Introduction to the CRISP Instruction Set Architecture

A. D. Berenbaum
D. R. Ditzel
H. R. McLellan

AT&T
Murray Hill, NJ 07974

Introduction

This paper presents the instruction set architecture used in the CRISP Microprocessor. The CRISP architecture contains a relatively small number of instructions and addressing modes, putting it in the category of RISC machines, though it is significantly different from most other RISC machines. These differences contribute to superior architectural and performance characteristics.

Since 1975 the Bell Labs C Machine Project has designed several computer architectures to support the C programming language[1] efficiently. These designs have evolved into the current C Machine instruction set architecture. The CRISP Microprocessor is one particular implementation of that architecture. Since we wish to describe the capabilities of a particular implementation in this paper, we will use "CRISP" in describing both the architecture and the implementation. While C Machines are designed to execute the C programming language particularly well, this by no means excludes the efficient execution of other languages.[2]

Motivation for the Design

There were several problems with the general purpose computers we were using. Despite several years of technology advances, the speed of machines had not significantly increased. On the other hand, the number of instructions and addressing modes had grown considerably. This large number of instructions and addressing modes made our compilers excessively complex, slow and likely to contain bugs. Measurements showed that overhead due to procedure calls constituted nearly half of the execution time of some programs, while accounting for only about 5% of the instructions executed. Measurements also showed that many of the more complex instructions were essentially unused, and the majority of the work was being done by a small number of relatively simple instructions. Machines seemed to have been designed for particular technologies and performance levels, and lacked scalability.

Our initial goal was to look for substantial performance increases through a synergistic design of the instruction set and innovative implementation techniques. Measurements were made on existing and proposed machines to gain insight on how to eliminate unneeded features and how to concentrate hardware on performance bottlenecks. One decision was to concentrate on a fast procedure call mechanism, and in particular to avoid the problem of saving and restoring a general register file. To create a match between the compiler and the hardware we wanted a simple and clean instruction set. We wanted compilers to be able to generate code easily, without requiring sophisticated optimization techniques. On the other hand, we designed the instruction set with provisions for a continually improving compiler technology.[3] Principles of computer architecture were applied where measurements of existing machines gave us little insight. For example, to facilitate high performance pipelined implementations the instruction set was streamlined and tailored to avoid side effects.

A 2½ Address Machine

CRISP's addressing structure is that of a 2½ address memory-to-memory style architecture. In this respect CRISP is quite different from register oriented one address load/store RISC machines. Memory-to-memory means that there are no visible general registers; operations access their operands directly via memory addresses. (There actually are data registers, they are simply hidden from the programmer.) Instructions may specify two memory operands. The first operand specifies a source, the second specifies an address to be used as both source and destination as in:

$$\mathbf{A \leftarrow A + B}$$

An additional "half address" comes from being able to specify a third address for a destination, though not a full third address. The result of an operation, if not placed back at the address of one of the sources, may be placed in a single location called the **Accumulator** as in:

$$\mathbf{Accumulator \leftarrow A + B}$$

An accumulator rather than a full third address was provided because our measurements showed that most of the use of a full third address destination field by the compiler was for a single scratch register. A single accumulator therefore provides most of the benefit without the cost of implementing a full third address. The accumulator is implemented not as a special machine register, it is simply a shorthand way of referencing the first variable location in the stack frame. The accumulator is always written as a 32-bit word by instructions using it as a destination. Since the accumulator is a word in memory, subsequent instructions can read or write individual bytes of that memory location. Most data-modifying instructions have both 2 address and 2½ address forms.

The Stack

A runtime stack frame is used to allocate space for local variables, incoming and outgoing arguments, and compiler generated temporaries. A stack pointer register maintains a pointer to the top of the current stack. The stack pointer is used to delimit the boundary between free and used space in the stack. To allocate space for a new procedure frame the stack is grown downward, toward address location zero. All variables in the stack can therefore be addressed with positive offsets from the current stack pointer. The word pointed to by the stack pointer is reserved for saving the PC return address during a procedure call. The accumulator is one word above the address pointed to by the stack pointer. A typical stack frame is shown in Figure 1.

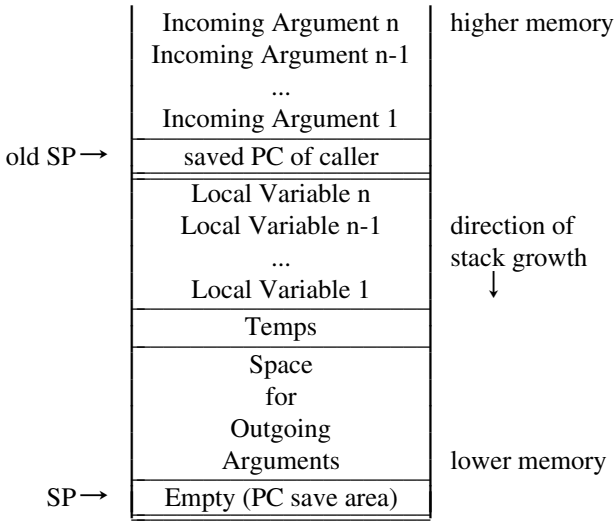


Figure 1. Stack Frame Layout

Processor Registers

CRISP has a small number of processor registers dedicated to specific functions. These are:

SP Stack Pointer. Holds the address of the top of the user stack.

ISP Interrupt Stack Pointer. Holds the address of the top of the interrupt stack.

MSP Maximum Stack Pointer. Delimits the bounds of data kept in the Stack Cache.[4]

VB Vector Base. Points to a table of distinct transfer addresses for kernel calls, interrupts, exceptions, and unimplemented instructions.

PSW Program Status Word.

Timer May be set to count clock ticks or instructions executed.

Pointers in these registers are forced to be aligned on four-word boundaries. The program counter is not an explicit machine register that may be read or written; software may only change its value through explicit jumps or procedure calls. The processor state consists of all the registers, though a user process needs only the **PSW**, **SP** and program counter for state.

Data Types

CRISP is a full 32-bit machine. The wordsize and addresses are 32-bits. Integer values may be 32-bit words, 16-bit halfwords or 8-bit bytes. All ALU operations are performed on 32-bit operands. Bytes and halfwords are either sign extended or zero filled to 32-bit values before an ALU operation takes place. If the size of the destination is less than 32-bits, then the least significant bits of the ALU are used for the result. Support for other datatypes, such as floating point, is provided through an off-chip co-processor interface channel.

CRISP is a byte-addressable machine. Bytes may be accessed on any boundary, though halfwords must be aligned on 16-bit boundaries, and words on 32-bit boundaries. The byte order is high-to-low, the same as the WE32100, IBM 360 and MC68000.

Addressing Modes

CRISP uses four addressing modes per operand for accessing most data. Memory operands may specify any of the available data types.

Immediate mode addressing allows a constant to be embedded in the instruction itself.

Absolute addressing uses an absolute address in the instruction to access data. Absolute addressing is typically used to reference global variables.

Stack Offset addressing develops the address of an operand by adding a constant offset in the instruction to the address of the current stack pointer. This addressing mode is used to access local variables, temporaries, and incoming and outgoing arguments.

Stack Offset Indirect addressing adds a constant offset in the instruction to the address of the current stack pointer. The word at this address is fetched and then used as an address to obtain the data operand.

Arithmetic and Logic Operations

The number of distinct data-modifying instructions is relatively small. Basic arithmetic is provided with signed **addition, subtraction, multiplication, division** and **remainder**. Multiplication, division and remainder are also provided for unsigned operands. Logic operations are provided with bitwise **and, or** and **exclusive or**. There are three shift operations, a **left shift**, **arithmetic right shift**, and **logical right shift**. The arithmetic right shift will repeat the sign bit into the most significant bit, whereas a logical right shift will fill the most significant bits with zeros. All the above operations are available in both 2 address and 2½ address forms, except for the unsigned versions of multiplication, division and remainder, which have only 2 address forms.

Data Movement

Data is transferred using a **move** instruction. Since the source and destination data sizes need not be the same, **move** can be used to convert data lengths. For example, a signed byte can be sign extended and moved into a halfword. For address computations, a **move address** works like a move, but the address of the source is stored in the destination.

Compares and Branches

Like many machines, CRISP uses separate compare and conditional branch instructions. Unlike other machines which compute comparisons based on the setting of Zero, Negative, Carry and Overflow condition code bits, CRISP uses a single flag bit called the True/False bit. The True/False bit is only modified by an explicit compare instruction, rather than as a side effect of every instruction. This simplifies compiler code generation[5] as well as assisting the implementation of branches. The conditional jump instruction can branch if the condition code bit is true, **ifTjmp**, or if the bit is false, **ifFjmp**. In addition the compiler may offer a prediction for which way the conditional jump is expected to go. Proper setting of the branch prediction bit may improve performance in some implementations. An unconditional jump, **jmp**, will always transfer to the branch target.

Four addressing modes are available for jump instructions. Similar to data addressing, branch targets may be specified with **absolute** or **stack offset** indirect addressing modes. **Program Counter relative** addressing is available instead of an absolute address if the offset fits in a short encoding form, to be described later. To facilitate the implementation of branch tables, a jump may use **absolute address indirect** mode.

Only three compare instructions are provided. Two operands may be compared for equality, **cmp.=**, for one being less than another when both operands are signed, **cmp.s<**, and for one being less than another when both operands are unsigned, **cmp.u<**. All other comparisons can be obtained by changing the order of the operands or inverting the sense of the conditional branch associated with the compare.

CRISP also has flags to detect carry and overflow conditions. These are of particular use for languages such as ADA that require detection of overflow. The carry and overflow bits are *sticky*, that is, once set they will remain set even if a subsequent arithmetic operation does not cause an overflow or carry. This means that an entire basic block can be executed and overflow needs to be checked only once. The carry and overflow bits are interrogated with the conditional jump instructions, **ifCjmp** and **ifOjmp**. Reading the carry or overflow flags, with either a conditional jump or by explicitly reading the PSW, will cause the respective bits to be cleared.

Procedure Calling

Four instructions are used for calling procedures, **call, enter, return** and **catch**. Each has one argument field. The **call** instruction saves the return address at the location pointed to by the current stack pointer, then jumps to the target address. The addressing modes available to a call are the same as for a jump. The first instruction inside the called procedure will be **enter**. The enter instruction subtracts its argument, the size of the new stack frame, from the current stack pointer to form a new stack pointer. This allocates space for new local variables, temporaries and outgoing arguments. To exit a procedure, the **return** instruction adds its argument, the size of the current stack frame, to the current stack pointer, returning the stack pointer to its value before the call took place, de-allocating the stack frame. Return then branches to the word pointed to by the new stack pointer, namely the return address. The **catch** instruction is the target of the return instruction, and is used to specify that the Stack Cache should be filled with as many entries as given in its operand. Much of the time the Stack Cache will be sufficiently full, and catch does not need to restore any entries. Details on the Stack Cache can be found elsewhere.[4]

Special Instructions

Three instructions are provided for the implementation of semaphores. Two-address forms of **add, and**, and **or** can be interlocked on the bus during their execution. Interlocked operations assert a "LOCK" pin on the microprocessor throughout the fetch and subsequent store of the operand. The original value of the destination operand is placed in the accumulator for the interlocked operations.

Off-chip processing is accomplished with a coprocessor instruction. The coprocessor instruction can send zero, one, or two source operands of 1, 2, 3 or 4 words to a coprocessor, and receive a result of 0, 1, 2, 3 or 4 words from the coprocessor.

On-chip special registers are accessed by prefixing an instruction with a 2-byte **cpu** register mode escape instruction. This instruction prefix modifies the addressing modes of the next instruction such that the encoding for signed halfword references will instead access internal registers. The machine registers may only be written by privileged processes; non-privileged processes may read the registers.

Instruction Encoding

The number of instructions and addressing modes to be encoded is relatively small, as shown by Table 1. The encoding of the instruction set affects not only code density, but also the ease with which instructions can be decoded, thereby affecting the efficiency of a pipelined implementation. An important consideration for pipelining was being able to determine the length of an instruction quickly. The length of all CRISP instructions is encoded in the first two bits. Some RISC microprocessors use a single fixed length instruction to avoid the need to determine instruction lengths. Unfortunately, this approach can result in poor code density. For some machines code density can be worse by a factor of two or more than

machines such as the VAX-11.

The issue of code compaction is important for several reasons. Better code density can lower bus traffic, and enable an on-chip instruction cache to be more effective by holding more instructions. Both of these can lead to better performance. As memory prices have dropped, programs have continued to grow. The cost of memory still dominates in some designs, poor code density can result in significantly increased system costs. For these reasons, and others, CRISP uses variable size instructions to obtain good code compaction.

The 2 and 2½ address instructions may require two 32-bit constants or addresses. This leads to the encoding shown in Figure 2. The first two bits, 11, encode the length as ten bytes. Six bits are used for the opcode. Source and destination addressing modes are each encoded in four bits for an immediate constant, plus the three memory addressing modes for each of the five datatypes.

If all instructions were encoded with the ten byte format, code space requirements would be prohibitively large. Fortunately, we know that most commonly used constants and offsets can be represented in fewer than 32-bits. Providing for two 16-bit fields, or a single 32-bit field for one address instructions yields the formats in Figure 3.

Instructions are compacted further still. To select smaller encodings, we compiled all of the UNIX source to determine what combinations of addressing modes and opcodes occurred most frequently. The 32 most commonly occurring 14-bit patterns for opcode and addressing modes were encoded in a single five-bit field. When these frequently occurring forms existed with operands of only a few bits we could encode an entire instruction in two bytes. The 2-byte format is shown in Figure 4.

The two byte form is distinguished by a single leading zero bit. Opcode and addressing modes are both encoded in a single 5-bit opcode field. For two operand instructions two 5-bit fields allow for constants in the range of −16 through 15, or a 5-bit word offset in the range 0 through 32. For branch instructions a 10-bit field is used for PC relative addressing. Instructions are aligned on half-word boundaries. This alignment allows PC relative addresses to have an extra bit of addressability compared to byte aligned instructions. About 80% of instructions typically fit in the 2-byte form.

Sharing an Address Space

It is desirable to have both user code and operating system code share the same address space. This is usually done by dividing the address space into large sections, for example, by having the high order address bit turned on for the operating system but not for the user. This allows the operating system easy means to access user process data. Unfortunately, the convenience of sharing a single address space might have caused code space to grow significantly because of encoding problems with absolute addresses.

In CRISP, as with many machines, the instruction encoding contains a 16-bit field for absolute addresses, in addition to full 32-bit addresses. If global variables and other static data can be placed in the first 64K bytes of memory, a shorter instruction form can be used, and the resulting code is smaller. However, if both the operating system and user are sharing the same address space, then only one can have its variables in the low section of memory. The other must use full 32-bit addresses, take more code space, and hence is likely to have lower performance. This is a problem common to many computer architectures. Some have no optimized way of accessing a global address, others use a base register that the operating system must set when the program context changes. Use of a base register is likely to lower performance or increase the complexity of the implementation.

CRISP solves this problem, and allows multiple processes in the same address space to utilize short absolute addresses effectively. In CRISP, a 16-bit address field is zero-extended only to bit 28 of the address. The top three bits, called the *Section ID*, are copied from the top three bits of the PC, effectively breaking the address space into eight half-gigabyte sections. Code in any section can use 16-bit absolute addresses to reference data starting within that section. Full 32-bit addresses are always available to reference data outside the section.

Operating System Interface

The operating system interface provides a mechanism for the kernel to handle system calls, interrupts, exceptions, illegal instructions, breakpoints and trace traps. The interface for CRISP follows a similar philosophy to that of the instruction set. The interface does as little as possible, letting the system programmer decide what actions must be done to service the event. The simplicity not only makes implementation easier and execution time faster, it also fits neatly into a UNIX environment.

System calls, interrupts and most exceptions are handled with the same sequence. The processor switches from the user Stack Pointer to the Interrupt Stack Pointer, allocates a new four word stack frame, and moves three words onto the stack: the current PSW, the relevant program counter address and a number. This number is either the argument of a **kcall** instruction, the interrupt vector or an internally-generated exception number. Execution is then transferred to an address pointed to by the Vector Base register. The Vector Base register points to a table of addresses, which are the starting addresses for the system call handler, the interrupt handler and the exception handler. After setting up the Interrupt Stack, the processor jumps to the appropriate handler, and sets the PSW to an appropriate mode (Kernel execution level, with interrupts disabled, etc.).

When the system call, interrupt or exception completes, the processor is left as if it is about to enter a procedure with three parameters (the three items placed on the Interrupt Stack). A complete assembly language interface for CRISP can be accomplished with two instructions: a **call** to the appropriate system function followed by a **kret** instruction. This matches well with UNIX, which maps all interrupts and system calls to a common function, so that individual handlers are called only after general kernel overhead is

dispatched.

A second mechanism is used only for handling unimplemented instructions. Instead of a full system call, an unimplemented instruction causes a procedure call, with the return address pointing to the unimplemented instruction and the handler address obtained from the table of addresses pointed to by the Vector Base register. The sequence does not force Kernel Mode, nor change the PSW in any way. It enables other implementations of the CRISP architecture that might lack some instructions to emulate them in software without the overhead of a full system call.

The RISC Philosophy

CRISP was intentionally designed with the RISC philosophy[6] in mind. Features have to be justified via measurements or necessity before they should be included in the instruction set. The instruction set and design decisions for CRISP were strongly influenced by program measurements. Most instructions take a small but uniform number of cycles, which insures the smooth flow of instructions in a pipelined implementation. Auto-increment addressing modes were rejected on this basis, for example. Functions that might lengthen cycle times were eliminated or modified.

CRISP is like other RISC machines in that it has a relatively small number of instructions and addressing modes. CRISP instructions are simple enough that they can be issued at a one cycle-per-instruction rate, like many RISC machines. Unlike some other RISC machines, however, CRISP uses variable length instructions and a memory-to-memory style architecture.

Summary

The instruction set presented for the CRISP microprocessor is a relatively simple, yet powerful model for the execution of programs. This simplicity benefits compiler writers, the designers of the CRISP Microprocessor, and those trying to debug hardware and software using CRISP. Compared to a load/store style architecture with fixed-length instructions, CRISP requires less code space and executes fewer instructions to complete a program. Characteristics of a particular implementation, such as the number of delay slots needed for delayed branches or loads have been specifically avoided. Perhaps CRISP's most unusual feature is the lack of visible data registers. This gives the architecture the flexibility to use a large number of registers effectively, without requiring them to be present in every implementation. The CRISP architecture, by compromising between extreme RISC implementations and lavish CISC features, permits high-end implementations to exploit large caches and aggressive compiler optimization while remaining compatible with low-end versions

References

1. B. W. Kernighan, D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).

2. D. D. Hill, ‘‘An Analysis of C Machine Support for Other Block Structured Languages,’’ *Computer Architecture News* **11**(4), pp. 6-16 (September 1983).

3. S. Bandyopadhyay, V. S. Begwani, R. B. Murray, ‘‘Compiling for the CRISP Microprocessor,’’ *Proceedings of the Spring 1987 COMPCON*.

4. D. R. Ditzel, H. R. McLellan, ‘‘Register Allocation for Free: The C Machine Stack Cache,’’ *Proc. of Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, pp. 48-56 (March 1982).

5. R. D. Russell, ‘‘The PDP-11: A Case Study of How Not to Design Condition Codes,’’ *Proceedings of the 5th Annual Symposium on Computer Architecture*, pp. 190-194 (April 1978).

6. D. R. Ditzel, D. A. Patterson, ‘‘The Case for the Reduced Instruction Set Computer,’’ *Computer Architecture News* **8**(7) (1980).



Figure 2. Ten byte instruction format.

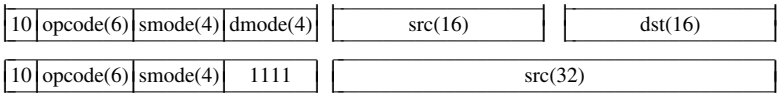


Figure 3. Six byte instruction formats.

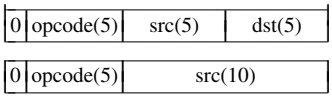


Figure 4. Two byte instruction formats.

Table 1. CRISP Instructions

Address Type	Name	Function	Addressing Modes	Data Types
2 and 2½ address	add sub mul quo and or xor shr ushl ushr	addition subtraction multiplication division bitwise and bitwise or bitwise exclusive or arithmetic shift right unsigned shift left unsigned shift right	Immediate Absolute Stack Offset	unsigned byte signed byte unsigned halfword
2 address	umul uquo urem cmp.= cmp.s< cmp.u< move mova addi andi ori	unsigned multiply unsigned divide unsigned remainder equality comparison signed less than comparison unsigned less than comparison move move effective address interlocked bitwise add interlocked bitwise and interlocked bitwise or	Stack Offset Indirect	signed halfword word
1 address	jmp ifTjmp ifFjmp ifCjmp ifOjmp call	unconditional jump conditional jump if True conditional jump if False conditional jump if Carry conditional jump if Overflow procedure call	Absolute Absolute Indirect Stack Offset Indirect PC Relative	
	kcall enter return catch	kernel call allocate new stack space de-allocate space and return restore Stack Cache	Immediate	
0 address	nop cpu kret	no operation internal register access kernel return		