INSTRUCTION SETS

A computer system's instruction set is the interface between the programmer/compiler and the hardware. Instructions in the instruction set manipulate components defined in the computer's *instruction set architecture* (ISA), which encompasses characteristics of the central processing unit (CPU), register set, memory access structure, and exception-handling mechanisms.

In addition to defining the set of commands that a computer can execute, an instruction set specifies the format of each instruction. An instruction is divided into various fields which indicate the basic command (opcode) and the operands to the command. Instructions should be chosen and encoded so that frequently used instructions or instruction sequences execute quickly. Often there is more than one implementation of an instruction set architecture. This enables computer system designers to exploit faster technology and components, while still maintaining object code compatibility with previous versions of the computer system.

Instruction sets began very simply and then became more complex as hardware gained complexity. By the 1980s, instruction sets had become sufficiently complex that a movement began to return to simpler instruction sets, albeit not the simplicity of the early machines. RISC (reduced instruction set computers) architectures were introduced, in contrast to the CISC (complex instruction set computers), which were then in vogue.

In addition to these *general-purpose* ISAs, special purpose architectures, such as vector and parallel machines, graphics processors, and digital signal processors (DSPs), require ISAs that capture their unique capabilities.

GENERAL-PURPOSE INSTRUCTION SETS

Instructions contain an opcode—the basic command to execute, including the data type of the operands—and some number of operands, depending on hardware requirements. Historically, some or all of the following operands have been included: one or two data values to be used by the operation

(source operands), the location where the result of the operation should be stored (destination operand), and the location of the next instruction to be executed. Depending on the number of operands, these are identified as one-, two-, three-, and four-address instructions. The early introduction of the special hardware register, the program counter, quickly eliminated the need for the fourth operand.

Types of Instructions

There is a minimum set of instructions that encompasses the capability of any computer:

- Add and subtract (arithmetic operations)
- Load and store (data movement operations)
- Read and write (input/output operations)
- An *unconditional* branch or jump instruction
- A minimum of two *conditional* branch or jump instructions [e.g., BEQ (branch if equal zero) and BLT (branch if less than zero) are sufficient]
- A halt instruction

Early computers could do little more than this basic instruction set. As machines evolved and changed, greater hardware capability was added, for example, the addition of multiplication and division units, floating-point units, multiple registers, and complex instruction decoders. Most instruction sets include, in addition to the minimum set already listed:

- System instructions such as operating system call and virtual memory management
- Traps and interrupt management instructions
- Instructions to operate on decimal or string data types
- Instructions to synchronize processors in multiprocessor configurations

Examples of basic and advanced instructions are given in the section "Representative Instruction Sets." Instruction sets expanded to reflect the additional hardware capability by combining two or more instructions of the basic set into a single, more complex instruction. The expanding complexity of instruction sets (CISCs) continued well into the 1980s until the introduction of RISC machines (see the subsection titled "RISC") changed this pattern.

Classes of Instruction Set Architectures

Instruction sets are often classified according to the method used to access operands. ISAs that support memory-to-memory operations are sometimes called SS architectures (for storage to storage), while ISAs that support basic arithmetic operations only in registers are called RR (register to register) architectures.

Consider an addition, $C = A + B$, where the values of A, B, and C have been assigned memory locations 100, 200, and 300, respectively. If an instruction set supports three-address memory-to-memory instructions, a single instruction,

```
Add C, A, B
```

would perform the required operation. This instruction would cause the contents of memory locations 100 and 200 to be added [by either moving the operands to registers in the arithmetic logic unit (ALU) or by performing the addition directly in memory, depending on the architecture] and store the result into location 300.

It is unlikely that an instruction set would provide this three-address instruction. One reason is that the instruction requires many bytes of storage for all the operand information and, therefore, is slow to load and interpret. Another reason is that later operations might need the result of the operation (e.g., if $A + B$ were a subexpression of a later, more complex expression), so it is advantageous to retain the result for use by subsequent instructions.

A two-address register-to-memory alternative might be:

```
Load    R1, A    ; R1 := A
Add     R1, B    ; R1 := R1 + B
Store   C, R1    ; C := R1
```

while a one-address alternative would be similar, with the references to R1 (register 1) removed. In the latter scheme, there would be only one hardware register available for use and, therefore, no need to specify it in each instruction. (The IBM 1620 and 7094 are example hardware.)

Most modern ISAs belong to the RR category and use general-purpose registers (organized either independently or as stacks) as operands. Arithmetic instructions require that at least one operand is in a register while "load" and "store" instructions (or "push" and "pop" for stack-based machines) copy data between registers and memory. ISAs for RISC machines (see the subsection titled "RISC") require both operands to be in registers for arithmetic instructions. If the ISA defines a register file of some number of registers, the instruction set will have commands that access, compute with, and modify all of those registers. If certain registers have special uses, such as a stack pointer, instructions associated with those registers will define the special uses.

The various alternatives that ISAs make available, such as

- Both operands in memory
- One operand in a register and one in memory
- Both operands in registers
- Implicit register operands such as an accumulator
- Indexed *effective address* calculation, for $A[i]$ sorts of references

are called the *addressing modes* of an instruction set. Addressing modes are illustrated in the section titled "Representative Instruction Sets," with examples of addressing modes supported by specific machines.

Issues in Instruction Set Design

There are many trade-offs in designing an efficient instruction set. The code density, based on the number of bytes per instruction and number of instructions required to do a task, has a direct influence on the machine's performance. The architect must decide what and how many operations the ISA will provide. A small set is sufficient, but leads to large programs. A large set requires a more complex instruction decoder. The number of operands affects the size of the instruction. A typical, modern instruction set supports 32 bit words, with 32 bit address widths, 32 bit operands, and dyadic opera-

tions, with an increasing number of ISAs using 64 bit operands. Byte, half-word, and double-word access are also desirable. If supported in an instruction set, additional fields must be allocated in the instruction word to distinguish the operand size. Implementation considerations such as pipelining are important to consider. Also, the ability of a compiler to map computations to a sequence of instructions must be considered for ISA design.

The number of instructions that can be supported is directly affected by the size of the opcode field. In theory, 2^n-1 (a 0 opcode is never used), where n is the number of bits allocated for the opcode, is the total number of instructions that can be supported. In practice, however, a clever architect can extend that number by utilizing the fact that some instructions, needing only one operand, have available space that can be used as an "extended" opcode. See the Representative Instruction Sets section for examples of this practice.

Instructions can either be fixed size or variable size. Fixed-size instructions are easier to decode and execute, but either severely limit the instruction set or require a very large instruction size, that is, waste space. Variable-size instructions are more difficult to decode and execute, but permit rich instruction sets. The actual machine word size influences the design of the instruction set. Small machine word size (see the subsection titled "DEC PDP-11" for an example machine) requires the use of multiple words per instruction. Larger machine word sizes make single-word instructions feasible. Very large machine word sizes permit multiple instructions per word (see the subsection titled "VLIW Instruction Sets").

Alternative General-Purpose ISAs

In the 1980s, CISC architectures were favored as best representing the functionality of high-level languages; however, later architecture designers favored RISC (reduced instruction set computer) designs for the higher performance attained by using compiler analysis to detect instruction level parallelism. Another architectural style, very large instruction word (VLIW), also attempts to exploit instruction level parallelism by providing multiple function units. In this section the instruction set characteristics of RISC and VLIW machines.

RISC. RISC architectures were developed in response to the prevailing CISC architecture philosophy of introducing more and more complex instructions to supply more support for high-level languages and operating systems. The RISC philosophy is to use simple instructions with extremely rapid execution times to yield the greatest possible performance (throughput and efficiency) for the RISC processor.

RISC designs try to achieve instruction execution times of one machine cycle per instruction by using instruction pipelines and load/store architectures.

The following simple CISC and corresponding RISC code examples display some of the basic differences between the two. Note that these codes are stylized rather than being examples of any specific machines.

```
LM    R6,R7,DATA   Load (multiple)
                   registers 6 and 7
                   beginning at the
                   location named DATA
```

```
Label: A     R6,DATA      Add the value in R6 to
                          the data value named
                          DATA
       BCT   R7,label     Decrement the value in
                          R7 and, if greater than
                          0, branch to location
                          "label"
```

Simple CISC Code Example

```
       LD    DATA,R6      }Two loads to perform
       LD    Count,R7     }the CISC LM
       LD    DATA,R8      No register-memory ops
                          in RISC
Label: ADD   R6,R8
       SUBi  R7,#1,R7     }Decrement and branch
       BGEZ  R7,label     }(BCT of CISC)
```

Corresponding RISC Code Example

On any machine, a series of steps is required in order to execute an instruction. For example, these may be: fetch instruction, decode instruction, fetch operand(s), perform operation, store result. In a RISC architecture, these steps are *pipelined* to speed up overall execution time.

If all instructions require the same number of cycles for execution, a *full* pipeline will generate an instruction per cycle. If instructions require different numbers of cycles for execution, the pipeline will necessarily delay cycles while waiting for resources. To minimize these delays, RISC instruction sets include *prefetch* instructions to help ensure the availability of resources at the necessary point in time.

Memory accesses require additional cycles to calculate operand address(es), fetch the operand(s), and store result(s) back to memory. RISC machines reduce the impact of these instructions by requiring that all operations be performed only on operands held in registers. Memory is then accessed only with load and store operations.

Load instructions fetch operands from memory to registers, to be used in subsequent instructions. Since memory bandwidth is generally slower than processor cycle times, an operator is not immediately available to be used. The ideal solution is to perform one or more instructions, depending on the delay required for the load, that are *not* dependent on the data being loaded. This effectively uses the pipeline, eliminating wasted cycles. The burden of generating effective instruction sequences is generally placed on a compiler and, of course, it is not always possible to eliminate all delays.

Lastly, branch instructions cause delays because the branch destination must be calculated and then that instruction must be fetched. As with load instructions, RISC designs typically use a delay on the branch instruction so they do not take effect until the one or two instructions (depending on the RISC design) immediately following the branch instruction have been executed. Again, the burden falls on the compiler to identify and move instructions to fill the one (or two) delay slots caused by this design. If no instruction(s) can be identified, a NOP (no op) has to be generated, which reduces performance.

VLIW Instruction Sets. VLIW architectures are formed by connecting a fixed set of RISC processors, called a cluster, and using only a single execution thread to control them all. Each

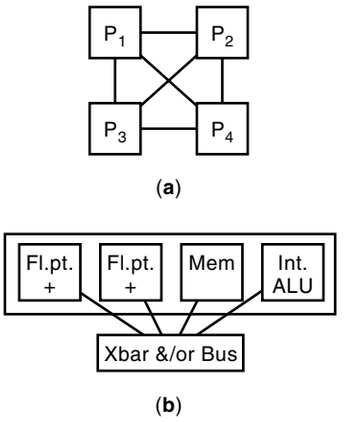


Figure 1. A generic VLIW machine. (a) A cluster of four VLIW processors; (b) A single VLIW processor.

RISC processor contains some number of parallel, pipelined functional units that are connected to a large memory and register bank using crossbars and/or busses. Each instruction has a field that corresponds to each of the functional units in a cluster and specifies the action of that unit. This generates a fine-grained parallelism, as compared with the coarse-grained parallelism of vector machines and multiprocessors. Figure 1 shows a "generic" VLIW computer and Fig. 2 shows an instruction word for such a machine.

To optimize code for a VLIW machine, a compiler may perform *trace scheduling* to identify the parallelism needed to fill the function units. Indirect memory references, generated by array indexing and pointer dereferencing, can cause difficulties in the trace. These memory references must be disambiguated, wherever possible, to generate the most parallelism.

SPECIALIZED INSTRUCTION SETS

The discussion above has focused on instruction sets for most general-purpose machines. Often the basic instruction set is augmented for efficient execution of special functions.

Vector Instruction Sets

Vector architectures, such as the original Cray computers, supplement the conventional scalar instruction set with a vector instruction set. By using vector instructions, operations that would normally be executed in a loop are expressed in the ISA as single instructions. Each vector instruction operates on an entire vector of registers or memory locations. In addition to the normal fetch-decode-execute pipeline of a scalar processor, a vector instruction uses additional vector pipelines to execute the vector instructions. In a vector instruction, the vector register's set of data is pipelined through the appropriate function unit. Categories of vector instructions include:

P_1			P_2			P_3			P_4		
F+	Fx	ALU	F+	Fx	ALU	F+	Fx	ALU	F+	Fx	ALU

Figure 2. A VLIW instruction word.

- Vector–vector instructions, where all the operands of the instruction are vectors. An example is an add with vector registers as operands and a vector register as result.
- Vector–scalar instructions, where the content of a scalar register is combined with each element of the vector register. For example, a scalar value might be multiplied by each element of a vector register and the result stored into another vector register.
- Vector–memory instructions, where a vector is loaded from memory or stored to memory.
- Vector reduction instructions, in which a function is computed on a vector register to yield a single result. Examples include finding the minimum, maximum, or sum of values in a vector register.
- Scatter–gather instructions, in which the values of one vector register are used to control vector load from memory or vector store to memory. Scatter uses an indirect addressing vector register and a base scalar register to form an effective address. Values in a data vector register corresponding to the indirect addressing vector register are stored to the calculated effective memory addresses. Similarly, a gather uses the indirect address register combined with a scalar base register to form a set of effective addresses. Data from those addresses are loaded into a vector data register.

SIMD Instruction Sets

Instruction sets for SIMD machines such as the CM-2, DAP, and MasPar MP series are conceptually similar to vector instruction sets. SIMD instructions also operate on aggregate data. However, rather than processing multiple pairs of operands through a functional pipeline, the SIMD machine has many identical processors, each operating in lockstep through a single instruction stream. The instructions may be SS, as in the CM-2, or RR, as in the MasPar machines. An important characteristic of SIMD instruction sets is the lack of branching instructions. Rather, these machines rely on the notion of *contextualization,* meaning each SIMD processor has its own unique "context" that determines whether or not it executes the current instruction.

Instructions exist in a SIMD instruction set to evaluate an expression and set the context to the result of the expression evaluation. Thus processors that evaluate the expression to true will execute subsequent instructions, while those that evaluate the expression to false will not. Naturally, there are some instructions that execute regardless of the context value, so that "context" can be set and reset during computation. SIMD instruction sets usually include reduce instructions, as described above for vector machines. In addition, some SIMD machines have scan instructions, which set up variable length vectors across the processor array on which reduce operations can be performed.

DSP Instruction Sets

The architecture of a digital signal processor (DSP) is optimized for pipelined data flow. Many DSPs for embedded applications support only fixed-point arithmetic; others have both fixed- and floating-point units; while still others offer multiple fixed-point units in conjunction with the floating-point processor. All these variations, of course, affect the in-

struction set of the DSP, determining whether bits in the instruction word are needed to specify the data type of the operands. Other distinguishing characteristics of DSP instruction sets include:

- Multiply-accumulate instruction (MAC), used for inner product calculations
- Fast basic math functions, combined with a memory access architecture optimized for matrix operations
- Low overhead loop instructions
- Addressing modes that facilitate FFT-like memory access
- Addressing modes that facilitate table look-up

Multimedia Instructions

Multimedia instructions are optimized to process images, graphics, and video data types. These instructions typically operate on 8-bit quantities, often in groups of 4 or 8, resembling VLIW or SIMD instructions. DSP-like capability may be provided with the inclusion of Multiply-accumulate on 8- or 16-bit data values. Many modern microprocessors include multimedia instructions to augment their instruction sets in order to support multimedia functions such as video decoding.

The multimedia extensions to the Intel Pentium instruction set have many DSP-like characteristics. An MMX instruction operates on data types ranging from 8 bits to 64 bits. With 8 bit operands, each instruction is similar to a SIMD instruction in that, during a single clock cycle, multiple instances of the instruction are being executed on different instances of data. The arithmetic instructions PADD/PSUB and PMULLW/PMULHW operate in parallel on either eight bytes, four 16 bit words, or two 32 bit double words.

The MMX instruction set includes a MAC instruction, PMADDWD, which does a multiply-add of four signed 16 bit words and adds adjacent pairs of 32 bit results. The PUNPCKL and PUNKCKH instructions help with interleaving words, which is useful for interpolation. The arithmetic instructions in the MMX instruction set allow for saturation, to avoid overflow or underflow during calculations.

Configurable Instruction Sets

Research into future generations of processors generalizes the notion of support for specialized operations. New designs call for *configurable logic* to be available so new instructions can be synthesized, loaded into the configurable logic, and thus dynamically extend the processor's instruction set. National Semiconductor's NAPA1000 is such a next-generation processor architecture. In conjunction with a conventional RISC processor, the NAPA chip contains an embedded field programmable gate array called the adaptive logic processor (ALP). By designing circuits for the ALP, a programmer can augment the instruction set of the RISC processor with arbitrary functionality. Control signals to activate the custom instructions are generated by memory-mapped writes to a communications bus, which connects the RISC processor with the ALP. Such architectures provide virtually unlimited, application-dependent extensibility to an ISA.

REPRESENTATIVE INSTRUCTION SETS

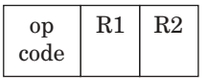
The details of five representative instruction sets are shown here. These are the IBM System 360, the PDP-11 minicomputer, the MIPS RISC computer, the Cray X-MP vector computer, and the Intel Pentium processor.

IBM System 360

The IBM System 360, introduced in April of 1964 with first delivery in April of 1965, was the first of the third-generation (integrated circuit) computers. The general acceptance of a 32 bit word and 8 bit byte come from this machine. The system 360 consisted of a series of models, with models 30, 40, 50, 65, and 75 being the best known. Model 20, introduced in November of 1964, had slightly different architecture from the others.

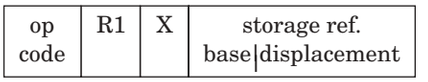
The 360 (any model) was a conventional mainframe, incorporating a rich, complex instruction set. The machine had 16 general-purpose registers (8 on the smaller models) and four floating-point registers. Instructions mainly had two addresses but 0, 1, and 3 were also permitted in some cases.

Instructions could be 2, 4, or 6 bytes in length, defining five addressing modes of instructions. Two-byte instructions were register-to-register (RR) instructions, consisting of:



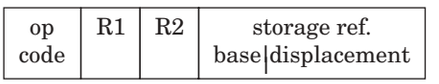
where the opcode is 1 byte, which specifies the operation to be performed, R1 is one of the 16 general-purpose registers that is a data source as well as the *destination* of the result of the operation, and R2 is one of the 16 general-purpose operations and is the second *source* of the data for the operation. At the completion of the operation, R1's value has been changed while R2 has the same value it did at the start of the instruction.

There were three modes of 4 byte instructions: register-indexed (RX), register-storage (RS), and storage-immediate (SI). RX instructions were of the form:



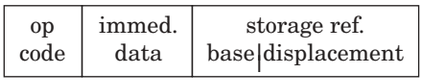
where the opcode is 1 byte, which specifies the operation to be performed, R1 is one of the 16 general-purpose registers and is either the instruction data source or destination, X is one of the 16 general-purpose registers used as an index added to the memory location specified, and the storage ref(erence) is a standard 360 memory reference consisting of a 4 bit base address and a 12 bit displacement value. So, for RX instructions, the memory location specified is base + displacement + index.

RS instructions had the form:



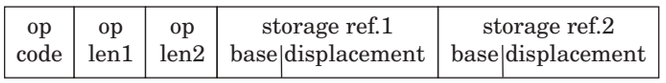
where the opcode is as for RX, R1, and R2 specify a *range* of general-purpose registers (registers "wrap" from R15 to R0), which are either the instruction data source(s) or destination, depending on the opcode, and the storage ref(erence) is the standard 360 memory reference, as specified above.

SI instructions had the form:



where opcode is as above, the storage ref(erence) is one of the instruction data values and is defined as above, and immed(iate) data is the second instruction data value. It is 1 byte and is the *actual* data value to be used, that is, the datum is not located in a register or referenced through a memory address.

The 6 byte instruction format was used for storage-to-storage (SS) instructions and looked like:



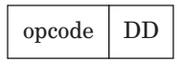
where the opcode is as before, op len1 is the length of the instruction result destination, op len2 is the length of the instruction data source and is only needed when packed-decimal data are used, and storage ref(erence)1 and storage ref(erence)2 are the memory locations of the destination and source, respectively. Table 1 contains a list of 360 opcodes along with the type (RR, RX, RS, SI, SS) of each operation.

DEC PDP-11

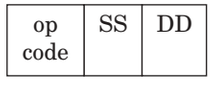
The DEC PDP-11 was a third-generation computer, and was introduced around 1970. It was a successor to the highly successful (also) third-generation PDP-8, introduced in 1968, which itself was a successor to second-generation PDP machines.

The PDP-11, and the entire PDP line, were minicomputers, loosely defined as machines with smaller word size and memory address space, and slower clock rate, than cogenerational mainframes. The PDP-11 was a 16 bit word machine, with eight general-purpose registers (R0 to R8), although R6 and R7 were "reserved" for use as the stack pointer (SP) and program counter (PC), respectively.

Instructions required one word (16 bits) with the immediately following one or two words used for some addressing modes. Instructions could be *single*-operand instructions:



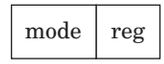
where the opcode is 10 bits, which specify the operation to be performed, and DD is the destination of the result of the operation; or *double*-operand instructions:



where opcode is 4 bits, which specify the operation to be performed, SS is the source of the data for the operation, and DD is the destination of the result of the operation.

Instructions operands could be either a single byte or a word (or words using indirection and indexing). When the operand was a byte, the leading bit in the opcode field was 1; otherwise, that bit was 0.

SS and DD each consist of a 3 bit register subfield and a 3 bit addressing mode subfield:



There are seven addressing modes, as shown in Table 2. Table 3 contains a list of PDP-11 opcodes.

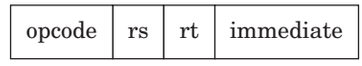
MIPS RISC Processor

The MIPS R-family of processors includes the R2000, 4000, and 10000. The R4000 and R10000 are 64 bit machines, but remain ISA-compatible with the R2000.

The MIPS RISC R2000 processor consists of two tightly coupled processors on a single chip. One processor is a 32 bit RISC CPU; the other (which will not be discussed in any detail) is a system control coprocessor that supports a virtual memory subsystem and separate caches for instructions and data. Additional coprocessors on higher performance members of the R-family include the floating-point coprocessor and a third coprocessor reserved for expansion.

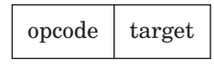
The RISC CPU is a 32 bit machine, containing 32 32 bit registers and 32 bit instructions and addresses. There are also a 32 bit program counter and two 32 bit registers for the results of integer multiplies and divide. The MIPS uses a five-stage pipeline and achieves an execution rate *approaching* one instruction per cycle. R2000 instructions are all 32 bits long and use only three instruction formats.

Immediate (I-Type) instructions consist of four fields in a 32 bit word.



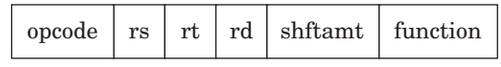
where opcode is 6 bits, rs is a 5 bit source register, rt is a 5 bit source or destination register or a branch condition, and immediate is a 16 bit immediate, branch displacement, or address displacement.

Jump (J-Type) instructions consist of two fields in a 32 bit word.



where opcode is 6 bits and target is a 26 bit jump address.

Register (R-Type) instructions consist of six fields in a 32 bit word.



where opcode, rs, and rt are as defined above for the I-Type instruction, rd is a 5 bit destination register specifier, shftamt is a 5 bit shift amount, and function is a 6 bit function field.

In addition to the regular instructions, the MIPS processor's instruction set includes coprocessor instructions. Coprocessor 0 instructions perform memory-management functions and exception handling on the memory-management coprocessor. These are I-type instructions.

Special instructions, which perform system calls and breakpoint operations, are R-type. Exception instructions

Table 1. IBM System 360 Instruction Set

Command	Mnemonic	Type	Command	Mnemonic	Type
Add register	AR	RR	Load multiple	LM	RS
Add	A	RX	Load negative register	LNR	RR
Add halfword	AH	RX	Load negative register (long)	LNDR	RR
Add logical register	ALR	RR	Load negative register (short)	LNER	RR
Add logical	AL	RX	Load positive register	LPR	RR
Add normalized register (long)	ADR	RR	Load positive register (long)	LPDR	RR
Add normalized (long)	AD	RX	Load positive register (short)	LPER	RR
Add normalized register (short)	AER	RR	Load PSW	LPSW	SI
Add normalized (short)	AE	RX	Load register (short)	LER	RR
Add packed	AP	SS	Load (short)	LE	RX
Add unnormalized register (long)	AWR	RR	Move immediate	MVI	SI
Add unnormalized (long)	AW	RX	Move character	MVC	SS
Add unnormalized register (short)	AUR	RR	Move numerics	MVN	SS
Add unnormalized (short)	AU	RX	Move with offset	MVO	SS
AND register	NR	RR	Move zones	MVZ	SS
AND	N	RX	Multiply register	MR	RR
AND immediate	NI	SI	Multiply	M	RX
AND character	NC	SS	Multiply halfword	MH	RX
Branch and link register	BALR	RR	Multiply register (long)	MDR	RR
Branch and link	BAL	RX	Multiply (long)	MD	RX
Branch on condition register	BCR	RR	Multiply packed	MP	SS
Branch on condition	BC	RX	Multiply register (short)	MER	RR
Branch on count register	BCTR	RR	Multiply (short)	ME	RX
Branch on count	BCT	RX	OR register	OR	RR
Branch on index high	BXH	RS	OR	O	RX
Branch on index low or equal	BXLE	RS	OR immediate	OI	SI
Compare register	CR	RR	OR character	OC	SS
Compare	C	RX	Pack	PACK	SS
Compare halfword	CH	RX	Read direct	RDD	SI
Compare logical register	CLR	RR	Set program mask	SPM	RR
Compare logical	CL	RX	Set storage key	SSK	RR
Compare logical immediate	CLI	SI	Set system mask	SSM	SI
Compare logical character	CLC	SS	Shift left double	SLDA	RS
Compare register (long)	CDR	RR	Shift left double logical	SLDL	RS
Compare (long)	CD	RX	Shift left single	SLA	RS
Compare packed	CP	SS	Shift left single logical	SLL	RS
Compare register (short)	CER	RR	Shift right double	SRDA	RS
Compare (short)	CE	RX	Shift right double logical	SRDL	RS
Convert to binary	CVB	RX	Shift right single	SRA	RS
Convert to decimal	CVD	RX	Shift right single logical	SRL	RS
Divide register	DR	RR	Start I/O	SIO	SI
Divide	D	RX	Store	ST	RX
Divide register (long)	DDR	RR	Store character	STC	RX
Divide (long)	DD	RX	Store halfword	STH	RX
Divide packed	DP	SS	Store (long)	STD	RX
Divide register (short)	DER	RR	Store multiple	STM	RS
Divide (short)	DE	RX	Store (short)	STE	RX
Edit	ED	SS	Subtract register	SR	RR
Edit and mark	EDMK	SS	Subtract	S	RX
Exclusive OR register	XR	RR	Subtract halfword	SH	RX
Exclusive OR	X	RX	Subtract logical register	SLR	RR
Exclusive OR immediate	XI	SI	Subtract logical	SL	RX
Exclusive OR character	XC	SS	Subtract normalized register (long)	SDR	RR
Execute	EX	RX	Subtract normalized (long)	SD	RX
Halt I/O	HIO	SI	Subtract normalized register (short)	SER	RR
Halve register (long)	HDR	RR	Subtract normalized (short)	SE	RX
Halve register (short)	HER	RR	Subtract packed	SP	SS
Insert character	IC	RX	Subtract unnormalized register (long)	SWR	RR
Insert storage key	ISK	RR	Subtract unnormalized (long)	SW	RX
Load register	LR	RR	Subtract unnormalized register (short)	SUR	RR
Load	L	RX	Subtract unnormalized (short)	SU	RX
Load address	LA	RX	Supervisor call	SVC	RR
Load and test	LTR	RR	Test and set	TS	SI
Load and test (long)	LTDR	RR	Test channel	TCH	SI
Load and test (short)	LTER	RR	Test I/O	TIO	SI
Load complement register	LCR	RR	Test under mask	TM	SI
Load complement (long)	LCDR	RR	Translate	TR	SS
Load complement (short)	LCER	RR	Translate and test	TRT	SS
Load halfword	LH	RX	Unpack	UNPK	SS
Load register (long)	LDR	RR	Write direct	WRD	SI
Load (long)	LD	RX	Zero and add packed	ZAP	SS

Table 2. Addressing Modes of the DEC PDP-11

Address Mode	Name	Form	Meaning
0	Register	Rn	Operand is in register n
1	Indirect register[a]	(Rn)	Address of operand is in register n
2	Autoincrement	(Rn)+	Address of operand is in register n (Rn) := (Rn) + 2 after operand is fetched[b]
3	Indirect autoincrement	@(Rn)+	Register n contains the address of the *address* of the operand: (Rn) := (Rn) + 2 after operand is fetched
4	Autodecrement	−(Rn)	(Rn) := (Rn) − 2 *before* operand is fetched[c]; address of operand is in register n
5	Indirect autodecrement	@ − (Rn)	(Rn) := (Rn) − 2 before operand is fetched; register n contains the address of the *address* of the operand
6	Index	X(Rn)	Address of operand is in X + (Rn); address of X is in the PC; (PC) := (PC) + 2 after X is fetched
7	Indirect index	@X(Rn)	X + (Rn) is the address of the *address* of the operand; address if X is in the PC; (PC) := (PC) + 2 after X is fetched

[a] "Indirect" is also called "deferred."
[b] If the instruction is a byte instruction and the register is *not* the SP or PC, (Rn) := (Rn) + 1.
[c] If the instruction is a byte instruction and the register is *not* the SP or PC, (Rn) := (Rn) − 1.

Table 3. PDP-11 Instruction Set

Command	Mnemonic	No. Operands	Command	Mnemonic	No. Operands
Add	ADD	2	Clear Z (= 0 condition)	CLZ	0
Add carry	ADC	1	Clear N (> or < 0 condition)	CLN	0
Add carry byte	ADCB	1	Clear C, V, Z, and N	CCC	0
Arithmetic shift right	ASR	1	Compare	CMP	2
Arithmetic shift right byte	ASRB	1	Compare byte	CMPB	2
Arithmetic shift left	ASL	1	Complement	COM	1
Arithmetic shift left byte	ASLB	1	Complement byte	COMB	1
Bit test	BIT	2	Decrement	DEC	1
Bit test byte	BITB	2	Decrement byte	DECB	1
Bit clear	BIC	2	Halt	HALT	0
Bit clear byte	BICB	2	Increment	INC	1
Bit set	BIS	2	Increment byte	INCB	1
Bit set byte	BISB	2	Jump	JMP	1
Branch not equal zero	BNE	1	Move	MOV	2
Branch equal zero	BEQ	1	Move byte	MOVB	2
Branch if plus	BPL	1	Negate	NEG	1
Branch if minus	BMI	1	Negate byte	NEGB	1
Branch on overflow clear	BVC	1	Rotate right	ROR	1
Branch on overflow set	BVS	1	Rotate right byte	RORB	1
Branch on carry clear	BCC	1	Rotate left	ROL	1
Branch on carry set	BCS	1	Rotate left byte	ROLB	1
Branch if gtr than or eq 0	BGE	1	Set C (carry condition)	SEC	0
Branch if less than 0	BLT	1	Set V (overflow condition)	SEV	0
Branch if greater than 0	BGT	1	Set Z (= 0 condition)	SEZ	0
Branch if less than or eq 0	BLE	1	Set N (> or < 0 condition)	SEN	0
Branch higher	BHI	1	Set C, V, Z, and N	SCC	0
Branch lower or same	BLOS	1	Subtract	SUB	2
Branch higher or same	BHIS	1	Subtract carry	SBC	1
Branch lower	BLO	1	Subtract carry byte	SBCB	1
Clear	CLR	1	Swap bytes	SWAB	1
Clear byte	CLRB	1	Test	TST	1
Clear C (carry condition)	CLC	0	Test byte	TSTB	1
Clear V (overflow condition)	CLV	0	Unconditional branch	BR	1

Table 4. MIPS RISC R2000 Instruction Set

Command	Mnemonic	Type	Command	Mnemonic	Operation Type
Add	ADD	R-type	Move from CP0	MFC0	I-type
Add immediate	ADDI	I-type	Move from coprocessor z	MFCz	R-type
Add immediate unsigned	ADDIU	I-type	Move from HI	MFHI	2 operand, R-type
Add unsigned	ADDU	R-type			
And	AND	R-type	Move from LO	MFLO	R-type
And immediate	ANDI	I-type	Move to coprocessor 0	MTC0	
Branch on coprocessor z false	BCxF	R-type	Move to coprocessor z	MTCz	I-type
Branch on coprocessor z true	BCxT	R-type	Move to HI	MTHI	R-type
Branch on equal	BEQ	I-type	Move to LO	MTLO	R-type
Branch on greater or equal zero	BGEZ	I-type	Multiply	MULT	R-type
Branch on greater or equal zero and link	BGEZAL	I-type	Multiply unsigned	MULTU	R-type
Branch on greater than zero	BGTZ	I-type	NOR	NOR	R-type
Branch on less or equal zero	BLEZ	I-type	OR	OR	R-type
Branch on less than zero	BLTZ	I-type	OR immediate	ORI	I-type
Branch on less than zero and link	BLTZAL	I-type	Store byte	SB	I-type
Branch on not equal	BNE	I-type	Store halfword	SH	I-type
Break	BREAK	I-type	Shift left logical	SLL	R-type
Cache	CACHE	I-type	Shift left logical variable	SLLV	R-type
Move control from coprocessor z	CFCx	I-type	Set on less than	SLT	R-type
Coprocessor operation z	COPz	I-type	Set on less than immediate	SLTI	I-type
Move control to coprocessor z	CTCz	I-type	Set on less than immediate unsigned	SLTIU	I-type
Divide	DIV	R-type	Set on less than unsigned	SLTU	R-type
Divide unsigned	DIVU	R-type	Shift right arithmetic	SRA	R-type
Double word move from C0	DMFC0	R-type	Shift right arithmetic variable	SRAV	R-type
Double word move to C0	DMTC0	R-type	Shift right logical	SRL	R-type
Exception return	ERET		Shift right logical variable	SRLV	R-type
Jump	J	J-type	Subtract	SUB	R-type
Jump and link	JAL	J-type	Subtract unsigned	SUBU	R-type
Jump and link register	JALR	J-type	Store word	SW	I-type
Jump register	JR	J-type	Store word from coprocessor z	SWCz	I-type
Load byte	LB	I-type	Store word left	SWL	I-type
Load byte unsigned	LBU	I-type	Store word right	SWR	I-type
Load halfword	LH	I-type	System call	SYSCALL	I-type
Load halfword unsigned	LHU	I-type	Probe TLB for matching entry	TLBP	R-type
Load upper immediate	LUI	I-type	Read indexed TLB entry	TLBR	R-type
Load word	LW	I-type	Write indexed TLB entry	TLBWI	R-type
Load word to coprocessor z	LWCz	I-type	Write random TLB entry	TLBWR	R-type
Load word left	LWL	I-type	Xor	XOR	R-type
Load word right	LWR	I-type	Xor immediate	XORI	I-type

cause a branch to an exception vector based on the result of a compare. These are R- and I-type instructions.

Table 4 gives the base instruction set of the MIPS RISC processor family. The 4000 and above also have an extended instruction set, which tightly encodes frequently used operations and provides access to 64 bit operands and coprocessors.

Pentium Processor

The Intel Pentium series processor has become the most prevalent of microprocessors in the 1990s. The Pentium follows the ISA of the 80 $\times$ 86 (starting with 8086). It uses advanced techniques such as speculative and out-of-order execution, once used only in supercomputers, to accelerate the interpretation of the $\times$86 instruction stream.

The original 8086 was a 16 bit CISC architecture, with 16 bit internal registers. Registers had fixed functions. Segment registers were used to create an address larger than 16 bits, so the address space was broken into 64 byte chunks. Later members of the $\times$86 family (starting with the 386) were true 32 bit machines, with 32 bit registers and a 32 bit address space. Additional instructions in the later $\times$86 instruction set made the register set more general purpose.

The general format of an "Intel architecture" instruction is shown in Fig. 3. As shown, the instructions are a variable number of bytes with optional prefixes, an opcode, an addressing-form specifier consisting of the ModR/M and Scale/Index/Base fields (if required), address displacement of 0 bytes to 4 bytes, and an immediate data field of 0 bytes to 4 bytes. The instruction prefixes can be used to override default registers, operand size, address size, or to specify certain actions on string instructions. The opcode is either one or two bytes, though occasionally a third byte is encoded in the next field. The ModR/M and SIB fields have a rather complex encoding. In general, their purpose is to specify registers (general-purpose, base, or index), addressing modes, scale factor, or additional opcode information. The register specifiers may

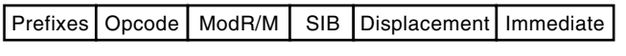


Figure 3. Intel architecture instruction format.

Table 5. Intel Architecture Instruction Set Summary

Command	Opcode	Command	Opcode
ASCII adjust after addition	AAA	Load global descriptor table register	LGDT
ASCII adjust AX before division	AAD	Load pointer to GS	LGS
ASCII adjust AX after multiply	AAM	Load interrupt descriptor table register	LIDT
ASCII adjust AL after subtraction	AAS	Load local descriptor table register	LLDT
ADD with carry	ADC	Load machine status	LMSW
Add	ADD	Assert LOCK num. signal prefix	LOCK
Logical AND	AND	Load string operand	LOD*
Adjust RPL field of selector	ARPL	Loop count (with condition)	LOOP*
Check array against bounds	BOUND	Load segment limit	LSL
Bit scan forward	BSF	Load task register	LTR
Bit scan reverse	BSR	Move data, registers	MOV*
Byte swap	BSWAO	Unsigned multiply	MUL
Bit test	BT	Two's complement negation	NEG
Bit test and complement	BTC	No operation	NOP
Bit test and reset	BTR	One's complement negation	NOT
Bit test and set	BTS	Logical inclusive OR	OR
Call procedure (in same segment)	CALL	Output to port	OUT*
Call procedure (in different segment)	CALL	Pop word/register(s) from stack	POP
Convert byte to word	CWB	Push word/register(s) onto stack	PUSH
Convert doubleword to Qword	CDQ	Rotate thru carry left	RCL
Clear carry flag	CLC	Rotate thru carry right	RCR
Clear direction flag	CLD	Read from model specific register	RDMSR
Clear interrupt flag	CLI	Read performance monitoring counters	RDPMC
Clear task-switched flag in CR0	CLTS	Read time-stamp counter	RDTSC
Complement carry flag	CMC	Input string	REP INS
Conditional move	CMOVcc	Load string	REP LODS
Compare to operands	CMP	Move string	REP MOVS
Compare string operands	CMP[S[W/D]]	Output string	REP OUTS
Compare/exchange	CMPXCHG	Store string	[REP] STOS
Compare/exchange 8 bytes	CMPXCHG8B	Compare string	REP[N][E] CMPS
CPU identification	CPUID	Scan string	[REP] [N][E] SCANS
Convert word to doubleword	CWD	Return from procedure	RET
Convert word to doubleword	CWDE	Rotate left	ROL
Decimal adjust AL after addition	DAA	Rotate right	ROR
Decimal adjust AL after subtraction	DAS	Resume from system management mode	RSM
Decrement by 1	DEC	Store AH into flags	SAHF
Unsigned divide	DIV	Shift arithmetic left	SAL
Make stack frame for proc.	ENTER	Shift arithmetic right	SAR
Halt	HLT	Subtract with borrow	SBB
Signed divide	IDIV	Byte set on condition	SETcc
Signed multiply	IMUL	Store global descriptor table register	SGTD
Input from port	IN	Shift left [double]	SHL[D]
Increment by 1	INC	Shift right [double]	SHR[D]
Input from DX port	INS	Store interrupt descriptor table register	SIDT
Interrupt type n	INT n	Store local descriptor table	SLDT
Single-step interrupt 3	INT	Store machine status word	SMSW
Interrupt 4 on overflow	INTO	Set carry flag	STC
Invalidate cache	INVD	Set direction flag	SDC
Invalidate TLB entry	INVLPG	Set interrupt flag	STI
Interrupt return	IRET/IRETD	Store task register	STR
Jump if condition is met	Jcc	Integer subtract	SUB
Jump on CX/ECX zero	JCXZ/JECXZ	Logical compare	TEST
Unconditional jump (same segment)	JMP	Undefined instruction	UD2
Load flags into AH register	LAHF	Verify a segment for reading	VERR
Load access rights byte	LAR	Wait	WAIT
Load pointer to DS	LDS	Writeback and invalidate data cache	WVINVD
Load effective address	LEA	Write to model-specific register	WRMSR
High level procedure exit	LEAVE	Exchange and add	XCHG
Load pointer to ES	LES	Table look-up translation	XLAT[B]
Load pointer to FS	LFS	Logical exclusive OR	XOR

Table 6. Cray X-MP Instruction Set

Command	CAL Syntax	Command	CAL Syntax
ADD scalar/vector	Vi Sj + Vk	Set vector length to 1	VL 1
ADD vector/vector	Vi Vj + Vk	Set vector mask to a value	VM Sj
ADD floating scalar/vector	Vi Sj + FVk	Set scalar to specified element of vector	Si Vj, Ak
ADD floating vector/vector	Vi Vj + FVk	Set specified element of vector to scalar	Vi, Ak Sj
AND scalar/vector	Vi Sj & Vk	Set scalar/vector based on vector mask	Vi Sj ! Vk & VM
AND vector/vector	Vi Vj & Vk	Set 0/vector based on vector mask	Vi # VM & VK
Clear vector mask	VM 0	Set vector/vector based on vector mask	Vi Vj ! Vk & VM
Clear specified element of vector	Vi, Ak 0	Set vector mask when zero	VM Vj, Z
Copy floating vector	Vi + FVk	Set vector mask when not zero	VM Vj, N
MULTIPLY floating scalar/vector	Vi Sj * FVk	Set vector mask when positive (>= 0)	VM Vj, P
MULTIPLY floating vector/vector	Vi Vj * FVk	Set vector mask when negative (<0)	VM Vj, M
MULTIPLY floating half precision scalar/vector	Vi Sj * HVk	Shift vector elements left (0 fill)	Vi Vj < Ak
MULTIPLY floating half precision vector/vector	Vi Vj * HVk	Shift vector elements left by 1 (0, fill)	Vi Vj < 1
MULTIPLY rounded floating scalar/vector	Vi Sj * RVk	Shift vector elements right (0 fill)	Vi Vj > Ak
MULTIPLY rounded floating vector/vector	Vi Vj * RVk	Shift vector elements right by 1 (0 fill)	Vi Vj > 1
MULTIPLY reciprocal iteration scalar/vector	Vi Sj * IVk	Shift pairs of vector elements left (0 fill)	Vi Vj, Vj < Ak
MULTIPLY reciprocal iteration vector/vector	Vi Vj * IVk	Shift pairs of vector elements left by 1 (0 fill)	Vi Vj, Vj < 1
Negate vector	Vi − Vk	Shift pairs of vector elements right (0 fill)	Vi Vj, Vj < Ak
Negate floating vector	Vi − FVk	Shift pairs of vector elements right by 1 (0 fill)	Vi Vj, Vj < 1
OR scalar/vector	Vi Sj ! Vk	Store from vector to memory (incr addr by spec. amt)	, A0, Ak Vj
OR vector/vector	Vi Vj ! Vk	Store from vector to memory (incr addr by 1)	, A0, 1, Vj
Population count vector	Vi PVj	SUBTRACT scalar/vector	Vi Sj − Vk
Population count parities vector	Vi QVj	SUBTRACT vector/vector	Vi Vj − Vk
Read vector mask	Si VM	SUBTRACT floating scalar/vector	Vi Sj − FVk
Read from memory to vector (incr addr by Ak)	Vi, A0, Ak	SUBTRACT floating vector/vector	Vi Vj − FVk
Read from memory to vector (incr addr by 1)	Vi, A0, 1	XOR scalar/vector	Vi Sj \ Vk
Reciprocal approximation floating vector	Vi/HVj	XOR vector/vector	Vi Vj \ Vk
Set vector length (VL)	VL Ak		

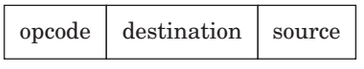
select MMX registers. The displacement is an address displacement. If the instruction requires immediate data, they is found in the final byte(s) of the instruction.

A summary of the Intel architecture instruction set is given in Table 5. The arithmetic instructions are 2-operand, where the operands can be two registers, register and memory, immediate and register, or immediate and memory. The jump instructions have several forms, depending on whether the target is in the same segment or a different segment.

Cray X-MP Vector Computer

The Cray X-MP was a pipelined vector processor consisting of two identical vector-extended RISC-based CPUs, which shared a common main memory and I/O subsystem. This discussion is limited to the vector instruction set only. Each processor had eight 64 bit vector registers and eight vector functional units: integer add, two logical, shift, population count/parity, floating point add, floating point multiply, and floating point reciprocal.

The X-MP was a vector-register (RR) architecture, performing all vector operations, with the exception of "load" and "store," in the vector registers. The alternative memory-memory vector architecture (SS) was used in some early machines, but has been discarded in favor of the RR architecture. Instructions were either two-address (source and destination):



or three-address (two sources and a destination):

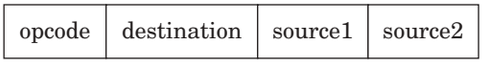


Table 6 shows the *vector* instruction set for a Cray X-MP. In the table, S = scalar register, V = vector register, and A = address register. An address register points to specific memory locations, or can be used as an index or offset. i, j, and k are used to indicate specific instances of these registers. The destination is always the first operand listed.

BIBLIOGRAPHY

N. Chapin, *360 Programming in Assembly Language,* New York: McGraw-Hill, 1968.

J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures,* Cambridge, MA: MIT Press, 1986.

A. Gill, *Machine and Assembly Language Programming of the PDP-11,* Englewood Cliffs, NJ: Prentice-Hall, 1978.

J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* San Mateo, CA: Morgan Kaufmann Publishers, 1990.

K. Hwang, *Advanced Computer Architecture,* New York: McGraw-Hill, 1993.

G. Kane, *MIPS RISC Architecture,* Englewood Cliffs, NJ: Prentice-Hall, 1988.

K. A. Robbins and S. Robbins, The Cray X-MP/Model 24, *Lecture Notes in Computer Science* #374, New York: Springer-Verlag, 1989.

MAYA GOKHALE
Sarnoff Corporation

JUDITH D. SCHLESINGER
IDA/Center for Computing Sciences