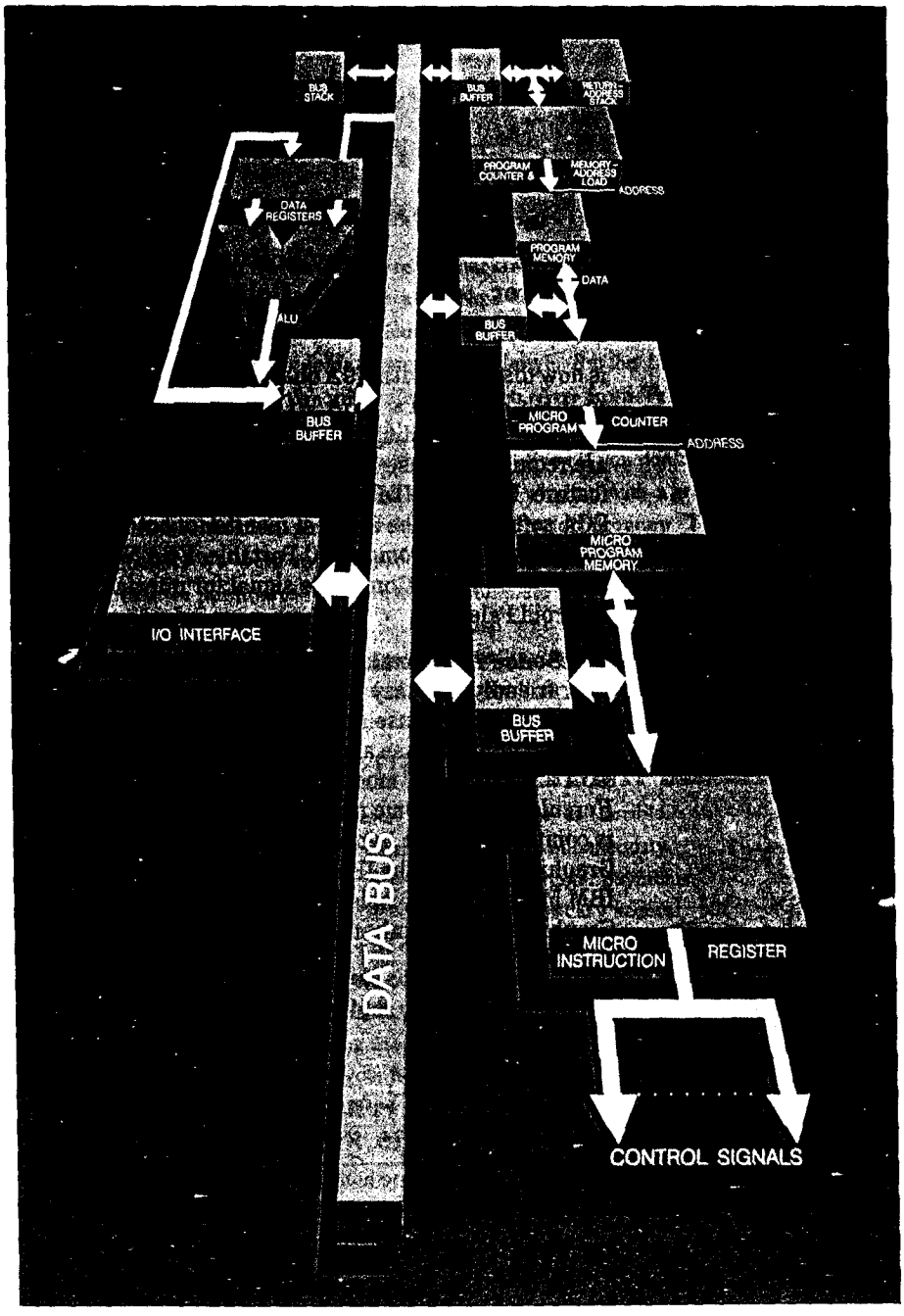

Figure 1: *A block diagram of a possible WISC machine implementation.*

The Past, Present, and Future of WISC

Constructing a hodgepodge of previously successful computer design techniques does not guarantee success. The WISC design criteria presented here represent a careful balance of often conflicting design requirements. That said, I will look at some past and current computers that inspired some of the WISC machine's unusual design features.

The Burroughs B1700, a microcoded machine, had a different instruction set for each language it supported: BASIC, FORTRAN, and COBOL/RPG-II. The tailored instruction set for each language resulted in smaller programs and much faster execution speed than that found on comparable machines of the time. But the complexity of the architecture for vari-

continued

able-width operand support made the machine expensive.

The current RISC II and MIPS processors (see "How Much of a RISC?" by Phillip Robinson on page 143) strive to achieve single-memory-cycle execution with the use of fixed instruction formats. Interestingly, the IBM RT PC and the Pyramid 90x computers use hybrid hard-wired/microcoded designs to allow for some complex instructions within a RISC framework.

One early reference to a stack machine was a design for a 1950s ALGOL language-specific processor known as ALCOR. While it was never built, it called for a two-stack machine that would have used one stack for operand storage and another stack for instruction storage.

More recently, the Novix NC4016 chip (see "Stack Machines and Compiler Design" by Daniel L. Miller on page 177) efficiently executes the dual-stack-based FORTH language with a hard-wired RISC architecture. The NC4016 is designed with single-cycle operation in mind and has low procedure-calling overhead due to the use of stacks, but it has a hard-wired instruction set like other RISC processors. Another stack-oriented processor, the MVP Microcoded CPU/16, combines hardware stacks with writable microprogram memory to allow redefinable instruction sets but is not optimized for single-memory-cycle instruction execution.

While none of the individual design features of WISC are new, I believe that implementing a true WISC machine will lead to discoveries about the nature of modern computer architectures and how to make them better. In the end, designing a more efficient computer architecture will lead to less expensive, more capable computers. ■

BIBLIOGRAPHY

Amsterdam, Jonathan. "Programming Project: Building a Computer in Software." BYTE, October 1985.

Bauer, F. L. "Between Zuse and Rutishauser—The Early Development of Digital Computing in Central Europe." *A History of Computing in the Twentieth Century*, N. Metropolis et al., eds. New York: Academic Press, 1980.

Colwell, R. P., et al. "Computers, Complexity, and Controversy." *Computer*, May 1977.

Fernandez, E. B., and T. Lang, eds. *Software-Oriented Computer Architecture (a Tutorial)*. Washington, DC: IEEE Computer Society Press, 1986.

Jennings, E. "The Novix NC4000 Project." *Computer Language*, October 1985.

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

Koopman, P. "MVP Microcoded CPU/16-Architecture." *The Journal of FORTH Applications and Research*, volume 4, number 2, 1986.

Meyers, G. J. *Advances in Computer Architecture*. New York: John Wiley & Sons, 1982.

Multinovic, V., ed. *Tutorial on Microprocessors and High-Level Language Computer Architectures*. Washington, DC: IEEE Computer Society Press, 1986.

Patterson, D. A., and C. H. Séquin. "A VLSI RISC." *Computer*, September 1982.

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

Ragan-Kelly, R., and R. Clark. "Applying RISC Theory to a Large Computer." *Computer Design*, November 1983.

Simpson, Richard O. "The IBM RT Personal Computer." *Inside the IBM PCs*, Fall 1986 BYTE.