my file

SCIENCE AND ENGINEERING RESEARCH COUNCIL
RUTHERFORD APPLETON LABORATORY

COMPUTING DIVISION

DISTRIBUTED INTERACTIVE COMPUTING NOTE 822

Report on visits to U.S. computer companies

Issued by
E V C Fielding

11 March 1983

DISTRIBUTION:

F R A Hopgood
R W Witty
K Robinson
C Prosser
P E Bryant
A S Williams
L O Ford
E V C Fielding
P J Smith

International Robomation/Intelligence
2281 Las Palmas Drive
Carlsbad CA 92008
Telephone: (619) 4384424

I met a Mr. Dave Davidson, who is the Corporate Communications Director. I'm not too sure what that means, but he was very friendly and supplied me with all their brochures and said he would be happy to provide further information should anyone in robotics in the U.K. require it.

The company is mainly concerned with marketing its "affordable robot". This is an air servo power robot arm controlled by a hierarchy of micros with a MC68000 as the master control. Each of its 5 axes is controlled by a 6803 and an additional 6803 acts as a safety control. The robot is referred to as a workstation too - not quite what I was looking for! I saw 3 or 4 of these robots being exercised and they look impressively powerful.

On file at RAL we have an article from Computer Design, January 1983, which describes IRI's "affordable vision system". This has been designed to be the eyes of a robot and is intended to complement their "affordable robot". The applications that they have in mind are: inspection of objects on a conveyor belt for rejction of imperfect articles by the robot; checking that the right object has been grasped and correctly oriented before milling operations, etc.

Unfortunately, the person responsible for developing the IRI P256 vision system was away in Germany. However, I did manage to speak briefly to a person who was programming the system and she gave me a quick demonstration of what they had.

The IRI P256 is well described in the accompanying brochures, so I shall try to describe it only briefly. It is a gray level image analysis system with a resolution of 256X256 pixels with 256 gray levels. It is based on a 12MHz MC68000 with

- 64K bytes of on-board fast SRAM (70ns) from which application programs are executed

- 32K bytes of on-board EPROM in which the real time monitor (RT/M) and some basic utilities (e.g. menu utilities for use with a light pen) are stored

- RS232 and RS422 interfaces

- the system bus is an IRI System Bus.

In addition to the host computer, there is an image digitizer with 4 TV camera inputs which may be selected by program; 256K bytes (4 frames) of frame buffer memory which is in the address space of the host computer; a monitor output; a preprocessor which includes histogram logic; and an optional coprocessor.

There are 2 DMA channels to frame buffer memory:

> The read-modify-write channel for image input and preprocessing. The preprocessor is inserted in the read-modify-write loop of frame buffer memory and so it can perform its functions during image input.
>
> The coprocessor DMA channel.

The preprocessor performs functions such as point transformation, equalization of gray-level distribution for better contrast, histogram calculation, etc.

The optional coprocessor acts as a hardware accelerator for software routines that could be executed in the host computer but 1x to 2x slower. Like the preprocessor, the coprocessor operates in SIMD (single instruction, multiple data) mode. It consists of a microprogrammable controller, systolic array processor, high-speed scratch pad memory and 32K bytes of on-board EPROM to contain the P256 subroutine library. The functions performed by the coprocessor are copying and scaling an image, image subtraction or addition, gradient and moment calculation and filtering by matrix convolution. The subroutine library, may be invoked by program or interactively by using a menu interface and a light pen.

The system which I was shown had no coprocessor, so its functions were being performed by software running on the host MC68000. I was unable to get any idea of when a coprocessor would be available, or how much it would cost. I was shown examples of histogram calculation and plotting, gradient calculation for edge detection, moment calculation and some convoluting and filtering. Most of these operations took a few seconds at least so it looked as if a coprocessor would be required in most real-time situations. The picture quality on the monitor looked much like that of a TV picture. Currently, programs to run on the 68000 are written in FORTH and compiled on a separate system and are down-loaded into the 68000 from cassette tape.

The future plan seems to be to produce 3 versions of the P256 workstation. These are a program development workstation, a test station and a production station.

The program development workstation is to provide a suitable environment under which to develop image analysis programs offline. The UNIX operating system is viewed as providing the ideal environment. An extended version of probably V7 with a real time core will be required - this certainly has not been implemented yet, and I doubt whether they do currently have a C compiler available despite the claims of the Computer Design article. The development workstation will have a terminal, printer, cassette tape and probably more memory.

The test station will be used for real-time testing of image analysis programs. This will probably require more RAM than a production station and a terminal, and preferably a printer. Programs can be loaded from a load terminal (cassette tape) or via the RS422 interface.

In the production station programs will either be loaded into RAM from a load terminal (cassette) or burned into EPROM and loaded from there.

I did not worry too much about collecting company details as this did not look like something we would be interested in as a general purpose personal workstation. If there is interest in the system for a particular application then further information and exact prices could be obtained fairly easily. The price of the basic system with host computer and digitizer and preprocessor boards seems to be $4995.

TeleSoft
10639 Roselle Street
San Diego CA 92121
Telephone: (619) 457-2700

I visited TeleSoft as I know some of the people who work there. I knew that they do not have personal computers with graphics capability, running UNIX. However, as they are on the list of 68000 vendors, I thought that it was probably worthwhile to visit, pick up their literature and to note down a few points.

The main concern of the company is to market their Ada compiler. This is as yet a compiler for a subset of Ada, but a full Ada compiler is under way and should be released by the 4th quarter of this year. The Ada compiler is available under TeleSoft's programming support environment on:

IBM 370 under CMS
Vax under VMS
IBM PC
HP 9836 under HPOS1
standalone on MC68000 for Q-bus and Multibus.

In addition, not described in the accompanying glossies, the Ada compiler is being implemented under UNIX - Berkeley 4.1 on the Vax and UniSoft UNIX on the MC68000.

On the hardware side, the company sells the T68KQ, a Q-bus compatible processor board with 8MHz MC68000 processor. They've also produced a multiple processor system, MPS. This permits up to 16 separate Q-bus based systems to share peripherals over a high speed bus - the O-bus. A further processor, the IOP, handles IO requests and low-level device control for shared devices. The other processors may run any operating systems and can have private peripherals. The device drivers for shared peripherals in these operating systems are modified to communicate with the O-bus.

The company is registered under Renaissance TeleSoftware Inc. but trades under the name TeleSoft.

Altos Computer Systems
2360 Bering Drive
San Jose CA 95131
Telephone: (408) 9466700

After a couple of days of telephoning Altos, I eventually spoke to a Linda Salmon in their International Marketing Division. She was determined that we should go throught their U.K. branch and gave me the following address and names:

Roger Llewellyn or Phil Harris
Altos Computer Systems Ltd.
Office Suite E
Manhattan House
High Street
Crowthorne
Berkshire
Telephone: 344677911
Telex: 849139

She said to come back to her if we were not satisfied by the U.K branch but she felt that they would be able to give us a demonstration and to answer questions relating to U.K. conditions.

Valid Logic Systems Inc.
650 North Mary Avenue
Sunnyvale CA 94086
Telephone: (408) 7731300

The person I met from Valid was Bob Lorentzen, their Product Marketing Manager. His first reaction when I outlined our requirements was that the general personal workstation aspects of their Scaldsystem (structured computer-aided logic design) were of low priority and that the system was targetted so specifically towards supporting logic design that it would potentially not be useful generally. However, their system seemed to come the closest to satisfying the requirements specification of all the systems I saw in the U.S., although what I saw of it did not make me think it matched the PERQ capabilities.

Lorentzen did not know the answers to some of my questions and when in doubt, left me with the impression that he guessed. So it may be worthwhile to arrange with their U.K. branch to see the Scaldsystem to verify details!

Marketing and support is done in the U.K. and the address and name of a person to contact is:

Tom Lawrence
Valid International
Berkshire House
56 Herschel Street
Slough SL1 1TP
Telephone: (0753) 820101

The Scaldsystem consists of a controller, called the S32 Computer System and up to 4 graphics Design Stations.

The S32 is based on the MC68000 processor (guessed 8MHz) and runs System III UNIX.

I was told that the system bus was Multibus and that it would be possible to upgrade the processor to the 68010, but this needs checking.

Up to 4 Mb of memory is supported, configurable in 1/2 Mb quantities.

There are a number of RS232 interfaces - I was unable to determine exactly how many, but there seems to be 1 available for dialup access, 2 or 3 for each design station (for keyboard and graphics and tablet) and 3 or 4 more for printer attachment etc. to the S32.

Ethernet is supported and there is a 56Kbit/sec bisynchronous interface to connect to an IBM 370 and a parallel interface for connection to a VAX. There is no IEEE488 interface, or X25

support. However they would try to assist with mounting X25 software if their system were bought together with their software. LucasFilm apparently buy their hardware without their software - the hardware only is marketed by AIM Technology in the U.S. The U.K. office would also handle any hardware-only purchases. Valid would support only what they had sold obviously.

Winchester disk storage is available in 33, 70, 240 and 420 Mbyte sizes, with a 1/2" streaming tape drive for backup.

The graphics design station is based on the Intel 8086 and has the following features:

20" raster scan display with resolution of 1024x768 points with 4 intensity levels. Landscape orientation only. The refresh rate is 60 times per second. There are no immediate plans for a colour display.

The display station has 272K bytes of display memory which contains the pixel map being displayed. A screen to be displayed has to be transferred into the display memory, either from the S32 memory or a host computer memory or disk.

The display station can be located up to 500 feet away from the S32.

The keyboard has a large number of programmable function keys.

Summagraphics tablet (13x11") and 4 button puck. A nice feature of the design station desk is that they attach the tablet underneath the desk top which makes for more surface workspace.

Software on the system consists of Unix System III and a graphics editor (ged) which cannot be unbundled. Additional application software can be unbundled and consists of logic design verification programs - compiler, timing verifier, logic simulator, a library of standard logic components etc. I saw a demonstration of the graphics editor only - another demonstration was in progress and this prevented a full demonstration as only one person was allowed to log in as demo at a time. The display reminded me very much of the Apollo display. I couldn't get a clear idea of how fast the graphics was from what I was shown. It was possible to alternate between a "Unix display" i.e. a dumb terminal type interface and a "graphics display" - the graphics editor menu down one side and the design being created occupying the rest of the screen. The graphics editor didn't look all that easy to use - a lot of use of function keys was made but there was some menuing - not popup or scrolling though. Text processing programs and editors available were ed, vi, nroff and troff.

Price in the U.K is probably 30000 - 35000 pounds for the design station and 40000 for the S32 controller. This excludes the software for which we would have to get a price from Valid International.

Company details - 65 people, 20000 square feet of office space in the U.S. Venture capital finance. They do all their own manufacture but buy in lots of the components. U.K. office was the last to open. They have systems out in the field.

Corvus Systems
2029 O'Toole Avenue
San Jose CA 95131
Telephone: (408) 9467700

At Corvus I met Bill Gitow, their International Sales Representative. Corvus products are marketed in the U.K. by Keene Computers, who also market the Sun. Corvus started out as a disk manufacturer and have now branched out into marketing their Omninet network and workstations as well.

The system I looked at was the Corvus Concept. This is based on the MC68000 processor. I could not discover what the clock rate is. Further specification details are:

Apple bus-compatible bus

Bit-mapped display, 35MHz, with a resolution of 720x560 points. The display can be used in either landscape or portrait orientation - requires a single switch to be flicked. I was warned by TeleSoft, who had a Corvus Concept on trial, that the graphics was unbelievably slow. The processor handles the screen refresh.

256K bytes of RAM is standard and this is extendible up to 512K.

There are 2 RS232 interfaces, 1 RS422 interface for Omninet and a IEEE 488 board can be plugged in.

Winchester disks come in 6 Mb, 12 Mb and 18 Mb sizes and up to 4 can be daisy-chained.

Mice and tablets are being considered. The keyboard has a number of programmable function keys.

There are floppy disk and video cassete recorder options too.

The display looks a lot like the PERQ but the graphics has only a fraction of the PERQ's speed.

The software is a Pascal system with a UCSD file structure. There is no multitasking so although up to 17 windows can be defined, only one is active at a time. They have a font editor which is inferior to the one on the PERQ (but modelled along similar lines). Fonts contain 255 characters and seem to have 16x16 pixel size characters. The first 136 characters in a font being the Ascii set. They have program editor and word processor called EdWord and Logicalc. There is no software support for networking although there is hardware to support this. They have a rudimentary graphics modelled on UCSD Turtlegraphics.

There do seem to be plans to add memory management and to use UNIX. Xenix is being considered.

The price for a station with 256K bytes of memory and software is $4995 - for 512K bytes it costs an additional $1000. Fortran EdWord and Logi-Calc all cost extra.

The company details are: started in 1979 and now consists of over 400 employees. The turnover was over 27 million dollars last year and should exceed 50 million this year.

Pyramid Technology
2471 East Bayshore Road
Suite 600
Palo Alto CA 94303
Telephone: (415) 4942700

Rick Rashid had passed on Len Ford's name to this company, as well as the fact that RAL had mounted UNIX on top of ACCENT. I arranged to see them to discover what they were doing and described what we had done in general terms. I explained that the work done by RAL was done under a collaboration agreement with ICL, who had rights to it.

The people I met were Ross Bott (a UCSD graduate) and, later on, his boss Rob Ragan-Kelly, who apparently went to university in Sweden. Not that Rob Ragan-Kelly looked all that recent a graduate however. The company gave the impression of being fairly new but I did not discover any definite company details, other than the fact that they were offering three jobs - with salaries in the $55000 - $60000 range, any ACCENT and UNIX experience being a definite advantage!!

They are in the process of building a new machine which is going to be called, originally, the Pyramid Machine. The machine is based on RISC (reduced instruction set machine) ideas - I've attached 3 papers on these which they gave me. It was explained briefly to me as follows: The idea is to reduce the number of instructions in a machine's instruction set to a minimum in order to reduce the logic on boards. The way in which an instruction set is reduced is to have large numbers of registers so that most operations are R -> R and memory accesses are minimized. This allows a very powerful single-chip computer to be produced. Compilers then have to be relied upon to produce good code as the reduced instruction set results in longer sequences of code being generated. In practice though, a simple sequence of instructions can sometimes be faster to execute than a single complex instruction which does the same thing.

The Pyramid machine details are company confidential. What they were able to tell me was: it is TTL-based, will have 1.5 - 3 times the power of a VAX 11/780, can operate in three modes - user/supervisor/kernel, and will support paging. Their machine will have a control stack of 16 frames of 32 registers which will be used by kernel and user. Procedure calls are very fast as on a call all that has to be done is to move a pointer to point to the next frame of registers available for the procedure to work with - the one above, as the frames are organized in a stack. Argument passing is done by having a window into the first 16 registers of the frame above the one currently being used and transferring argument values to these registers before the call.

The machine is designed to be a multiprocessor with up to 4 processors. Their aim is to have a UNIX-like environment. As they require an operating system which has been designed to be distributed they plan to remove as much of the UNIX kernel as is possible while still retaining an external UNIX appearance. The present aim is to implement ACCENT with

UNIX sitting above it.

I was asked to pass on various questions regarding collaboration and information exchange: eg. possibility of a couple of people from RAL working with them for month(s); whether there was any ICL interest in collaboration with them and us; the possibility of one of them visiting the U.K. I shall inform ICL of these details - Bob Hopgood suggested that Richard Stonehouse or Chris Barfield should be contacted.

RISC I: A REDUCED INSTRUCTION SET VLSI COMPUTER

DAVID A. PATTERSON and CARLO H. SEQUIN

Computer Science Division
University of California
Berkeley, California

ABSTRACT

The Reduced Instruction Set Computer (RISC) Project investigates an alternative to the general trend toward computers with increasingly complex instruction sets: With a proper set of instructions and a corresponding architectural design, a machine with a high effective throughput can be achieved. The simplicity of the instruction set and addressing modes allows most instructions to execute in a single machine cycle, and the simplicity of each instruction guarantees a short cycle time. In addition, such a machine should have a much shorter design time.

This paper presents the architecture of RISC I and its novel hardware support scheme for procedure call/return. Overlapping sets of register banks that can pass parameters directly to subroutines are largely responsible for the excellent performance of RISC I. Static and dynamic comparisons between this new architecture and more traditional machines are given. Although instructions are simpler, the average length of programs was found not to exceed programs for DEC VAX 11 by more than a factor of 2. Preliminary benchmarks demonstrate the performance advantages of RISC. It appears possible to build a single chip computer faster than VAX 11/780.

INTRODUCTION

A general trend in computers today is to increase the complexity of architectures commensurate with the increasing potential of implementation technologies, as exemplified by the complex successors of simpler machines. Compare, for example, VAX 11[1] to PDP-11, IBM System/38[2] to IBM System/3, and Intel iAPX-432[3] to 8086. The consequences of this complexity are increased design time, increased design errors, and inconsistent implementations.[4] We call this class of computers, complex instruction set computers (CISC).

Investigations of VLSI architectures[5] indicated that one of the major design limitations is the delay-power penalty of data transfers across chip boundaries and the still-limited amount of resources (devices) available on a single chip. Even a million transistors does not go far if a whole computer has to be built from it.[6] This raises the question as to whether the extra hardware needed to implement CISC is the best way to use this "scarce" resource.

The above findings led to the Reduced Instruction Set Computer (RISC) Project. The purpose of the project is to explore alternatives to the general trend toward architectural complexity. The hypothesis is that by reducing the instruction set, VLSI architecture can be designed that uses the scarce resources more effectively than CISC. We also expect this approach to reduce design time, the number of design errors, and the execution time of individual instructions.

Our initial version of such a computer is called RISC I. To meet our goals of simplicity and effective single-chip implementation, we placed the following "constraints" on the architecture:

1. Execute one instruction per cycle. RISC I instructions should be about as fast as, and no more complicated than, micro instructions in current machines such as PDP-11 or VAX. Furthermore, this simplicity makes microcode control unnecessary. Skipping this extra level of interpretation appears to enhance performance while reducing chip size.

2. All instructions are the same size. This again simplifies implementation. We intentionally postponed attempts to reduce program size.

3. Only load and store instructions access memory; the rest operate between registers. This restriction simplifies the design. The lack of complex addressing modes also makes it easier to restart instructions.

4. Support high-level languages (HLL). An explanation of the degree of support follows. Our intention is always to use high-level languages with RISC I.

RISC I supports 32-bit addresses, 8-, 16-, and 32-bit data, and several 32-bit registers. We intend to

0149-7111/81/0000/0443$00.75 © 1981 IEEE

examine support for operating systems and floating-point calculations in successors to RISC I.

It would appear that such constraints would result in a machine with substantially poorer code density or poorer performance or both. In spite of these constraints, the resulting architecture competes favorably with other state-of-the-art machines such as VAX 11/780. This is largely because of an innovative new scheme of register organization we call *overlapped register windows*.

SUPPORT FOR HIGH-LEVEL LANGUAGES

Clearly, new architectures should be designed with the needs of high-level language programming in mind. It should not matter whether a high-level language system is implemented mostly by hardware or mostly by software, provided the system hides any lower levels from the programmer.[7] Given this framework, the role of the architect is to build a cost-effective system by deciding what pieces of the system should be in hardware and what pieces should be in software.

The selection of languages for consideration in RISC I was influenced by our environment; we chose C and Pascal languages, because there is a larger user community and considerable local expertise. Given the limited number of transistors that can be integrated into a single-chip computer, most of the pieces of a RISC high-level language system are in software, with hardware support for only the most time-consuming events.

To determine what constructs are used most frequently and, if possible, what constructs use the most time in average programs, we looked first at the frequency of classes of variables in high-level language programs. Figure 1 shows data collected by Goldwasser for Pascal language[8] and by Cohen and Soiffer for C language.[9]

The most important observation was that integer constants appeared almost as frequently as components of arrays or structures. What is not shown is that over 80% of the scalars were local variables and over 90% of the arrays or structures were global variables.

We also looked at the relative dynamic frequency of high-level language statements for the same eight programs; the ones with averages over 1% are shown in Figure 2. This information does not tell what statements use the most time in the execution of typical programs. To answer that question, we looked at the code produced by typical versions of each of these statements. A "typical" version of each statement was supplied by W. Wulf (private communication, Nov. 1980) as part of his study on judging the quality of compilers. We used C compilers for VAX, PDP-11, and 68000 to determine the average number of instructions and memory references. By multiplying the frequency of occurrence of each statement with the corresponding number of machine instructions and memory references, we obtained the data shown in Figure 3, which is ordered by memory references.

The data in these tables suggests that the procedure CALL/return is the most time-consuming operation in typical high-level language programs. The statistics on operands emphasizes the importance of local variables and constants. RISC I attempts to make each of these constructs efficient, implementing the less-frequent operations with subroutines.

BASIC ARCHITECTURE OF RISC I

The RISC I instruction set contains a few simple operations (arithmetic, logical, and shift) that operate on registers. Instructions, data, addresses, and registers are 32 bits. RISC instructions fall into four categories (Figure 4): arithmetic-logical (ALU), memory access, branch, and miscellaneous. The execution time of a RISC I cycle is given by the time it takes to read a register, perform an ALU operation, and store the result back into a register. Register 0, which always contains 0, allows us to synthesize a variety of operations and addressing modes.

Load and store instructions move data between registers and memory. These instructions use two CPU cycles. We decided to make an exception to our constraint of single-cycle execution rather than to extend the general cycle to permit a complete memory access. There are eight variations of memory access instructions to accommodate sign-extended or zero-extended 8-bit, 16-bit, and 32-bit data. Although there appears to be only one addressing mode, *index plus displacement*, *absolute* and *register indirect* addressing can be synthesized using register 0 (Figure 5). (Using one register to always contain 0 dates back at least to CDC-6600 in 1964. It has also appeared in more recent designs.[10])

Branch instructions include CALL, return, conditional and unconditional jump. The conditional instructions are the standard set used originally in PDP-11 and are found in most 16-bit microprocessors today. Most of the

innovative features of RISC are found in CALL, return, and jump; they will be discussed in subsequent sections.

Figure 6 shows the 32-bit format used by register-to-register instructions and memory access instructions. For register-to-register instructions, DEST selects one of the 32 registers as the destination of the result of the operation, which itself is performed on the registers specified by SOURCE1 and SOURCE2. If IMM equals 0, the low-order 5 bits of SOURCE2 specify another register; if IMM equals 1, SOURCE2 expresses a sign-extended, 13-bit constant. Because of the frequency of occurrence of integer constants in high-level language programs, the immediate field has been made an option in every instruction. SCC determines if the condition codes are set. Memory access instructions use SOURCE1 to specify the index register and SOURCE2 to specify the offset. One other format, which combines the last three fields to form a 19-bit PC-relative address, is used primarily by the branch instructions.

Although comparative measurements of benchmarks are the real test of effectiveness, the examples in Figure 5 show that many of the important VAX instructions can be synthesized from simple RISC addressing modes and operation codes. Remember that register 0 (*r*0) always contains 0; specifying *r*0 as a destination does not change its value.

Register Windows

The previously mentioned investigations on using high-level languages indicate that the procedure CALL may be the most time-consuming operation in typical high-level language programs. Potentially, RISC programs may have an even larger number of calls, because the complex instructions found in CISCs are subroutines in RISC. Thus, the procedure CALL must be as fast as possible, perhaps no longer than a few jumps. The RISC *register window* scheme comes close to this goal. At the same time, this scheme also reduces the number of accesses to data memory.

Using procedures involves two groups of time-consuming operations: saving or restoring registers on each CALL or return, and passing parameters and results to and from the procedure. Because our measurements on high-level language programs indicate that local scalars are the most frequent operands, we wanted to support the allocation of locals in registers. Baskett[11] and Sites[12] suggested that microprocessors keep multiple banks of registers on the chip to avoid register saving and restoring. Thus, each procedure CALL results in a new set of registers being allocated for use by that new procedure. The return just alters a pointer, which restores the old set. A similar scheme was adopted by RISC I; however, some of the registers are not saved or restored on each procedure CALL. These registers (*r*0 through *r*9) are called *global* registers.

In addition, the sets of registers used by different processes are overlapped to allow parameters to be passed in registers. In other machines, parameters are usually passed on the stack with the calling procedure using a register (frame pointer) to point to the beginning of the parameters (and also to the end of the locals). Thus, all references to parameters are indexed references to memory. Our approach is to break the set of window registers (*r*10 to *r*31) into three parts (Figure 7). Registers 26 through 31 (HIGH) contain parameters passed from "above" the current procedure; that is, the calling procedure. Registers 16 through 25 (LOCAL) are used for the local scalar storage exactly as described previously. Registers 10 through 15 (LOW) are used for local storage and for parameters passed to the procedure "below" the current procedure (the called procedure). On each procedure CALL, a new set of registers, *r*10 to *r*31, is allocated; however, we want the LOW registers of the "caller" to become the HIGH registers of the "callee." This is accomplished by having the hardware overlap the LOW registers of the calling frame with the HIGH registers of the called frame; thus, without moving information, parameters in registers 10 through 15 appear in registers 25 through 31 in the called frame. Figure 8 illustrates this approach for the case in which procedure *A* calls procedure *B*, which calls procedure *C*.

Multiple register banks require a mechanism to handle the case in which there are no free register banks available. RISC I handles this with a separate register overflow stack in memory and a stack pointer to it. Overflow and underflow are handled with a trap to a software routine that adjusts that stack. Because this routine can save or restore several sets of registers, the overflow/underflow frequency is based on the local variations in the depth of the stack rather than on the absolute depth. The effectiveness of this scheme depends on the relative frequency of overflows and underflows; studies by Halbert and Kessler[13] indicate that overflow will occur in less than 1% of the calls with only 4 to 8 register banks. (Other machines, such as BBN C/70, contain register banks, but they do not overlap their windows.)

The final step in allocating variables in registers is handling the problem of pointers. Pointers to variables require that variables have addresses. Because registers do not normally have addresses, one could let the compiler determine what variables have pointers and put such variables in memory. This precludes separate compilation, slows down access to these variables, and is beyond state-of-the-art compiler technology found in most companies and universities. RISC I solves that problem by giving addresses to the window registers. If we reserve a portion of the address space, we can determine, with one comparision, whether an address points to a register or to memory. Because the only instructions to access memory are load and store, and they take an extra cycle already, we can add this feature without reducing the performance of the load and store instructions. This permits the use of straightforward compiler technology and still leaves a large fraction of the variables in registers.

Delayed Jump

The normal RISC I instruction cycle is just long enough to execute the following sequence of operations:

1. Read a register
2. Perform an ALU operation
3. Store the result back into a register

We increase performance by prefetching the next instruction duing the execution of the current instruction. This introduces difficulties with branch instructions. Several high-end machines have elaborate techniques to prefetch the appropriate instruction after the branch,[14] but these techniques are too complicated for a single-chip RISC. Our solution was to redefine jumps so that they do not take effect until after the following instruction; we refer to this as the *delayed jump*. (This approach to branching dates back to MANIAC I in 1952 and is now commonly used in microprogramming.)

The delayed jump allows RISC I always to prefetch the next instruction during the execution of the current instruction. The machine language code is suitably arranged so that the desired results are obtained. Because RISC I is always intended to be programmed in high-level languages, we will not "burden" the programmer with this complexity; the burden will be carried by the programmers of the compiler, the optimizer, and the debugger.

To illustrate how the delayed branch works, Figure 9a shows a sequence of instructions, which, in machines with normal jumps, would be executed in the order 100, 101, 102, 105, ... To get that same effect in RISC I, we would have to insert NOP (Figure 9b). In this case, the sequence of instructions for RISC I is 100, 101, 102, 103, 106, ... In the worst case, every jump could take two instructions. The RISC I software, however, includes an optimizer that tries to rearrange the sequence of instructions to perform the equivalent operations without NOP. Such an optimized RISC I sequence is 100, 101, 102, 105, ... (Figure 9c). Because the instruction following a jump is always executed, and the jump at 101 is not dependent on the ADD at 102, this sequence is equivalent to the original program segment in Figure 9a.

EVALUATION

We will now evaluate the register window scheme, the delayed branch, and the overall performance of RISC I.

Register Windows

The results of running two benchmarks have shown that the window registers have been effective in reducing the cost of using procedures. The puzzle and quicksort programs, discussed below, are highly recursive routines. Figure 10 shows the maximum depth of recursion, the number of register window overflows and underflows, and the total number of words transferred between memory and the RISC CPU as a result of the overflows and underflows. It also shows the memory traffic caused by saving and restoring registers in VAX. For this simulation, we assumed that half of the registers were saved on an overflow and half were restored on an underflow. We found that for RISC I, an average 0.37 words were transferred to memory per procedure invocation for the puzzle program and 0.07 for quicksort. Note that half of the data memory references in quicksort were the result of the CALL/return overhead of VAX.

We also compared the performance of the RISC I procedure mechanism to that of more traditional machines. We chose VAX, PDP-11, and M68000 as representatives of modern computers. Figure 11 shows the numbers of instructions, their total sizes in bytes, and the numbers of register accesses and data memory accesses for these three computers and for RISC I. The data was collected by looking at the code generated by C compilers for these four machines for procedure CALL

and return statements, assuming that two parameters are passed and requiring that 3 registers must be saved. It appears that this scheme reduces the cost of using procedures significantly.

This scheme also reduces off-chip memory accesses. In traditional machines, generally 30% to 50% of the instructions access data memory, with not more than 20% of the instructions being register-to-register.[15] Because RISC I arithmetic and logical instructions cannot access memory, it might be expected that even a higher fraction of the instructions would be data transfer. This was not the case. The static frequencies of RISC I instructions for nine typical C programs show that less than 20% of the instructions were loads and stores, and more than 50% of the instructions were register-to-register. RISC I has successfully changed the allocation of variables from memory into registers. This indicates that RISC I requires a lower number of the slower off-chip memory accesses. It also indicates that complex addressing modes are not necessary to obtain an effective machine.

Delayed Jump

The performance of our scheme can be evaluated by counting the number of NOP instructions in a program. Static figures before optimization show that in typical C programs, about 18% of the instructions are NOP instructions inserted after jump instructions. A simple peephole optimizer built by students reduced this to about 8%. The optimizer did well on unconditional branches (removing about 90% of NOP instructions), but not so well with conditional branches (removing only about 20% of NOP instructions). This optimizer was improved to replace NOP by the instruction at the target of a jump. This technique can be applied to conditional branches if the optimizer determines that the target instruction modifies temporary resources; for example, an instruction that only modifies the condition codes. In quicksort, this removes all NOP instructions except those that follow return instructions. The dynamic effectiveness of the delayed branch must now include the number of NOP instructions plus the number of instructions after conditional branches that need not be executed for a particular jump condition. The total percentages of either type of instruction for three programs discussed below are 7%, 22%, and 4%.

Overall Performance

To judge the effectiveness of the RISC I architecture, we compared it with VAX, because it is an efficient and a popular modern machine, and PDP-11, because it was the first machine with a C compiler and many persons assume that it is an ideal C machine. (This assumption is not valid. Although the development of C language was somewhat influenced by the architecture of PDP-11, most features of C came from B language, which was an interpreted language not tailored to any architecture.) Figure 12 and 13 compare the static numbers of instructions and the static sizes for 11 typical C programs for the three machines. The compilers used are similar: the VAX and RISC C compilers are both based on the UNIX portable C compiler[16] the compiler for PDP-11 is based on the Ritchie C compiler.[17] Experiments comparing the Ritchie and Portable C compilers for PDP-11 have shown that the average difference in the size of generated code is within 1% (S. C. Johnson, private communication, Feb. 1981).

We found that on the average, RISC uses only two-thirds more instructions than VAX and about two-fifths more than PDP-11, in spite of the fact that RISC I has simple instructions and addressing modes. The most surprising result was that the RISC programs were only about 50% larger than the programs for the other machines even though size optimization was virtually ignored.

Our main goal for RISC I was to obtain good performance; thus dynamic results are the most interesting. We used a C program developed by F. Baskett (private communication, Nov. 1980) called "puzzle." This program is essentially a recursive bin-packing program that solves a three-dimensional puzzle. It displays many features of typical programs, except that there are less than 0.2% procedure calls, the call stack gets deep (20 nested procedure calls), and there are a relatively large number of loops. There are several versions of this program. Version A, which we received from Baskett, accesses arrays with subscripts and does not declare register variables. (Register variables are hints, supplied by the programmer, to the C compiler that this variable will be used frequently and should be kept in a register). We produced version B by converting some local variables into register variables. In version C, we changed the way arrays are accessed from using subscripts to using pointers. The dynamic information about each version of this program is shown in Figures 14 and 15 . The statistics of VAX came from an instruction trace program developed by Henry.[18]

RISC I statistics came from a simulator developed by Tamir.

The results of running the recursive quicksort program are also shown in Figure 14. This program sorts 2,600 fixed-length character strings. The only unusual feature of this program is that it has relatively more memory references than most programs. The execution of this program results in 1,713 multiply operations and 1,712 divide operations, which are subroutines in RISC I.

There is much important information in Figure 14. The first is that it made no difference to RISC whether we used version A or B of the puzzle program. This is because the architecture makes it relatively simple for a compiler to allocate local scalars in registers, so there is no need for a language to give hints telling which should be used. Thus, a one-pass Pascal compiler, which does not normally allocate registers for machines like VAX, would likely allocate variables in registers for RISC I and, therefore, result in the same relative memory traffic as version A of the puzzle program.

Note that most commercial compilers do little optimization. For example, even a three-pass, optimizing Pascal compiler for DEC 10 does not allocate locals or parameters in registers.[19] It is unreasonable for architects to expect, in the near future, sophisticated optimization from production quality compilers.

RISC I was successful in reducing the number of data accesses substantially in all programs. The number of instruction words accessed, however, increased. This is because of the number of NOP instructions executed and the inefficient encoding of RISC I instructions. We expect that successors to RISC I could reduce this difference.

The final, and perhaps most important, figure of merit is execution time. This was easy to determine for VAX 11/780, but difficult for RISC I as we do not have any hardware. Our execution time was based on low-level circuit simulations of early RISC I designs. Using student circuit designers, we estimated that a RISC cycle is 400 nsec: 100 nsec to read one of 135 registers, 200 nsec to perform a 32-bit addition, and 100 nsec to store the result in one of 135 registers. We can argue that this is both optimistic and pessimistic: it is optimistic because it is unlikely that students can successfully build something that fast in their first pass, and it is pessimistic because it is likely that an experienced IC design team could build a much faster machine. Nevertheless, the student-technology single-chip RISC I may still be faster than VAX 11/780 for all benchmarks mentioned previously.

We must mention that although our results are encouraging, they are estimates based upon simulations of only two programs. Further benchmarks must be finished before we can accurately characterize the performance of RISC I.

MEMORY INTERFACE

In most computers, the interface to memory is a main performance bottleneck, so this point must be given special consideration. In our discussions and simulations, we assumed that we can access main memory in a single RISC CPU cycle. Depending on the assumptions that we make for our CPU cycle time, and the size of the main memory, this assumption may be too optimistic. We thus reworked our benchmarks also under the assumption that two CPU cycles are required to access data memory. Performance degraded [illegible] 10%, because the register window scheme redu[illegible] the number of off-chip data references. Data references do not constitute a problem, but allowing two cycles to fetch instructions out of memory would reduce performance by almost a factor of 2.

Clearly, this memory interface will be an increasingly critical point as the intrinsic speed of CPU increases with technologic advances. Accesses to memory can be forced to come mainly from on-chip, either with a large register file or with an on-chip cache and associated memory hierarchy.[6]

An on-chip cache would be beneficial for RISC. It is sometimes forgotten that a cache is ineffective if it is too small. In our opinion, an effective data cache would have to be quite a bit larger than our planned register file, especially if it was to provide the same number of ports as the register file. More-complicated tra[illegible]ion and decoding might even strech the basic CPU cycle time. Given the limited amount of circuitry we can place onto a chip at this point, and given the university environment and our student designers, a register file is clearly the safer way to go.

Although the problem of data accesses has been alleviated by the large number of registers and the effective window scheme, the number of instruction fetches has actually increased because of the simplicity of individual instructions. Instruction fetches from main memory are indeed a major speed-limiting factor. An instruction cache is a desirable commodity. Because

there is no need for CPU to write into this cache, its controller can be simpler than that of a data cache. We decided that RISC I should not be burdened with the design of a full-blown on-chip cache, but an instruction cache would definitely be a good idea for the next-generation RISC.

SUMMARY

From our limited experience based on the results of a few small programs, it appears that the reduced instruction set computer is a promising style of computer design. We have convinced ourselves that complicated addressing schemes are not a vital part of high-throughput machines. The register window scheme appears to make significant contributions toward the performance of our architecture and should be seriously considered in other machines.

We have taken out most of the complexity of modern computers without sacrificing much in code density while improving performance. The loss of complexity has not reduced the functionality of RISC; the chosen subset, especially when combined with the register window scheme, emulates more complex machines. It also appears we can build a single-chip computer much sooner than the traditional architectures. We are encouraged by these results and have begun the design of a single-chip RISC I as part of a multiterm class project.

ACKNOWLEDGMENTS

This research was sponsored by the Defense Advance Research Projects Agency (DoD), ARPA order No. 3803, and monitored by Naval Electronic System Command under contract No. N00039-78-G-0013-0004.

The RISC Project has been sustained by a large number of students. We would like to thank all those in the Berkeley community who have helped to push RISC from a concept to an engineering experiment. The contributions of the following persons were important to RISC: C statistics by E. Cohen and N. Soiffer; Pascal statistics by S. Goldwasser; C compiler initially by D. Doucette and K. Shoens with extensive revisions by R. Campbell; RISC 0 optimizer by D. Fitzpatrick; RISC I optimizer by R. Campbell; assembler by R. Campbell and later revised by Y. Tamir; RISC 0 simulator by R. Campbell, E. Lock, and M. Hakam; RISC I simulator by Y. Tamir; ISPS description by G. Corcoran; window scheme based on an idea of F. Baskett, but designed by D. Halbert and P. Kessler; and LSI timing and suggested LSI implementation by M. Katevenis. We would also like to thank L. Dickman, D. Ditzel, R. Hyerle, M. Katevenis, J. Ousterhout, D. Presotto, D. Ungar, and K. Van Dyke for their suggestions on this paper.

REFERENCES

[1] W. D. Strecker. VAX-11/780: A virtual address extension to the DEC PDP-11 family, *Proceedings of NCC* (June 1978), 967-980.

[2] B. G. Utley et al. In *IBM System/38 Technical Developments* (GS80-0237), 1978, 1-110

[3] S. Colley et al. The object-based architecture of the Intel 432, *COMPCON* (Feb. 1981).

[4] D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer, *Computer Architecture News*, 8 (15 Oct. 1980), 25-33.

[5] D. A. Patterson, E. S. Fehr, and C.H. Séquin. Design considerations for the VLSI processor of X-tree. *The 6th Annual International Symposium on Computer Architecture* (April 1979).

[6] D. A. Patterson and C. H. Séquin. Design considerations for single-chip computers of the future, *IEEE Journal of Solid-State Circuits*, SC-15 (Feb. 1980), 44-52; and *IEEE Transactions on Computers*, C-29 (Feb. 1980), 108-116. (Joint special issue on microprocessors and microcomputers.)

[7] D. R. Ditzel and D. A. Patterson. Retrospective on high-level language computer architecture, *The 7th Annual International Symposium on Computer Architecture* (May 1980), 97-104.

[8] S. Goldwasser. Dynamic Pascal statistics (in progress, Sept. 1980).

[9] E. Cohen and N. Soiffer. Static and dynamic statistics of C "CS 292R Final Reports" (University of California at Berkeley, 1980), 101-140.

[10] S. C. Johnson. A 32-bit processor design (Computer science technical report No. 80), Bell Laboratories, 1979.

[11] F. Baskett. A VLSI Pascal machine (Public lecture), University of California, 1978.

[12] R. L. Sites. How to use 1000 registers, *Caltech Conference on VLSI* (Jan. 1979).

[13] D. Halbert and P. Kessler. Windows of overlapping register frames, "CS 292R Final Reports" (University of California at Berkely, 1980), 82-100.

[14] D. Morris and R. N. Ibbett. *The MU-5 Computer System* (Springer-Verläg, 1979).

[15] W. C. Alexander and D. B. Wortman. Static and dynamic characteristics of XPL programs, *Computer*, 8 (Nov. 1975), 41-46.

[16] S. C. Johnson. A portable compiler: Theory and practice, *Proceedings of the Fifth Annual ACM Symposium of Programming Languages (Jan. 1978), 97-104.*

[17] D. M. Ritchie. A tour through the UNIX C compiler (Unpublished), 1975.

The MIPS Machine

John Hennessy, Norman Jouppi, John Gill, Forest Baskett,
Alex Strong, Thomas Gross, Chris Rowen, and Judson Leonard

Departments of Electrical Engineering and Computer Science
Stanford University

Abstract

MIPS is a new single chip VLSI processor architecture that obtains high performance by means of a simplified instruction set, similar to those found in microengines. The processor is a fast pipelined engine implemented in a single NMOS chip. MIPS uses software solutions to several traditional hardware problems, such as providing pipeline interlocks.

Introduction

MIPS (Microprocessor without Interlocked Pipe Stages) is a general purpose processor architecture designed to be implemented on a single VLSI chip. The main goal of the design is high performance in the execution of compiled code. The architecture is experimental since it makes a radical break with the trend of modern computer architectures. The basic philosophy of MIPS is to present an instruction set that is a compiler-driven encoding of the microengine. Thus, little or no decoding is needed and the instructions correspond closely to microcode instructions. The processor is pipelined but provides no pipeline interlock hardware; this function must be provided by software.

The MIPS architecture provides a fast and simple instruction set. This approach is currently used in the RISC project[1]. It is directly opposed to the approach taken by architectures such as the VAX. However, there are significant differences between the RISC and MIPS approaches. The major differences revolve around the philosophy of the architecture. The RISC architecture is simple both in the instruction set and the hardware needed to implement that instruction set. Although the MIPS instruction set has a simple hardware implementation (i.e., it requires a minimal amount of hardware control), the user level instruction set is not as straightforward, and the simplicity of the user level instruction set is secondary.

In the area of software, the RISC project relies on a straightforward instruction set and straightforward compiler technology. The RISC instruction set design is based on this compiler technology. MIPS will require more sophisticated compilers from which it will gain significant performance benefits.

MIPS is designed for high performance. To allow the user to get maximum performance, the complexity of individual instructions is minimized. This allows the execution of these instructions at significantly higher speeds. To take advantage of simpler hardware and an instruction set that easily maps to the microinstruction set, additional compiler-type translation is needed. This compiler technology makes a compact and time-efficient mapping between higher level constructs and the simplified instruction set. The shifting of the complexity from the hardware to the software has several major advantages.

First, the complexity is paid for only once during compilation. When a user runs his program on a complex architecture, he pays the cost of the architectural overhead each time he runs his program. Second, energy is concentrated on the software, rather than on constructing a complex hardware engine, which is hard to design, debug, and efficiently utilize. Software is not necessarily easier to construct, but the VLSI environment makes hardware simplicity important.

The design of a high performance VLSI processor is dramatically affected by the technology. Among the most important design considerations are the effect of pin limitations, available silicon area, and size/speed tradeoffs. Pin limitations force the careful design of a scheme for multiplexing the available pins, especially when data and instruction fetches are overlapped. Area limitations and the speed of off-chip intercommunication require choices between on- and off-chip functions as well as limiting the complete on-chip design. With current state-of-the-art technology, either some vital component of the processor (such as memory management) must be off-chip, or the size of the chip will make both its performance and yields unacceptably low. Choosing what functions are migrated off-chip must be done carefully so that the performance effects of the partitioning are minimized. In some cases, through careful design, the effects may be eliminated at some extra cost for high speed off-chip functions.

Speed/complexity/area tradeoffs are perhaps the most important and difficult phenomena to deal with. Additional on-chip functionality requires more area, which also slows down the performance of every other function. This occurs for two equally important reasons: additional control and decoding logic increases the length of the critical path (by increasing the number of active elements in the path) and each additional function increases the length of internal wire delays. In the processor's data path, these wire delays can be substantial, since they accumulate both from bus delays, occurring when the data path is lengthed, and control delays, occurring when the decoding and control is expanded or when the data path is widened. In the MIPS

CH1739-2/82/0000-0002$00.75 © 1982 IEEE

architecture, we have attempted to control these delays; however, they remain a dominant factor in determining the speed of the processor.

The microarchitecture

Design philosophy

The fastest execution of a task on a microengine would be one in which all resources of the microengine were used at a 100% duty cycle performing a nonredundant and algorithmically efficient encoding of the task. The MIPS microengine attempts to achieve this goal. The user instruction set is an encoding of the microengine that makes a maximum amount of the microengine available. This goal motivated many of the design decisions found in the architecture.

MIPS is a load/store architecture, i.e. data may be operated on only when it is in a register and only load/store instructions access memory. If data operands are used repeatedly in a basic block of code, having them in registers will prevent redundant load/stores and redundant addressing calculations; this allows higher throughput since more operations directly related to the computation can be performed. The only addressing modes supported are immediate, based with offset, indexed, or base shifted. These addressing modes may require fields from the instruction itself, general registers, and one ALU or shifter operation. Another ALU operation available in the fourth stage of every instruction can be used for a (possibly unrelated) computation. Another major benefit derived from the load/store architecture is simplicity of the pipeline structure. The simplified structure has a fixed number of pipestages, each of the same length; instructions move between stages in unison. Because the stages can be used in varying (but related) ways, the pipline utilization improves.

Although MIPS is a pipelined processor, it does not have hardware pipeline interlocks. The five-stage pipeline contains three active instructions at any time; either the odd or even pipestages are active. The major pipestages and their tasks are shown in Table 1.

Table 1: Major pipestages and their functions

Stage	Mnemonic	Task
Instruction Fetch	IF	Send out the PC, increment it
Instruction Decode	ID	Decode instruction
Operand Decode	OD	Compute effective address and send to memory if load or store, use ALU
Operand Store/ Execution	OS/ EX	Store: write operand/ Execution: use ALU
Operand Fetch	OF	Load: read operand

Interlocks required because of pipeline dependencies are *not* provided by the hardware. Instead, these interlocks must be statically provided where they are needed by a *pipeline reorganizer*. This has two major benefits. First, a more regular and faster hardware implementation is possible since it does not have the usual complexity associated with a pipelined machine. Hardware interlocks cause small delays for all instructions, regardless of their relationship on other instructions. Also, interlock hardware tends to be very complex and nonregular[2,3]. The lack of such hardware is especially important for VLSI implementations, where regularity and simplicity are important.

Second, rearranging operations at compile time is better than delaying them at run time. With a good pipeline reorganizer, most cases where interlocks are avoidable should be found and taken advantage of. This results in performance better than that of a comparable machine with hardware interlocks, since the use of resources will not be delayed. In cases where this is not detected or is not possible, no-ops must be inserted into the code. This does not slow down execution compared to a similar machine with hardware interlocks, but does increase code size. The shifting of work to a reorganizer would be disadvantageous if it took excessive amounts of computation. It appears this is not a problem for our first reorganizer.

MIPS has one instruction size, and all instructions execute in the same amount of time (one data memory cycle). This choice simplifies the construction of code generators for the architecture (by eliminating many nonobvious code sequences for different functions) and makes the construction of a synchronous, regular pipeline much easier. Additionally, the fact that each macroinstruction is a single microinstruction of fixed length and execution time means that a minimum amount of internal state is needed in the processor. The absence of this internal state leads to a faster processor and minimizes the difficulty of supporting interrupts and page faults.

Resources of the microengine

The major functional components of the microengine include:

- ALU resources: A high speed, 32-bit carry lookahead ALU with hardware support for multiply and divide; and a barrel shifter with byte insert and extract capabilities.
- Internal bus resources: Two 32-bit bidirectional busses, each connecting almost all of the functional components.
- On chip storage: Sixteen 32-bit general purpose registers.
- Memory resources: Two memory interfaces, one for instructions and one for data. Each part of the memory resource can be 100% utilized (subject to packing and instruction space usage) because one instruction fetch and either one store or load form data memory can occur simultaneously.
- A multistage PC unit: An incrementable current PC with storage of one branch target as well as four previous PC values. These are required by the pipelining of instructions and interrupt and exception handling.

The instruction set

All MIPS instructions are 32-bits. The user instruction set is a compiler-based encoding (i.e., code generation efficiency is used to choose alternative instructions) of the micromachine. Multiple simple (and possibly unrelated) instruction pieces are packed together into an instruction word. The basic instruction pieces are: ALU pieces, load/store pieces, control flow pieces, and special instructions.

The ALU pieces are all register/register (2 and 3 operand formats). They all use less than 1/2 of an instruction word. Included in this category are byte insert/extract, two bit Booths multiply step, and one bit nonrestoring divide step.

Load/store pieces load and store memory operands. They use between 16 and 32-bits of an instruction word. When a load instruction is less than 32-bits, it may be packaged with an ALU instruction, which is executed during the Execution stage of the pipeline.

Control flow pieces include straight jumps and compare instructions with relative jumps. MIPS does not have condition codes, but includes a rich collection of set conditional and compare and jump instructions. The advantages of this approach over a condition code approach have been discussed and empirically demonstrated[4]. The set conditional instructions provide a powerful implementation for conditional expressions. They set a register to all 1's or 0's based on one of 16 possible comparisons done during the Execution stage. The compare and jump instructions are direct encodings of the micromachine: the effective operand decode stage computes the address of the branch target and the Execution cycle does the comparison. All branch instructions have a delay in their effect of one instruction; i.e., the next sequential instruction is always executed.

The special instructions support procedure and interrupt linkage. The procedure linkage instructions also fit easily into the micromachine format of effective address calculation and register-register computation instructions.

MIPS is a word-addressed machine. This provides several major performance advantages over a byte-addressed architecture[4]. First, the use of word addressing simplifies the memory interface since extraction and insertion hardware is not needed. This is particularly important, since instruction and data fetch/store are in a critical path. Second, when byte data (characters) can be handled in word blocks, the computation is much more efficient. Finally, the effectiveness of short offsets from base register is multiplied by a factor of four.

MIPS does not directly support floating point arithmetic. For applications where such computations are infrequent, floating point operations implemented with integer operations and field insertion/extraction sequences should be sufficient. For more intensive applications a numeric co-processor similar to the Intel 8087 would be appropriate.

Systems Issues

The key systems issues are the memory system, and internal traps and external interrupt support.

The memory system

The use of memory mapping hardware (off-chip in the current design) is needed to support virtual memory. Modern microprocessors (e.g., Motorola 68000) are already faced with the problem that the sum of the memory access time and the memory mapping time is too great to allow the processor to run at full speed. This problem is compounded in MIPS; the effect of pipelining is that a single instruction/data memory must provide access at approximately twice the normal rate (for 64k RAMS).

In MIPS, we obtain this increased memory bandwidth by using an instruction cache. This cache forces separation of code and data, but this separation is a regular practice on many machines; in the MIPS system it allows us to significantly increase performance. Because the instruction memory can be treated as read-only memory (except when a program is being loaded), the cache control is relatively simple. The use of an instruction cache allows increased performance by providing more time during the critical instruction decode pipe stage.

Faults and Interrupts

The MIPS architecture will support page faults, externally generated interrupts, and internally generated traps (arithmetic overflow and software gencated). The necessary hardware to handle these discontinuities within the instruction stream in a pipelined architecture are usually large and complex[2,3]. Furthermore, this is an area where the lack of sufficient hardware support makes the construction of systems software impossible. However, because the MIPS instruction set is not interpreted by a microengine (with its own state), hardware support for page faults and interrupts is significantly simplified.

To handle interrupts and page faults correctly, two important properties are required. First, the architecture must ensure correct shutdown of the pipe, without executing any faulted instructions (such as the instruction which page faulted). Most present microprocessors (e.g., Motorola 68000, Zilog Z8000, and the Intel 8086) cannot perform this function correctly. Second, the processor must be able to correctly restore the pipe and continue execution as if the interrupt or fault had not occurred.

These problems are significantly eased in MIPS by the location of writes within the pipe stages. In MIPS all instructions which can page fault do not write to any storage, either registers or memory, before the fault is detected. The occurrence of a page fault need only turn off writes generated by this and any instructions following it which are already in the pipe. These following instructions also have not written to any storage before the fault occurs. The instruction preceding the faulting instruction is guaranteed to be executable or to fault in a restartable manner

even after the instruction following it faults. The pipeline is drained and control is transferred to a general purpose exception handler. To correctly restart execution, two instructions need to be reexecuted. A multistage PC tracks these instructions and aids in correctly executing them.

Software Issues

The two major components of the MIPS software system are compilers and *pipeline reorganizers*[5]. The input to a pipeline reorganizer is a sequence of simple MIPS instructions or instruction pieces generated without taking the pipeline interlocks and instruction packing features into account. This relieves the compiler of the task of dealing with the restrictions that are imposed by the pipeline constraints on legal code sequences. The reorganizer reorders the instructions to make maximum use of the pipeline while enforcing the pipeline interlocks in the code. It also packs the instruction pieces to maximize use of each instruction word. Lastly, the pipeline reorganizer handles the effect of branch delays[6].

Since all instructions execute in the same time and most instructions generated by a code generator will not be full MIPS instructions, the instruction packing can be very effective in reducing execution time. In fully packed instructions, e.g., a load combined with an ALU instruction, all the major processor resources (both memory interfaces, the ALU, busses and control logic) are used 100% of the time.

The optimal packing of instructions is obviously a hard problem (at least NP-complete); however, we are investigating heuristics that we believe will have acceptable running times, yet will produce nearly optimal code in most cases[5, 6].

To show the effectiveness of these optimizations, we ran versions of a program that does reorganization, packing, and branch delay elimination of three input programs. The input programs consist of an implementation of computing Fibbonacci numbers and two implementations of the Puzzle benchmark[7]. All the programs were written in C and compiled to instruction pieces by a version of the Portable C Compiler. The data in Table 2 shows the improvements in static instruction counts.

Table 2: Cumulative improvements with postpass optimization

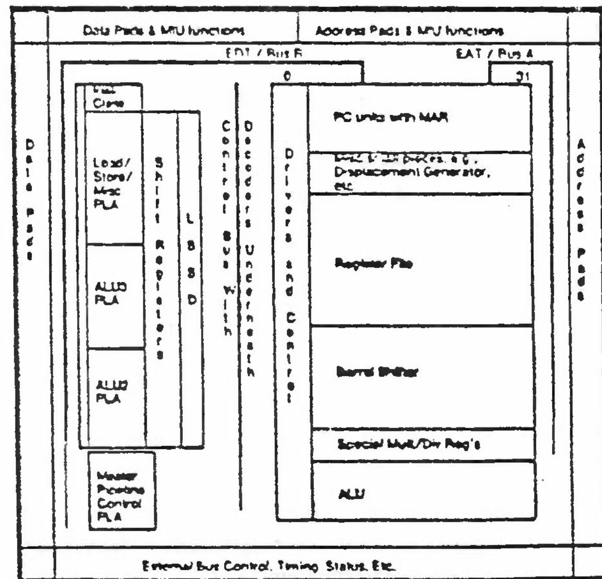
Optimization	Fibbonacci	Puzzle 0	Puzzle 1
None (no-ops inserted)	63	843	1219
Reorganization	63	834	1113
Packing	56	776	992
Branch delay	50	634	791
Total Improvement	20.6%	24.8%	35.1%

Implementation

The primary components of the implementation are the data path design and the control system. Because MIPS requires only simple decoding of instructions into control bits for the data path, the control section is relatively uncomplicated. However, to allow sufficient time to drive control lines and maintain the basic clock speed, the control bits must be predecoded (even this process must occur relatively quickly). To accomplish this several parallel PLA's are used, one for each type of instruction piece. The PLA's are cycled to compute all the control bits during the Instruction Decode cycle of each instruction. Outputs of one of the PLA's is used to choose between the outputs of the other PLA's. A pipline control PLA is used to sequence the processor, handle exceptions, and interface with the external world.

Figure 1 shows the floorplan of the chip. The dimensions of the chip are approximately 6.8 by 7.0 mm with a minimum feature size of 4 μ (i.e., $\lambda = 2\ \mu$). The chip area is heavily dedicated to the data path as opposed to control structure, but not as radically as in RISC implementation.

Figure 1: MIPS Floorplan



We plan to use an 84-pin leadless chip carrier to package the processor. This allows the use of 32 pins for each of the address and data busses. This package also dissipates 2 watts which exceeds our current estimates of power requirements. Our current design calls for the use of a relatively slow instruction cache and a fast main memory, compared to other machines. To accommodate standard bus structures, slower main memory and full speed operation, a data cache may also be necessary.

The data path

The data path is based around a two bus structure. The pitch of the data path is 33λ. The primary components of the data path are the ALU, the barrel shifter, the register file, and the multi-stage PC-unit.

One key component of our data path is a fast 32-bit carry-lookahead ALU. The ALU does an addition with carry while simultaneously doing a logical operation (any one of the 16 Boolean functions of two variables). Either of the input operands may be complemented, allowing subtract and reverse subtract operations. Either the arithmetic or the logical result is chosen for the final result. The ALU sets conditions to indicate zero, negative, carry-out, and overflow (obtained by exclusive-oring carries into and out of the high-order bit of the result).

The basic cell to produce the next-higher propagate and generate signals from four lower-order propagates and generates uses only two gate delays. These gates are short-channel, high-power devices to reduce the effects of stray and wiring capacitances on the basic RC delays. These cells, which logically are the nodes of a binary tree, are replicated in a straight line, and two-layer routing interconnects them in the required tree structure (metal is used for the long direction to virtually eliminate diffusion delay). This arrangement has the advantage that doubling the number of bits merely requires 2 more wiring channels for the next propagate and generate layer, plus half a wiring channel for another layer of intermediate carries (quadrupling the number of bits requires one more carry channel), a total of about 10λ. An optional superbuffer can be included in any bit slice for driving the carry-out for that slice; these are put into those cells with high fanout and long carry lines, the top two layers of the tree. With the Mead-Conway design rules (and a modified implant rule), the ALU is about 300λ high. Using buried rather than butting contacts, shortens the length somewhat.

A test chip for the 32-bit version of the ALU was fabricated at Xerox over the summer at a λ of 2 microns. With inputs stable, the carry-in to carry-out delay was 55 ns (including the output pad driver). Because of fabrication defects, the worst-case timing could not be ascertained.

The barrel shifter selects any contiguous 32-bit field from the 64 bit word made from the abutment of two words, one from each of the busses. Input multiplexers allow both 32-bit words to be taken from the same bus, for rotates. Arithmetic and logical shifts are supported by making one word either zero or the extended sign bit. Byte insert and extract logic is also included. A byte is extracted from a 32-bit word by shifting it to the lowest eight bit positions and masking out the remaining 24 bits. A byte is inserted by shifting the insert byte from the lowest position up to the indicated position and then combining it with the three other bytes of the target word via a byte multiplexer.

The barrel shifter is a two level switch array. The three most significant bits of the shift amount are used to select a 35-bit window, aligned at a nibble boundary, out of the 64-bit input formed by logically concatenating the left and right input busses. This intermediate value is then used as input to a second rank of multiplexors, controlled by the low order shift amount bits, which selects the output function from the intermediate window. The intermediate value is developed on a 35-bit bus which runs perpendicular to the input and output busses, allowing both ranks of multiplexors to be distributed to appropriate sites in the array, rather than routing their inputs and outputs. Select lines run diagonally through the array to control the multiplexing.

The register file is a simple two-port structure. Each bit is represented by a pair of inverters. Each register may be loaded from either of the busses or may be read onto either of the busses. In addition, if a register is not loaded in a given cycle, its data is fed back for refresh. In general, reads are done on one cycle, and loads or refresh on the other. However, in one case it is also necessary to read a register onto the bus going to the data pins in the same clock phase, as data may be loaded into some other register from the ALU result.

The program counter contains the current virtual address of the next instruction to execute, a history of past instruction addresses, and a possible future address that may be loaded conditionally. The program counter proper consists of the current instruction pointer, the increment mechanism with lookahead carry, a shift register containing the last four PC values, and a possible future value. The four previous values are needed to backtrack and restart instructions should a fault occur. The future value supports the branch instructions. The branch address for a conditional branch is calculated in the OD cycle for a given instruction and stored in the future PC register. Only if the specified condition is satisfied is that register jammed into the PC register itself. A given PC can come from one of three sources, from the future PC shift register, from the incrementer (the normal source), or from the PC register itself for refresh. The PC may be read out onto the data bus for address calculation, into the memory address register (MAR) to send it to the address pins, or into the first element of the shift register.

The lookahead carry half adder examines the current PC in four-bit sections and indicates whether carry should be propagated. This carry is combined with the carry from the fourth previous bit to generate the next carry-in. The PC also contains the MAR itself. This is a dynamic latch which may be loaded from either the PC or a data bus. A superbuffer driver drives the metal lines out to the pads, which increase the drive but contain no additional latches.

Only half the S (State and Surprise) register resides physically within the data path. It contains all the other state of the machine outside the program counter, plus a code field for traps and faults. The trap code-field is loaded from the data path on conditional or unconditional traps. The state bits include a field indicating the types of faults occurring in a given cycle, the type of cycle in which they occur, and two flag fields. These two flag fields indicate the current and previous CPU status. The field includes user/supervisor mode bits, the interrupt enable flag, and the

overflow enable flag. On context change, the two fields are interchanged. This state word may be read or modified only by supervisor mode processes, or by interrupts, traps, and hard faults (overflow or page fault).

Several smaller pieces are also included in the data path. A displacement register contains the displacement field of the most recently fetched instruction. It may be used as one of the sources of address calculations. Similarly, small constants, to be used in ALU operations, may be inserted into the data path and sign extended.

The data path has been laid out with standard Mead-Conway type design rules. The two data busses which run the length of the PC, register file, and barrel shifter, were chosen to be metal, while the control lines are polysilicon. This is based on the observation that the control lines are less than half the length of the busses and that they must connect to poly gates in any event. Naturally, the power busses run parallel to the data busses in the PC and register file. Each bit shares Vdd with one neighbor and ground with the other. The barrel shifter required only ground internally. Since the ALU is at the far end of the data path, both busses need not run through it. Control signals and the lookahead carry paths are allowed to run in metal. This contributes significantly to the speed of arithmetic operations.

Present status and conclusions

The key components of the MIPS implementation are the data path, the control system, and the software. The data path components are all designed and specified; a data path layout is near completion and should be sent for fabrication during January 1981. The ALU has been fabricated and works. The control system consists of a number of parallel PLA's and their associated drivers and control bus. A SLIM[8] program for designing the control PLA's has been written and the PLAs have been generated. Code generators have been written for both C and Pascal. These code generators produce simple instructions, relying on a pipeline reorganizer to enforce pipeline interlocks and branch delays, and to pack instruction pieces. A first version of the pipeline reorganizer and an instruction level simulator are being used for dynamic benchmarking.

Early estimates of performance indicate that we should achieve approximately 2 MIPS (using the Puzzle program[7] as a benchmark) compared to other architectures executing compiler generated code. We expect to have more accurate and complete benchmarks available in the near future. Table 3 compares the MIPS processor to the Motorola 68000, running the Puzzle benchmark written in C with no optimization or register allocation. MIPS uses 32-bit integers, while the 68000 uses 16-bit integers. The Portable C Compiler (with different target machine descriptions) generated code for both processors. The execution time numbers for MIPS are an accurate approximation.

Table 3: Comparing 68000 and MIPS estimated performance

	68000	MIPS
Transistors	65,000	25,000
Clock	8 MHz	8 MHz
Data path	16 bits	32-bits
Instructions (Static)	1300	634
Instruction Bytes	6360	2638
Execution Time(sec)	26.8	6.5

Acknowledgments

The MIPS project has been supported by the Defense Advanced Research Projects Agency under contract # MDA903-79-C-0680. Thomas Gross is supported by an IBM Graduate Fellowship.

Many people have contributed to the MIPS project. Among the most important other contributors are: Wayne Wolf, data path design; and Steven Przybylski, fault and trap system.

References

1. Patterson, D.A. and Sequin C.H., "RISC-I: A Reduced Instruction Set VLSI Computer," *Proc. of the Eighth Annual Symposium on Computer Architecture*, Minneapolis, Minn., May 1981, .

2. Lampson, B.W., McDaniel, G.A. and S.M. Ornstein, "An Instruction Fetch Unit for a High Performance Personal Computer," Tech. report CSL-81-1, Xerox PARC, January 1981.

3. Widdoes, L.C., "The S-1 Project: Developing high performance digital computers," *Proc. Compcon*, IEEE, San Francisco, Feburary 1980, .

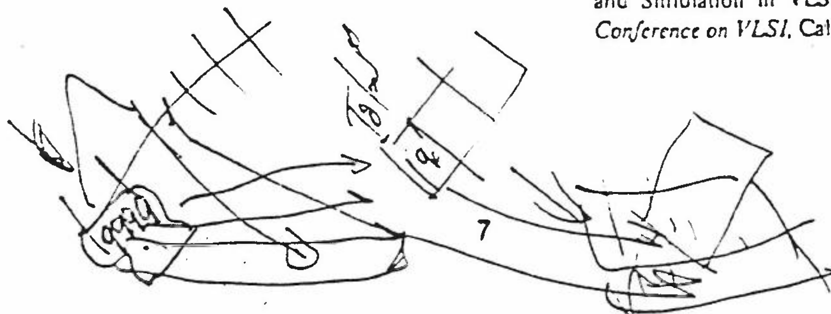
4. Hennessy, J.L., Jouppi, N., Baskett, F., Gross, T.R., and Gill, J., "Hardware/Software Tradeoffs for Increased Performance," *Sym. on Architectural Support for Programming Languages and Operating Systems*, ACM, March 1982, .

5. Hennessy, J.L. and Gross, T.R., "Code Generation and Reorganization in the Presence of Pipeline Constraints," *Proc. Ninth POPL Conference*, ACM, January 1982, .

6. Hennessy, J.L. and Gross, T.R., "Optimizing Branch Delays," Tech. report, Computer Systems Lab., Stanford University, 1981.

7. Baskett, F., "Puzzle: an informal compute bound benchmark", Widely circulated and run.

8. Hennessy, J.L., "A Language for Microcode Description and Simulation in VLSI," *Proc. of the Second Caltech Conference on VLSI*, Caltech, January 1981, .



Running RISCs

John K. Foderaro, Korbin S. Van Dyke, and David A. Patterson
Computer Science Division, University of California, Berkeley

Last year, we described the design of RISC I (Fitzpatrick et al. 1981), a 44,500-transistor 32-bit microprocessor. At that time, we had not yet received our first chips. Now we can tell a complete story, including a moral and a happy ending, in which promptness is punished, Murphy's Law is proved, and perseverance and patience are rewarded.

RISC stands for *Reduced Instruction Set Computer*, a new class of simpler computers promising higher performance using simpler hardware. Examples of RISCs are the IBM 801 (Radin 1982), the Berkeley RISC I (Patterson and Séquin 1981) and the Stanford MIPS (Hennessy *et al.* 1982). RISC I was the first chip built as part of a new graduate curriculum of the Electrical Engineering and Computer Sciences Department at the University of California, in which students propose and evaluate architectural concepts, learn Mead-Conway design methods, form teams to build the system, and then test their design.

RISC CAD

In our environment, design tools dictate design style. Because we had only five students, and had to complete a 32-bit computer in 23 weeks, we had to rely on programs to increase productivity. We used existing programs whenever possible, building our own only when other solutions were not available. We designed RISC I at two levels: a low-level mask description and a high-level functional description. The masks were entered via *Caesar*, a color graphics layout editor developed by John Ousterhout (Ousterhout 1981). We used Clark Baker's *DRC* program to check for layout errors. We counted on visual inspection to discover the few layout errors that *DRC* overlooked (implant-to-gate spacing, gate overhang, and implant overlay around gates).

The functional description was written in *Slang*, a LISP-based simulation language created by John Foderaro during the development of RISC I (Van Dyke 1982). The most difficult parts of our logical design were the timing and the miscellaneous gates to drive the control lines. Because we expected to have more errors in this area than in the data path, a program that would simulate a description of the control circuitry to discover timing and control errors was more important. The RISC I *Slang* description, including all control lines, the PLA's, and miscellaneous control logic, explicitly corresponds to about 2000 transistors in RISC I: *Slang* simulates the rest of the chip—42,500 transistors corresponding to the registers, ALU, and shifter—at a high level. We debugged the description by running about a dozen small RISC I programs, called diagnostics, on the *Slang* description of RISC I. Limited time kept us from using formal fault-coverage models to decide whether these diagnostics adequately exercised the chip.

Two more programs linked these two descriptions. *Mextra*, a circuit extraction program created by Dan Fitzpatrick, takes mask descriptions and derives a transistor-level description. *Esim*, a switch-level simulator created by Chris Terman, simulates the derived description. *Slang* "swallowed" *Esim* to monitor the values of several dozen interesting nodes in both the functional- and switch-level simulations. This multi-level simulation found dozens of disagreements between the two levels of simulation—errors that would have kept RISC I from working.

RISC Fabrication

On June 22, 1981, the RISC I design, using single poly and single metal layers with no buried contacts, was complete: it passed all software checks. RISC I then became part of the experiment to see whether commercial silicon foundries would provide fabrication for small-volume custom designs.

When inexperience is combined with ambition, you get a very large chip. The standard fabrication services were giving fast turnaround to small chips using 5-micron minimum features. Even using 4-micron features, RISC I measures 10.3 x 7.75 mm (406 x 305 mil). Fortunately, the MOSIS Implementation Service at USC/ISI agreed to use RISC I to explore the problems of fabricating large chips at 4 microns. Alan Bell and Lynn Conway of Xerox PARC also offered to fabricate our design, and we replied with a copy of the CIF file on July 17, 1981.

Although Xerox and MOSIS selected different mask-makers, both selected the same vendor for fabrication. The Xerox masks arrived first, but promptness was not rewarded. A new vendor employee ran the wafers through the line using the wrong process. The MOSIS masks arrived later and were sent through the right process steps, but the poly lines were too wide because of problems with one of the polysilicon processing steps. As predicted by Murphy's Law, two more events at the vendor's site kept us from receiving the chips until November: a management reorganization and a fire in the ventilation system.

In November, Michael Arnold finished *Lyra*, a new layout-rule checker (Arnold and Ousterhout 1982). *Lyra* discovered four layout errors—gate overhang and implant-to-gate spacing—overlooked by *DRC*. These probably would not have kept the chip from working, but they might have reduced the yield. The folly of visual inspection was illustrated while verifying the errors. This early version of *Lyra* gave the location and layer of the error; nevertheless, three people, using *Caesar* to explore the design, needed more than two hours to find the errors. The

current version of *Lyra* pinpoints the error and gives complete error messages.

MOSIS offered a new run in December 1981; therefore, we submitted the corrected CIF to be fabricated by a second vendor. This was an experimental run on a research line to try 4-micron Mead-Conway designs. Thus, the need for a second run in April 1982 did not surprise us. Wafers with good processing arrived at Berkeley in May 1982. MOSIS started another run with the original vendor in late January, and we also received their wafers in May.

In the meantime, because of the processing problems, the original vendor refabricated chips from the original masks (M17A) for Xerox. We received chips from this run in January. Table 1 shows the chronology of RISC I fabrication.

Date	Event	Name
April 1980	Experimental Architecture Class	CS292R
September 1980	Mead/Conway Class	CS248
January 1981	VLSI Systems Class	CS292X
April 1981	VLSI 'Testing' Class	CS292Y
June 22, 1981	CIF sent to MOSIS	M17M
July 17, 1981	CIF sent to Xerox PARC	M17A
October 22, 1981	Chips from Xerox PARC	M17A
October 25, 1981	Chips from MOSIS	M17M
November 1981	*Lyra* finds layout errors	
December 4, 1981	Submit new CIF to MOSIS	M1DT
January 7, 1982	Chips from Xerox PARC via refab	M17A
January 29, 1982	MOSIS sends new CIF	M21Z
February 9, 1982	MOSIS M1DT failed fab	M1DT
April 1982	MOSIS refab	M1DT
May 1982	Wafers from MOSIS	M21Z,M1DT
June 11, 1982	RISC I runs first program on board	M17A/CS251
June 15, 1982	Design error discovered	M1DT
July 2, 1982	Find 4 RISC I die without yield errors	M21Z

TABLE 1. History of RISC I.

RISC Testing

Although the fabrication delays were longer than we had expected, building the hardware and software to test the chips kept us busy. The tester receives bit patterns for a test from a host computer, applies these bit patterns to the inputs of the chip under test, and sends the resulting output bit patterns back to the host computer for analysis.

The tester was simply a buffer that could drive and record any pattern of 64 bits every 250 ns. Figure 1 shows a photograph of the tester hardware. This tester provided variable-speed clocks and power supplies, and could repeat a test indefinitely (although it could record only the last 1024 entries). We needed programs on the VAX to prepare the test patterns and massage the test results.

Once again *Slang* came to the rescue, as shown symbolically in Figure 2. *Slang* uses the test programs and computes the correct patterns for the tester and the correct results for those patterns. *Slang* then checks the results from the real system.

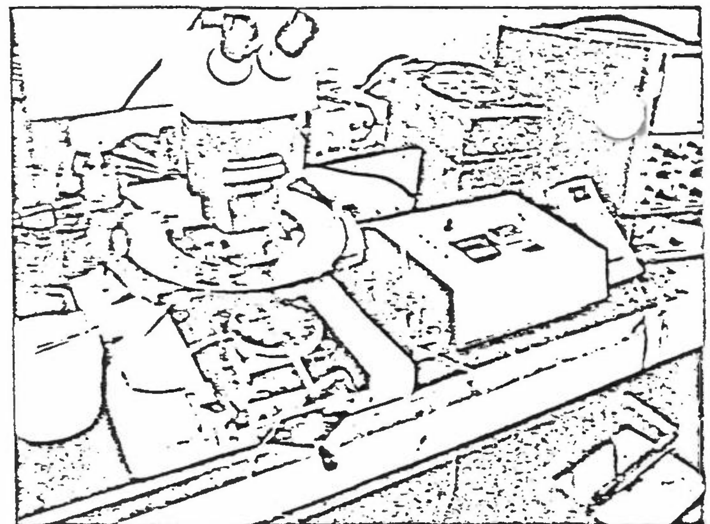


FIGURE 1. Rucker and Kolls 250 probe station with tester (white box on the right).

RISC Results

Out test plan was founded on pessimism. We planned to drive Scan-In-Scan-Out (SISO) hardware to test each block. There are 5 SISO loops in RISC I, one each for the shifter, ALU input, ALU output, the program counters, and control. We tested the first chips from MOSIS (M17M), but no chip emerged with all SISO loops working. We depended on a functioning control loop to test some other modules in the chip, but this loop rarely functioned. The SISO loops were routed after the main area was laid out, resulting in very long poly lines which were potentially more susceptible to yield errors.

Following the same plan, we tested the second batch from Xerox during the winter. Testing proceeded slowly, owing to both the debugging of the RISC I testing software and hardware, and to the other educational requirements of Foderaro and Van Dyke (the only RISC I designers still at Berkeley). We never found working SISO loops on this batch of chips, but two chips displayed "signs of life" when programs were fed to them. Further testing showed that in spite of yield flaws or design errors, these chips performed most of the intended functions. A few bits of the data path were stuck at 0 or 1; nevertheless these chips could still execute some instructions.

Van Dyke designed and built a board for RISC I including the miscellaneous "glue" around the CPU, the memory, the I/O, and the memory management elements. A Xerox refab chip, even with some of the upper bits stuck high, successfully ran the first RISC I program on June 11, 1982. Figure 3 shows a photograph of that board. The first RISC I program rea[d] characters from the terminal, changed the text, and wrote th[e] characters back out.

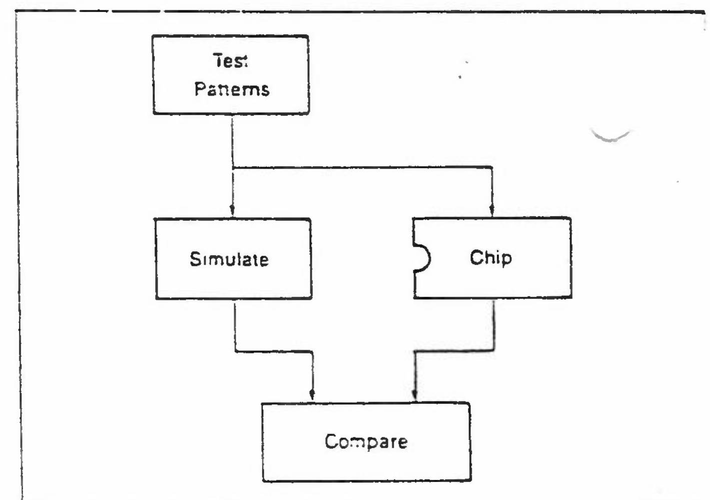


FIGURE 2. Use of Slang in testing.

Because we considered the logical design to be largely co[r]rect, we ignored the SISO loops and began running diagnost[ic] programs on the chips. Before this, we were testing diced ar[...]

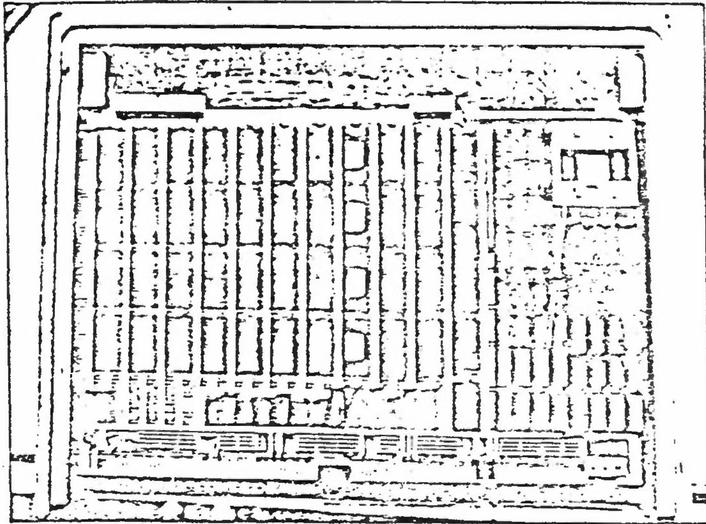


FIGURE 3. Photo of RISC I board.

Machine	Speed MHz	wait states	Language	search	sieve	puzzle	acker
				Time (milliseconds)			
8086	5	0	Pascal	7.3	764	44000	11100
iAPX-432	8	4	Ada	4.4	978	45700	47800
MC68000	8	2	C	4.7	740	37100	7800
Average				5.5	827	42300	22200
RISC I	1.5	0	C	2.5	698	23500	16000

TABLE 2. Execution time of four microprocessors on four programs.

bonded chips. As the students' "lifetimes" were running short, we skipped the dicing and bonding stages and tested whole wafers from the second round of MOSIS fabrication. Figure 1 also shows the probe station we connected to our VLSI tester. These new chips ran the diagnostic programs used to verify our original design. We (foolishly) created new diagnostics and uncovered a design error associated with the optional setting of condition codes on the load and shift instructions. In defiance of historical precedent for solving a design problem by turning it around and calling the problem a new architectural "feature," we decided to fix this error by modifying the RISC I assembler. (This was possible because ALU operations properly set all condition codes, whereas load and shift instructions do not set the negative condition bit. The patch consists of inserting an arithmetic test instruction when a conditional jump needs the N condition from a load or shift operation.) At this writing, we have tested about 40 chips from the second MOSIS wafers. We have found four fully functional chips, (although the SISO circuitry has not been verified yet), giving a 10% yield. This is a better yield than we had expected from such a large chip.

The fastest of these chips runs all diagnostics at 1.5 MHz at room temperature, using the probe card with a floating substrate bias, or 2 μsec per RISC I instruction. This rate was calculated by running the tester at 4-μsec per instruction, with the gap between the three non-overlapped clock phases being as large as the clock. We had based our original performance projections for RISC I on a .4-μsec register-register instruction, derived from the .4-μsec register-register operation of the 10-MHz Motorola MC68000 and the .3-μsec register-register operation of the 10-MHz National NS16032.

Several factors caused this difference. To understand the speed of this fabrication process, we measured the speed of the ring oscillator on the test strip (which is routinely inserted by MOSIS). It ran at 11 MHz. Previous chips processed with the same design-rule features have run the same oscillator at 20 MHz. The later stages of design involved connecting cells, and we concentrated on logical correctness rather than circuit speed. Although we followed the rule of avoiding long unbuffered lines, we had no tool to check for such mistakes. We recently re-examined the design, and found four long clocked control lines that *SPICE* predicts will limit the maximum clock speed to 4 MHz. Furthermore, many of our diagnostics can be run with a 3-MHz clock, suggesting that only a few RISC I instructions (CALL and LOAD) are limiting the performance. Finally, because we still have 200 more chips to test, we may well find faster RISC I's.

Table 2 compares commercial microprocessors to a 1.5-MHz RISC I (including the assembler changes to fix the error). Hansen *et al.* (1982) ran programs on a 5-MHz 8086 with no wait states on an Intellec MDS III development system, an 8-MHz MC68000 with two wait states on a Dual Systems Corporation Dual 8312, and a simulator of the newest version (release 3) of the Intel 432/800, (an 8-MHz, 4-wait state system). As the table shows, a 1.5-MHz RISC I runs these programs a bit faster than do current commercial microprocessors.

A Retrospective Look

Hindsight lets us see our mistakes and offer warnings for future designers. First, don't rely on visual inspection to catch any layout errors. (A second layout-rule checker would have found our mistakes in the first masks.) Our second mistake was incomplete diagnostics. A few more diagnostics would likely have found our only design error. SISO proved difficult to use; if we had written SISO diagnostics, we would have noticed the difficulty and changed the chip.

Although we were concerned with performance, the lack of a simulator between analog-level *SPICE* and switch-level *Esim* precluded performance-tuning of the complete design. Existing higher-level timing analyzers, such as *MOTIS-C* and *LOGIS*, were not integrated into the UNIX environment; perhaps more importantly, they required a new description of the design which could not easily be extracted from the other descriptions. Manufacturers apparently have budgets for hours of *SPICE* runs on CRAY-1's—a luxury not likely to be found soon in academe. We believe higher-level timing analyzers have a promising future.

We hope this article has made it clear that the work required to build hardware and software to test a chip of this size approached the amount of effort required to design it. If we had started over, we would have used more resources on this tedious but important chore. Many people were working on methods to improve chip design, but very few are working on testing. This research area is ripe for new ideas.

The short student "lifetime" requires fast-turnaround silicon. If we have chips with good processing within 3 months after design, we can rely on students to test and perhaps even to improve the chips. If it takes a year, students can rarely enjoy that important advantage. Therefore, one must balance ambi-

tion with die size. MOSIS can provide a 6-week turnaround for standard-size chips that use the standard process (Cohen and Tyree 1982). Furthermore, even special runs have problems with designs larger than 8 mm on a side, because the makers of vendor software (to compensate the masks for the process) never dreamed of a chip that big.

Fortunately, hindsight has also shown successes. For first silicon, the chip had acceptable yield. (One terrible possibility was that the chip might be too large ever to result in a working die.) The working chips testify to the quality of the design tools. (For more information about these tools, see *University Scene* in this issue.) Perhaps the most unusual aspect of this approach to design was that we kept all knowledge of the chip on-line in "program-understandable" form. Furthermore, the on-line description superseded any written documentation. Because the description was finished before the layout was done, we could write a few programs to ask *Slang* what was connected to a given line, or what nodes should be connected. Designers usually keep a set of official logic diagrams for the chip in a project notebook, and derive all other descriptions of the chip from those logic diagrams. We believe that if we had done this, the chip would have taken longer to build and contained more errors, because you can't compare a sheet of paper with an extracted circuit using a program. Unlike the procedure used by most systems, control circuitry was drawn "on-line" using specifications from our *Slang* simulator description.

Perhaps the most unusual aspect of this approach to design was that we kept all knowledge of the chip on-line in "program-understandable" form. Furthermore, the on-line description superseded any written documentation.

The bottom line of the RISC I effort is that as part of the graduate curriculum, students designed and evaluated an architecture, learned VLSI design methods, built new CAD tools, and tested their design. The end product, a 44,000-transistor integrated circuit, had one minor design error, worked on the first good silicon, and ran diagnostic programs faster than commercial microprocessors.

Acknowledgements

Carlo Séquin supervised the master's-degree projects by Wing-Cho Feng and Bob Cmelik, who built the initial VLSI tester hardware and software. The tester built by Jim Beck, which we used to test RISC I, is a revision of this design. John Foderaro and Korbin Van Dyke spent several months testing RISC I, and John wrote many programs to automate testing. Jim Peek did the recent *SPICE* studies of RISC I performance.

We thank Danny Cohen, Lee Richardson, Vance Tyree, and the rest of the MOSIS crew at USC/ISI; and Alan Bell, Alan Paeth, Gaetano Borriello, and Lynn Conway of Xerox PARC for their cooperation and hard work exploring this new field. Their commitment made RISC I possible.

We also thank Alan Bell, Barbara Borske, Danny Cohen, Lynn Conway, Dan Fitzpatrick, Paul Losieben, John Ousterhout, Jim Peek, Lee Richardson, Carlo Séquin, and Jerry Werner for their suggestions on this article.

This research was sponsored in part by Defense Advance Research Projects Agency (DoD), ARPA Order No. 3803, and monitored by the Naval Electronic System Command under Contract No. N00039-81-K-0251. Our thanks also go to Duane Adams, Paul Losleben, Robert Kahn, and DARPA for their foresight in providing resources that let universities attempt projects involving high risk.

References

Arnold, M., and J. Ousterhout. 1982. "Lyra: A New Approach to Geometric Layout Rule Checking," *Proceedings of the 19th Design Automation Conference*, Las Vegas, NV.

Cohen, D. and V. Tyree. July/August 1982. "Quality Control from the Silicon Brokers Perspective," *VLSI DESIGN*.

Fitzpatrick, D.T., J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Séquin, R.W. Sherburne, and K.S. Van Dyke. Fourth Quarter 1981. "A RISCy Approach to VLSI," *VLSI DESIGN*.

Hansen, P.M., M.A. Linton, R.N. Mayo, M. Murphy, and D.A. Patterson. June 1982. "A Performance Evaluation of the Intel iAPX 432," *Computer Architecture News*.

Hennessy, J., N. Jouppi, F. Baskett, A. Strong, T. Gross, C. Rowen, and J. Gill. February 1982. "The MIPS Machine," *Proceedings of COMPCON Spring*, San Francisco, CA.

Ousterhout, J. Fourth Quarter 1981. "Caesar: An Interactive Editor for VLSI Circuits," *VLSI DESIGN*.

Patterson, D.A., and C.H. Séquin. May 1981. "RISC I: [illegible]duced Instruction Set VLSI Computer," *Proceedings of the [illegible]th International Symposium on Computer Architecture*.

Radin, G. March 1982. "The 801 Minicomputer," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*.

Van Dyke, K.S. June 1982. *SLANG: A Logic Simulation Language*, M.S. Report, U.C. Berkeley, Berkeley, CA.

Afterword

We thought the readers of *VLSI DESIGN* might like to know what happened to the Berkeley authors who appeared in the Fourth Quarter 1981 issue. Most graduates joined small start-up companies and, in one way or another, are capitalizing on their IC design experience. First, the students:

Dan Fitzpatrick is finishing his Ph.D. thesis, and working on CAD tools at CADLINC in Palo Alto.

John Foderaro, the only RISC I designer still at Berkeley, plans to finish his Ph.D. in symbolic computation next year. He won the 1982 Dimitri Angelakos award as the person who gave the most help to his fellow students.

Manolis Katevenis is working on RISC II, and doing his Ph.D. dissertation on VLSI computer architecture.

Howard Landman finished his M.S. thesis, and is now building CAD tools at Metheus in Portland, Oregon.

Jim Peek is finishing his M.S. thesis, and is now building a VLSI graphics chip at CADLINC.

Zvi Peshkess finished his M.S. thesis, and is designing analog chips for a communications system at Silicon Systems in Tustin, California.

Bob Sherburne is doing his dissertation on VLSI computer constructs, and also working with Katevenis on RISC II.

Korbin Van Dyke finished his M.S. thesis and is now a VLSI systems engineer at VLSI Technology in San Jose, California. Korbin is probably one of the few people in the world who has investigated architecture, designed a microprocessor, tested the chip, built memory and I/O boards for the chip, written programs for the chip, and seen it all work.

Now the faculty:

Carlo Séquin, chairman of the Computer Science Division, has been teaching seminars on VLSI design across the U.S. He was named a Fellow of the IEEE this year.

John Ousterhout is teaching the VLSI layout class at Berkeley. In addition to supervising new tools such as *Lyra*, he is beginning to work on his next CAD system. His first system, *Caesar*, has been distributed to 70 universities and companies.

Dave Patterson is teaching the Experimental Architecture and VLSI Systems classes that produced RISC I. He is leading the design of an instruction cache for RISC II, and is looking for a new vehicle to investigate cost-effective, VLSI-based software systems. He was awarded the 1982 Distinguished Teaching Award by the Academic Senate of the University of California.