THE CLIPPER® PROCESSOR: INSTRUCTION SET ARCHITECTURE AND IMPLEMENTATION

Intergraph's CLIPPER microprocessor is a high performance, three chip module that implements a new instruction set architecture designed for convenient programmability, broad functionality, and easy future expansion.

WALTER HOLLINGSWORTH, HOWARD SACHS, and ALAN JAY SMITH

The Intergraph CLIPPER[1] employs a new high performance computer architecture currently implemented as a three chip module, consisting of a processor chip and two cache and memory management unit (CAMMU) chips (see Figure 1); the processor is also available separately. It uses a new "simplified" instruction set and satisfies the key aspects of RISC designs [41] (limited and simple instruction set; maximum use of registers and minimal references to memory; and emphasis on optimizing the instruction execution pipeline). The machine has a 32-bit architecture, with a 32-bit bus data path, 32-bit registers, 32-bit data paths on chip, and a separate 32-bit virtual address space for the system and each user address space. There are nine addressing modes, permitting memory addresses to be computed from most of the useful combinations of the program counter, register contents and/or a displacement of 12, 16, or 32 bits. Instructions are 2, 4, 6, or 8 bytes long, with their length, address mode, and opcode specified in the first two bytes for efficient decoding. Data types include bytes, halfwords, words (32 bits), longwords (8 bytes), and single (4 bytes) and double (8 bytes) precision floating point. Three user visible register sets are available: 16 user and 16 supervisor general purpose 32-bit registers, and 8 floating point registers of 64 bits each. There are also the usual control registers (program counter, program status word, system status word) and some internal registers used by the processor. Eighteen traps are implemented and 128 system calls are provided. Floating point operations conform to the IEEE 754 standard [6].

The CLIPPER microprocessor uses caching and virtual memory as the standard mode of operation. The associated CAMMU chips each contain a 4 Kbyte cache, a translation lookaside buffer (TLB), and a translator. One CAMMU is used for instruction references and the other for data; the CAMMUs not only provide caching, but also implement protection, detect page faults, and watch the system bus to ensure multiple cache consistency. A full 32-bit address space is provided for the operating system and for each user process; the address space is *not* partitioned via high order address bits.

The floating point unit is on the CLIPPER processor chip. Instruction execution is pipelined with up to five instructions in the pipeline. Interlocks and dependency checks are provided in the pipeline hardware, so that no compiler inserted no-ops are needed for correct operation. Some complicated operations and diagnostics are implemented as instruction sequences in a small, on-chip ROM, called the Macro Instruction ROM (MIROM); all other instructions are hardwired. No microcode is used. The machine has 168 instructions, of which 101 are directly hardwired.

Two versions of the CLIPPER processor have been introduced. The C100, first available in 1986 from Fairchild Semiconductor, was implemented in 2 micron CMOS, was 167 K square mils, and used 132,000 transistors. The C300, available in 1988 from Intergraph Corporation, is implemented in 1.5 micron CMOS, is 285K square mils, and uses 174,000 transistors. Per-

[1] The trademark *CLIPPER* was chosen to reflect the principal architect and general manager's preference for spending his weekends sailing [34].

CLIPPER is a trademark of Intergraph Corporation.

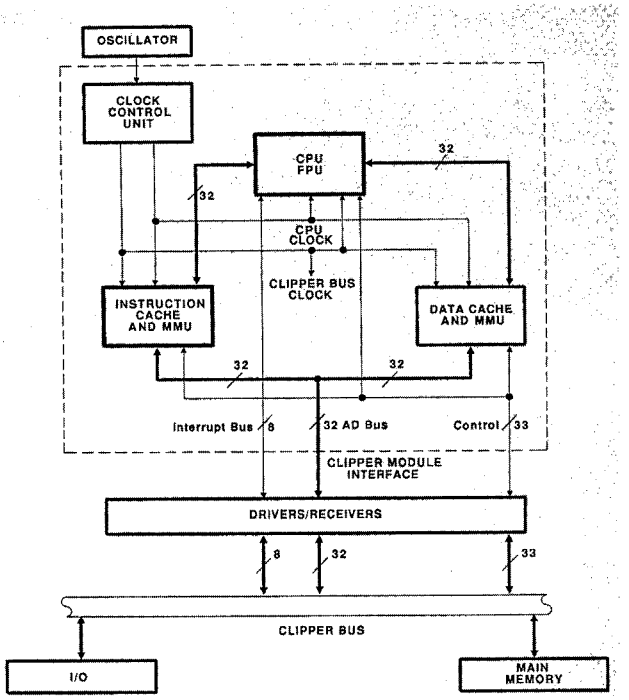

FIGURE 1. Schematic Diagram of CLIPPER

formance measurements show that the C100 implementation is 3 to 15 times as fast as a Vax 11/780 (averaging more than five times faster) and is somewhat faster than a Vax 8600. The C300 is about twice as fast as the C100. The peak execution rate in CLIPPER instructions for the C100 is 33 MIPS and 50 MIPS for the C300. Additional information on CLIPPER is available in [3] and [13].

Motivation and Design Philosophy

CLIPPER was designed and built to fulfill the need for a very high performance, microcomputer chip-based computer. The immediate applications for such a processor are in high performance workstations and "super-minicomputer" shared machines. To introduce some historical perspective, the highest performance commercial mainframe in 1976 was the IBM 370/168, which for the kind of workloads expected on CLIPPER (C, Fortran, Pascal), had performance comparable to that of the C100 CLIPPER.

When the CLIPPER project began 1982–83, no existing commercial computer architecture permitted a high performance implementation on a microprocessor chip with the necessary instruction set and architectural features. At the time, architectures available on microprocessors failed to permit high performance implementations, and most other architectures failed to be easily implementable on a chip or to provide a reasonable range of features. There were also commercial barriers to using existing architecture. The decision was thus made to design a new instruction set architecture, using the previous experience of the designers and the latest thinking in the computer architecture research community.

Fashions in computer architecture have varied widely over the last few years, changing from the "baroque" or "rococo" of the 1970s to the "minimalist" 1980s. It was widely believed in the 1970s that hardware would be very cheap, and software difficult and expensive; therefore as much functionality as possible should be moved to the hardware, resulting in complex architectures such as the DEC Vax [10, 24]. The problems with such a complex architecture are that it is very difficult to obtain good performance as a function of the amount of logic needed, it is difficult to get compilers to actually generate instructions that use the machine features, and the machine is hard (time consuming, expensive) to design, build, and debug [18, 19].

The popular thinking in computer architecture shifted in the 1980s toward very simple architectures, as originally implemented in the Cray machines (CDC 6400, 6600, 7600), studied and implemented in the IBM 801 [36], and further studied and popularized by the RISC project at Berkeley [32, 33] and the MIPS project at Stanford [19]. The essence of such machines is a simplified instruction set, which permits a hardwired implementation, a very simple instruction encoding which permits rapid decoding and effective pipelining, a load/store architecture, which greatly simplifies the control logic, and effective use of registers to cut memory traffic. Some such machines, such as RISC [32, 33] and MIPS [19] have carried these concepts to their limits by requiring fixed length instructions, almost all of which execute in one cycle. The fixed length instructions result in a significant increase in code size [14], increasing memory traffic and cache miss ratios. The single cycle execution requirement increases the machine cycle time; CLIPPER has more compact code and a shorter cycle time than such very simple machines. Some discussion of the RISC/CISC issues appear in [7] and [14].

The choice was thus made to design a new instruction set architecture (ISA). The instructions, the module design, and the functional partitioning were chosen to permit mainframe level performance, and to permit future compatible mainframe implementations. The continuing and increasing adoption of the easily ported UNIX* [37] as the standard operating system for academic, software development, and workstation environments made a new ISA commercially feasible.

Outline and Context

It is possible to describe a "computer" at many levels. The *instruction set architecture (ISA)* refers to the computer instruction set as expressed in binary or in assembly language and its functions; the ISA is usually described in the "principles of operation" manual. We

* UNIX is a trademark of AT&T Bell Laboratories.

use the term *design architecture* to refer to the highest level description of an implementation, i.e., the block diagram and parameter level. Below that are gate and circuit level descriptions.

This article focuses primarily on CLIPPER's instruction set architecture, and examines the design architecture and related issues such as performance, design tradeoffs, design implications, and areas for possible future expansion.

MEMORY ARCHITECTURE AND DATA TYPES

Memory Architecture

First, we'll provide a *brief* overview of the memory architecture of the CLIPPER microprocessor. A much more detailed description, including a discussion of the CAMMU, is provided in [3].

Address Space

In normal operation, CLIPPER uses virtual memory, although unmapped (real memory) mode is also possible. The supervisor and each user process has its own 32-bit virtual address space, defined by the PDO (page directory origin) register in the CAMMU, which contains the physical memory address of the base of the first level of the page map for the process. The page map is implemented in two levels: the first level is the page directory, and the second level contains the page tables. The page size is 4 Kbytes, which is large enough for efficient I/O [38], keeps the TLB miss ratio down, and provides enough unmapped bits that set selection in the 4 Kbyte caches can be effectively overlapped with translation [39]. The page size is also small enough to avoid unreasonable levels of internal fragmentation. No address bits are used to partition the address space, as in the Vax and MIPS machines [9], so such a partitioning isn't an obstacle to increased address space size as technology evolves.

Caching

Two cache and memory management chips (see Figure 1) provide most of the support for the memory architecture; one is used for data and the other for instructions; each is connected to the processor by its own 32-bit address/data bus. Each CAMMU has a TLB and a translator. The TLB is set associative with 128 entries organized as 64 sets of 2 elements each. Protection is provided on a page basis, with each page table entry specifying permission for the process to read, write, and/or execute from the page in supervisor and/or user state; protection bits are cached in the TLB. Page faults, protection faults, and memory errors are detected by the CAMMU and a trap code is returned to the processor for supervisor action.

Each CAMMU also contains a 4 Kbyte cache memory, organized as 128 sets of two 16-byte lines. The caching policy (copy back, write through, uncacheable) is defined on a per page basis and can vary from page to page; caching policy bits are attached to each page table and TLB entry. The CAMMU is capable of "watching" the system bus and acting to maintain cache consistency when there are multiple CAMMUs on the bus and/or when I/O operations reference data resident in the local cache. Specifically, shared data is marked "shared" and is cached write-through. Bus operations labeled as "I/O" or "shared write-through" are recognized by the CAMMU. I/O reads to lines that are dirty in the cache are preempted and the cache supplies the data. Single word I/O writes and shared write-throughs on the bus update the local copy, if any, and quad-word writes invalidate the local copy.

The low order eight pages of the supervisor address are permanently mapped by the CAMMU to provide access to Boot ROM (residing on the system bus), I/O, which is addressed via reads and writes to memory addresses, and low main memory. Trap and interrupt vectors reside in low memory. The CAMMUs are controlled by reads and writes to the I/O region of memory.

Bit Ordering

The C100 model of the CLIPPER was designed to use a consistent, "*little endian*" [23], numbering system for bits, bytes, and words, in which the most significant bit is in the highest numbered bit of the highest numbered byte, and internally, CLIPPER remains little endian. Figure 2 shows the instruction formats, in which the bit, byte, and word numbering may be observed. The "first parcel" is the first two bytes of the instruction stream; the remaining bytes of the instruction or the bytes of the following instruction(s) will appear in the second, third, and fourth parcels. This numbering system is also used in the Dec VAX, Intel 80386, and National 32000 [21]. This contrasts with the System/370 [22] in which the most significant bit is the lowest numbered bit of the lowest numbered byte; bits, bytes and words are numbered in increasing order from left to right, with the MSB at the left. The Motorola 68000 also uses a "big endian" scheme, but numbers bits in the opposite order from bytes and words [28].

In the C300 version, CLIPPER can function in either a little-endian or big-endian mode, although internally the little-endian-ness is retained. The appropriate byte order is selected at power-up time by tying a pin to either +5v or ground. When operating in big-endian mode, CLIPPER internally reverses the order of half words in the instruction buffer, reverses the order in which double word operands are loaded/stored, and changes the byte and half-word addressing to reference the correct byte or half word within a word. As a result, data can be exchanged with a big-endian machine without reversing the bytes or changing the byte numbering. It also facilitates upgrading low performance (big-endian) machines with higher performance, CLIPPER-based products. (In contrast, when data is exchanged between a Vax and an IBM 370, bytes must be explicitly swapped.)

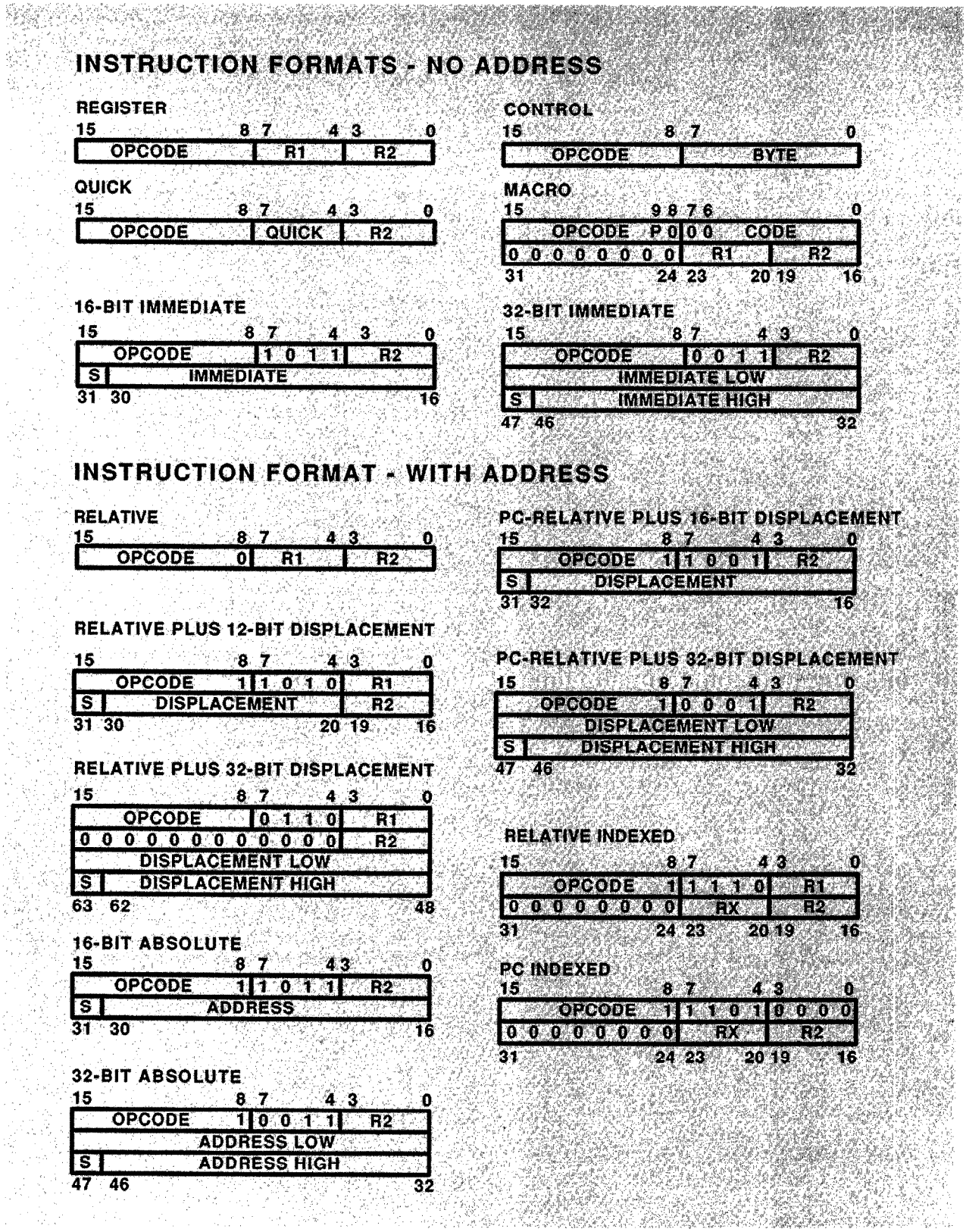

FIGURE 2. CLIPPER's Instruction Formats

Data Types

The selection of data types represents a compromise between apparent functionality, which is enhanced by a large number of data types, and implementability, which is easiest when the number of types is small. The data types supported by the CLIPPER architecture include signed and unsigned bytes, half words (2 bytes), words (4 bytes) and long words (8 bytes). There are also single and double precision (4 and 8 bytes, respectively) floating point numbers. This set of data types is sufficient to implement programming languages such as C, Fortran, and Pascal with direct hardware support provided for most language operations. (Initially, as suggested in [18], little support for bytes or half words was intended, but further examination of programming needs showed that more direct hardware support was required.)

At this time CLIPPER does not provide decimal numbers, strings, or precision beyond that of long words or double precision floating point as hardware specified data types. Strings can be easily implemented via software; CLIPPER also provides three string manipulation instructions (move, compare, fill) as Macro ROM sequences. Extended precision can be obtained via software when needed.

CLIPPER also imposes *alignment restrictions* on data items, as do other RISC and RISC-like processors. All data items must be stored on a boundary which is a multiple of its size [29]. This restriction generally causes little difficulty, and simplifies the processor implementation considerably. For CLIPPER, there is no implementation problem with *line crossers* (fetch or store requests spanning a pair of cache lines) or *page crossers* (fetch or store requests spanning a page boundary), since line and page crossers are impossible for data loads and stores. Instructions can span page boundaries, but no problem occurs since the instruction stream is fetched sequentially, four (aligned) bytes at a time.

REGISTERS AND MODES OF OPERATION

User and Supervisor General Purpose Registers

There are two sets of 16 general purpose registers (GPRs), one referenced by user mode programs and one by supervisor mode programs. The mode of the program is determined by a bit in the system status word (SSW). Two privileged instructions allow data transfers between user and supervisor registers.

Using separate user and supervisor register sets speeds up interrupt and trap handling, and makes CLIPPER especially suitable for real time applications, since registers don't need to be stored or restored when interrupts occur. The selection of 16 registers was determined by several factors, including the number of bits conveniently available for register addressing and the fact that 16 registers represent a good tradeoff; 16 registers are enough for local working storage without inducing unreasonable overhead for saving and restoring them at procedure call time. The C compiler provided by Intergraph [29] saves and restores only those registers that have been modified, and passes the first two arguments in registers. For comparison, we note that both the Vax and the IBM 370 have 16 GPRs. Lunde's results [25] suggest that 8 to 10 registers are almost always sufficient. Analyses in [14] show that with intra-procedure register allocation, no improvement in load/store traffic is obtained with more than 16 registers; even with interprocedural register allocation, minimal improvement is obtained with more than 16 registers. Eight registers, however, are too few.

The idea of register windows was first proposed by Baskett and was implemented in the Berkeley RISC project [32]; the motivation was that loads and stores due to procedure calls and returns could be avoided by simply moving to a new set of registers, using shared registers to pass parameters and results. Analyses in [14] show that with fewer than 100 registers, interprocedural register allocation results in less memory traffic than register windows; even with a total of 256 registers, register windows only outperform interprocedural register allocation by a small amount. Large register sets, such as those used in register windows, however, have a number of disadvantages [18, 33]: they require substantial chip area, only a small fraction of the registers are in use at any one time, process switching time is much larger since all registers need to be stored and restored, and larger register files are slower due to distance and circuit drive requirements. Register windows also require a mechanism to address across windows, so that nonlocal variables can be referenced [41]. For some projects (RISC II, SOAR), register access time has been a primary determinant of cycle time [14]. The decision, therefore, to use 16 user and 16 supervisor GPRs seems to be fully justified.

Floating Point Registers

CLIPPER provides a set of eight double precision floating point (FP) registers accessible in both user and supervisor states; floating point instructions refer to these. This is similar to the IBM 370 design, in which there are four FP registers. Eight registers provide sufficient storage for temporary operands, whereas four are insufficient in the absence of memory to register operations other than load and store. Four registers are clearly insufficient to permit interprocedural register allocation. (For non-numerically intensive programs, Lunde found that three floating point registers were usually sufficient. We expect a workload that is more numerically intensive than that analyzed by Lunde.)

Processor Status Registers

Three additional program addressable registers are provided, the *program counter* (*PC*), the *program status word* (*PSW*), and the *system status word* (*SSW*). The **program counter** contains the address of the instruction about to be *issued*, i.e., the instruction in the pipeline that will be released and allowed to modify the processor state (write into a register or store a result). The internal registers containing addresses of instructions following

the currently issued instruction in the pipe are not user addressable.

The **program status word (PSW)** is primarily used to hold status information (condition codes, trap codes) and to set those aspects of the processor state that the user process is permitted to modify, such as floating point trap enables. Four bits of *condition code* are provided (negative, zero, overflow, carry), and five bits of floating point exception status, as required by IEEE 754 standard, are also available. Six bits are used to enable/disable floating point traps, and two more to specify the floating point rounding mode. A trace trap bit is available. Four bits are used to record program traps (e.g., trace trap, illegal operation), and four more to record system trap types (memory error, page fault, etc.). The PSW may be read or written by the user process.

The last status register is the **system status word (SSW).** The SSW is used, among other things, to record the interrupt number and level, to enable interrupts, to set the mode (user/supervisor) and to set the protection key. The SSW may only be written in supervisor state. Its use is further described in [3].

INSTRUCTION FORMATS AND ADDRESSING MODES

Addressing Modes

The CLIPPER microprocessor has a *load/store architecture*; i.e., most of the references to memory are via load and store instructions in contrast to both the IBM 370 and DEC Vax which make extensive use of their register/memory operations (370 RX type instructions) and memory-to-memory (370 SS type) instructions. Eliminating most RX and SS instructions substantially simplifies the processor implementation by eliminating control logic and especially by simplifying recovery from traps and interrupts such as page faults and memory errors. As noted in [41], all modern, simplified architectures are load/store. The lack of RX and most SS-type instructions increases CLIPPER code size above that for such densely encoded CISC (complex instruction set computer) processors such as the Vax, the National 32000 and the Intel 80386, but provides considerably denser code than RISC processors such as the SUN Sparc and the IBM ROMP. (CLIPPER does have some SS operations implemented in the MIROM.) For RISC-I [32], a ⅔ increase in number of instructions over the Vax was observed, using a very primitive compiler for RISC. Table I shows static code sizes (the size of the text segment of the object file) for a number of standard benchmarks compiled on a number of machines; data in [14] shows that static and dynamic code sizes are very closely correlated. There are two advantages to small code sizes: there is less memory traffic, which is a limiting factor in most multiprocessor designs, and cache miss ratios are lower, since working sets are smaller; see [14] for analyses and comparative miss ratios.

For *load* and *store* instructions, CLIPPER provide nine addressing modes, which appear in Figure 2. These nine address modes represent those judged to be important for convenient programming plus those that "come for *free*;" i.e., those that can be trivially generated with the logic and data paths already available. For a 32-bit architecture, a register + 32-bit displacement mode (relative with 32-bit displacement) is very useful. The long 32-bit displacement eliminates the aggravating addressability problem posed by the 12-bit displacement of the IBM 370. The register + 12-bit displacement mode saves 4 bytes, if only a short displacement is

TABLE I. Code Size of Several Standard Benchmarks (in bytes)[a]

	Code Sizes							
Machine	doduc.f	livermore loops.f	linpack.f	whet.c	smith.f	dhry.c	NAS Kernels.f	Average Ratio
Vax (Unix 4.3 BSD)	86832	24908	9016	1900	6448	1276	21404	
	(1.0)	(1.0)	(1.0)	(1.0)	(1.0)	(1.0)	(1.0)	1.0
IBM PC/RT (ACIS 1.0)	*	41696	16792	*	10872	1632	*	
	—	(1.67)	(1.86)	—	(1.69)	(1.28)	—	1.63
Sun 3/280 (Sun Unix 4.2/3.4)	134372	29572	9388	2600	7064	1616	23372	
	(1.55)	(1.19)	(1.04)	(1.37)	(1.10)	(1.27)	(1.09)	1.23
Sun 4 (Sun Unix 4-3.2)	141280	29944	13104	3152	12248	1792	30976	
	(1.63)	(1.20)	(1.45)	(1.66)	(1.90)	(1.40)	(1.45)	1.53
Sequent (32032)	95396	*	6960	2024	5876	1368	16516	
	(1.10)	—	(0.77)	(1.07)	(0.91)	(1.07)	(0.77)	0.81
Sequent (80386)	100112	*	7264	2224	8556	1384	16776	
	(1.15)	—	(0.81)	(1.17)	(1.33)	(1.08)	(0.78)	1.05
CLIPPER	114680	24584	8976	1904	7816	1376	24456	
	(1.32)	(0.99)	(1.00)	(1.00)	(1.21)	(1.08)	(1.14)	1.11

[a] Code size in bytes is shown for the text part of the object file for the compiled version of each of several standard benchmarks: Doduc, Livermore Loops [27], Linpack [11], Whetstone [8], NAS Kernels [1], and a synthetic benchmark used by one author (Smith). In parentheses below the size is the ratio of that size to the size of the code for the VAX, using the compiler distributed with 4.3BSD UNIX. The average ratio (arithmetic average of the ratios) is shown at the right. In each case, the smallest code size obtained is shown; code sizes vary with the level of optimization. An asterisk means that the program would not compile.

needed, and the relative (register with no displacement) mode requires two bytes less. Register + displacement addressing is often used for array and stack references, and local variables.

Absolute addressing is provided with 16-bit or 32-bit address constants. Absolute addressing is typically used for references (e.g., calls) to independently compiled code segments, and in the 16-bit form, for references to low memory and within small programs.

A PC-relative address mode would have been very useful in the IBM 370 [35], and such modes are provided by CLIPPER. The PC can be used with 16- or 32-bit displacement or with a register (GPR) displacement. Most of the time, the short displacement should be sufficient; in [35] 99 percent of the branches were expressible in 16 bits or less as an offset from the PC. PC relative addressing is used primarily for branches and the PC + GPR mode for computed gotos and case statements.

Finally, a two register address mode (relative indexed) is provided, which facilitates addressing when both the base and index addresses are in registers, as well as when an array is passed as a parameter.

Four important aspects of the way the address mode is specified are evident in Figure 2. First, the address mode and opcode are *always* defined in the first instruction parcel (first two bytes), so there is no (slow) sequential decoding of the instruction; subsequent bytes can be immediately routed (as to the adder) without further examination. This encoding provides many of the supposed advantages of fixed length instructions that are used in RISC and MIPS. Second, 4 bits are used to specify the addressing mode, and only 8 of the 16 possible combinations are currently assigned, leaving the remainder available for future extensions. Third, there is no indirect addressing mode, a mode which is very difficult to implement efficiently. Finally, some of the address modes result in unused bits in some fields, which could be used in the future to generate more than 32 bits of virtual address.

To estimate the frequency of use of the various addressing modes, we examined data from the literature. In [35], addressing calculations for System/370 RX type instructions used no register 1.1 percent of the time, one register 85.6 percent of the time, and two registers 13.3 percent of the time; the RX type instruction forms an effective address as the sum of a 12-bit displacement and the contents of up to two registers. Data in [12] indicates that for the Vax, 61 percent of the operand addresses were displacement + register, and 23 percent were just register. Displacements from a register were most often one byte long. For the PDP-11 [31], most of the operand addresses were specified in a register (with or without increment or decrement), and most of the remainder were displacement + register. Based on the data cited and further data in [16] and [44], we would expect the *relative* {(*R*)}, *relative with 12-bit displacement* {(*R*) + *disp*}, *and PC Relative with 16-bit Displacement* {(*PC*) + *disp*} to account for the bulk of the address mode use. In fact, as shown in Table VII for one (unrepresentative) benchmark, those address modes are common, as are also the *PC Relative with 32-bit Displacement* {(*PC*) + *disp*)} *and Relative Indexed* {(*RX*) + (*R*)}. The former is appropriate to large programs, such as Spice, and the latter for numerical programs making many array references. We again note that many of the address modes provided "*come for free*"; e.g., the relative address mode is a displacement mode with no displacement. If each address mode had required significant additional logic, fewer modes would have been justified or included.

Instruction Formats

Figure 2 shows the available instruction formats. Those instructions using addresses have already been discussed; next we'll comment on instructions which do not contain memory addresses.

Register-to-register instructions are specified in two bytes. Register-immediate operations can be specified in 2, 4, or 6 bytes, depending on the size of the immediate constant. Immediate constants are often small; 69 percent of the immediate operands can be encoded in 4 or fewer bits and 96 percent in 8 or fewer bits [18]; the corresponding figures from [16] are 60 percent and 70 percent. The availability of the *quick* format (which provides a 4-bit unsigned constant) and the 16-bit immediate format aid code density.

The *control* opcode is used when the operation requires a small (8-bit) constant only, as for the *calls* (system call) instruction. The *macro* opcodes are used to invoke operations implemented via instruction sequences in the on-chip ROM, such as the string move (*movc*) instruction.

INSTRUCTION SET

The CLIPPER instruction set is fairly conventional and reflects the experience of its designers with respect to two factors: what is needed for convenient and efficient programmability, and what can be easily implemented in hardware. Table II shows the set of opcodes. Most of the entries are self-explanatory, and we will discuss only those that are interesting or worth explaining.

Floating Point

The CLIPPER microprocessor is unusual in that its floating point unit is on the processor chip; the floating point execution unit is also used to compute the integer multiplication, division and mod operations. Floating point arithmetic operations are performed as specified in the IEEE 754 standard. As noted earlier, there is a separate set of eight floating point registers, and all floating point operations are register to register. The floating registers may loaded or stored from/to main memory, or from/to the general purpose registers.

Branches and Condition Codes

The approach chosen for CLIPPER for controlling program execution is that of condition codes, which are set

TABLE II. Operations and Opcodes

Instruction Type	Variants and Op Codes
Load	address (loada), byte (loadb), byte unsigned (loadbu), double floating (loadd), floating status (loadfs), halfword (loadh), halfword unsigned (loadhu), immediate (loadi), quick (loadq), single floating (loads), word (loadw)
Store	byte (storb), double floating (stord), halfword (storh), single floating (stors), word (storw)
Move	double floating (movd), double floating to longword (movdl), longword to double floating (movld), word (movw), processor register to word (movpw), single floating (movs), supervisor to user (movsu), user to supervisor (movus), single floating to word (movsw), word to processor register (movwp), word to single floating (movws)
Add	double floating (addd), immediate (addi), quick (addq), single floating (adds), word (addw), word with carry (addwc)
Subtract	double floating (subd), immediate (subi), quick (subq), single floating (subs), word (subw), word with carry (subwc)
Multiply	double floating (muld), single floating (muls), word (mulw), word unsigned (mulwu), word unsigned extended (mulwux), word extended (mulwx)
Divide	double floating (divd), single floating (divs), word (divw), word unsigned (divwu)
Negate	double floating (negd), single floating (negs), word (negw)
Modulus	word (modw), word unsigned (modwu)
Scale-by	double floating (scalbd), single floating (scalbs)
Convert	double floating to single (cnvds), double floating to word (cnvdw), rounding double to word (cnvrdw), rounding single to word (cnvrsw), single floating to double (cnvsd), single floating to word (cnvsw), truncating double to word (cnvtdw), truncating single to word (cnvtsw), word to double floating (cnvwd), word to single floating (cnvws)
And Or Exclusive-or Not	immediate (andi), word (andw) immediate (ori), word (orw) immediate (xori), word (xorw) word (notw), quick (notq)
Shift arithmetic Shift logical Rotate logical	immediate (shai), longword (shal), word (shaw), longword immediate (shali) immediate (shli), longword (shll), word (shlw), longword immediate (shlli) immediate (roti), longword (rotl), word (rotw), longword immediate (rotli)
Compare	double floating (cmpd), immediate (cmpi), quick (compq), single floating (cmps), word (cmpw)
Test and set	(tsts)
Compare characters Initialize characters Move characters	(cmpc) (initc) (movc)
Pop word Push word	(popw) (pushw)
Save registers r*n* . . . r14 Save floating registers f*n* . . . f7 Save user registers	(savew*n*) (saved*n*) (saveur)
Restore registers r*n* . . . r14 Restore floating registers f*n* . . . f7 Restore user registers	(restw*n*) (restd*n*) (restur)
Branch conditional Branch floating exception	(b*): less than (bclt), less than or equal (bcle), equal (bceq), greater than (bcgt), greater or equal (bcge), not equal (bcne), less than unsigned (bcltu), less or equal unsigned (bcgtu), greater or equal unsigned (bcgeu), not carry (bnc), carry (bc), overflow (bv), not overflow (bnv), negative (bn), not negative (bnn), floating unordered (bfn) (bf*): floating any exception (bfany), floating bad result (bfbad)
Call Call supervisor Return from subroutine Return from interrupt	(call) (calls) (ret) (reti)
Trap on floating unordered Wait for interrupt No operation	(trapfn) (wait) (noop)

by one instruction and read and used by a subsequent instruction; this is similar to what is done on the IBM 370. Using condition codes for branching yields better performance and less complexity than an instruction that both tests and branches.

Four standard condition codes—N (negative), Z (zero), V (overflow) and C (carry)—are set in the PSW after certain operations. There are five floating point exception signalling codes: FX (floating inexact), FU (floating underflow), FD (floating divide by zero), FV (floating overflow), and FI (floating invalid op). Compare instructions normally set the N and Z flags; since the compare is executed by performing a subtraction, V and C may also be set.

There are two standard branch instructions. *Branch on condition* tests the NZVC PSW bits; the list of possibilities is shown in Table II. The *branch on floating exception* tests either for any exception or for a bad result (floating invalid, divide by zero, overflow). Branch instructions use the standard addressing modes, as defined in Figure 2, where the R2 field holds the condition code field that specifies the type of branch.

Implemented directly in the hardwired instruction set are the *call* and *return* (*ret*) instructions. The call instruction decrements the stack pointer (defined by the register in the R2 field), pushes the address of the next instruction onto the stack, and then loads the PC with the target address. *Return* reverses the process.

Macro Instructions

The CLIPPER processor chip includes a small ROM (known as the *Macro Instruction ROM*), which holds various useful code sequences. The MIROM contents are regular instructions, *not microcode*. Microcode requires a two-level decode [19] (instructions need to be decoded into microinstructions, and then decoded and executed), and microcoded machines tend to be slower than hardwired ones. Approximately half of the MIROM is devoted to diagnostic code to be used for chip testing and sorting during manufacturing. The remainder implements complex operations that are often found as single (usually microcoded) instructions on CISC machines. Implementing these functions as MIROM sequences increases code density and readability, instruction fetch penalties (misses, sequential fetch delays) and memory traffic decrease, and less instruction cache space is used. The MIROM concept has other advantages: (1) new instructions can be easily added; and (2) custom versions of the processor can be easily designed and implemented.

A Macro instruction actually represents a branch into the ROM; the instruction fetch unit starts fetching instructions from the ROM at the address specified by the macro opcode. Next, we'll briefly discuss the instructions implemented in the MIROM; the operation of the MIROM is described in more detail later.

Instructions to save and restore general registers (save registers (*savew***n**), restore registers (*restw***n**), *save floating registers* (*saved***n**), and *save user registers* (*saveur*)) are implemented in the MIROM as a sequence of consecutive store (or load) operations, starting from a given register number and continuing through register 14. The floating point register saves and restores are implemented similarly.

Three string (storage to storage) instructions are currently implemented in the MIROM: *movc* (copy a string of characters from/to nonoverlapping fields), *initc* (initialize a string with the contents of a register; primarily used for clearing buffers), and *cmpc* (compare two character strings). These instructions may be interrupted and restarted.

All of the conversion operations, and negate floating, scale by, and load floating status (see Table II) are implemented in the ROM.

The *return from interrupt* (*reti*) instruction restores the processor state after trap or interrupt processing. The *wait for interrupt* (*wait*) instruction causes the processor to halt pending the arrival of an enabled interrupt. The interrupt routine then determines whether to continue execution.

Test and Set

The cost and performance advantages of multiple microprocessor computer systems sharing a common memory are currently quite compelling [40]. The *Test and Set* (*tsts*) instruction is the instruction chosen for CLIPPER to implement the locks used in multiprocessor and multiprocess synchronization. As a single, indivisible operation, it loads the contents of a main memory location into a specified GPR, and sets bit 31 of the given main memory word to 1. Indivisibility is achieved by making the lock word noncacheable, and holding the main memory bus for the entire operation (which is a read/modify/write). A processor may either loop, continually testing the lock until it is released, use the *wait* instruction to sleep, or task switch. Test and set is also used by the IBM 370 and the M68000; the Vax provides seven instructions for locking and synchronization, some of which are equivalent to test and set. Test and set locks may be either cacheable or noncacheable. If they are cacheable, the local copy is updated and any remote copies are invalidated; in any case, the *tsts* operation always references main memory.

Opcode Assignment

As shown earlier in Figure 2, the high order byte of the first parcel of each instruction *always* contains the instruction opcode. As noted earlier, this greatly facilitates rapid execution, by always permitting immediate instruction decode. The assignment of bits to opcodes is shown in Figure 3. Of the possible 256 operation codes available from 8 bits, 85 instructions (including sets of instructions) are defined, and 104 of the bit combinations are used. (Some opcodes used to implement instructions that may be executed only from the MIROM are not shown in Figure 3.) That leaves over 140 possible opcodes for future expansion. In general, we have

Instruction Opcode/Mnemonic Summary

MSB \ LSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	noop															
1	movwp	movpw	calls	ret	pushw		popw									
2	adds	subs	addd	subd	movs	cmps	movd	cmpd	muls	divs	muld	divd	movsw	movws	movdl	movld
3	shaw	shal	shlw	shll	rotw	rotl			shal	shall	shll	shlll	rotl	rotll		
4					call				b				br			
5																
6	loadw		loada		loads		loadd		loadb		loadbu		loadh		loadhu	
7	storw		tsts		stors		stord		storb				storh			
8	addw		addq	addl	movw		loadq	loadl	andw			andl	orw			orl
9	addwc	subwc		negw					mulw	mulwx	mulwu	mulwux	divw	modw	divwu	modwu
A	subw		subq	subl	cmpw		cmpq	cmpl	xorw			xorl	notw		notq	
B					macro		macro									
C																
D																
E																
F																

Macro Instruction Code Field (Opcode B4)

MSB \ LSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	savew0	savew1	savew2	savew3	savew4	savew5	savew6	savew7	savew8	savew9	savew10	savew11	savew12	movc	initc	cmpc
1	restw0	restw1	restw2	restw3	restw4	restw5	restw6	restw7	restw8	restw9	restw10	restw11	restw12			
2	saved0	saved1	saved2	saved3	saved4	saved5	saved6	saved7	restd0	restd1	restd2	restd3	restd4	restd5	restd6	restd7
3	cnvsw	cnvrsw	cnvtsw	cnvws	cnvdw	cnvrdw	cnvtdw	cnvwd	cnvsd	cnvds	negs	regds	scalbs	scalbd	trapfn	loadfs

FIGURE 3. Assignment of Bits to Opcodes

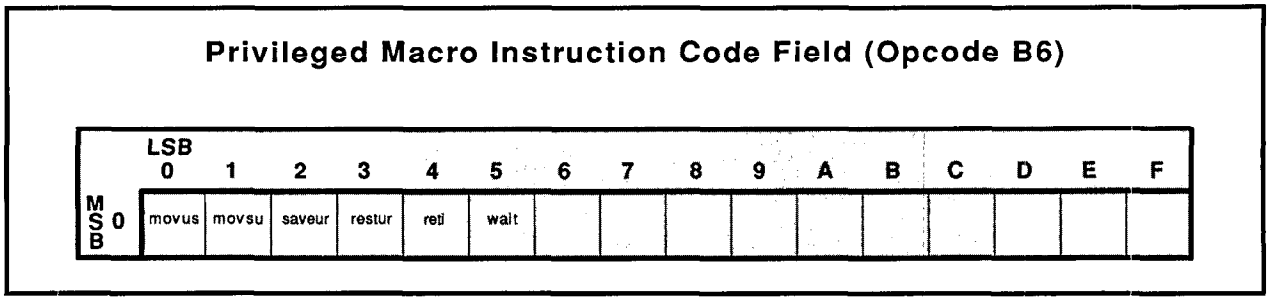

FIGURE 3. (*Continued*)

made a conscious effort to allow the CLIPPER architecture to evolve with user needs and technology trends; reserving a significant number of opcodes is one part of that effort.

INTERRUPTS, TRAPS AND SUPERVISOR CALLS

The CLIPPER microprocessor provides for 402 exception conditions: 18 hardware traps, 128 programmable supervisor calls and 256 vectored interrupts. The number of hardware traps can be expanded to 128.

A *trap* is an exception that relates to a condition of a single instruction, e.g., page fault, memory error, overflow, etc. *Interrupts* are events signalled by devices external to the CLIPPER module.

Intrap and Return Sequences

The recognition by the hardware of a trap or interrupt causes entry to a macro instruction sequence, *INTRAP*, which in noninterruptible mode performs a context switch to supervisor mode, stores the PC, PSW, and SSW on the supervisor stack, and transfers control to the trap or interrupt handler through the *vector table*. The vector table is a table in low memory containing two-word entries; each entry contains the address of the trap or interrupt handler and the new SSW. The *reti* (return from interrupt) sequence is a noninterruptible sequence which restores the system to the correct user or supervisor environment. Interrupts and traps are prioritized, with logic within the processor giving service to the highest priority event. Traps are permitted during interrupt and trap handling but result in an unrecoverable fault; page fault traps must be avoided during fault handling.

Traps

When a trap occurs, all instructions prior to the trapping instruction are completed (including those in the floating point unit), and all instructions that follow the trapping instruction are flushed from the pipeline.

Traps can be classified into several groups: data memory, floating point arithmetic, integer arithmetic, instruction memory, illegal operation, diagnostics, and supervisor calls.

Data memory and *instruction memory traps* include *correctable and uncorrectable memory errors, page faults,* and *protection faults.* In each case, the CAMMU recognizes the exception and maintains copies of the protection bits taken from the page table entries in the TLB.

The five *floating point arithmetic traps* are *invalid operation, inexact result, overflow, underflow,* and *divide by zero.* There are trap enable flags for each of these in the PSW, as well as exception flags in the PSW which are set when the corresponding events occur. An overall floating point trap enable flag (also in the PSW) can be used to disable all floating point traps.

The *trace trap* causes a trap at the end of the current instruction. A MIROM sequence is considered to be a single instruction for tracing purposes. Tracing is disabled on entry to the INTRAP sequence and trace trap handler.

Supervisor calls are implemented as traps triggered by the *calls* instruction. There are potentially 128 supervisor call codes; the CLIX® system (the Intergraph port of Unix) [30] uses approximately 60 of them.

Interrupts

Interrupts are signalled externally to the processor and appear as signals on the interrupt pins of the system bus. An interrupt is taken only when no traps are pending except the trace trap, interrupts are enabled, all instructions currently in the pipeline have completed, and string instructions have either completed or have saved sufficient state to be able to restart. (Long string instructions periodically test for pending interrupts, and if there are any, save their state and permit the interrupt to be processed.) With the exception of the string instructions, interrupts are not accepted during MIROM sequences.

There are 16 prioritized interrupt levels, with 16 interrupts of equal priority within each level. Interrupt processing can be interrupted by an event of higher priority.

DESIGN ARCHITECTURE

As explained earlier, the term *design architecture* refers to the architectural implementation at a fairly high level. Figure 4 shows the major components of the CLIPPER processor and the major interconnections in a simplified fashion. Somewhat more detail is shown in

® CLIX is a trademark of Intergraph Corporation.

Figure 5. As can be seen from those figures, the processor is divided into six major sections: the *instruction bus interface* (including an instruction prefetch buffer), the *macro instruction unit*, the *instruction control unit*, the *floating point unit*, the *integer execution unit*, and the *data bus interface*. Table III shows the fraction of the chip area occupied by various processor sections; the remainder of the area is occupied by other minor components or empty space.

Instruction Bus Interface

The instruction bus (described in more detail in [3]) is a bi-directional 45-line bus connecting the CPU chip to the Instruction CAMMU. The interface contains receivers (RCV) and drivers (DRV), and a 64-bit (8-byte) instruction buffer on the processor chip. Instructions are prefetched into this buffer, and are then fed into the instruction control unit as needed. A branch never hits in this buffer because there is no mechanism to detect that a branch target address is within the buffer; on a successful branch, the instruction buffer is cleared. The Instruction CAMMU contains its own instruction counter, and will feed the next 4 bytes of the instruction stream into the instruction buffer every time the next instruction line of the instruction bus is clocked. While within a cache line, the ICAMMU can deliver 4 bytes every 2 CPU cycles (60 ns), and the CPU can at its maximum rate execute 2 bytes (one parcel, or one 2-byte instruction) every CPU cycle (30 ns).

A multiplexor (MUX) that can accept instructions from either the instruction buffer or the Macro Instruction ROM and feed them to the instruction control unit is also associated with the instruction bus interface.

Macro Instruction Unit

The Macro Instruction ROM (MIROM) is an on-chip ROM (1 K entries × 47 bits) that implements complicated instructions as sequences of simpler *hardwired* instructions; the opcode for the MIROM implemented instruction is effectively a branch target address into the ROM; the MIROM does *not* contain microcode. Each entry in the MIROM contains two instruction parcels plus the next instruction address and a stop bit.

The set of legal opcodes for ROM instructions is a superset of the standard instruction set, including, for example, the conditional branch within the MIROM itself; those ROM-only instructions are not shown in Table II or Figure 3.

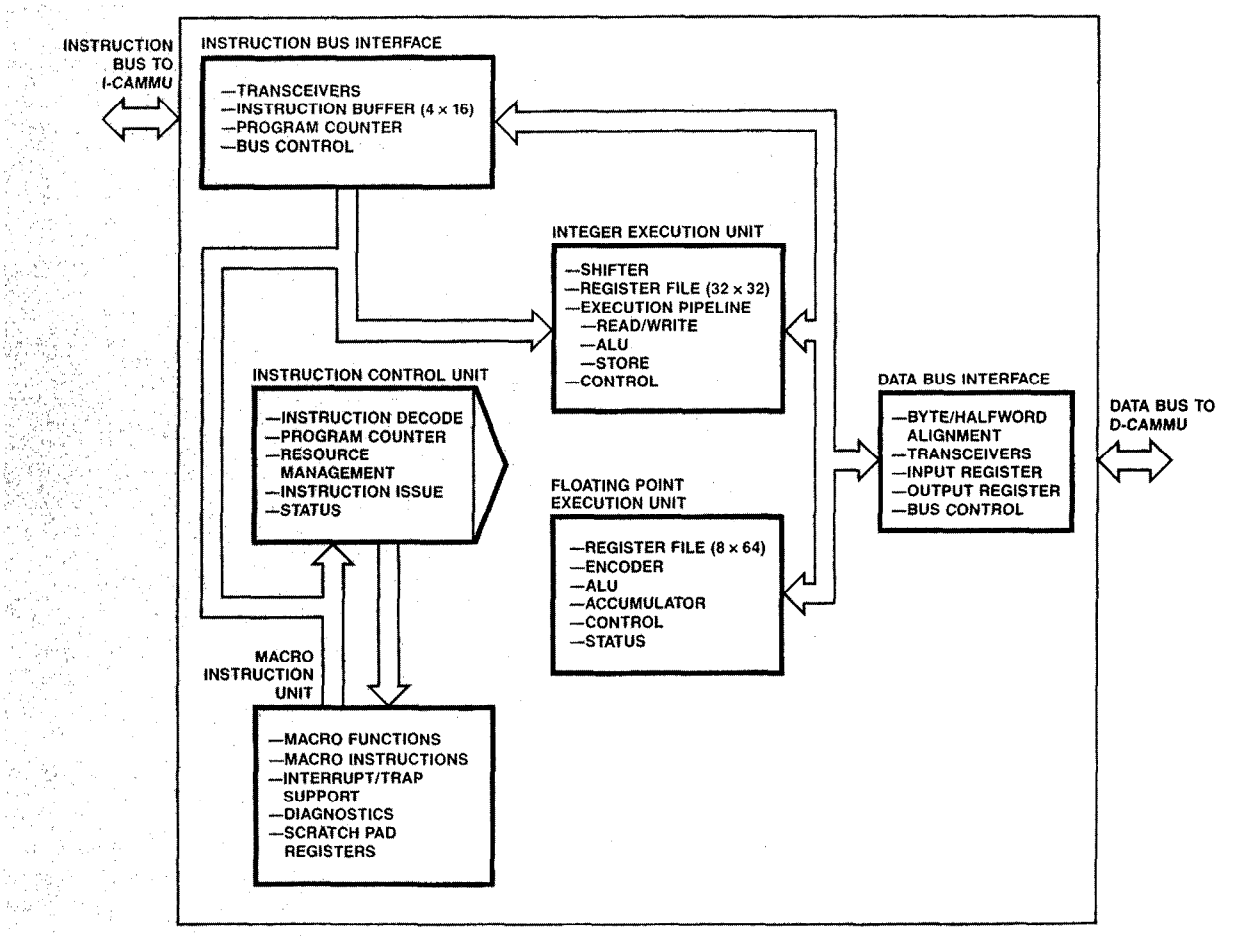

FIGURE 4. Simplified Diagram of CLIPPER's Major Components

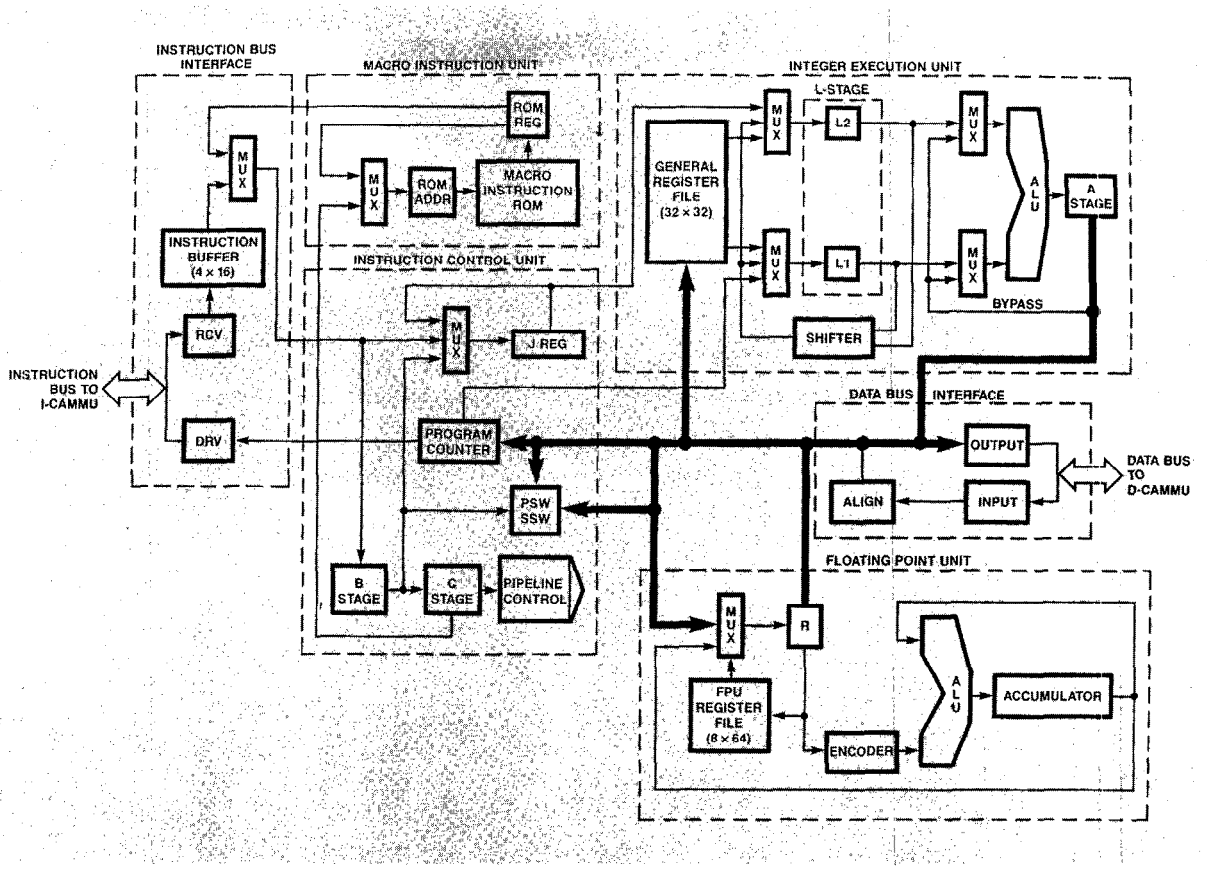

FIGURE 5. Detailed Diagram of CLIPPER's Major Components

In addition to the regular registers, there are 16 scratch registers (12 regular and 4 floating point) accessible only from instructions in the MIROM. The instructions in the MIROM also have a mechanism to reference the registers specified by the R1 and R2 fields of the Macro instruction (see Figure 2).

Integer Execution Unit

The integer execution unit contains the general register file (16 user GPRs, 16 supervisor GPRs, and 12 scratch registers), the shifter, and the ALU. The register file has three ports, permitting two reads and one write during the same machine cycle.

The shifter implements the shift and rotate instructions and is designed as a serial double bit shifter. Single and double bit shifts occur in one cycle; larger shifts require multiple cycles. Data in [20] shows that for a particular System/370 workload, only 1.9 percent of all shifts were for more than 3 bits.

The ALU (arithmetic/logic unit) implements integer addition and subtraction, bitwise logical operations, and register-to-register transfers. The address mode additions are also performed by the ALU; each requires only one pass through the ALU, since no address computation requires more than one add.

Floating Point Unit

CLIPPER is unusual among current microprocessors in having its floating point unit (FPU) on chip. Multiplication uses a Booth algorithm [2] which produces products iteratively, two bits per clock cycle for single precision (2 bits/3 cycles for double precision) in the C100 and 8 bits per cycle in the C300. Typically, one clock time is needed for round and one (3 in the C300) for normalize. Division uses a nonrestoring shift and subtract algorithm, producing 1 bit per three clocks in the C100 and 8 bits per seven clocks in the C300. Associated with the FPU is the floating point register file, which contains eight regular and four scratch-pad 64-bit floating point registers; the latter are accessible only from code running in the Macro Instruction ROM. The floating point unit is also used to perform integer multiply and divide.

The floating point unit operates in parallel with respect to the rest of CLIPPER. Although only one floating point operation can be executed at a time, operations that neither use the FPU nor rely on its output can be issued steadily while the FPU completes the current operation. As a result, much of the execution time for floating point operations will overlap that of other instructions.

TABLE III. Area Allocations for Functions on CLIPPER Chip

Processor Section	Fraction of Area C100	C300
Floating Point Unit	0.25	0.507
(Floating Point Control)	(0.067)	(0.082)
Execution Unit	0.187	0.096
(Register File)	(0.05)	(0.019)
(ALU)	(0.053)	(0.016)
ROM	0.056	0.025
Program Counter	0.013	0.006
Instruction Buffer	0.014	0.023
Branch Logic	0.041	0.032
B-stage Control Logic	0.074	0.037
C-stage (Execution) Logic	0.083	0.062
Data Memory Interface	0.026	0.022
Status Logic (PSW, SSW, Trap and Cond Codes)	0.048	0.032
Other (interconnect, misc and unused)	0.208	0.158

Floating point exceptions may be out of sequence with respect to the rest of the instruction stream. When a floating point trap occurs, the address of the floating point instruction may be recovered from a special register; the PC value pushed on the system stack can potentially be quite far from the address of the trapping instruction.

Data Bus Interface

The data bus interface consists principally of receiver and driver circuits for the data bus, and a shifter for aligning byte and half word operands. It is connected to all of the major functional units of the CPU via the S-bus so it can receive and deliver operands in the most expeditious manner.

Instruction Control Unit and CPU Pipeline

The heart of the CLIPPER processor is the instruction control unit (ICU), which is responsible for decoding instructions and controlling instruction execution. The ICU is shown in Figure 5, and the instruction execution pipeline is shown in Figure 6.

The ICU has several components. The program counter contains the address of the instruction about to be issued; to *issue* an instruction means to allow it to run to completion (i.e., modify registers or memory), provided no traps occur. Figure 6 shows two boxes, called the "B stage" and "C stage." Each consists of a set of decoding logic and registers for holding partially decoded instructions and the corresponding instruction address. The B stage is responsible for instruction decoding and resource management; resource management keeps track of which functional units are busy and allows instructions to advance to the issue stage only if the necessary units are available. The C stage holds the fully decoded instruction, and controls the operation of the integer execution unit and the floating point unit. The J register (Figure 5) is used to hold immediate values (including address offsets and address constants). The PSW and SSW registers are also located in the ICU.

There can be one instruction in each of the B and C stages. Shown preceding the B stage (Figure 6) is the instruction buffer (IB), which holds 4 parcels (8 bytes) of instructions, or up to four instructions.

The last stage of the pipeline consists of parallel integer and floating point execution units. These two execution units can operate in parallel, with one active instruction in the FPU and one instruction in each of the three stages of the integer execution unit (IEU). Those three stages are operand fetch (L stage), arithmetic (A stage: ALU or shifter) and operand write (O stage—to either registers or elsewhere). It takes three cycles for an instruction to pass through the IEU—one to read from the registers into the ALU, one to pass through the ALU or shifter, and one to write the results. There is a bypass from the output of the ALU to the input, so that results can be immediately reused in the next instruction.

LAYOUT, AREA, AND PHYSICAL PARAMETERS

Table III shows the fraction of the chip used for various purposes. The C100 (and C300) are implemented respectively using 2-micron (1.5-micron) CMOS, with two levels of metal interconnect with a 6.5 micron (5.2 micron) pitch, one polysilicon level with 2.0 micron (1.5 micron) gates and a 4.0 micron (3.2 micron) pitch, a 250 Å thick gate oxide, and 2.0 micron contacts and vias. Transistor switching speeds range from 0.5 ns (0.35 ns) to 3.0 ns, depending on gate size and load. The chip dissipates 0.5 (1.5) watts. The processor cycle time is 30 ns (20 ns), which is also the minimum time to execute an instruction. The power supply is required to provide 0 and +5 volts. The processor chip has 132 (144) pins. The chip size is 10.55 × 10.24 (13.45 × 14.12) millimeters; the package is 0.9 in.^{2} (1.025) and is surface mounted.

PERFORMANCE

CLIPPER was conceived of and designed as a high performance processor, and design decisions and tradeoffs have been made whenever possible to achieve higher performance. That high performance has indeed been achieved is evident from the instruction execution times shown in Table IV. The minimum instruction execution time is one CPU cycle time, or 30 ns in the C100 and 20 ns in the C300. The peak program execution rate is thus 50 MIPS on the C300.

Benchmark results have been obtained both from real machines running current software and from an instruction set timing simulator. The simulator shows an average of 5 to 6 clock cycles per instruction including memory delays for typical integer programs on the C100. That works out to about 5 to 7 MIPS on the C100 and 1.8 to 2.0 times that for the C300.

Table V shows the results of the Dhrystone [43], Whetstone [8], Linpack [11], Livermore Loops [27], Stanford, Smith and Doduc benchmarks on the C100 (33 MHz) and C300 (50 MHz) CLIPPER, the Vax 8600,

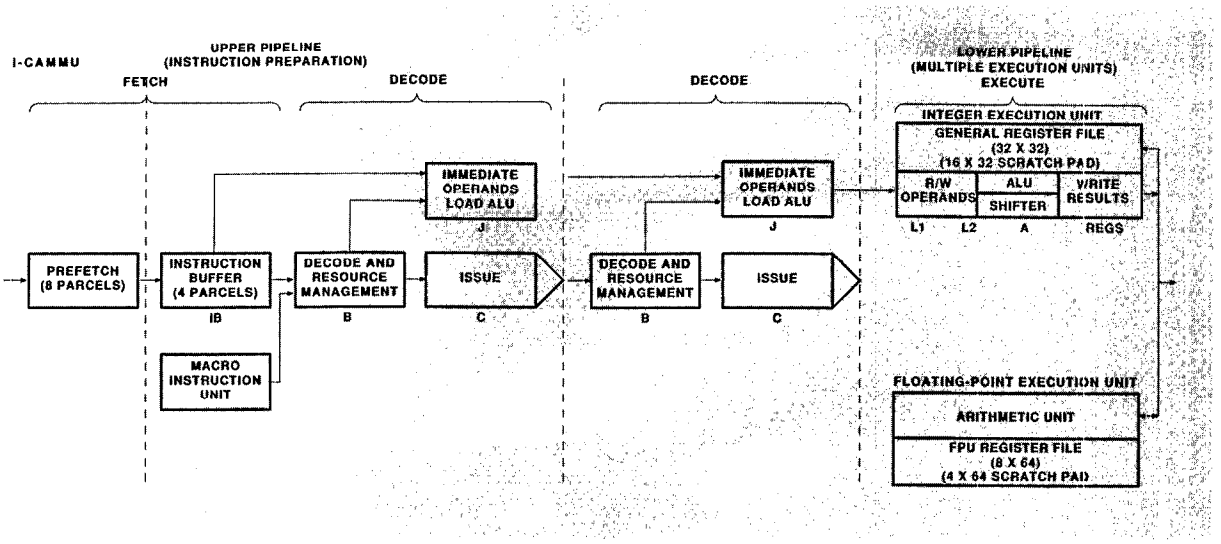

FIGURE 6. CLIPPER's Instruction Execution Pipeline

8800, and 11/785, and the SUN 3/50 (with 68881), 3/280 (with 68881), 386i/250 (with 80387) and 4/280. Whetstone and Dhrystone are in C; the others are in Fortran. All runs were with *unoptimized code*; published data usually shows optimized results. All runs were made by one of the authors personally, using the same source code in all cases, and should be comparable. Results have been normalized to the Vax 8600, since we no longer have access to a Vax 11/780. The Vax 11/780 is typically considered to be a 1 MIPS (millions of instructions per second) machine, and the Vax 8600 is approximately four times as fast, or 4 MIPS. (Actually, the Vax has a CISC instruction set, and thus generally runs at about 0.5 MIPS [12]. The Vax 11/780 runs about as fast as an IBM System/370 machine running at 1 MIPS on a scientific workload.)

While there is considerable variation among the various benchmarks, the C100 CLIPPER is approximately 1.3 times as fast as a Vax 8600, or a little over 5 MIPS. The C300 CLIPPER is about 2.5 times as fast as a Vax 8600, or about 10 MIPS. Performance ratings of all machines shown would be higher with fully optimized code.

Hardware Monitor Measurements

A limited number of programs have been run on a C100 CLIPPER and measured with a hardware monitor, and also traced. Here we summarize the measurements taken from an execution of the SPICE circuit simulator on an MOS memory cell circuit. SPICE is a large double precision numerical program, and the results are not representative for other workoads.

The execution time was 8.64 seconds at 33MHz; 1.37 seconds of system time and 7.27 seconds of user time. The instruction cache miss ratio in user state was 14 percent and the data cache miss ratio 10 percent; for system state, the miss ratios were 2.3 percent and 3.9 percent. User state data references were 69 percent read and 31 percent write; in supervisor state, the figures were 54 percent and 46 percent. Instructions were 85 percent fixed point, 12 percent point, 10 percent branch and call, and 3 percent other. The percentages of the most common instructions in user state are shown in Table VI. 33 percent of the branches were unconditional, and 67 percent were conditional. The frequencies of the various address modes are shown in Table VII. Data types for compares were 47 percent quick, 32 percent double, 13 percent word, and 9 percent immediate. Floating point instructions were 28 percent add double, 31 percent subtract double, 30 percent multiply double and 9 percent divide double. 11.7 percent of the instructions were "quick" types.

Performance versus Cycle Time and Cycles/Instruction

For a given instruction set architecture, CPU performance is inversely proportional to the product of cycle_time and cycles/instruction. CLIPPER achieves its high level of performance via a careful tradeoff of these two factors, rather than forcing all instructions to execute in one cycle, as is suggested by many RISC proponents [18, 33, 36].

The disadvantage to the single cycle per instruction approach is that not all instructions are equally complex, and the cycle time must accommodate the longest single cycle instruction; conversely, partitioning an instruction into a larger number of sequential phases provides more possibilities for overlap. For these reasons, the CLIPPER designers chose to implement the instruction set in the manner of a traditional mainframe, whereby the longer and more complex instructions are permitted more cycles to complete. The CPU cycle time in the C100 (30 ns) was chosen as a design goal, on the basis that the technology available at the time of chip

TABLE IV. Execution Times for Common Instructions

Instruction	C100		C300	
	Time (Clocks)	Time (ns)	Time (Clocks)	Time (ns)
Add Word	1	30	1	20
Logical	1	30	1	20
Move Word	1	30	1	20
Load Word/Pop	4–6	120–180	3–5	60–100
Store Word/Push	8	180	5	100
Branch (not taken)	4	120	4	80
Branch (taken)	7–9	210	7–9	140–180
Multiply word	26	720	14	280
Floating Add Single	25	750	12	240
Floating Add Double	27	810	13	260
Floating Multiply Single	24	720	13	260
Floating Multiply Double	69	2070	17	340
Floating Divide Single	110	3300	50	1000
Floating Divide Double	183	5490	71	1420

Note: Floating point times vary; figures given are averages.

TABLE V. Benchmark Results

Unoptimized code in all cases. (Quality of optimizers varies; published results are usually from optimized code.) Each set of data (with one exception) consists of two rows. The first row of each pair contains the raw performance numbers. The second row of the pair shows the data normalized to the Vax 8600 (first column). In the second row, in all cases, bigger is better. The row "average ratio" represents the (arithmetic) average speed ratio of the machines to the Vax 8600.

Benchmark	Machine								
	VAX 8600	VAX 11/785	CLIPPER C100	SUN 3/50	SUN 3/280	VAX 8800	SUN 4/280	CLIPPER C300	SUN 386i
Dhrystone 1.1	5893	1889	8670	1732	3942	7936	10489	17005	7246
(dhrystones)	1.0	.32	1.47	.30	.67	1.35	1.78	2.89	1.23
Whetstone (double)	1648	600	2048	617	909	2352	2765	5259	1202
(kilowhetstones)	1.0	.36	1.24	.38	.55	1.43	1.68	3.19	.73
Whetstone (single)	1935	692	3429	696	972	2667	4000	7026	1587
(kilowhetstones)	1.0	.36	1.77	.36	.50	1.38	2.07	3.63	.82
Linpack	.536	.218	.792	.077	.105	.777	.585	1.40	.195
(mfLops)	1.0	.41	1.48	.14	.20	1.45	1.09	2.61	.36
Livermore Loops	.521	.198	.529	.070	.095	.617	.484	.990	.185
(mfLops)	1.0	.38	1.02	.13	.18	1.18	.93	1.90	.36
Stanford (ms)	863	2356	561	2435	1143	582	698	347	694
Composite—non-FP	1.0	.37	1.50	.35	.76	1.48	1.24	2.49	1.24
Stanford (ms)	1538	4274	1132	4630	2481	1119	1163	675	1685
Composite—FP	1.0	.36	1.34	.33	.62	1.37	1.32	2.28	.91
Smith (sec)	236	589	176	708	412	156	208	102	287
	1.0	.40	1.34	.33	.57	1.51	1.13	2.31	.82
Smith (ave)	1.0	.308	1.23	.470	.385	1.51	1.23	2.35	.698
Doduc (sec)	140	355	203	639	449	95	110	83.9	359
	1.0	.39	.84	.22	.31	1.47	1.27	1.67	.39
Average Ratio	1.0	.37	1.32	.30	.48	1.41	1.37	2.53	.76

Figures for Dhrystones [43], Whetstones [8], Linpack [11] and Livermore Loops [27] are given in number of units per second. "mfLops" is "millions of floating point operations per second." Figures for Stanford, Smith and Doduc are given in units of time. Linpack matrices with leading dimension of 201. Livermore loops mflops are harmonic mean, vector length 468. The "Smith (ave)" line gives a weighted average of subcomponents of the Smith benchmark, normalized to the same weighted average for the Vax 8600 (paper in preparation).

TABLE VI. Instruction Frequencies, Spice Program

Instruction	Percentage	Cumulative
load double floating	11.58	11.58
load address	9.87	21.45
load word	9.71	31.16
add word (fixed)	7.65	38.81
branch conditional	6.57	45.38
store double floating	5.42	50.80
subtract quick	5.26	56.06
shift arithmetic immediate	4.19	60.25
move word	3.84	64.09
multiply double	2.63	66.72
compare quick	2.53	69.25
subtract double	2.52	71.77
load immediate	2.43	74.20
add quick	2.41	76.61
move double	2.28	78.89
add double	2.26	81.15
store word	1.69	82.84
load quick	1.48	84.32
move longword to double	1.29	85.61
branch	1.28	86.89
compare double	1.26	88.15
return	0.95	89.10
call	0.95	90.05
compare immediate	0.91	90.96
push word	0.74	91.70
divide double	0.72	92.42
compare word	0.66	93.08
and immediate	0.51	93.59

fabrication would permit the basic instructions (e.g., add, logical operations) to complete in one cycle. Longer instructions were allowed to take as many cycles as necessary, and the appropriate hardware support was placed on-chip to ensure that the instructions executed correctly in the presence of traps, interrupts, and data and register dependencies.

As a result, in 1986 it was possible to build a 33 Mhz part and in 1988, a 50 MHz part. This compares with speeds of about 16 MHz for the initial Sparc implementation (1987), and 8 MHz for the initial MIPS Corp. implementation (1986). The minimum instruction time for those machines is one cycle, so the peak instruction rate of CLIPPER is substantially higher.

Performance Improvement

There are two approaches to improving the performance of an implementation of a given instruction set architecture. The first is technology scaling, by which faster technology and denser packaging (or a smaller chip) permit the machine to run faster without any changes in the design architecture or even in the circuit diagram.

For the most part, performance improvements in scaling from one technology (e.g., 2-micron CMOS) to another (e.g., 1.5-micron CMOS) are independent of the actual absolute value of the cycle time. The cycle time in a machine is limited by the longest signal path (including gate delays) within a cycle; halving the longest path almost halves the cycle time. CLIPPER has already improved in performance significantly through the scaling and semiconductor process improvements that occurred in going to the C300, which also has a much improved floating point unit relative to the C100, as well as other minor functional changes.

In considering the performance of CLIPPER, the factor most strictly limiting performance on a high performance microprocessor is the memory interface [26]. As is discussed in more detail in [3], CLIPPER is most strictly limited by memory delays, despite the two buses (one each for instructions and data), and the fact that those busses are short and that each is dedicated to communication between a pair of chips. In scaling any processor, the limiting factor will continue to be the memory interface, which does not scale as well as other aspects of the machine.

The other approach to improved performance is a redesign which decreases the number of cycles per instruction. In general, this can be accomplished by the use of more logic. This type of redesign has already occurred in going from the C100 to the C300, as is shown in Table IV. There we see that by redesigning the floating point unit, floating instruction times have decreased significantly. Similar improvements are possible in other multicycle instructions. In comparison, the Amdahl 470V/6 required 5 to 6 cycles per instruction, and that was roughly halved for the 580. The DEC Vax 11/780 needed about 10 cycles per instruction [12], which was reduced to about 6 cycles for the 8600 [15]; the cycle time was only reduced from 200 ns to 80 ns, but the total performance was improved by a factor of almost five. The next versions of CLIPPER will be complete reimplementations with the mean number of cycles per instruction reduced substantially.

CONCLUSIONS

The Intergraph CLIPPER microprocessor was designed from scratch to provide high performance, cost effectiveness, convenient programmability, and an architecture that can be expanded as technology improves and the art of computer architecture design advances.

Among the important characteristics of CLIPPER are a load/store, fully hardwired architecture, full feature

TABLE VII. Frequencies of Address Modes for Spice Benchmark

Address Mode	Frequency (all instructions)	Frequency (all address modes)
12(r1)	15.66	31.48
32(pc)	10.77	21.65
(r1)	8.04	16.16
16(pc)	7.85	15.78
[rx](r1)	4.82	9.69
32(r1)	1.62	3.26
abs32	0.75	1.51
[rx](pc)	0.23	0.46
Total	49.74	

instruction set with complex instructions implemented in an on-chip ROM, an instruction set encoding that permits very fast decode, compact code, very fast cycle time, a sophisticated pipeline, on-chip floating point, and high performance. To minimize the costs of using CLIPPER in a product, CLIPPER is available as a small module containing the processor, two cache and memory management units, and the clock; thus the user doesn't have to build his own cache or memory management system. Opcodes and address modes have been left available, so that the instruction set and address space may be easily expanded.

We believe that CLIPPER represents a good set of design choices.

TRADEOFFS AND EXTENSIONS

Instruction Set Choice

Why Not "Pure RISC"? The current research trend in computer architecture is to design machines with extremely simple instruction sets. The term *RISC*, named after the Berkeley RISC project [32], is sometimes taken to mean a machine with a simple, load/store architecture; it can also be used to refer to a machine with a specific set of "features," including fixed length 32-bit instructions, single cycle execution, and register windows. This specific set of features is only one means to high performance; as is noted in [32]: "we somewhat artificially placed the following design constraints on the hardware."

We decided not to use register windows for several reasons. Flynn [14] shows that register windows do not improve memory traffic in comparison to good register allocation mechanisms, and the large number of registers can increase the cycle time [19, 33]. Fixed length instructions increase code size, memory traffic and cache miss ratios; data in Table I demonstrate that CLIPPER code is reasonably compact. The benefits of fixed length instructions are obtained in CLIPPER by placing the opcode and address mode in the first parcel (two bytes) of the instruction, so that efficient decoding is possible. Single cycle execution means that the cycle must be long enough to accommodate the longest single cycle instruction; CLIPPER allows instructions to take as many cycles as necessary. As with all RISC designs, CLIPPER is a load/store architecture, which only increases the code size slightly (see Table I), while greatly simplifying the pipeline control and interrupt and trap handling. Finally, the benefits of the more useful complex instructions (such as copy string and fill buffer) are obtained through the use of the MIROM; no complexity has been added to the implementation or the actual hardwired instruction set.

Why Not "More CISC", and What We Chose Not To Include.

There is a certain intellectual appeal to taking commonly needed software functions and implementing them in single instructions. Extreme examples are instructions to manipulate queues and compute polynomials, but we can include such reasonable operations as the three memory address instruction in this class. There are several problems with this approach. First, the number of gates available on a chip in current technology is not sufficient to implement these instructions entirely in hardware; microcode would have been required. Existing microcoded machines tend to be slow. (see [19, 33, 41] for extensive and detailed discussions of this issue.)

A natural form of computation is memory-to-register, register-to-memory, or memory-to-memory, but such instructions are not provided for three reasons:

1. It is very simple to generate the corresponding code sequences. Very few extra instruction bytes are needed, since the total number of operand specifiers is almost the same.
2. There is usually little savings in execution time, since the same sequence of operations must occur.
3. There is *considerable* additional complexity, because of the difficulty of handling memory traps and interrupts, especially page faults. In particular, if there are multiple memory references per instruction, then there can be multiple page faults; an extreme case occurs with the M68000 which permits an indirect indexed address mode.

Some complicated instructions seen in the IBM/370 and DEC Vax (e.g., translate, translate and test, edit, queue, polynomial, etc.) were omitted because of their substantial complexity, and the fact that the same functionality can be reasonably implemented in software. In practice, a compiler is seldom able to generate these instructions even when they are needed. All existing studies show that a small number of opcodes account for the large majority of all instructions executed; (see e.g., [5, 35]). For many of the same reasons, we omitted complicated branch instructions (such as decrement, test, and branch if less than zero).

Protection domains were limited to those possible from the protection bits assigned to page frames (see [3] for further discussion), since very few operating systems are prepared to take advantage of ring-structured protection domains or similarly complex designs. Likewise, a segmented address space was avoided, due to the inflexibility it imposes on the use of memory, including the impediments it provides to increases in the address space size, and the fact that the same functionality is obtained by protection bits on pages. General purpose registers were selected over dedicated registers (e.g., index, data, and address registers) for programming flexibility and generality.

A compatibility mode is not necessary in CLIPPER since it is not an upward compatible extension of an existing architecture. Not having to provide this feature greatly simplified the design, avoided undesirable architectural compromises, and permitted increased performance.

Extended precision arithmetic was not considered to be sufficiently useful at the time CLIPPER was designed to justify the area required to implement it in hardware. Extended precision is currently implemented with instruction sequences, and opcodes are available to implement extended precision in the hardwired instruction set in the future.

Possible Additions

One of the limiting factors in the design of a microprocessor is the silicon area available and the area required for each gate. For that reason, some features originally considered were deferred until future CLIPPER versions. For example, a delayed branch can reduce the pipeline penalty due to successful branches. The problem with a delayed branch is that of saving the state when a trap (e.g., a page fault) or

interrupt occurs between the time the branch decision is made (the delayed branch instruction) and the time that it takes effect (one or two instructions later). Because of its sophisticated pipeline, the existing CLIPPER chips simply don't have the space to implement this feature. (This is in contrast to various RISC chips which can easily include a delayed branch because each instruction executes in one cycle, thus only a very small amount of state has to be saved when a page fault occurs.) In addition to the delayed branch, a delayed load and vector instructions are also possible future enhancements.

Pipeline Control

CLIPPER is pipelined, and the pipeline is fully hardware controlled, with all interlocks (including checks for register dependencies) enforced with hardwired logic. This is in contrast to designs such as MIPS [4], where the compiler must reorganize code and insert no-ops as necessary. Some claim [19] that hardware pipeline control increases the cycle time, but this is disputed by others [33]. We chose to use hardware control deliberately because (1) it is a burden to require that the compiler understand the pipeline and inserts no-ops as necessary; (2) it is an unreasonable burden on the assembly language programmer and/or code generator to overcome the lack of hardware; (3) the implications of (1) and (2) are that without interlocks, code will tend to be "buggy"; (4) compilers and programs become implementation dependent; instead of just depending on the instruction set architecture, they depend on the precise features of the pipeline for correctness. Object code is thus not portable between different implementations of the same instruction set architecture. We regard (4) to be the most serious of these problems.

On-Chip Cache or Larger Instruction Buffer

Considerable study was devoted to the question of whether CLIPPER should have an on-chip cache or a significantly larger instruction buffer than the current 8 bytes. We do not have space here to discuss the reasons for the existing choice in detail (see [3]), but the basic problem is that given the limited chip area, we were unable to put enough cache or buffer on the chip to yield a useful performance improvement. For example, the 68030 (which was available in 1987) has only 256 bytes of instruction cache and 256 bytes of data cache; such small caches are of little use, and much of the potential utility of the small instruction cache is obtained from the 8-byte instruction buffer used in CLIPPER. In addition, there are problems of virtual versus real addressing, synonyms, cache flushing, and cache consistency [39]. A 2- or 3-level cache (on chip, CAMMU chip, cache board) is a future possibility.

Address Space Size

Almost any shortcoming in a computer architecture can be overcome except too small an address space; this is the reason that DEC was finally forced to design the Vax ("virtual address extension") as a replacement for the PDP-11. CLIPPER provides a flat, uniform (not partitioned) 32-bit address space. Because of the availability of additional address modes, it will be possible to define modes which produce more than 32 bits of virtual address. More than 32 bits of physical addressing can be obtained by changing the format of the page tables. These changes are straightforward and require few user programs to undergo conversion. We expect that within 10 or 15 years, both physical and virtual addresses will need more than 32 bits.

Better Multiprocessor Cache Consistency

As explained in [3], the CLIPPER CAMMU implements a bus watch cache consistency protocol; it watches memory transactions on the bus, and maintains cache consistency in a system with multiple CPUs and shared writeable areas of memory. The algorithm implemented requires that shared writeable data be marked, and thus the CAMMU need only take action when the reference is marked shared. Because the present CAMMU can do only one thing at a time, consistency operations interfere with normal CPU access, and thus the use of this mode should be minimized. With improved technology, we expect that it will be possible to implement a much more sophisticated bus interface, with a dual-ported cache directory and an optimized consistency algorithm such as is described in [42].

Acknowledgments. The Advanced Professor Division of Intergraph Corporation (formerly part of Fairchild Semiconductor) consists of over one hundred people, including those doing architecture, software, circuits, CAD, marketing, and manufacturing, all of whom contributed to this project. We want to especially note and thank Vern Brethour, James Cho, Rich Dickson, Duncan Gurley, John Kellum, Kevin Kissell, David Neff, and Ray Ryan, all of whom had major design and implementation roles throughout most of the project.

Alan Jay Smith's research in computer architecture and computer system performance is supported in part by the National Science Foundation under grants CCR-8202591 and MIP-8713274. Some research results obtained under this funding are presented in this article.

REFERENCES

1. Bailey, D.H., and Barton, J.T. The NAS Kernel Benchmark Program, NASA Tech. Memo. 86711, August 1985.
2. Cavanagh, J. *Digital Computer Arithmetic—Design and Implementation.* McGraw-Hill, New York, 1984.
3. Cho, J., Smith, A.J., and Sachs, H. The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPER, UC Berkeley CS Division Tech. Rep. UCB/CSD 86/289. March, 1986.
4. Chow, F., Himmelstein, M., Killian, E., and Weber, L. Engineering a RISC compiler system. In *Proceedings of the IEEE Compcon*, San Francisco, Calif., March, 1986, pp. 132–137.
5. Clark, D., and Levy, H. Measurement and analysis of instruction use in the VAX-11/780. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Austin, Texas, Apr. 1982), pp. 9–17. (Also *Computer Architecture News 10*, 3.)
6. Cody, W.J., Coonen, J.T., Gay, D.M., Hanson, K., Hough, D., Kahan, W., Karpinski, R., Palmer, J., Ris, F.N., and Stevenson, D. A Proposed radix- and word-length-independent standard for floating point arithmetic. *IEEE Micro* 4, 4 (Aug. 1984), 86–100.
7. Colwell, R.P., Hitchcock C.Y., III, Jensen, E.D., Brinkley Sprung, H.M., and Kollar, C.P. Computers, complexity and controversy. *IEEE Comp. 18*, 9 (Sept. 1985), 8–19.
8. Curnow, H.J., and Wichman, B.A. A synthetic benchmark. *Comput. J. 19*, 1 (Feb. 1976), 43–49.
9. DeMoney, M., Moore, J., and Mashey, J. Operating system support on a RISC. In *Proceedings of the IEEE Compcon* (San Francisco, Calif., March, 1986), pp. 138–143.
10. Digital Equipment Corp., *VAX Architecture Handbook*, 1981.
11. Dongarra, J.J. Performance of various computers using standard linear equations software in a Fortran environment. *Comp. Arch. News 13*, 1 (Mar. 1985), 3–11.

12. Emer, J.S., and Clark, D.W. A characterization of processor performance in the VAX-11/780. In *Proceedings of the 11th Annual Symposium on Computer Architecture* (Ann Arbor, Mich., June 1984), pp. 301–309.
13. Fairchild. *CLIPPER 32-bit Microprocessor User's Manual.* Prentice Hall, Englewood Cliffs, N.J., 1987.
14. Flynn, M., Mitchell, C., and Mulder, J. And now a case for more complex instruction sets. *IEEE Comp.* (Sept. 1987), 71–83.
15. Fossum, T., McElroy, J., and English, W. An Overview of the VAX 8600 System, *Digital Tech. J. 1* (Aug. 1985), 8–23.
16. Grochowski, E.T. An Instructor Tracer for the Motorola 68010, MS Project Report, Computer Science Division, EECS Dept., University of California, Berkeley, Calif., May, 1986.
17. Hansen, P.M., et al. A performance evaluation of the Intel iAPX 432, *Comp. Arch. News 10*, 4 (June 1982), 17–26.
18. Hennessy, J., Jouppi, N., Baskett, F., Gross, T., and Gill, J. Hardware/software tradeoffs for increased performance. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (Sigarch Computer Architecture News, *10*, 2, March, 1982), pp. 2–11.
19. Hennessy, J. VLSI processor architecture, *IEEETC C-23*, 12 (Dec. 1984), 1221–1246.
20. Huck, J. Comparative analysis of computer architectures, Computer Systems Laboratory Tech. Rep. 83-243, May, 1983, Stanford University, Stanford, Calif.
21. Hunter, C., and Farquhar, E. Introduction to the NS16000 architecture. *IEEE MICRO 4*, 2 (Apr. 1984), 26–47.
22. IBM Corporation. IBM Systems/370 Principles of Operation, Form Number GA22-7000-5, IBM, Poughkeepsie, New York, 1976.
23. Kirrmann, H. Data format and bus compatibility in microprocessors. *IEEE MICRO 3*, 4 (Aug. 1983), 32–47.
24. Levy, H., and Eckhouse, R. *Computer Programming and Architecture: The VAX-11*, Digital Press, Bedford, Mass., 1980.
25. Lunde, A. Evaluation of Instruction Set Processor Architecture by Program Tracing. Tech. Rep., Dept. of Computer Science, Carnegie-Mellon University, July, 1974.
26. Mateosian, R. System considerations in the NS32032 design. In *Proceedings of the NCC*, 1984, pp. 77–81.
27. McMahon, F. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range, Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, December, 1986.
28. Motorola Corporation *16-Bit Microprocessor User's Manual.* 3rd Edition, 1982.
29. Neff, D. C compiler implementation issues on the CLIPPER microprocessor. In *Proceedings of Compcon* (San Francisco, Calif., Mar. 1986), pp. 196–201.
30. Neff, L. CLIPPER microprocessor architecture overview. In *Proceedings of Compcon.* (San Francisco, Calif., Mar. 1986), pp. 191–195.
31. Neuhauser, C.J. Analysis of the PDP-11 Instruction Stream. Tech. Rep. 183, Computer Systems Laboratory, Stanford Electronics Laboratories, Stanford University, Stanford, Calif. 94305, February, 1980.
32. Patterson, D.A., and Sequin, C.H. A VLSI RISC, *IEEE Comp. 16*, 9 (Sept. 1982), 8–20.
33. Patterson, D. Reduced instruction set computers. *Commun. ACM 28*, 1 (Jan. 1985), 8–21.
34. Perry, T. At work or play, he's the captain. *IEEE Spect. 24*, 6 (June 1987), 56–59.
35. Peuto, B., and Shustek, L. An instruction timing model of CPU performance. In *Proceedings of the 4th Annual Symposium on Computer Architecture* (College Park, Md., Mar., 1977), pp. 165–178.
36. Radin, G. The 801 Minicomputer. *IBM J. Res. Devel. 27*, 3 (May, 1983), 237–246.
37. Ritchie, D.M., and Thompson, K. The UNIX timesharing system. *Commun. ACM 17*, 7 (July, 1974), 365–375.
38. Smith, A.J. Input/output optimization and disk architecture: A survey. *Perform. Eval. 1*, 2 (1981), 104–117.
39. Smith, A.J. Cache memories. *Comp. Surv. 14*, 3 (Sept. 1982), 473–530.
40. Smith, A.J. Problems, directions and issues in memory hierarchies. In *Proceedings of the 18th Annual Hawaii International Conference on System Sciences* (Honolulu, Jan. 2–4, 1985), pp. 468–476. Also available as UC Berkeley CS Report UCB/CSD84/220.
41. Stallings, W. Reduced instruction set computer architecture. In *Proceedings of the IEEE 76*, 1 (Jan. 1988), pp. 38–55.
42. Sweazey, P., and Smith, A.J. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (Tokyo, June, 1986), pp. 414–423.
43. Weicker, R.P. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM 27*, 10 (Oct. 1984), 1013–1030.
44. Wiecek, C.A. A case study of VAX-11 instruction set usage for compiler construction. In *Proceedings of the Symposium on Architecture Support for Programming Languages and Operating Systems.* Palo Alto, Calif., March, 1982.

CR Categories and Subject Descriptors: C.1.1 **[Processor Architectures]**: Single Data Stream Architectures—*Pipeline Processors, SISD Architectures, Von Neumann Architectures*; C.4 **[Performance of Systems]**; C.5 **[Computer System Implementation]**: Microprocessors
General Terms: Design, Measurement, Performance
Additional Key Words and Phrases: CLIPPER, Instruction Set

ABOUT THE AUTHORS:

WALT HOLLINGSWORTH works for Intergraph in Palo Alto, California, and is one of the CLIPPER architects. He has over 29 years of computer design experience beginning with navigation computers at Honeywell, the Sigma mainframes of SDS/Xerox, and has also worked at Telefile and Cray Research. He received a BSEE from Lamar Tech in 1960. Author's Present Address: Intergraph Corporation, Advanced Processor Division, 2400 Geng Road, Palo Alto, CA 94301.

HOWARD SACHS is Vice President and General Manager of Intergraph Corporation's Advanced Processor Division. He has extensive experience in computer design, and is the chief architect of Intergraph's CLIPPER microprocessor. Author's Present Address: Intergraph Corporation, Advanced Processor Division, 2400 Geng Road, Palo Alto, CA 94301.

ALAN JAY SMITH received a B.S. degree in electrical engineering from the MIT, and M.S. and Ph.D. degrees in computer science from Stanford University. He is currently a Professor in the Computer Science Division of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, where he has been on the faculty since 1974, and was vice chairman of the EECS department from July, 1982–June, 1984. His research interests include the analysis and modeling of computer systems and devices, computer architecture, and operating systems. Author's present address: Alan Jay Smith, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA 94720.