COLLECTED WISC PAPERS

June 1987

WISC Technologies, Inc.

La Honda, CA 94020

First Printing June 1987
Second Printing June 1987

WISC Technologies, Inc.
Box 429 Star Route 2
La Honda, CA 94020

PREFACE

WISC Technologies, Inc. was incorporated in the State of California in March, 1987. The Company is dedicated to the development of new technologies in computer software and hardware design. These papers describe the work we have done.

The original CPU/16 was shown at the San Francisco Computer Faire in 1986. We were pleased that *BYTE* noted our product in their What's New section of the June 1986 issue.

At the 1986 Rochester Convention, two papers were presented on the history and architecture of the product.

BYTE invited two papers from Phil Koopman, Jr. The first was in their January 1987 issue featuring Programmable Hardware. The second was in their April 1987 issue featuring Instruction Set Strategies.

At the 1987 Rochester Forth Conference with the theme Computer Architectures, Glen B. Haydon presented a paper entitled "A Unification of Software and Hardware; A New Tool for Human Thought " and Phil Koopman Jr. presented an invited paper entitled "Writable Instruction Set, Stack Oriented Computers: The WISC Concept" .

These papers are collected in this publication to provide convenient access to the background history and the problems addressed by WISC Technologies, Inc. in their development of computer architectures to implement the WISC concepts.

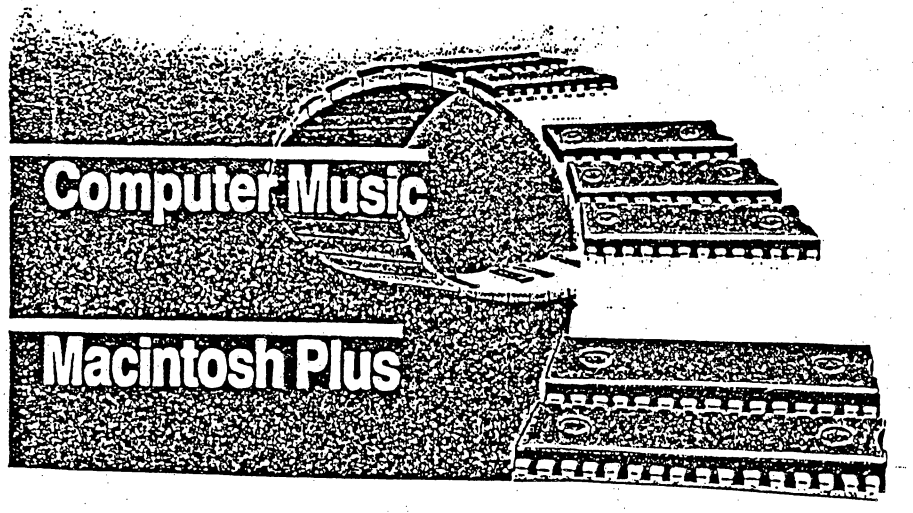
The WISC CPU/16 and WISC CPU/32 are available for immediate delivery.

June 1987 P. K. and G. B. H.

CONTENTS

$3.50 JUNE 1986

BYTE
THE SMALL SYSTEMS JOURNAL



WHAT'S NEW

Microcoded IBM PC Board

Designed for building customized processors, the MVP Microcoded CPU/16 from Mountain View Press is an add-on board for the IBM PC that implements a high-speed microcoded processor. A wire-wrapped prototype of the board, which MVP demonstrated at the West Coast Computer Faire in April, ran one FORTH test program 50 times faster than an IBM PC alone. According to the company, the processor can execute over 2 million stack operations per second.

The card's 74-chip design includes a 16-bit ALU, two hardware stacks, an interface to the IBM PC, 128K bytes of static memory, a program counter, two 16-bit data registers, and room for 256 microcoded processor instructions. Each microcoded instruction is defined by up to eight 32-bit user-definable microcode instructions.

An Engineering Prototype Kit is available for $1500, and a printed circuit board version should be available this month. MVP includes the following software with the wire-wrap kit: MVP FORTH/16, a word-oriented FORTH that executes directly in the processor; the MVP-FORTH Programmer's Kit; a Number Extensions package; a microcode assembler; a cross-compiler; a set of diagnostic programs, and source code for all the preceding software.

For more information, contact Mountain View Press Inc., POB 4656, Mountain View, CA 94040, (415) 961-4103.

Inquiry 558.

MVP MICROCODED CPU/16

HISTORY

Glen B. Haydon
Haydon Enterprises
Box 429 Route 2
La Honda, CA 94020

Phil Koopman Jr.
20 Cattail Lane
No. Kingstown, RI 02852

INTRODUCTON

The MVP-MICROCODED CPU/16 design resembles that conceived in the ALCOR project in developing an ALGOL translator utilizing multiple hardware stacks combined with the powerful techniques of a freely microcodable processor implemented in discrete components. In the present form of the CPU/16 design, the user is free to structure the processor according to application requirements for optimal efficiency.

HISTORY

The ALCOR project was led by Samelson and Bauer during the 1950s. Its goal was to provide a direct method for translation of ALGOL. They conceived of a hardware design with two "cellars", one was to hold operational characters and the other to hold numbers. In modern terminology these would be called stacks. They are hardware storage devices based on a last in first out scheme. A block diagram of their concept, Figure 1, has been included in several papers.

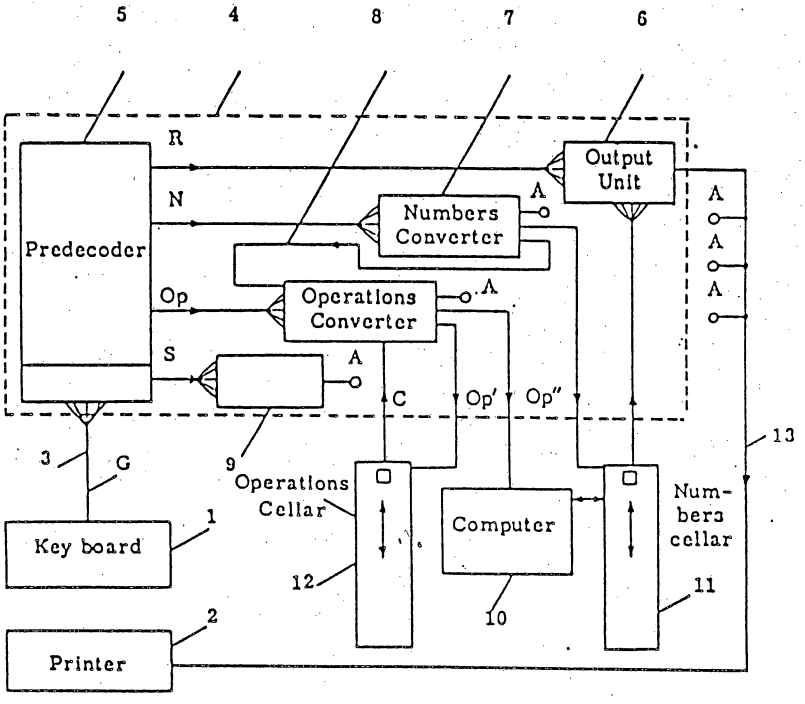


FIG. 1.

It appears that a hardware implementation of the ALCOR design was never completed. Computer processor designs took another direction. A stack operation was often included but with the stack memory mapped into a portion of the system's memory. Such stacks are usually used to store the return location for subroutine calls and sometimes to preserve other values.

In the late 1960s, Charles Moore designed a scheme of programming also using two stacks. One stack contained the return addresses of successive subroutine calls and the other stored interim data values during computation. Unfortunately, he could find no hardware designed to fulfill his needs and resorted to emulating such a processor. Such emulations are available on many systems today. They are known as a FORTH kernel.

A second consideration in the CPU/16 design is similar to that adopted by Seymore Cray. In his Cray computer design, he used discrete components. His claim was that it was the only way to get speed. Of course the Cray design utilized many other features but the basic idea was that faster processors could be implemented utilizing simple components.

The Cray computers used a Data General Eclipse as a host giving access to the outside world. In a similar manner the CPU/16 uses an IBM compatible as a host providing I/O to the outside world. With only minor changes, the CPU/16 could use any common microcomputer.

Finally, the concept of microcoding a simple processor has been utilized in many different ways. Specific microcodable devices have been designed and are commercially available. Examination of these devices suggested that we could design a simple processor with discrete components which could be microcoded and provide even greater versatility and speed.

RESULTS

The end result of these ideas is presently operational and available in kit form. It provides an ideal tool for exploring the potential of the design and as a learning medium. Unfortunately, many people are reluctant to undertake a wire wrapping exercise requiring 30 to 40 hours. However, utilizing the single stepping capabilities from the host, any portion of the processor can be exercised step by step. There is no better way to learn at first hand the capabilities of a multistack microcodable processor.

WORK IN PROGRESS

Now that the CPU/16 design of the kit has stabilized, the next step is to produce that design on printed circuit boards to be placed in the IBM PC compatible. A problem with such a board is that it is no longer simple to change a wire corresponding to a bit in the microcode. The printed circuit board is no longer the experimental tool at the hardware level.

The CPU/16 design is currently being laid out and wire wrapped on a pair of S-100 system boards. It will run with an implementation of MVP-FORTH on an S-100 bus system. There is also interest in implementing the CPU/16 design utilizing the Apple II series of computers as the host.

NEXT GENERATION

Where to from here? The 16-bit bus of the present design is limited to 16-bits of address space. By addressing on word boundaries, the system can address 128K bytes. But without some form of bank switching, virtual memory or some other technique, the size of memory is limited. Intel has overcome this limitation by utilizing several base segments from which addresses can be indexed. This is in essence a form of bank switching although it has been efficiently implemented.

The next bus size to consider is 32-bits wide. Intermediate numbers of address bits can be used but efficiency dictates the next size limitation at twice the size of the 16-bit limit. Using a 32-bit bus in a manner analogous to the current 16-bit design and adding a number of enhancements, a significant further increase in performance is anticipated. Also a billion 32-bit words (4-giga-bytes) of contiguous memory could be addressed without some form of bank switching. Part of the engineering prototype for a CPU/32 based on these considerations is already completed and functional.

The CPU/16 kit is an ideal hardware system with which to study other architectures. For example, the design is clearly not a RISC machine as currently described. However, by addressing items in the dedicated stack memory with optimized microcode, it would be possible to treat stack RAM as an array of registers and emulate a RISC design. In such an implementation, the RISC architecture could be thought of as a subset of the capabilities of the CPU/32 processor.

LANGUAGES

The MVP CPU design lends itself to the efficient implementation of a wide variety of high level languages. For example Smalltalk-80 uses approximately 100 primitives each of which could be implemented in microcode. The design would be ideal for implementation of a p-code machine.

FORTH has been used in the initial phases of this work. FORTH is, after all, an emulation of the hardware design. The language has the advantage of ease of interactive programming and access to all hardware components. The diagnostic suite, micro-assembler and cross-compiler were easily developed with a minimum of effort. The language also provides a versatile facility for programming many applications.

However, with the desirability of making the system compatible with other existing programs, it would be desirable to have a common operating system available. One route to a popular operating system would be to first implement the C language. Already, one group is working to implement Small C. With a full implementation of C, the entire UNIX system could be added.

With the versatility of a microcodable processor, the development of new languages tailored to specific applications becomes more reasonable. The languages of LISP and PROLOG are just a beginning in the field of artificial intelligence and they have been implemented in FORTH. It should be relatively easy to move such implementations to the newly designed processor.

CONCLUSIONS

The MVP CPU design provides flexibility in designing and using hardware to solve many application problems. In addition, many high level languages could be implemented on such a system with excellent efficiency. Initially, FORTH has been chosen as the as the host and processor language. As such, the system complements a variety of other commercially available implementations of FORTH in hardware. The MVP CPU series of products provides flexibility for experimentation and tailoring the processor to specific application and a tool for teaching and testing a variety of hardware processor designs.

The kit would make an ideal starting point for a comprehensive computer science course sequence. Such a sequence might start with the building of the kit as a microcodable processor. That might be followed with the writing of the software for a compiler and an operating system. The series might conclude with a significant application utilizing the tools which were developed.

The fundamental philosophy has been to examine programming requirements of the application at hand, and design the hardware accordingly. It is a shame to have the hardware limitations drive the programming and limit the solution of the application. The present design is a stage in the evolution of hardware to solve problems. Perhaps more than two stacks would be desirable in some applications. Once a design is found for a specific application, the next step would be to cast that design in silicon. But don't get the cart before the horse.

BIBLIOGRAPHY

Bauer. Fredrich l., Between Zuse and Rutishauser- The Early Development of Digital Computing in Central Europe., in A History of Computing in the Twentieth Century, N. Metropolis, J. Howlet, and Gain-Carlo Rota, Editors, Academic Press 1980.

Note: This volume is a treasury of historical ideas which are unknown to many workers in various branches of computer science today.

MVP MICROCODED CPU/16 ARCHITECTURE

Phil Koopman Jr.
20 Cattail Lane
No. Kingstown, RI 02852

Glen Haydon
Haydon Enterprises
Box 429 Route 2
La Honda, CA 94020

ABSTRACT

The MVP Microcoded CPU/16 is a 16-bit coprocessor board that directly executes high level stack-oriented programs. The CPU/16 may be micro-programmed to execute any stack-oriented language. FORTH was used as the initial implementation language to reduce development time and costs.

INTRODUCTION

Modern computer languages and compilers rely heavily on the concept of the push-down stack. However, conventional computers are optimized for register-oriented operations and impose large memory access time penalties when using stacks residing in main memory. The CPU/16 stack-oriented coprocessor can improve the performance of a personal computer to equal that of a much more expensive mini-computer for programs that make heavy use of stacks.

The MVP Microcoded CPU/16 was designed as a "low tech" exploration tool for stack-oriented processing. The result is an inexpensive commercial system that:

1) Uses simple, inexpensive, commonly available components.
2) Minimizes hardware and software development tool costs.
3) Fits the basic system onto a single IBM compatible Personal Computer expansion board (13" x 4").
4) Maximizes flexibility and minimizes complexity.
5) Achieves a 20 to 50 times speed improvement over 8088 MVP FORTH.

SYSTEM ARCHITECTURE

The CPU/16 is implemented in only 74 ICs (with 8k words of program memory), with no custom or semi-custom chips required. 74xx and 74LSxx series ICs provide all logic functions, with 120ns CMOS static RAMs for microcode and program memory.

Figure 1 shows the architectural structure of the CPU/16. All data paths are 16 bits wide.

The CPU/16 plugs into an IBM compatible personal computer as a one-slot expansion board. The host interface on the CPU/16 allows the personal computer to alter registers and memory as well as single-step programs at the microcode or macrocode level. When the CPU/16 is in

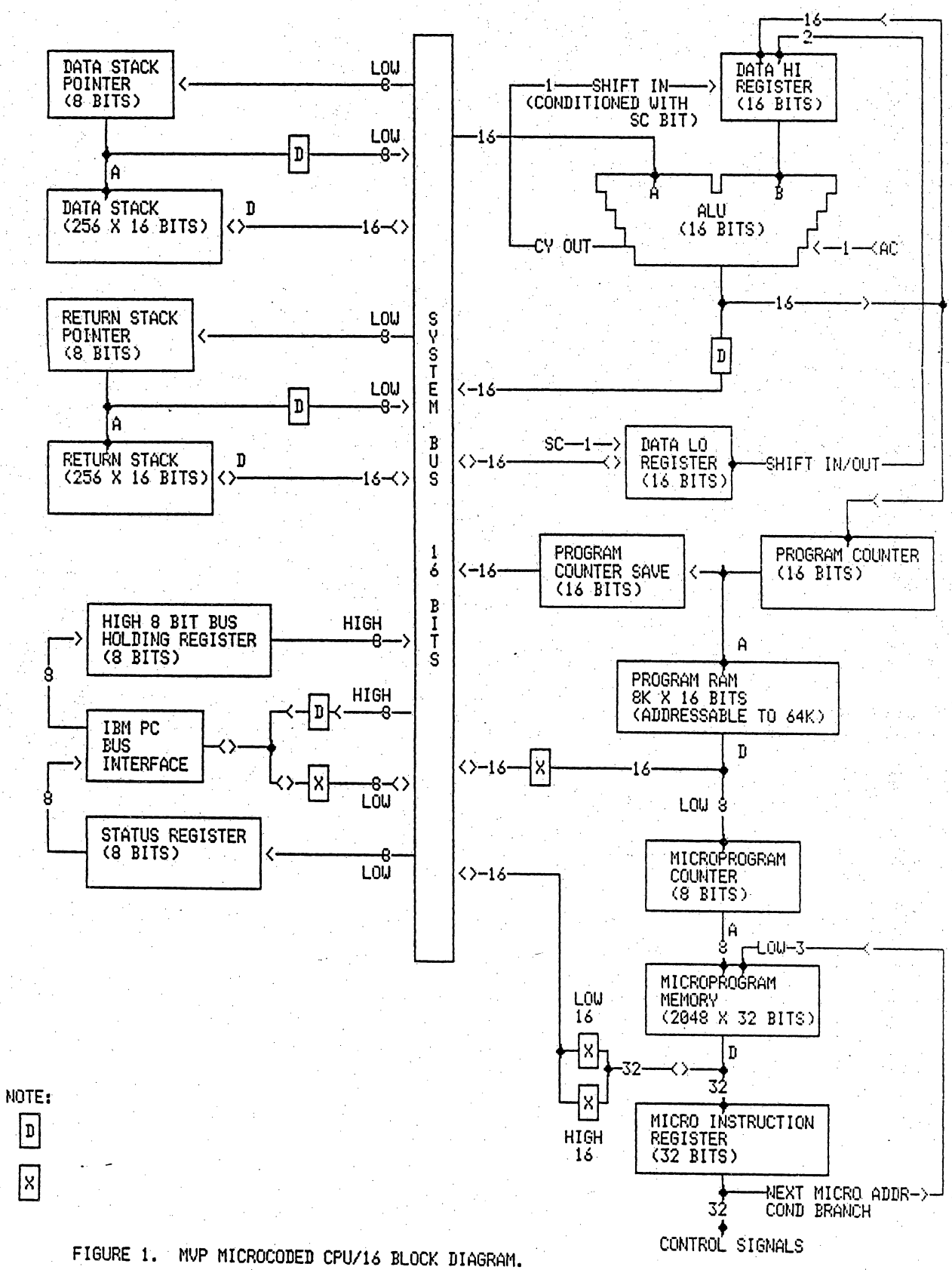


FIGURE 1. MVP MICROCODED CPU/16 BLOCK DIAGRAM.

"master" mode, the personal computer waits for the CPU/16 to request I/O service through the status register.

The return stack and data stack are hardware stacks with 8-bit pointers addressing 256 elements of stack memory. The stacks may be accessed and pointers incremented or decremented in a single clock cycle.

The ALU is built from 74LS181 chips, and has two shift registers to hold intermediate results. The Data Hi register and the Data Lo register can be shifted together as a 32-bit register for multiplication and division. The Data Hi register normally contains the top data stack element.

Program memory is organized as 64k words of 16 bits. All but the last 256 words may be used for program memory. A 16-bit program counter is used for all memory access addressing. The separate memory address bus from the program counter allows overlapped instruction fetching and execution. Program memory expansion beyond 8k words requires a daughter-board.

Micro-program memory is organized as 2k words of 32 bits. The microcode bit format is typical of modern horizontally microcoded machines. The micro-program counter and micro-instruction register allow overlapped fetching and execution of micro-instructions. Conditional microcode branches and microcode looping are accomplished by manipulation of the low order 3 bits of the micro-program address. If, during macro-instruction decoding, the highest 8 bits of a macro-instruction are not all 1, the microprogram counter is forced to all zero's, executing a DOCOL subroutine call. If the highest 8 bits are all 1, then one of 256 possible microcoded primitives is executed.

SOFTWARE SUPPORT

FORTH was picked as the CPU/16's development language for its efficiency, its simplicity of compiler implementation, and its friendly interactive environment with easy access to hardware resources. The CPU/16 supporting software includes a host control program, a microcode assembler, and a FORTH cross-assembler, as well as the FORTH microcode and kernel for the CPU/16 implementation.

The host program, microcode assembler, and cross-compiler are written in 8088 MVP-FORTH. The CPU/16 currently uses an MVP-FORTH kernel that differs in functionality from the 8088 MVP-FORTH version in that it uses word-oriented instead of byte-oriented memory addressing. In addition to FORTH, the CPU/16 is capable of supporting other programming languages such as Modula 2, Pascal, Lisp, and C. Any compiler implemented in machine-independent MVP-FORTH can be quickly installed on the CPU/16.

Current applications available on the CPU/16 include double-precision and quad-precision integer arithmetic and single-precision floating point math packages.

PERFORMANCE

The CPU/16 runs at a 4.77 MHz micro-cycle rate. An "average" microcoded primitive executes in 3 clock cycles (630 ns). This provides approximately a 20 to 50 times speed increase over 8088 MVP-FORTH programs operating at the same clock speed.

Since only half of the micro-program memory is required for the MVP-FORTH implementation, custom-written microcoded primitives may be added to a user's application to increase the speed of commonly used words. As an example, software stack manipulation words:

```
: INC[@]  ( PTR-ADDR -> N )   DUP  @ @  1  ROT +! ;
: DEC[!]  ( N PTR-ADDR -> )   -1  OVER +!  @  ! ;
```

can each be implemented in 10 micro-cycles (2.10 us), a speed increase of greater than 300% over high-level definitions. The listing for INC[@] is given as an example of CPU/16 microcode:

```
177 OPCODE: INC[@]     ( ADDR -> N )
0 :: SOURCE=ALU  ALU=B    DEST=PC  ;;    \ PC <- ADDR
1 :: SOURCE=ALU  ALU=-1   DEST=DLO ;;    \ DLO <- -1
2 :: SOURCE=RAM  ALU=A+1  DEST=DHI ;;    \ DHI <- POINTER+1
3 :: SOURCE=ALU  ALU=B    DEST=RAM ;;    \ POINTER <- DHI
4 :: SOURCE=DLO  ALU=A+B  DEST=PC  INC[MPC]  ;;  \  PC <- PTR
5 ::  JMP=000  ;;     \ WAIT FOR RAM ACCESS, JMP TO NEXT PAGE
178 CURRENT-PAGE !
0 :: SOURCE=RAM           DEST=DLO ;;    \ DLO <- DATA
1 :: SOURCE=PCSAVE ALU=A+1 DEST=PC ;;    \ RESTORE PC
2 :: SOURCE=DLO ALU=A DEST=DHI DECODE ;;  \ T.O.S. <- DATA
3 ::    END ;;                  \ JMP TO NEXT INSTRUCTION
```

FUTURE DEVELOPMENTS

Future developments for the CPU/16 will focus on broadening the range of languages and application programs available. Potential applications for a stack-oriented processor include: artificial intelligence, computer graphics, image processing, real-time control, and efficient execution of modern computer languages.

The CPU/16 is the first in a family of stack-oriented processors. A 32-bit general-purpose stack-oriented processor with greater speed and memory addressability is currently in development.

CONCLUSIONS

The MVP Microcoded CPU/16 is a high performance, general-purpose stack-oriented processor. A "low tech" approach has yielded significant speed improvements over current microprocessors at a modest cost. Compatibility with existing MVP-FORTH systems allows for easy porting of existing software to a high performance environment.

Microcoded Versus Hard-wired Control

A comparison of two methods for implementing the control logic for a simple CPU

Phil Koopman

THE INSTRUCTION decoding and execution control sections of modern computers are prime areas for using programmable hardware. Two of the most widely used methods for designing CPU control sections in microprocessors, minicomputers, and mainframes are microcode and hard-wired logic. Each method has its advantages, and both are natural applications for programmable hardware devices.

Architectural Description

I'll start by giving the specifications for a simple computer architecture, then walk through the implementation of this architecture using both microcoded and hard-wired design strategies. While both approaches require the same description and specification groundwork, they use different schemes to generate control signals.

I will examine the CPU architecture of Toy, a fictitious computer designed especially for this article. The CPU has an accumulator (ACC), an arithmetic logic unit (ALU), an instruction register (IR), a program counter (PC), some random-access memory (RAM), and some control logic. Figure 1 is a block diagram of the Toy architecture. All data paths are 16 bits wide with 12-bit memory-address paths. You can directly implement the ALU, ACC, IR, PC, multiplexer, and RAM sections of Toy using commonly available chips. Toy's control-logic section will require detailed design and the use of customized hardware or a large number of combinatorial logic gates.

The Toy instruction format shown in figure 2 consists of a 4-bit op code and a 12-bit address field. The 16 implemented op codes are shown in table 1. Op codes 8 through 15 do not make use of the instruction's address field.

Since Toy is a single-accumulator machine, the instructions ADD, SUB, AND, OR, and XOR combine the contents of a memory location with the accumulator and return the result to the accumulator. The instructions STORE and LOAD transfer the accumulator to and from RAM. The instructions NOT, INC, DEC, and ZERO operate on the accumulator alone. While JMPZ is the only branching instruction, you can program an unconditional branch by following ZERO with a JMPZ. Finally, the four unused op codes act as null operations (NOPs) to eliminate the annoyance of dealing with illegal op codes.

Control Logic

The control-logic section translates the op-code bit patterns into CPU-control and timing signals. Figure 1 shows the op-code inputs to the control-logic unit and the control-signal outputs required to run the rest of the CPU. The signals ALU0 through ALUCIN control the ALU. (I based the bit assignments on those for the 74181 ALU chip. See *The TTL Data Book*, listed in the Bibliography.) If ALUMODE is a 1, then the ALU will perform a logical operation; if it's a 0, the ALU will perform an arithmetic operation. ALU0 through ALU3 control which arithmetic or logic operation the ALU is performing. ALUCIN acts as the carry-in for the ALU.

When the signal CLOCK[ACC] is a 1, the ACC register is loaded with the value of its inputs at the rising edge of the system clock. This is usually referred to as "clocking in" the contents of the ACC. When the signal CLOCK[IR] is a 1, the contents of the IR are clocked in from the RAM output. This is the mechanism used to decode the next op code. When ADDR=IR is a 1, the RAM address multiplexer places the contents of the IR address field onto the RAM address bus. When it is a 0, the PC is used to address RAM. I use the descriptor ADDR=PC to mean ADDR=IR is 0. When CLOCK[PC] is a 1 and the ACC is 0, the PC is loaded from the IR address field. When INC[PC] is a 1, the program counter is incremented by 1 at the end of the current clock cycle. When WRITE[RAM] is a 1, the RAM cell addressed by the RAM address bus is loaded with the output of the ALU; when this signal is a 0, the ALU is driven from the output of RAM.

Functional Specifications

Now for the heart of how the Toy instruction set is implemented. In the Toy CPU, all instructions can be executed in just one or two clock cycles. Table 2 shows the actions required to complete each op code's function. Those actions in table 2 that are

continued

By day, Phil Koopman (20 Cattail Lane, North Kingston, RI 02852) is a U.S. Navy submariner and engineering duty officer; by night, he designs computer hardware, software, and microcode.

not the control signals shown in figure 1 are macros for the ALU control bits whose value is given in table 3. Let's examine some representative op codes in detail.

The STORE op code stores the contents of ACC into RAM. For the first cycle of this instruction, the low 12 bits of the IR address RAM. The ALU routes the ACC contents through without modification, then writes them out to RAM.

STORE requires two clock cycles since RAM is being used for accessing a data value during the first clock cycle. The second clock cycle is the same for all two-cycle instructions; it is simply a decoding of the next op code.

The contents of the RAM address pointed to by the PC are put onto the RAM address bus to fetch the op code. They are then clocked into the IR, and

continued

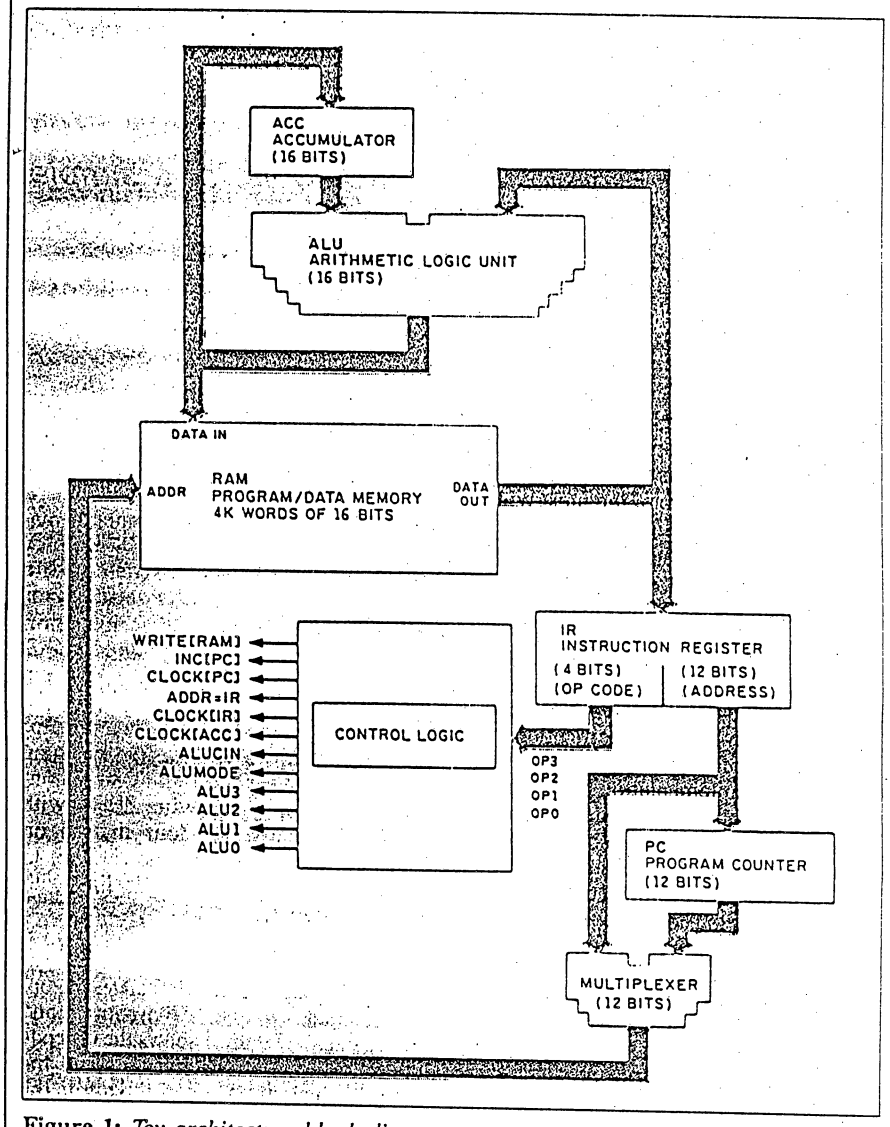


Figure 1: *Toy architecture block diagram.*

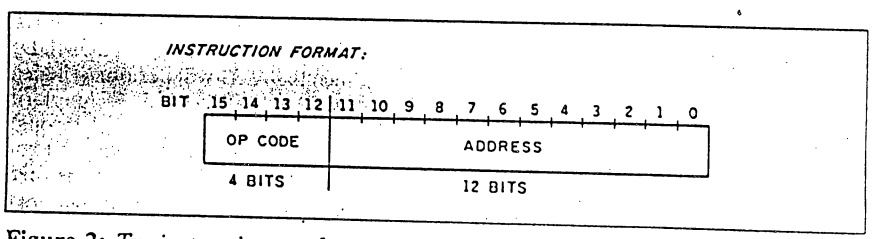


Figure 2: *Toy instruction set format.*

Table 1: *Toy instruction set.*

Op code	Operation	Description
0	STORE	store accumulator in RAM at address
1	LOAD	load ACC from RAM at address
2	JMPZ	jump to address if ACC is zero
3	ADD	add RAM to ACC
4	SUB	subtract RAM from ACC
5	OR	logical OR RAM into ACC
6	AND	logical AND RAM into ACC
7	XOR	logical XOR RAM into ACC
8	NOT	logical one's complement into ACC
9	INC	add 1 to ACC
10	DEC	subtract 1 from ACC
11	ZERO	place 0 in ACC
12	NOP	null operation — unused op code
13	NOP	null operation — unused op code
14	NOP	null operation — unused op code
15	NOP	null operation — unused op code

Table 2: *Toy functional specification. Note that ADDR=PC is equivalent to the ADDR=IR signal being 0. Also, I have used descriptive macro names for the ALU control bits (see table 3).*

Op code	Operation	Cycle	Specification
0	STORE	1	ADDR=IR ; ALU=ACC ; WRITE[RAM]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
1	LOAD	1	ADDR=IR ; ALU=RAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
2	JMPZ	1	CLOCK[PC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
3	ADD	1	ADDR=IR ; ALU=ACC+RAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
4	SUB	1	ADDR=IR ; ALU=ACC−RAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
5	OR	1	ADDR=IR ; ALU=ACCorRAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
6	AND	1	ADDR=IR ; ALU=ACCandRAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
7	XOR	1	ADDR=IR ; ALU=ACCxorRAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
8	NOTA	1	ALU=notACC ; CLOCK[ACC] ; ADDR=PC ; CLOCK[IR] ; INC[PC]
9	INCA	1	ALU=ACC+1 ; CLOCK[ACC] ; ADDR=PC ; CLOCK[IR] ; INC[PC]
10	DECA	1	ALU=ACC−1 ; CLOCK[ACC] ; ADDR=PC ; CLOCK[IR] ; INC[PC]
11	ZERO	1	ALU=0 ; CLOCK[ACC] ; ADDR=PC ; CLOCK[IR] ; INC[PC]
12–15	NOP	1	ADDR=PC ; CLOCK[IR] ; INC[PC]

finally the PC is incremented so that it is pointing to the next op code.

JMPZ accomplishes a conditional branch by loading the contents of the PC with the address in the IR. For this to be a conditional branch, the control signal to the PC loader must be ANDed with a signal that is only true if all the bits of the ACC are 0. Since the PC is loaded with the new instruction address at the end of the first clock cycle, the second cycle is a normal decoding instruction for this new address, identical to the second cycle of STORE.

Table 3: *Macros for the ALU control bits (based on bit assignments in the 74181 ALU chip).*

Macro	ALU0	ALU1	ALU2	ALU3	ALUMODE	ALUCIN
ALU = ACC	1	1	1	1	1	x
ALU = RAM	0	1	0	1	1	x
ALU = ACC + RAM	1	0	0	1	0	0
ALU = ACC − RAM	0	1	1	0	0	1
ALU = ACC OR RAM	0	1	1	1	1	x
ALU = ACC AND RAM	1	1	0	1	1	x
ALU = ACC XOR RAM	0	1	1	0	1	x
ALU = NOT ACC	0	0	0	0	1	x
ALU = ACC + 1	0	0	0	0	0	1
ALU = ACC − 1	1	1	1	1	0	0
ALU = 0	1	1	0	0	1	x

Table 4: *Control signal value specification.*

Values for first clock cycle of each instruction

Control signal	Op code 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ALU0	1	0	x	1	0	0	1	0	0	0	1	1	x	x	x	x
ALU1	1	1	x	0	1	1	1	1	0	0	1	1	x	x	x	x
ALU2	1	0	x	0	1	1	0	1	0	0	1	0	x	x	x	x
ALU3	1	1	x	1	0	1	1	0	0	0	1	0	x	x	x	x
ALUMODE	1	1	x	0	0	1	1	1	1	0	0	1	x	x	x	x
ALUCIN	x	x	x	0	1	x	x	x	x	1	0	x	x	x	x	x
CLOCK[ACC]	0	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0
CLOCK[IR]	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
ADDR=IR	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
CLOCK[PC]	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
INC[PC]	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
WRITE[RAM]	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Values for second clock cycle of each instruction

Control signal	Op code 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ALU0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALU1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALU2	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALU3	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALUMODE	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALUCIN	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
CLOCK[ACC]	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x
CLOCK[IR]	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x
ADDR=IR	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x
CLOCK[PC]	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x
INC[PC]	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x
WRITE[RAM]	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x

The single-clock-cycle instructions, such as NOTA, do not require a RAM access for an operand. This means that the usual second-cycle decoding sequence can occur during the same clock cycle as the ALU operation that modifies the ACC contents. In the case of NOTA, the RAM input to the ALU is ignored while the ALU computes the one's complement (logical inverse) of the current ACC contents.

Control-Logic Outputs

Table 4 gives a complete listing of all the control-logic output values that you need to specify the Toy functional description. Each X corresponds to a signal whose value does not matter, either because the controlled resource is unused (as in the ALU signals for op code 2) or because the second clock cycle is unused for op codes 8 to 15. These "don't-care" signals become crucial when you are designing hard-wired control circuitry.

Hard-wired Control

A CPU designed with hard-wired control uses random logic such as AND, OR, and NOT gates and either flip-flops or counters to decode each op code and control the processing flow. The hard-wired design process usually consists of identifying all the states needed to implement the instruction set, then deriving the Boolean logic equations required to control the computer's resources for each step.

Figure 3 shows the hard-wired implementation of the functional specifications given in table 4. It requires a controller with two states: first clock cycle and second clock cycle. The flip-flop in figure 3 is forced to the CLOCK1 state whenever a new instruction is clocked into the IR and changes to the CLOCK2 state whenever the IR is not clocked.

The most tedious part of a hard-wired control design is creating the logic gate networks to decode instructions into control signals. I have derived the required logic equations shown in figure 4 from the functional specifications in table 4. Figure 5 shows the Karnaugh map for deriving the first equation (ALU0) in figure 4. (See W. Fletcher's *An Engineering Approach to Digital Design* [Prentice-Hall, 1980] for a discussion of Karnaugh maps.)

The don't-care conditions are vital in reducing the complexity of the gate networks, since they allow freedom to ignore some op-code bits or state bits to minimize decoding logic. A good example of a don't-care condition is the ALU control signals; they do not depend on whether the controller is currently in the CLOCK1 or CLOCK2 mode.

continued

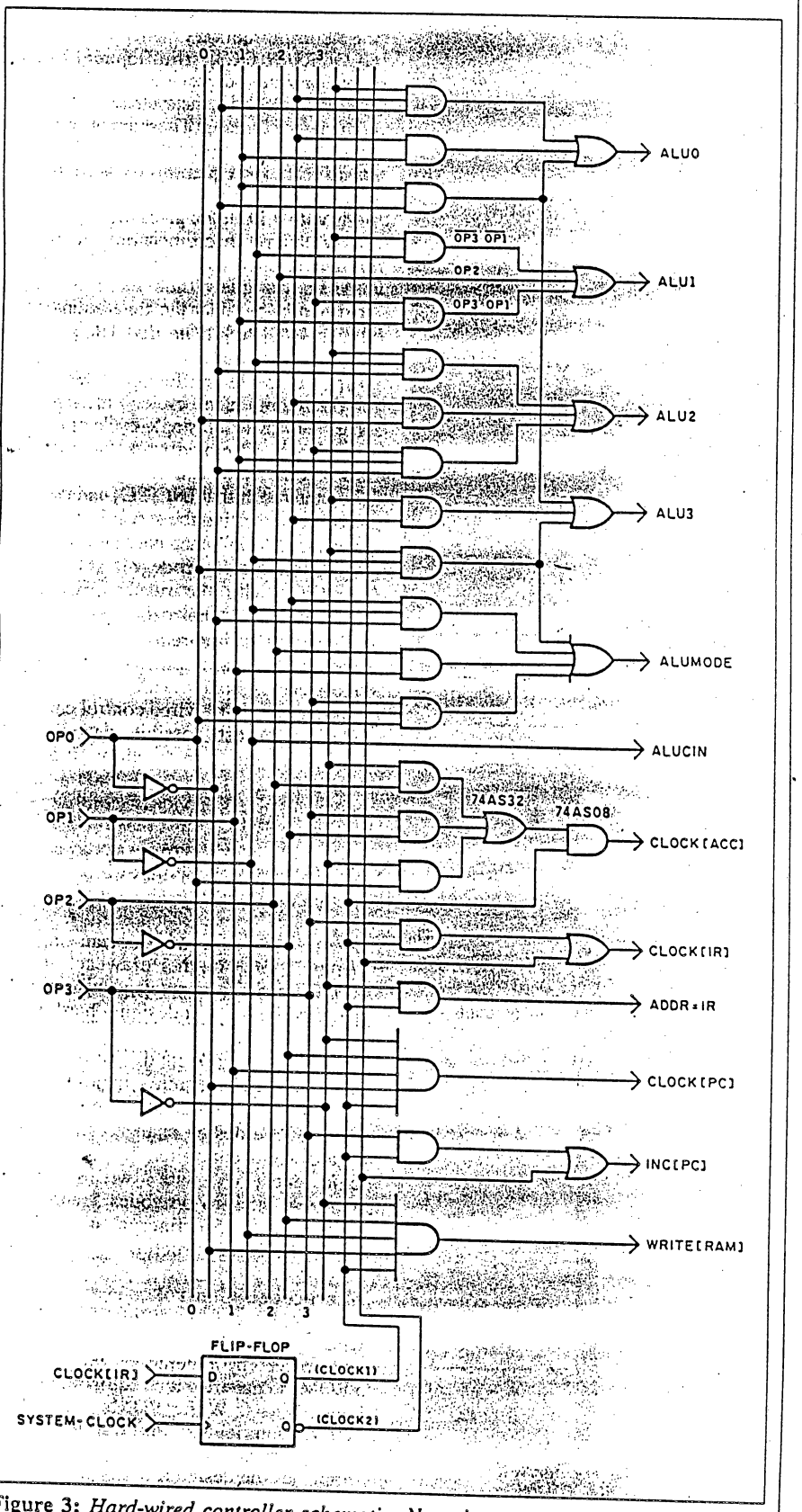


Figure 3: *Hard-wired controller schematic. Note that none of the ALU signals depend on whether the controller is in the CLOCK1 or CLOCK2 mode.*

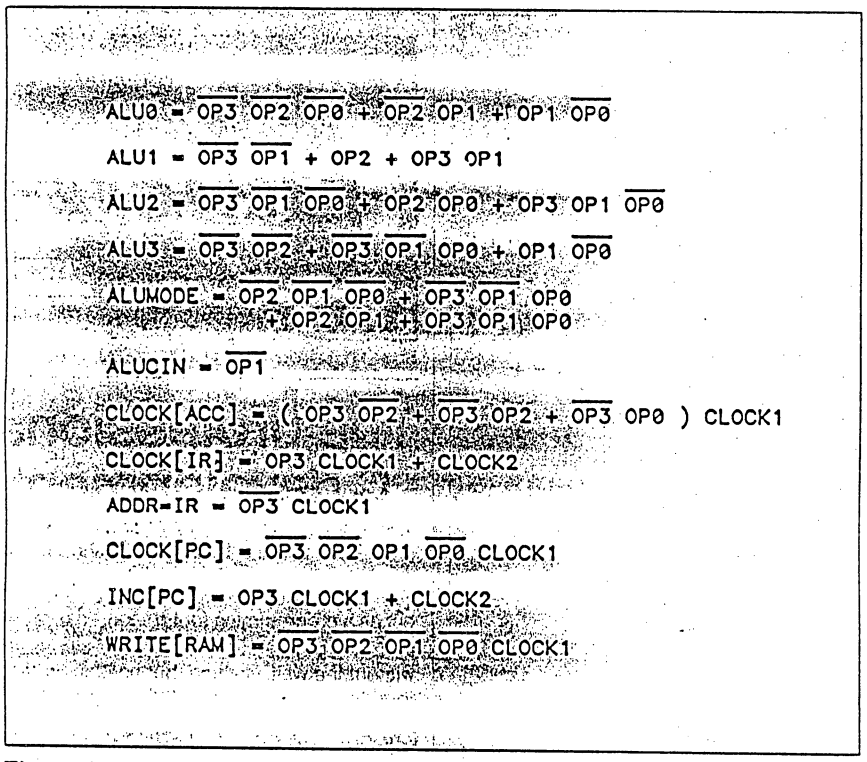


Figure 4: *Logic equations for Toy's hard-wired implementation.*

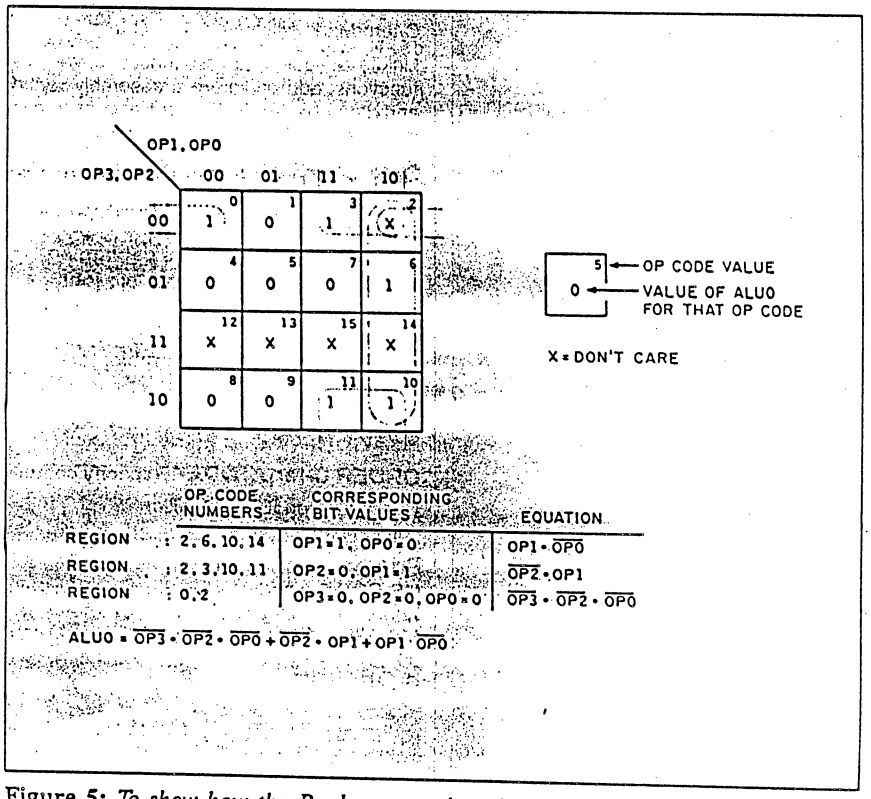


Figure 5: *To show how the Boolean equations in figure 4 were derived from table 4, here is the Karnaugh map used to minimize the ALU0 Boolean equation. The Xs are the don't-care bits, and the number in the upper right corner of each box is the op code.*

To implement the hard-wired controller, the complementary outputs of the CLOCK1/CLOCK2 flip-flop and the inputs from the current op code in the IR are fed throughout the system by the lines at the left of figure 3. These inputs are then fed through logic gate combinations specified by the equations in figure 4. You can implement these logic-gate combinations with TTL logic gates or, if you want to save board space, program them into hardware, such as a PAL.

As an example of how these decoding gates work, consider the generation of the signal INC[PC]. The INC[PC] signal should be a 1 for op codes 8 to 15 on the first clock cycle and for op codes 0 to 7 on the second clock cycle. But, since op codes 8 to 15 are all single-cycle op codes, any signals generated from them during the second cycle can be ignored. This gives the result that INC[PC] can be 1 for all op codes during the second cycle. The logic for INC[PC] then becomes the AND of the highest op-code bit (OP3) and CLOCK1, with the result ORed with CLOCK2.

Because the time required for a signal to pass through a simple logic gate is only a few nanoseconds with most current technologies, hard-wired control can provide the fastest possible decoding of machine language instructions. It also is the most flexible design method for specifying unique and complex control flows within a CPU because the designer can specify any decoding gate combinations and any control-flow hardware.

One drawback to using hard-wired control methodology is that it requires a considerable amount of Boolean algebra manipulation. Another drawback is that the CPU must be completely and correctly specified before you design a hard-wired control unit.

Any additions or modifications to the specification can require a major redesign of the control unit. If you want a feel for the impact a design change can have on a hard-wired controller, try redoing the logic equations with two op codes switched, such as op codes 5 and 9, or with op code 15 defined as a two-cycle logical NAND instruction.

Microcoded Control

Microcoded design differs from hard-wired design in that the control-logic gates are replaced by a memory array (usually a ROM) to generate the required control-logic signals. While ROMs are slower than random logic within the same price and performance categories, using a ROM simplifies the design process and significantly reduces time and costs for implementing a CPU control circuit.

Figure 6 shows the schematic for a

microcoded control circuit for Toy. The op code and a flip-flop similar to the one used in the hard-wired controller are fed in as an address to the microprogram ROM. The outputs of the ROM directly drive the control signals for the CPU. Each ROM location contains the proper bit settings to control a single clock cycle of an op code's execution, as shown in figure 7.

The control signals for the first cycle of each op code are placed in the even memory addresses (which are addressed when the flip-flop in the controller outputs a 0 for the first clock cycle), and the second cycle op codes are placed in odd memory addresses. I have arbitrarily assigned the value 0 to all don't-care bits from table 4 and copied the rest of the bits directly from table 4 to figure 7.

The main advantage to microcoded control is that it lets the designer change the CPU's functional description by changing the bits in any ROM address without having to redesign the machine's logic-decoding gate structure. Microcoded machine design also lends itself to simply structured, low-component-count computers such as those built using bit-slice technology. Most modern microproces-

continued

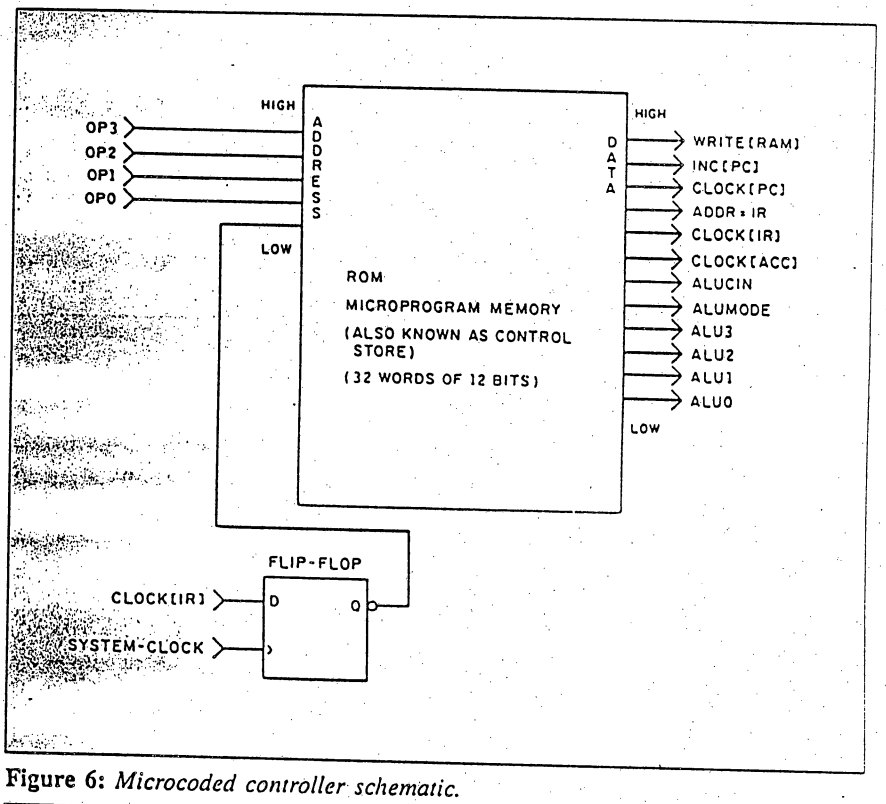


Figure 6: *Microcoded controller schematic.*

OP CODE	ROM ADDRESS	WRITE[RAM]	INC[PC]	CLK[PC]	ADDR[IR]	CLK[ACC]	ALU[ACC]	ALUOUT	A3	A2	A1	A0
0	0	1	0	0	1	0	0	0	1	1	1	1
	1	0	1	0	0	1	0	0	0	0	0	0
1	2	0	0	0	1	0	1	0	1	1	0	1
	3	0	1	0	0	1	0	0	0	0	0	0
2	4	0	0	1	1	0	0	0	0	0	0	0
	5	0	1	0	0	1	0	0	0	0	0	0
3	6	0	0	0	1	0	1	0	0	1	0	0	1
	7	0	1	0	0	1	0	0	0	0	0	0
4	8	0	0	0	1	0	1	1	0	0	1	1	0
	9	0	1	0	0	1	0	0	0	0	0	0
5	10	0	0	0	1	0	1	0	1	1	1	1	0
	11	0	1	0	0	1	0	0	0	0	0	0
6	12	0	0	0	1	0	1	0	1	1	0	1	1
	13	0	1	0	0	1	0	0	0	0	0	0
7	14	0	0	0	1	0	1	0	1	0	1	1	0
	15	0	1	0	0	1	0	0	0	0	0	0
8	16	0	1	0	0	1	1	0	1	0	0	0
	17	0	0	0	0	0	0	0	0	0	0	0
9	18	0	1	0	0	1	1	1	0	0	0	0	0
	19	0	0	0	0	0	0	0	0	0	0	0
10	20	0	1	0	0	1	1	0	0	1	1	1	1
	21	0	0	0	0	0	0	0	0	0	0	0	0
11	22	0	1	0	0	1	1	0	1	0	0	1	1
	23	0	0	0	0	0	0	0	0	0	0	0	0
12	24	0	1	0	0	1	0	0	0	0	0	0	0
	25	0	0	0	0	0	0	0	0	0	0	0	0
13	26	0	1	0	0	1	0	0	0	0	0	0	0
	27	0	0	0	0	0	0	0	0	0	0	0	0
14	28	0	1	0	0	1	0	0	0	0	0	0	0
	29	0	0	0	0	0	0	0	0	0	0	0	0
15	30	0	1	0	0	1	0	0	0	0	0	0	0
	31	0	0	0	0	0	0	0	0	0	0	0	0

Figure 7: *Contents of ROM for the microcode.*

sors and large computers use microcoded design techniques because the design costs associated with hard-wired control are too high.

In some cases, a computer will use RAM instead of ROM for its microcoded memory, providing a "writable control store." A sophisticated programmer can use this to modify and extend the machine's instruction set for special applications. By using multiple sets of ROM or RAM within a machine, the programmer can make a computer emulate more than one machine-code instruction set for different computing environments.

The method of microcoding I used in Toy is called horizontal microcoding, since each bit of the ROM directly feeds a control line for the CPU. A hybrid design method known as vertical microcoding compacts some control signals together to save ROM bits. It then uses decoding logic much like that used by the hard-wired approach to regenerate the signals.

In general, hard-wired control is used for computer designs that are simple or that require fast execution speeds, while microcoded control is used in complex computer designs to keep design costs low. Both design methods can implement CPUs that are much more complex than the Toy architecture. ■

BIBLIOGRAPHY

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

The TTL Data Book, volume 2, Dallas, TX: Texas Instruments Inc., 1985, pages 3-712.

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.

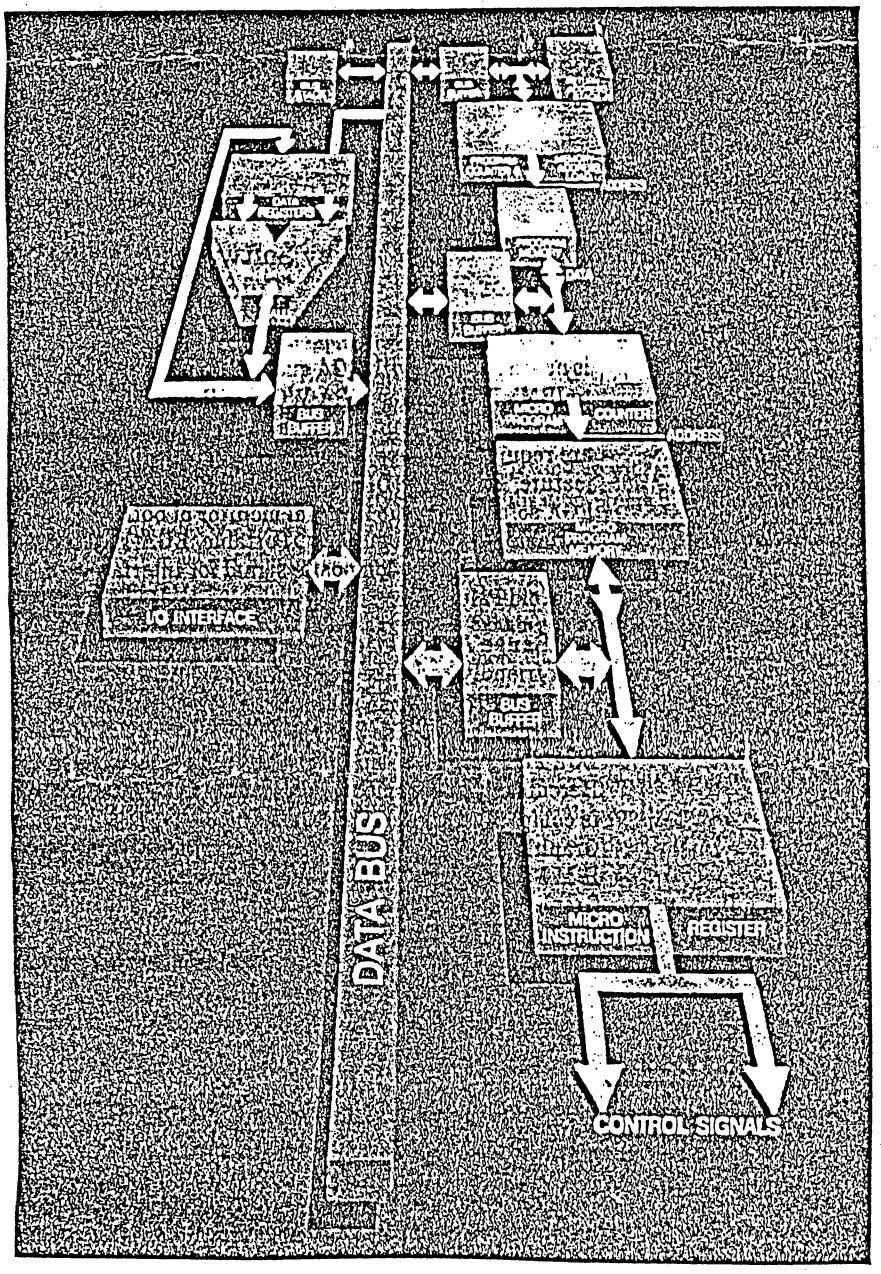


Figure 1: *A block diagram of a possible WISC machine implementation.*

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

Graphics by Imagebuilder Design

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.

A UNIFICATION OF SOFTWARE AND HARDWARE; A NEW TOOL FOR HUMAN THOUGHT

Glen B. Haydon
WISC Technologies, Inc.
La Honda, CA 94020

The following discussion briefly develops a philosophical basis with which to unify the hardware and software tools of a computer development system. The result is an improved match between software and hardware.

The nature of the human mind and thought processes are not understood. However, there appears to be a mismatch between human thought and the rapidly growing use of computers as tools to help men think. Software engineers and hardware engineers seem to be working in different directions. If we could unify the software and hardware of computers along new lines, we might find a better tool to aid us in our intellectual endeavours. Perhaps a unification of software and hardware would provide a better model to simulate part of the activities of the human brain.

Origins of Language

The development of speech and natural languages produced a tool for the development of human thought. In an interesting paper by James Cooke Brown and William Greenhood entitled "PATERNITY, JOKES AND SONG: A POSSIBLE EVOLUTIONARY SCENARIO FOR THE ORIGINS OF MIND AND LANGUAGE " , (*Cultural Futures Research*, Vol VIII, No.2, Winter 1983/84), a new perspective to the development of natural languages is presented. The paper is a long one and carefully argued with many references.

The origins begin with the development of speech as a tool for communication. Along with communication has come the internal activity of the mind, thinking. In the development of language, the burden of disambiguation grows geometrically with every increase in sentence length. The development of grammar attempts to accomplish the disambiguation.

A Logical Language – LOGLAN

In his FORWARD to *LOGLAN 1: A LOGICAL LANGUAGE* , 3rd Ed. (The Loglan Institute, Inc. 1975, 1701 Northeast 75th Street, Gainesville, FL 32601) James Cook Brown begins:

"At the beginning of the Christmas Holidays, 1955, I sat down before a bright fire to commence what I hoped would be a short paper on the possibility of testing the social psychological implications of the Sapir-Whorf hypothesis [relating lanugage to thought]. I meant to proceed by showing that the construction of a tiny model language, with a grammar borrowed from the rules of modern logic, taught to subjects of different nationalities, in a laboratory setting, under conditions of control, would permit a decisive test. I have been writing appendices for that paper ever since. ... "

And now, over thirty years later, the appendices continue to develop. The language became known as LOGLAN. It was described in the literature, in the June 1960 issue of *Scientific American*. Books and publications have continued over the years. Within the past 5 years the language has been refined with a completely unambiguous machine parsable grammar. Currently, a number of minor revisions to the language are being summarized and a new publication should be forthcoming before long.

History of Computing

Several years ago, Hans Nieuwenhuyzen called my attention to two books. The first was *A HISTORY OF COMPUTING IN THE TWENTIETH CENTURY*, (N. Metropolis, J. Howlet and Gian-Carlo Rota, Editors, 1980 Academic Press.) Computers have changed with time. Originally, von Neumann thought of the computer as a number cruncher. Perhaps it was Turing who showed that computers can be symbol-manipulating machines. The hardware design of computers started from these perspectives. Early programming languages dealt with methods trying to use the newly developed hardware to solve real problems.

The second book was *HISTORY OF PROGRAMMING LANGUAGES*, (Richard L Wexelblat, Editor, 1981, Academic Press). The history traces the development of many languages to bridge the gap between real problems and the tools provided with computer hardware. The computer language, FORTRAN was developed as a numerical scientific number cruncher and continues to this day as a major programming language for scientific computation. Other languages which immediately followed were also number crunchers. These were batch processing languages. On-line languages were devised nearly a decade later.

Business applications with number storage and crunching came later. The introduction of string and list processing followed. It was always a problem to make the newer application requirements fit on hardware designed for number crunching. At best, the fit has not been optimal.

Thus the problems addressed with computer hardware expanded from number crunching to assisting in other areas of human thinking and problem solving. As software engineers developed languages, the importance of a divide and conquer approach became apparent. Structured programming became the tool of software engineers. Libraries of program modules were developed. However, the hardware techniques of number crunching do not lend themselves to efficient execution of structured programs requiring sequences of subroutine calls to a variety of modules.

Progress in Hardware Design

In conjunction with the developing languages, the hardware engineers made great strides to support the computational applications addressed by the early languages. The hardware design has been oriented to improving the speed of execution of sequential operations.

In hardware development there has been a trade off between the speed and semantic content of the operations and the physical limitations of the speed of memory access. The increased complexity of instructions increased semantic content of each operation, but with many operations taking many machine cycles. Other techniques have been developed to increase the speed of memory access.

In an alternate approach to increasing hardware speed, hardware designers have tried to reduce the number of operations with each instruction, each of which would then require

only a single processing cycle. Many registers are used rather than slower machine memory to further increase speed.

In the course of these hardware engineering efforts, little attention has been given to efficient subroutine calls.

Progress in Software Design

Software designs have taken other directions. Compilers were developed to translate the newer languages to the machine language of the hardware. Modern language optimizing compilers have many different ways of handling subroutine calls. Not infrequently, when speed is required, the subroutine is simply duplicated in line. Though longer, such machine code will run faster.

Compilation is essentially a batch process. Often multiple passes through the source code are required. Batch processes are slow. A program needs to be completely recompiled to test it. It used to be that such batch programs took overnight to run. Compilers have been designed to run ever faster, but they still require minutes to process. Program development is inhibited by the slow turn-around of batch processing.

With structured programming, it would be desirable to have an instantaneous turn-around on tests of new procedures as they are written. A software development system should also have instantaneous turn-around on tests of connected structures in building the final program. The conventional development systems requiring a compile, load and go for each test is not conducive to good software development.

The Hardware-Software Mismatch

The sequential methods of hardware design are mismatched with structured programming. Sequential methods are also a mismatch with the thought processes of the software developer. The process is almost a random jumping of ideas in the process of thinking. Structured programming seems to be better matched with the thinking process. As such it provides a tool for simulation and study of thought processes. For example: What are the differences between left and right hemisphere processes?

Computer software is divided into smaller and smaller procedures. The process is similar to the divide and conquer process of problem solving. As programs are written, regardless of the language used, they tend to follow a process of natural thought. A translator is required to take a programming language following thought processes and structured programming, and produce machine code which can be run inefficiently on hardware designed to run sequentially.

Unification of Hardware and Software

A rethinking of the hardware design is necessary to better match the direction of software development. Rather than sequential efficiency, what is needed is subroutine call efficiency. It would be ideal if subroutine calls could come for free. This is one of the results of the ideas presented in Phil Koopman Jr's invited paper at this conference. Some of those ideas are summarized here.

Stack oriented Machines

Samelson and Bauer described an ALGOL translator using multiple stacks. (See *A HISTORY OF COMPUTING IN THE TWENTIETH CENTURY* referred to above.) Though a US patent was issued on a full wiring diagram, no hardware was built. At the time, they turned to implementing their ideas in software. Prior to the recent work of Phil Koopman Jr, hardware designers of general purpose processors have not adopted the stack concepts in developing hardware better suited to structured programming.

It is time to adopt the proposals of Samelson and Bauer. An efficient multiple hardware stack machine will contribute to a functional unification of hardware and software. Such a hardware design provides for subroutine calls with no cost in processor time. It contrasts dramatically with the time penalty for subroutine calls.

Writable Control Store

Machine operations should have the semantic content optimized according to the specific requirements of new applications in the software development process. This can be done by using software control of hardware components with writable control store machines. The process divides the hardware components into smaller pieces and allows the software engineer to assemble their functions into optimal operations according to the application requirements.

In the history of computers, writable control structures have been used. Bit slice technology with writable instructions are available but have not been widely exploited.

A Unified Design

A rethinking of hardware design, has led to a writable instruction set computer (WISC) interfaced with multiple dedicate hardware stacks as proposed by Samelson and Bauer.

The first results of such a rethinking of hardware design were presented and discussed at the 1986 Rochester Forth Conference by Phil Koopman Jr and Glen B. Haydon. The design was available then as a wire-wrapped kit. The design is now available on a pair of printed circuit boards.

Also at the 1986 Rochester Forth Conference, Phil Koopman Jr demonstrated the operation of his initial design of an enhanced system. During the past year the design has undergone several itterations. At this, the 1987 Rochester Forth Conference, Phil Koopman Jr is presenting an invited paper in which he details his concepts of the problems and implementation of a hardware design to solve the problems.

Conclusions

I have endeavored to review some of the more philosophical ideas leading to a better match between the computer tools available and the human thought processes. The result has been a unification of structured programming of software engineering with the necessary hardware to run such software efficiently.

To me, one of the greatest potential powers of modern computers is the ability to simulate problems. Perhaps the unification of software and hardware will provide an improved tool to better understand man's way of thinking and problem solving.

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.

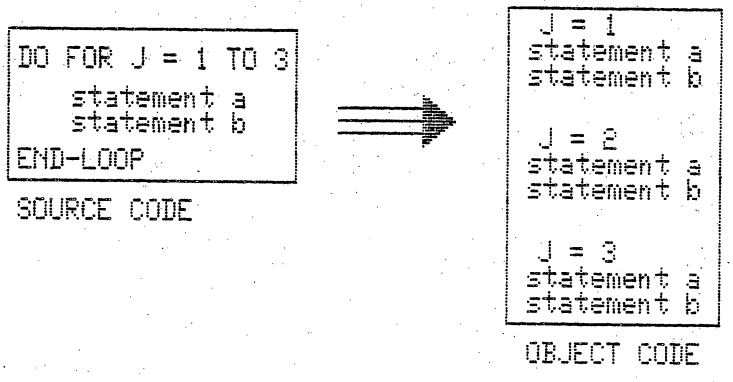


Figure 1a. Unrolling Loops.

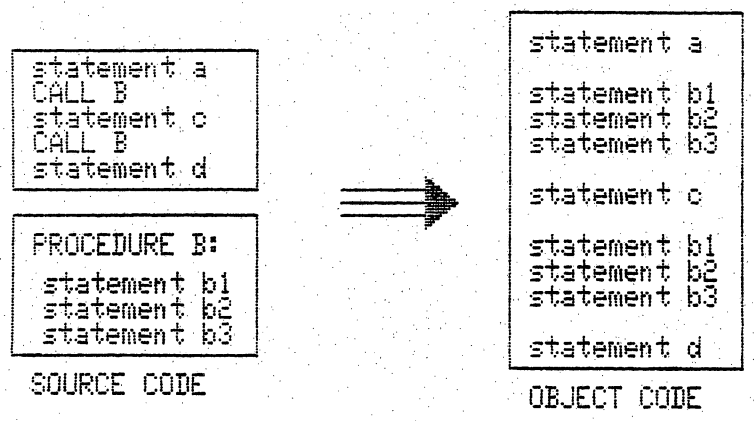


Figure 1b. Expanding procedures in-line.

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.

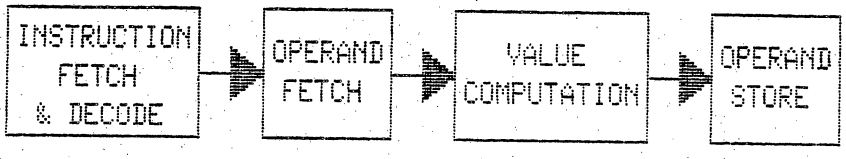


Figure 2a. Pipelining.

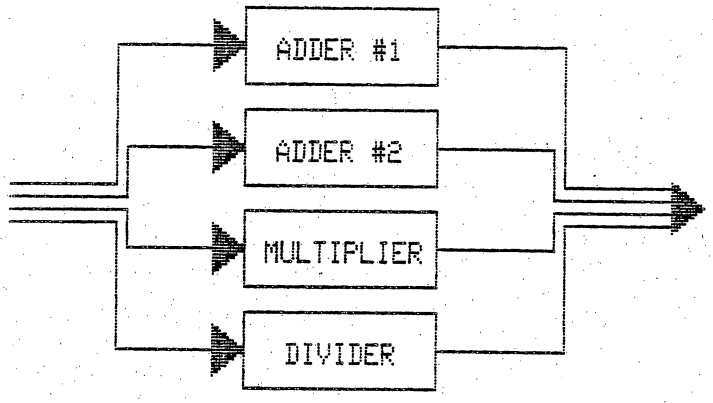


Figure 2b. Parallelism.

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

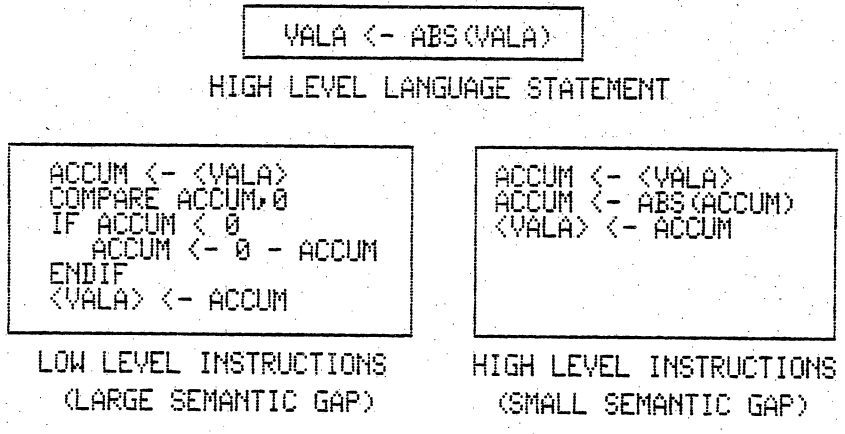


Figure 3. Semantic Gap.

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.

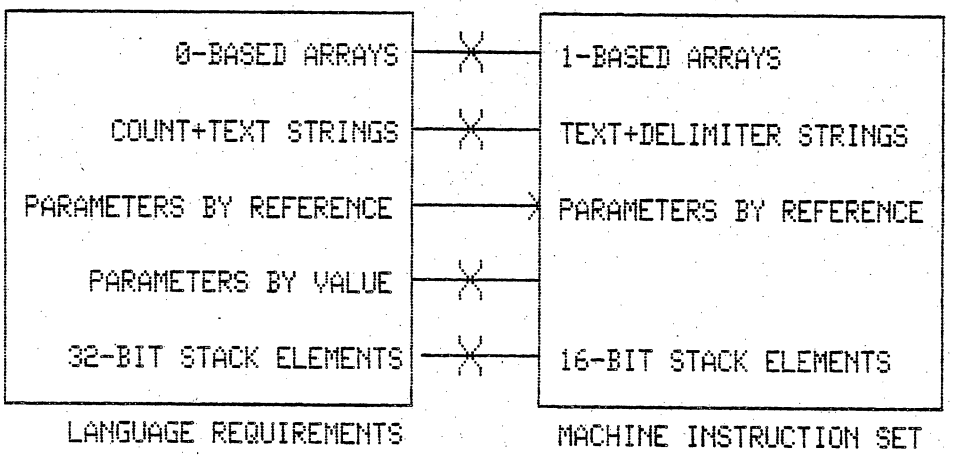


Figure 4. Semantic mismatch.

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.

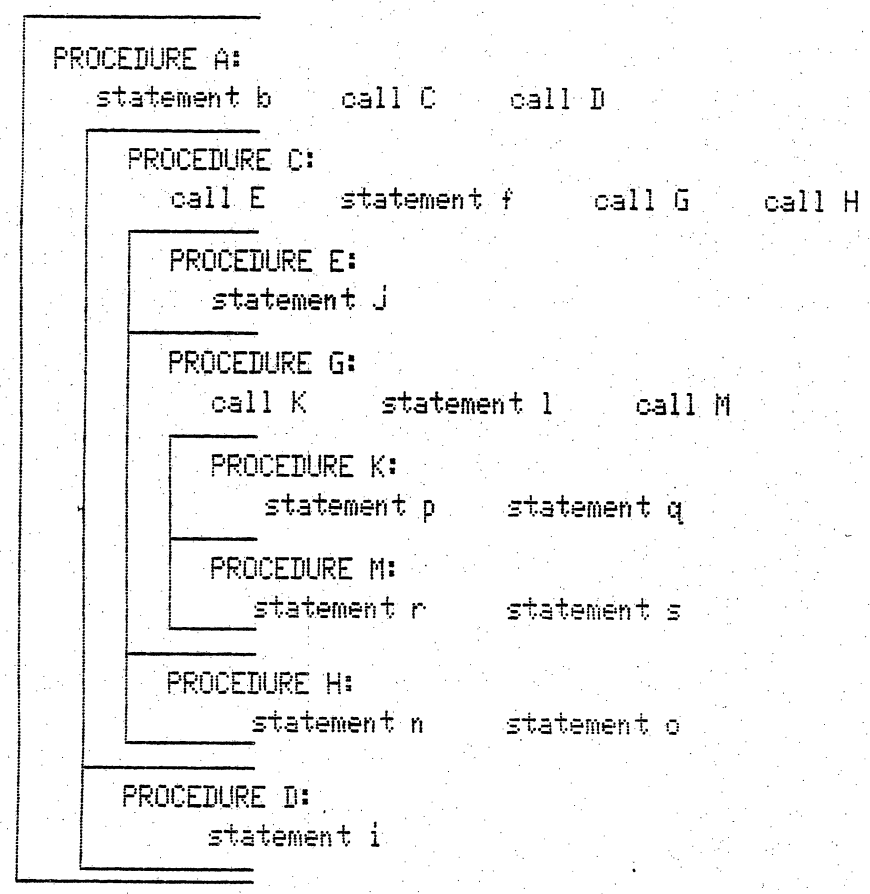


Figure 5. A typical sequential program

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

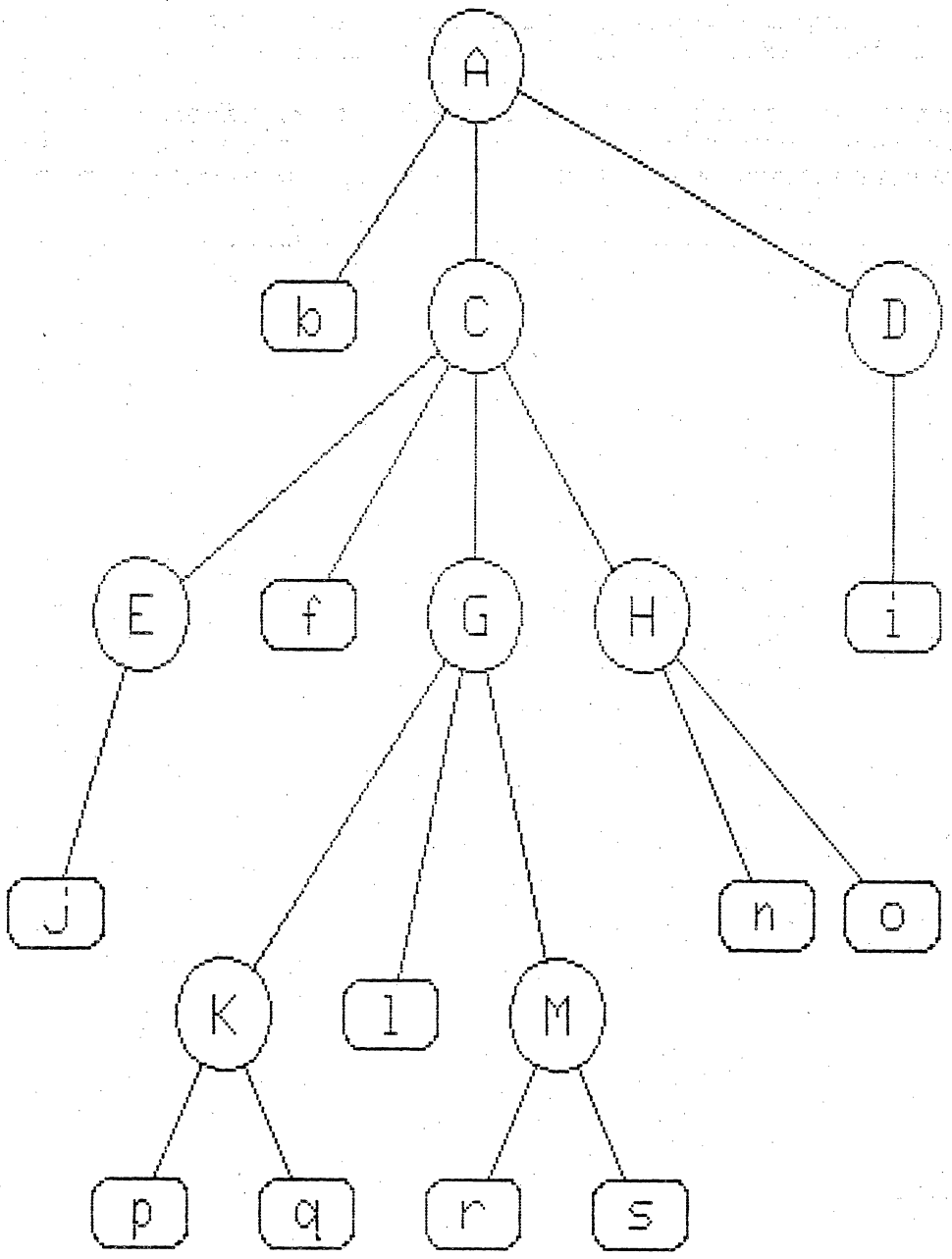


Figure 6. A typical program tree

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.

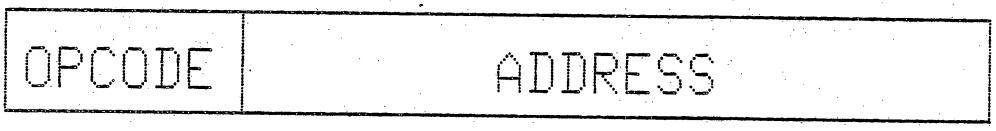


Figure 7. Generic WISC instruction format.

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.

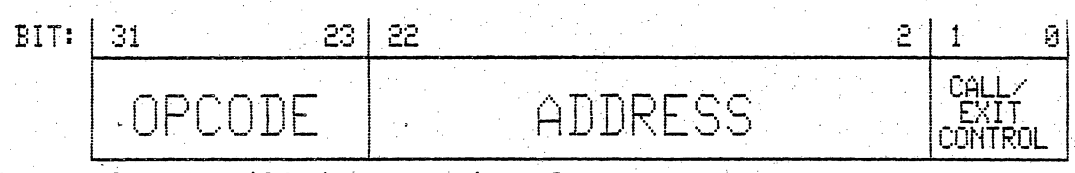


Figure 8. CPU/32 instruction format.

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

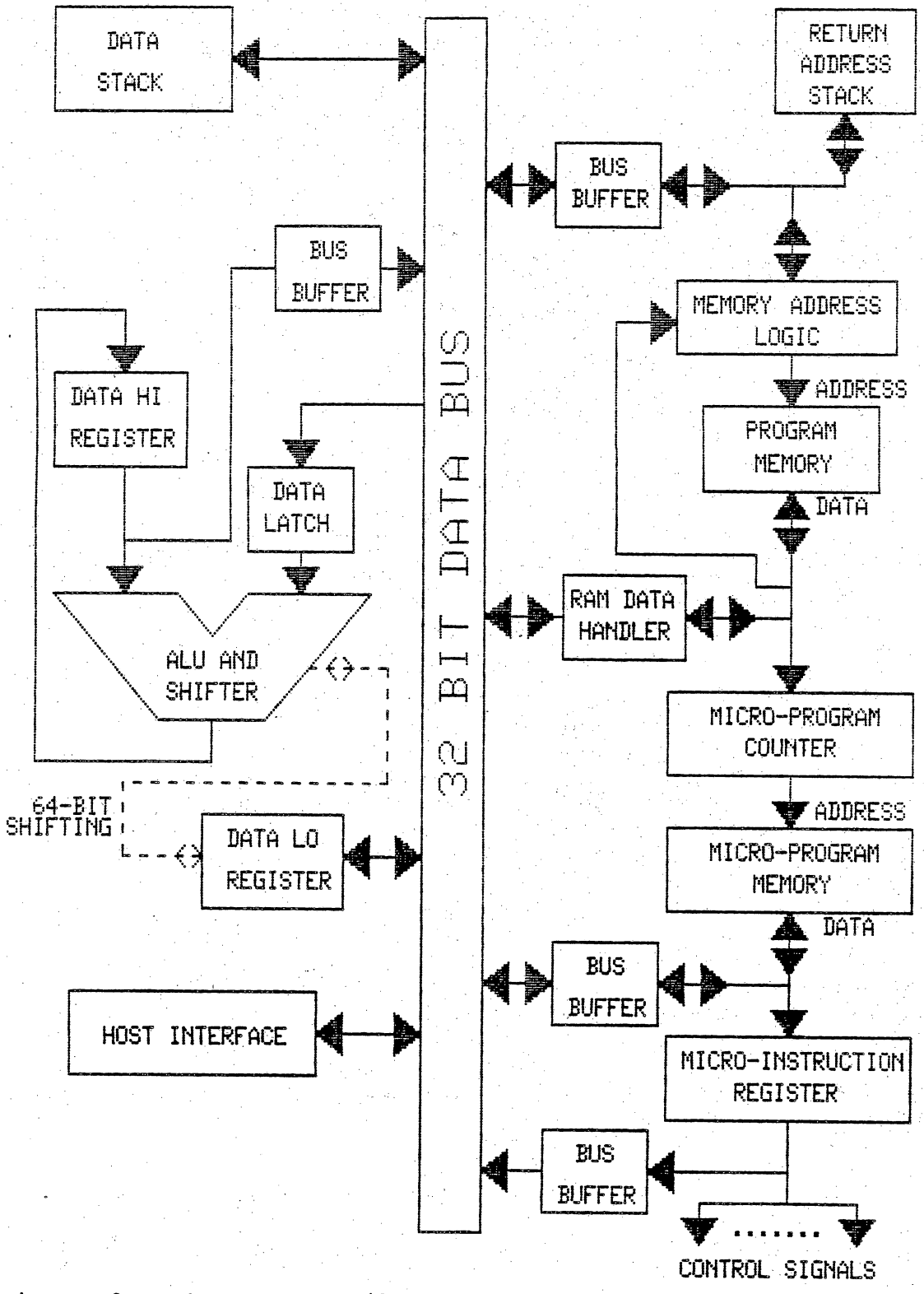


Figure 9. The WISC CPU/32.

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.

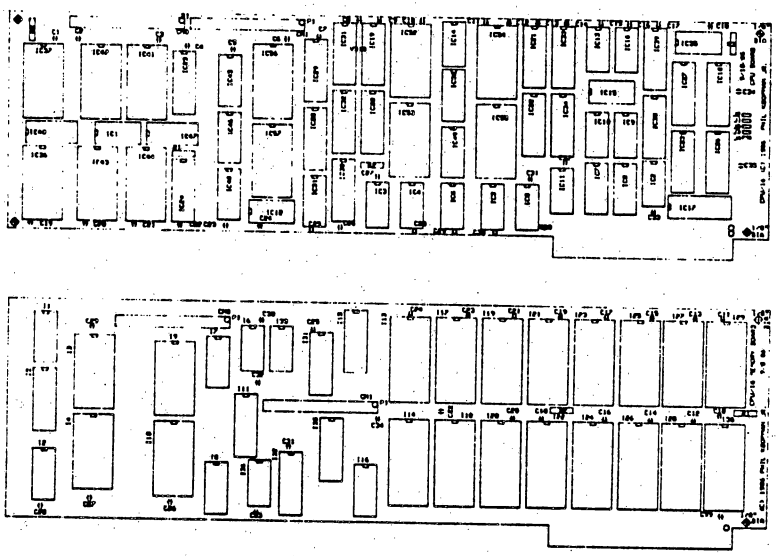
P. Schulthess, "Reduced High-Level-Language Instruction Set", IEEE Micro, June 1984, pp. 55-67.

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.

Stack-Oriented WISC Machine



Stack-oriented, writable instruction set computers, WISC, are for forward-looking project planners searching for state of the art techniques to solve a wide variety of problems. Solutions are easy to formulate, implement, and test with the CPU/16 combination of hardware and software.

The writable instruction set gives a new tool to the project team. It provides the ability to custom design — with software — an optimal set of hardware functions. When efficiently programmed, stack-oriented WISC machines can execute programs faster than conventional machines based on complex instruction set computers (CISC) or reduced instruction set computers (RISC). This versatile new technique encourages development of fully integrated hardware and software systems to solve each new problem.

CPU/16 is a high-speed, stack-oriented WISC machine that includes a processor and memory on two printed circuit boards populated with common TTL components. The boards run as a master processor in an IBM PC, XT, or AT host. Microcode for the WISC processor is written easily with the microassembler, and is loaded from the host along with the application program before the master takes over. Control can be returned to the host at any time, freeing it to execute other programs in a normal manner.

Assembled and tested CPU/16 boards are available, complete with all documentation and software to create customized, high-speed processors. With them a programmer or engineer can implement and test solutions via modifiable microcode. Additional hardware and software in development will expand a growing family of stack-oriented WISC products.

The CPU/16 processor occupies two slots in the IBM PC, XT, AT, and compatibles. Package includes microassembler, cross-compiler, diagnostic programs (all with source code) and complete schematics.

WISC Technologies • Box 429, Star Route 2 • La Honda, California 94020 • USA