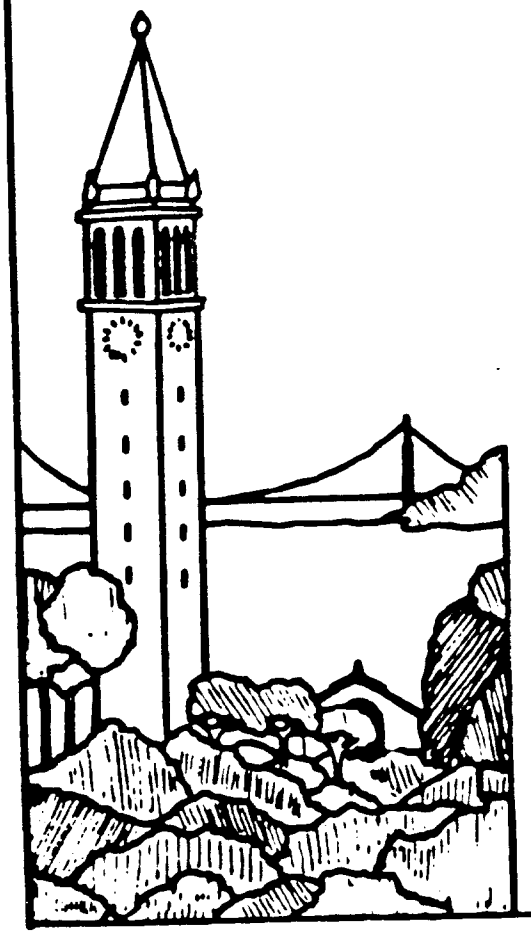
THE FAIRCHILD CLIPPER: Instruction Set Architecture and Processor Implementation

Walter Hollingsworth
Howard Sachs
Alan Jay Smith



Report No. UCB/CSD 87/329

February 11, 1987

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The Fairchild CLIPPER™+: Instruction Set Architecture and Processor Implementation

Walter Hollingsworth*, Howard Sachs* and Alan Jay Smith†#

Abstract

The Fairchild CLIPPER microprocessor is a new high performance three chip module consisting of a microprocessor chip and two cache and memory management (CAMMU) chips, mounted on a small PC board. CLIPPER implements a new instruction set architecture which has been designed for high performance, convenient programmability, broad functionality and sufficient architectural "openness" to permit future evolution and a variety of implementations.

In this paper, we (a) describe the instruction set architecture of CLIPPER, (b) describe the chip design architecture and the interesting features of the implementation, and (c) consider in some detail the reasons for various design decisions and tradeoffs. Performance estimates are provided. Possible future directions for both performance and instruction set architecture are outlined. Some comments on the RISC vs. CISC issue are given.

+CLIPPER is a trademark of Fairchild Semiconductor Corporation

*Fairchild Semiconductor Corporation, 4001 Miranda Avenue, Palo Alto, Ca., 94304.

†Computer Science Division, EECS Dept., University of California, Berkeley, Ca. 94720

#Research by Professor Smith in computer architecture and computer system performance is supported in part by the National Science Foundation under grant CCR-8202591, and by the Defense Advance Research Projects Agency (DoD), under Arpa Order No. 4871, Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089 Some research results obtained under this funding are presented in this paper.

1. Introduction

1.1. Summary of Features

The Fairchild CLIPPER[+†] employs a new high performance computer architecture implemented initially 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 instruction set which is "simplified" and "RISC-like" but not RISC. 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 for 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 there is provision for 128 system calls. Floating point operations conform to the IEEE 754 standard [Cody84].

The CLIPPER microprocessor has been designed with virtual memory as the standard mode of operation. The associated CAMMU chips each contain a 4Kbyte 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 in order 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. A few 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

---

+ CLIPPER is a Trademark of Fairchild Semiconductor Corporation

†The trademark "CLIPPER" was chosen in a reflection of the preference of the principal architect and program manager for spending his weekends sailing.

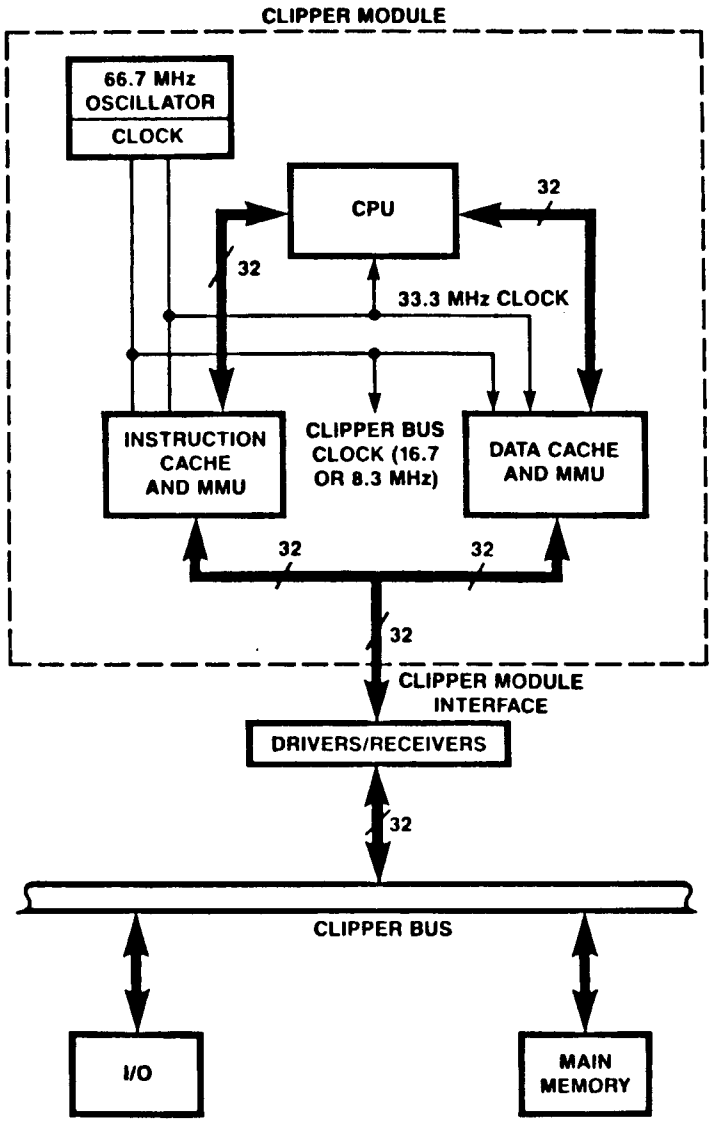


Figure 1. CLIPPER Module and Interface

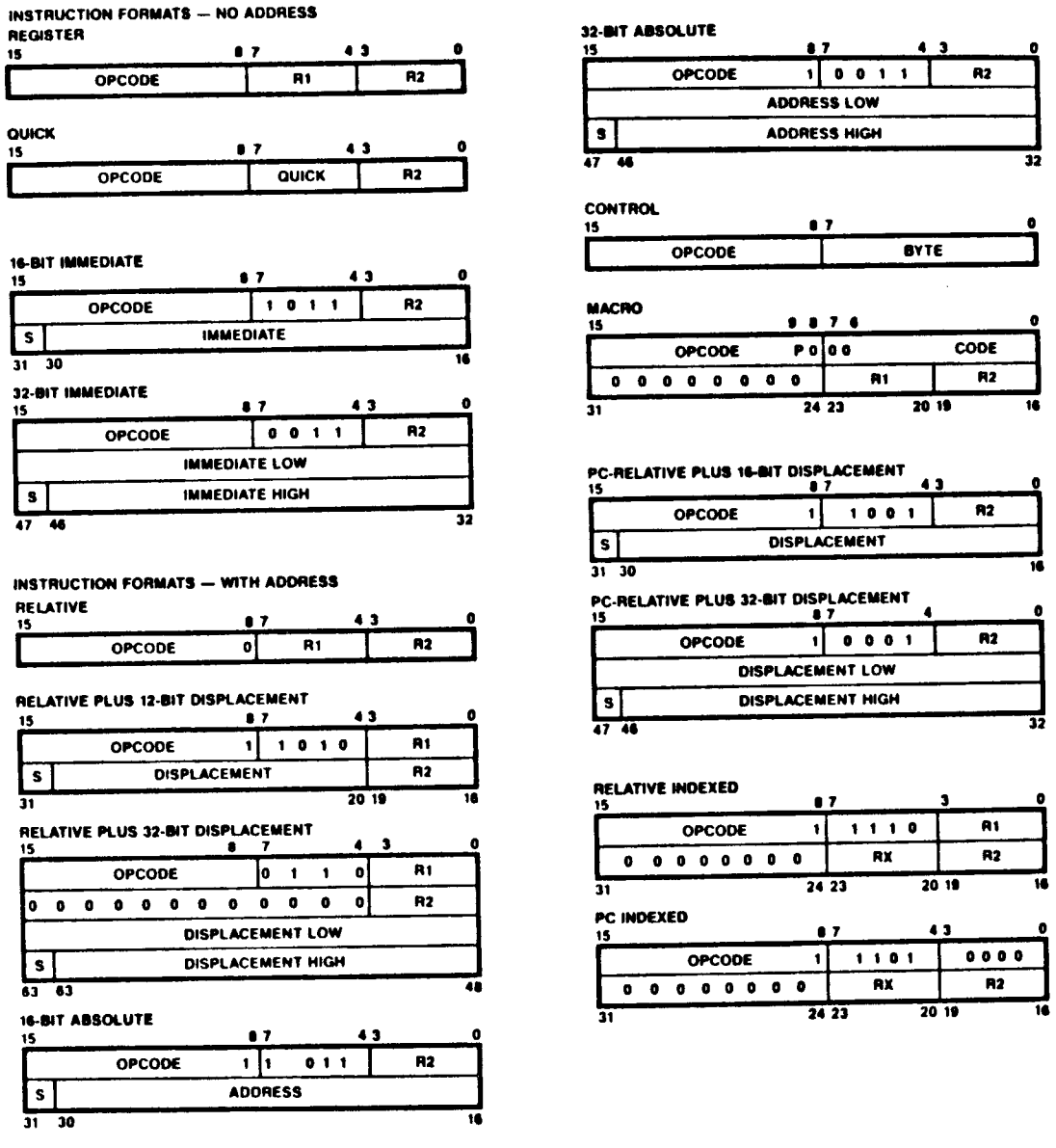


Figure 2. Instruction Formats and Address Modes

microcode is used. The machine has 168 instructions, of which 101 are directly hardwired.

The processor chip is implemented in 2 micron CMOS, is 156K square mils and uses 132,000 transistors. Performance estimates show that the current implementation is somewhat faster than a VAX 8600, which is itself generally referred to as a "4-MIPS" machine; CLIPPER is thus a 5 MIPS computer; the peak execution rate in CLIPPER instructions is 33 MIPS. Additional information on CLIPPER is available in [Fair86, Cho86].

1.2. Motivation and Design Philosophy

The decision to design and build CLIPPER was made in the belief that there existed and exists the need for a very high performance computer based on a microprocessor chip. The immediate applications for such a processor are in high performance workstations and for use in "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 CLIPPER.

It is the belief of the CLIPPER designers that no existing commercial computer architecture in 1982-3 met the requirements of: (a) permitting a high performance implementation (b) on a microprocessor chip (c) with the necessary instruction set and architectural features. Architectures then available on microprocessors failed to permit high performance implementations, and most other architectures failed to either be easily implementable on a chip or failed to provide a reasonable range of features. There were also commercial barriers to the use of an 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 in the 1970s to the minimalist 1980's. It was widely believed in the 1970's that hardware would very cheap, software was difficult and expensive, and that therefore as much functionality as possible should be moved to the hardware. The result was complex architectures such as the DEC VAX [DEC81, Levy80]. 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, and the machine is hard (time consuming, expensive) to design, build, and debug [Henn82,84].

The popular thinking in computer architecture shifted in the 1980s toward very simple architectures, as originally implemented in the Cray designed machines (CDC 6400, 6600, 7600), studied and implemented in the IBM 801

[Radi81] and further studied and popularized by the RISC project at Berkeley [Patt85] and the MIPS project at Stanford [Henn84]. Such machines permit high performance implementations and rapid design and development but are less than ideal in terms of programmability; one becomes very dependent on sophisticated software technology to obtain good performance and guarantee correct operation. There is a saying attributed to Einstein [Lamp83], to the effect that "everything should be made as simple as possible, but no simpler." Our feeling was that the "pure RISC" type architectures provided insufficient features and functionality for a commercial product, and that equivalent performance advantages were also available in a carefully designed architecture of "moderate" complexity. Some discussion of the RISC/CISC issue appears in [Colw85].

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™* [Ritc74] as the standard operating system for academic, software development and workstation environments made the decision to use a new ISA commercially feasible.

1.3. 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 function; the ISA is usually described in the Principles of Operation manual. We use the term *design architecture* to refer to the highest level of description of an implementation, i.e., the block diagram and parameter level. Below that are gate and circuit level descriptions.

This paper is primarily directed at the instruction set architecture of CLIPPER, with some of the material concerned with the design architecture and related issues such as performance, design tradeoffs, design implications and areas for possible future expansion.

A brief summary of the memory architecture of CLIPPER is provided in the next section. Registers and modes of operations are discussed in section 3, instruction formats and addressing modes in section 4, and the instruction set itself in section 5. Interrupts and traps are considered in section 6. Section 7 presents the essentials of the design architecture and implementation. Performance is discussed in section 8 and design tradeoffs and possible extensions are reviewed in section 9.

*UNIX is a trademark of AT&T Bell Laboratories

2. Memory Architecture and Data Types

2.1. Memory Architecture

In this section, we 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 [Cho86], and that paper should be read as the companion to this one.

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, with the first level referred to as the page directory and the second level containing the page tables. The page size is 4Kbytes, which is large enough for efficient I/O [Smit81], keeps the TLB miss ratio down and provides enough unmapped bits that set selection in the 4Kbyte caches can be effectively overlapped with translation [Smit82]. 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 is done in the VAX and MIPS machines [Demo86], so such a partitioning doesn't constitute an obstacle to increased address space size as technology evolves.

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 connected to the processor by its own 32-bit bus. Each CAMMU has a TLB (translation lookaside buffer) and a translator. The TLB is set associative with 128 entries organized as 64 sets of two 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 4Kbyte 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 CPUs on the bus and/or when I/O operations reference data resident in the local cache.

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.

Originally, CLIPPER was designed to use a consistent, *"little endian"* [Kirr83], numbering system for bits, bytes and words, in which the most significant bit is in the highest numbered bit of the highest numbered byte, has been defined, 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 the same as is used in the DEC VAX and National 32000 [Hunt84]. This contrasts with the System/370 [IBM76] 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 [Moto82].

CLIPPER has been enhanced so that in its current version, it can function in either a little endian or big-endian mode. The appropriate byte order is selectable at power-up time by tying a pin to either +5v or ground. When operating in big-endian mode, CLIPPER does the following: (a) reverses the order of half-words in the instruction buffer, (b) reverses the order in which double word operands are loaded/stored, (c) changes the byte and half word addressing so as to reference the correct byte or half word within a word. When operating in big-endian mode, CLIPPER can function effectively in a system with big endian processors and/or data files created by big-endian machines.

2.2. 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 [Henn82], little support for bytes or half words was intended, but further examination of programming needs showed that more direct hardware support was required.)

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

CLIPPER also imposes *alignment restrictions* on data items. All data items must be stored on a boundary which is a multiple of its size [Neff86a]. This restriction generally causes little difficulty, and considerably simplifies the processor implementation. 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.)

3. Registers and Modes of Operation

3.1. 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 SSW. There are two privileged instructions that allow data transfers between user and supervisor registers.

The use of separate user and supervisor register sets speeds up interrupt and trap handling. 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 Fairchild [Neff86a] saves and restores only those registers that have been modified. For comparison, we note that both the VAX and the IBM 370 have 16 GPRs. Lunde's results [Lund74] suggest that 8-10 registers are almost always sufficient.

3.2. Floating Point Registers

CLIPPER provides a set of eight double precision floating point registers accessible in both user and supervisor states; floating point instructions refer to these. This is similar to the IBM 370 design; in that machine there are four FP registers. Eight registers seem to provide sufficient storage for temporary operands, whereas four seem insufficient in the absence of memory to register operations other than load and store. (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.)

3.3. 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). Not user addressable are the internal registers containing addresses of instructions following the currently issued instruction in the pipe.

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 Std. 754, 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 [Cho86].

4. Instruction Formats and Addressing Modes

4.1. Addressing Modes

The CLIPPER microprocessor uses primarily a *load/store architecture*; i.e. most of the references to memory are via load and store instructions. This is contrast to both the IBM 370 and DEC VAX which make extensive use of their register/memory operations (370 RX type instructions) and their memory to memory (370 SS type) instructions. The elimination of 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. Without the RX and SS type instructions, we expect that CLIPPER code will be slightly larger in size than that for the IBM 370 and VAX, but since all operands must always be specified, the increase should be small. Because we have variable length instructions with a variety of addressing modes, CLIPPER experiences a much smaller increase in code volume than the RISC [Patt85] and MIPS [Mous86] processors, both of which use only fixed length 32-bit instructions. For RISC-I [Patt81], a 2/3 increase in number of instructions

over the VAX was observed, using a very primitive compiler for RISC.

For *load* and *store* instructions, CLIPPER provides 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. can be trivially generated given 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 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.

It has been observed [Peut77] that a PC-relative address mode would have been very useful in the IBM 370, and such modes are provided by CLIPPER. The PC can be used with a 16 or 32 bit displacement or with a register (GPR) displacement. Most of the time, the short displacement should be sufficient; in [Peut77], 99% 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 when an array is passed as a parameter.

It is important to note four aspects of the way the address mode is specified, as shown in figure 2. First, the address mode is *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 much of the supposed advantages of fixed length instructions such as 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; the remainder are available for future extensions. Third, there is no indirect addressing mode, a mode which is very difficult to implement efficiently. Finally, we note that 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 note data from the literature. In [Peut77], it was found that for the workload examined

there, addressing calculations for System/370 RX type instructions used no register 1.1% of the time, one register 85.6% of the time, and two registers 13.3% 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 [Emer84] indicates that for the VAX, 61% of the operand addresses were displacement+register, and 23% were just register. Displacements from a register were most often one byte long. For the PDP-11 [Neuh80], 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 [Groc86] and [Wiec82], we expect the *relative {(R)}, relative with 12-bit displacement {(R)+disp}, and the PC Relative with 16-bit Displacement {(PC)+disp}* to account for the bulk of the address mode use.

4.2. Instruction Formats

Figure 2 shows the available instruction formats. Those instructions using addresses have been discussed above; here we 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; in [Henn82], is reported that 69% of the immediate operands can be encoded in 4 or fewer bits and 96% in 8 or fewer bits; the corresponding figures from [Groc86] are 60% and 70%. The availability of the *quick* format (which provides a 4-bit unsigned constant) and the 16-bit immediate format should greatly 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 those used to invoke operations implemented via instruction sequences in the on-chip ROM, such as the string move (*movc*) instruction.

5. Instruction Set

The CLIPPER instruction set is fairly conventional and reflects the experience of the designers with respect to two factors: what is needed for convenient and efficient programmability, and what can be easily implemented in hardware. Table 1 shows the set of opcodes, grouped in such a way as to minimize the redundant listing of the same opcode for various data types. Most of the entries there are self explanatory, and in this section we discuss only those operations that are either interesting or worth explaining.

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	immediate (andi), word (andw)
Or	immediate (ori), word (orw)
Exclusive-Or	immediate (xori), word (xorw)
Not	word (notw), quick (notq)
Shift Arithmetic	immediate (shai), longword (shal), word (shaw) longword immediate (shali)
Shift Logical	immediate (shli), longword (shll), word (shlw) longword immediate (shlli)
Rotate Logical	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)

Table 1a - Operations and Opcodes

Instruction Type	Variants and Op Codes
Compare Characters	(cmpc)
Initialize Characters	(initc)
Move Characters	(movc)
Pop Word	(popw)
Push Word	(pushw)
Save Registers *rn*...r14	(savew*n*)
Save Floating Registers *fn*...f7	(saved*n*)
Save User Registers	(saveur)
Restore Registers *rn*...r14	(restw*n*)
Restore Floating Registers *fn*...f7	(restd*n*)
Restore User Registers	(restur)
Branch Conditional	(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)
Branch Floating Exception	(bf*): floating any exception (bfany), floating bad result (bfbad)
Call	(call)
Call Supervisor	(calls)
Return from Subroutine	(ret)
Return from Interrupt	(reti)
Trap on Floating Unordered	(trapfn)
Wait for Interrupt	(wait)
No Operation	(noop)

Table 1b - Operations and Opcodes

5.1. Floating Point

The CLIPPER microprocessor is unusual in placing its floating point unit 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 8 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.

5.2. Branches and Condition Codes

The approach chosen for CLIPPER for controlling program execution is that of condition codes, which are set by one instruction and read and used by a subsequent instruction; this is similar to what is done on the IBM 370. The use of condition codes for branching yields better performance and less complexity than an instruction which both tests and branches.

There are four standard condition codes: N (negative), Z (zero), V (overflow) and C (carry), which 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, it is also possible that V and C may be set.

There are two standard branch instructions. *Branch on condition* tests the NZVC PSW bits; the list of possibilities is shown in table 1. 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.

5.3. Macro Instructions

The CLIPPER processor chip includes a small ROM (known as the *Macro Instruction ROM*), which holds various useful code sequences. 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 which 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) decrease, and less instruction cache space is used.

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. In this section, we briefly discuss those instructions implemented in the MIROM; the operation of the MIROM is described in more detail in section 7.2.

Instructions to save and restore general registers (save registers (*savew***n**), restore registers (*restw***n**), *save floating registers (savedn)* 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. These are *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 1) are implemented in the ROM.

The *return from interrupt (reti)* instruction restores the processor state after trap or interrupt processing, and is discussed in more detail in section 6.1. 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.

5.4. Test and Set

The cost and performance advantages of multiple microprocessor computer systems sharing a common memory are currently quite compelling [Smit85]. The *Test and Set (tsts)* instruction is the instruction chosen for CLIPPER for the implementation of locks to be used in multiprocessor and multiprocess synchronization. As a single, indivisible operation, it (a) loads the contents of a main memory location into a specified GPR, and (b) sets bit 31 of the given main memory word to 1. Indivisibility is achieved by (a) making the lock word non-cacheable, and (b) 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, may use the *wait* instruction to sleep, or may 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.

5.5. Opcode Assignment

As shown earlier in figure 2, the high order byte of the first parcel of each instruction contains the instruction opcode. The assignment of bits to opcodes is shown in figure 3.

The important observation to be made from figure 3 is that 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. (Not shown in figure 3 are some opcodes used to implement instructions which may be executed only from the MIROM.) That leaves over 140 possible opcodes for future expansion. In general, we have made a conscious effort to allow the CLIPPER architecture to evolve with user needs and technology trends, and reserving a significant number of opcodes is one part of that effort.

6. 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 at some future time.

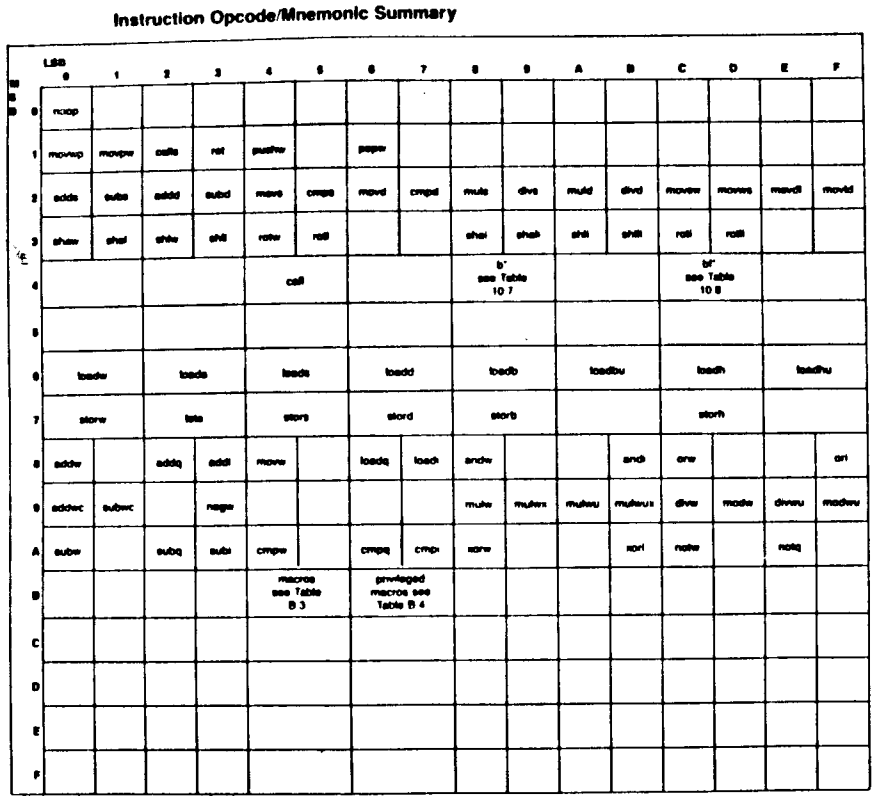
A *trap* is an exception that relates to a condition of a single instruction, e.g., page fault, memory error, overflow, etc. *Interrupts* are events signalled by devices external to the CLIPPER module.

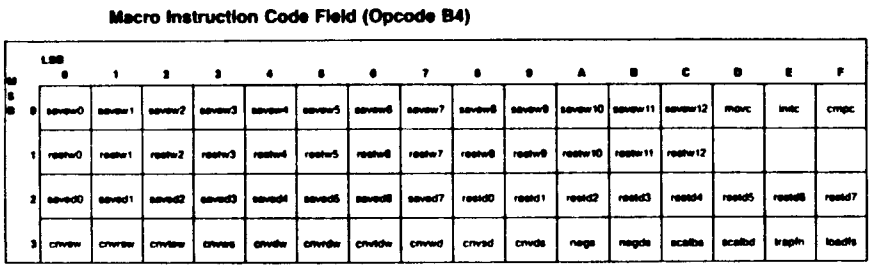
6.1. Intrap and Return Sequences

The recognition by the hardware of a trap or interrupt causes entry to a macro instruction sequence, *INTRAP*, which in noninterruptable 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 2-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 non interruptable 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.

6.2. Traps

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





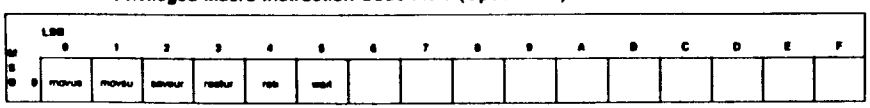


Figure 3. OpCode Bit Assignment

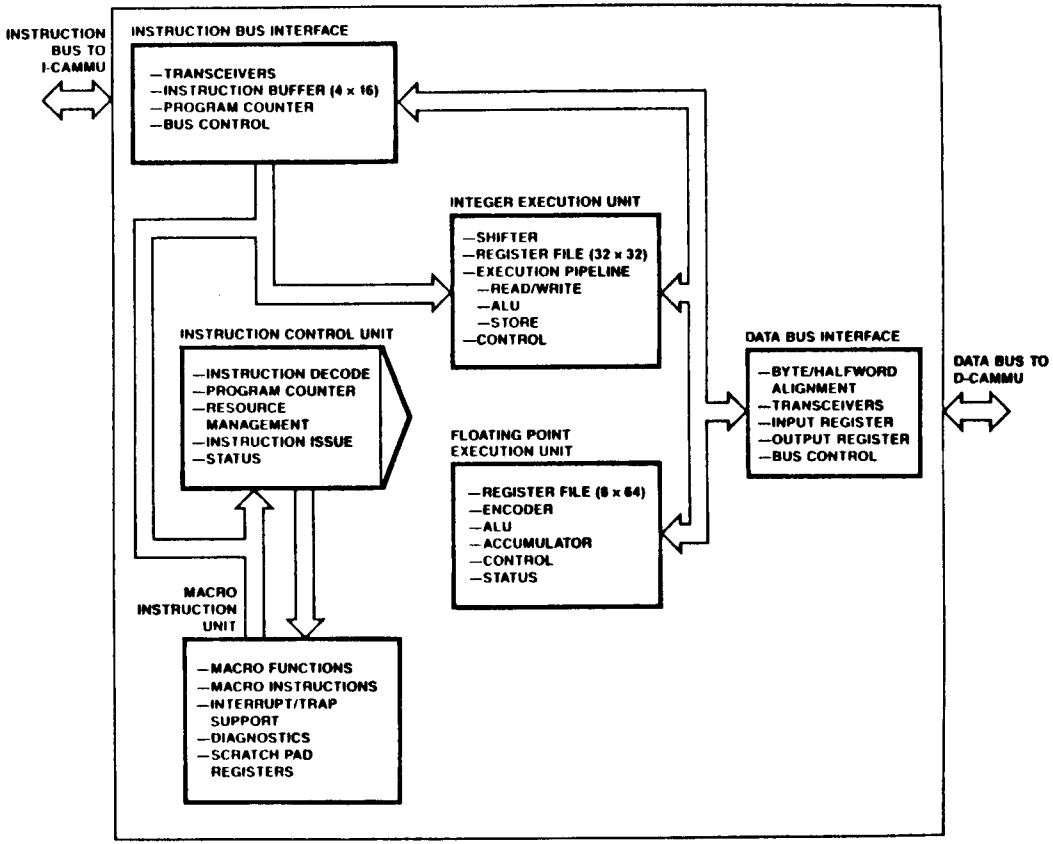


Figure 4. Processor Functional Units and Data Paths

It is possible to classify traps 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 exception is recognized by the CAMMU which maintains in the TLB copies of the protection bits taken from the page table entries.

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, and also exception flags in the PSW which are set when the corresponding events occur. There is an overall floating point trap enable flag (also in the PSW) which may be used to disable all floating point traps.

The *trace trap* causes a trap at the end of the current instruction. An 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 Fairchild port of Unix) [Neff86b] uses approximately 60 of them.

6.3. 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: (a) no traps are pending except the trace trap, (b) interrupts are enabled, (c) all instructions currently in the pipeline have completed, and (d) string instructions have either completed or have saved sufficient state to be able to restart. (Long string instructions will periodically test for pending interrupts, and if there are any, will 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.

7. Design Architecture

As explained earlier, the term *"design architecture"* refers to the architectural implementation at a fairly high level. We discuss the design architecture of the CLIPPER CPU in this section.

*CLIX is a trademark of Fairchild Semiconductor Corporation

Figure 4 shows the major components of the CLIPPER processor and the major interconnections in a simplified fashion. Somewhat more detail is shown in figure 5. As can be seen from those figures, the processor is divided into 6 principal 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*. We discuss each of these in this section. Table 2 shows the fraction of the chip area occupied by various processor sections; the remainder of the area (to the total of 100%) is occupied by empty space or other minor components.

7.1. Instruction Bus Interface

The instruction bus (described in more detail in [Cho86]) is a bi-directional 46-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. Instructions are prefetched into this buffer, and are then fed into the instruction control unit as needed. A branch never hits in this buffer, as 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 pulsed. While within a cache line, the ICAMMU can deliver 4 bytes every 2 CPU cycles (60ns), and the CPU can at its maximum rate, execute 2-bytes (one parcel, or one 2-byte instruction) every CPU cycle (30ns).

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

7.2. Macro Instruction Unit

The Macro Instruction ROM (MIROM) is an on-chip ROM (1K entries x 47 bits) which 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. 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 1 or figure 3.

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).

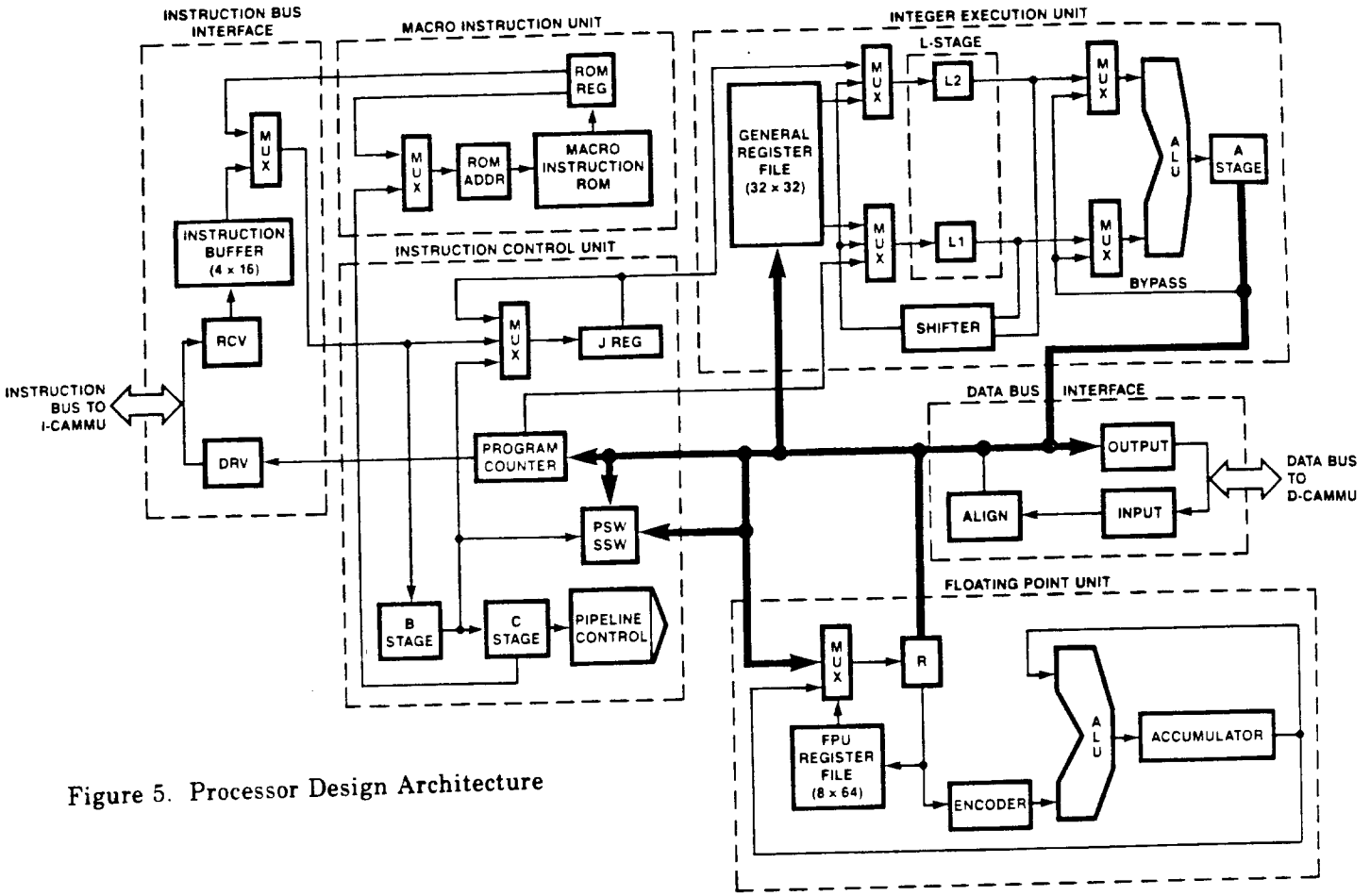
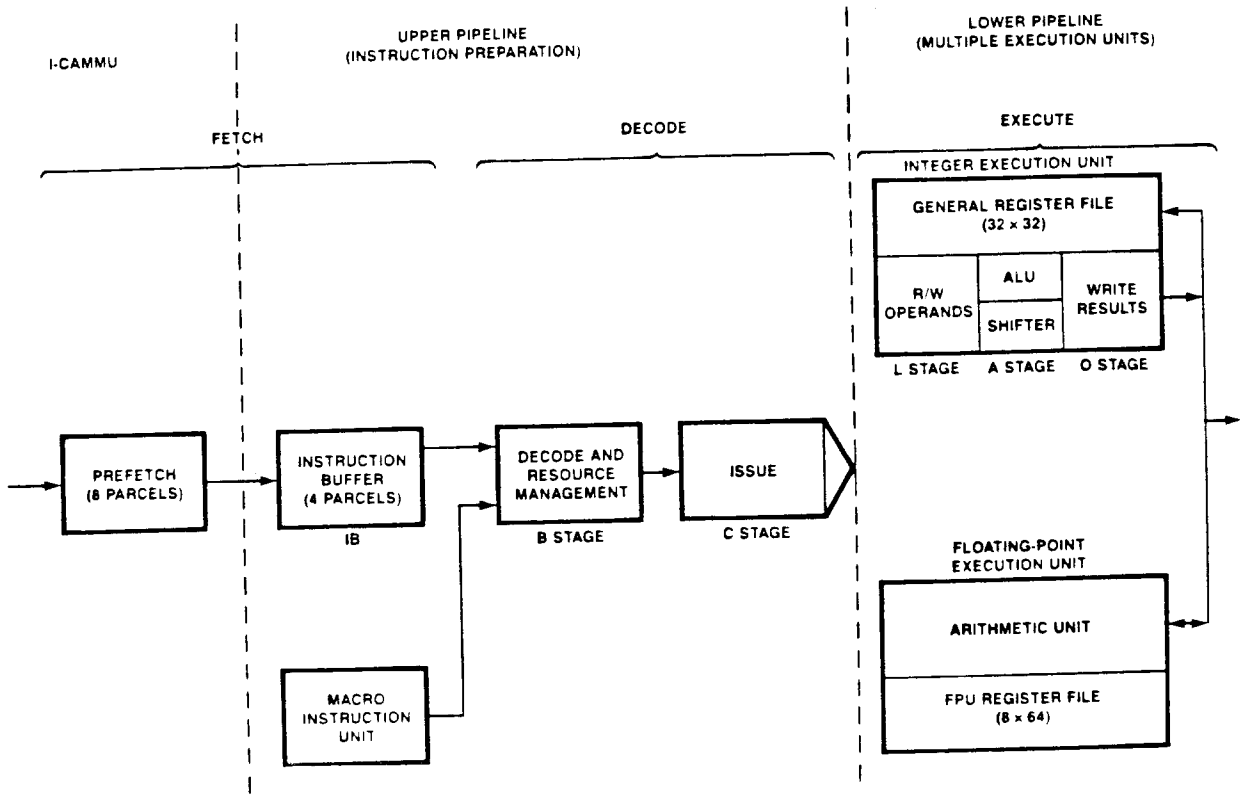


Figure 5. Processor Design Architecture

Figure 6. Pipeline Structure



7.3. 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 [Huck83] shows that for his System/370 workload, only 1.9% 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.

7.4. Floating Point Unit

CLIPPER is unusual among current microprocessors in having its floating point unit on chip. Multiplication uses a Booth algorithm [Cava84] which produces products iteratively, two bits per clock cycle. Typically, one clock time is needed for round and one for normalize. Division uses a nonrestoring shift and subtract algorithm, producing 1 bit per clock. Associated with the FPU is the floating point register file, which contains eight regular and four scratch 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 in execution at any one time, operations which neither use the FPU nor rely on its output can be issued steadily while the FPU completes the current operation. The result is that much of the execution time for floating point operations will overlap that of other instructions.

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 be quite far from the address of the trapping instruction.

7.5. 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, (shown in bold in figure 5) so that it can receive and deliver operands in the most expeditious manner.

7.6. 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 reader should also note figure 6, which diagrams the operation of the instruction execution pipeline.

In the ICU are 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. Shown in figure 6 are 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). Also located in the ICU are the PSW and SSW registers.

There can be one instruction in each of the B and C stages. Shown preceding the B stage 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.

7.7. Layout, Area, and Physical Parameters

A photograph of the CLIPPER CPU chip, on which functional areas are indicated, is shown in figure 7. Table 2 shows the fraction of the chip used for various purposes. The chip is implemented using 2-micron CMOS, with two levels of

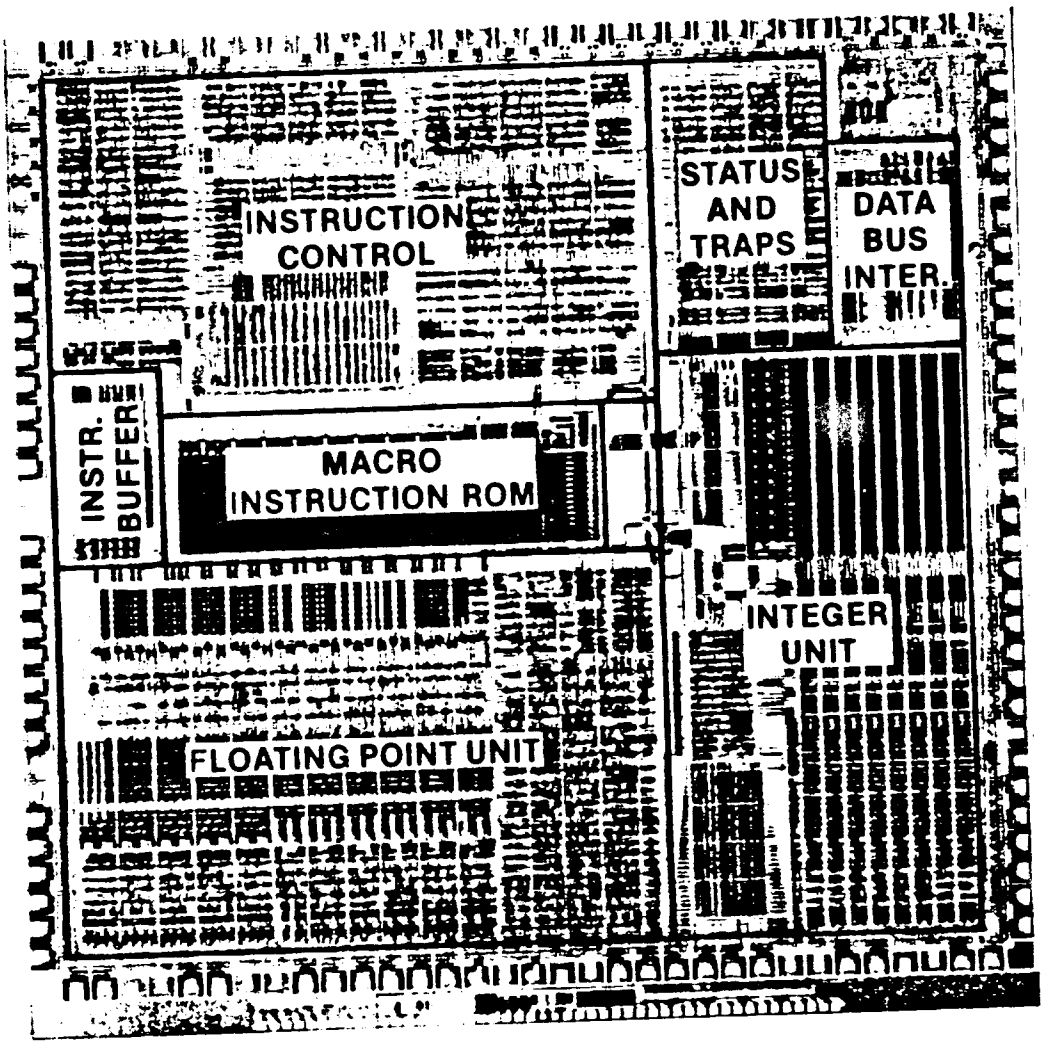


Figure 7. Photograph of CLIPPER Processor Chip, with functional units labeled.

metal interconnect with a 6.5 micron pitch, one polysilicon level with 2.0 micron gates and a 4.0 micron pitch, a 250 Angstroms thick gate oxide, and 2.0 micron contacts and vias. Transistor switching speeds range from .5ns to 3.0ns, depending on gate size and load. The chip dissipates 0.5 watts. The processor cycle time is 30ns, 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 pins. The overall chip size is 185,000 square mils; the package is .9 in. sq. and is surface mounted.

8. Performance

CLIPPER was conceived of and designed as a high performance processor, and as has been noted throughout this paper, design decisions and tradeoffs have been made whenever possible in the direction of higher performance. That high performance has indeed been achieved is evident from the instruction execution times shown in table 3. As can be seen, the minimum instruction execution time is one CPU cycle time, or 30 ns. The peak program execution rate is thus 33 MIPS.

Benchmark timings have been obtained both via an instruction set timing simulator and from runs on a real machine using early versions of the various compilers. The simulator shows an average of 5-6 clock cycles per instruction including memory delays, or about 5-7 MIPS (measured in CLIPPER instructions).

Because the power of various instruction sets varies greatly, simply quoting a MIPS figure is not very meaningful. For that reason, various standard benchmarks have been run on a real CLIPPER. The CLIPPER system used had a relatively slow memory (2 wait states), and the compilers used have been available from less than a year (C) to less than a month (Fortran), so a high performance CLIPPER "box", using mature software, should do *considerably* better than the results presented here. The compiler version under current development (but not yet released) shows 10% to 20% better performance on the existing hardware. All of the results presented below were run on production hardware (during January, 1987) at Fairchild, by the same people using the same code under the same conditions, and should be comparable and accurate.

Table 4 shows the results of the Dhrystone [Weic84], Whetstone [Curn76], Linpak [Dong83] and Berkeley [Hans82] benchmarks. It can be seen that CLIPPER is about 25% faster than the VAX 8600 (VMS 4.1) on the Dhrystone benchmark and is about nine times the speed of a VAX 11/780. (Versions 1.0 and 1.1 refer to the two versions of the Dhrystone benchmark.) For the Whetstone benchmark, CLIPPER is about 3 times the speed of the VAX 11/780 (with the floating point accelerator) and about 7.5 - 9.5 times as fast as the VAX 11/785 (without an FPA) under Ultrix. Table 4 shows that CLIPPER is about 3.15 times as fast as the VAX 11/780 (VMS 3.7) on single precision Linpack and is about 3.11

Processor Section	Fraction of Area
Floating Point Unit	.25
(Floating Point Control)	.067
Execution Unit	.187
(Register File)	.05
(ALU)	.053
ROM	.056
Program Counter	.013
S-Bus	.031
Instruction Buffer	.014
Branch Logic	.041
B-stage Control Logic	.074
C-stage (Execution) Logic	.083
Data Memory Interface	.026
Status Logic (PSW, SSW), Trap and Cond Codes)	.048

Table 2
Area Allocations for Functions

Instruction	Time (Clocks)	Time (ns)
Add Word	1	30
Logical	1	30
Move Word	1	30
Load Word / Pop	4-6	120-180
Store Word / Push	6	180
Branch (not taken)	4	120
Branch (taken)	7-9	210
Multiply Word	24	720
Floating Add Single	12 (ave)	360
Floating Add Double	14 (ave)	420
Floating Multiply Single	24 (ave)	720
Floating Multiply Double	69 (ave)	2070
Floating Divide Single	94 (ave)	2820
Floating Divide Double	182 (ave)	5460

Table 3
Execution Times for Common Instructions

Dhrystones (Dhrystones per second)

	CLIPPER CLIX V.3	VAX 8600 VMS 4.1	VAX 780 VMS 3.1-FPA	VAX 785 Ultrix-No FPA
Version 1.0				
regs	9140	7332	1025	1965
noregs	9140	7409	1012	1965
Version 1.1				
regs	8006	6465	908	1800
noregs	8006	6413	920	1800

Whetstones (Millions of Whetstones per second)

single	3.380	4.530	1.120	0.350
double	2.008	2.650	0.716	0.267

LINPAK (Megaflops per second)

CLIPPER (Fortran) (BLAS)	CLIPPER (coded) (BLAS)	VAX 8600 VMS	VAX 780 VMS FPA	VAX 785 Ultrix No FPA
single precision, arrays with leading dimension of 201				
.6242	.8078	.8692	.2008	.2030
.6242	.7923	.8583	.1979	.2020
.6338	.8078	.8583	.2014	.1981
.6348	.7969	.8637	.2032	.2013
double precision, arrays with leading dimension of 201				
.3815	.5024	.4905	.1244	.06509
.3780	.5024	.4976	.1215	.06478
.3780	.5024	.4976	.1239	.06519
.3787	.5049	.4908	.1232	.06498

Berkeley Benchmarks (Milliseconds per iteration)

	CLIPPER	VAX 8600	VAX 780	VAX 785
acker	358	1084	4650	2960
puzzle	1090	1125	4755	7420
search	0.14	0.263	0.504	0.787
sieve	33.2	39.1	117.6	172.8

Table 4
Benchmark Results

times as fast with double precision. (The VAX Linpack benchmarks were all run using the standard VMS Fortran library. The "Fortran BLAS" results are with the Fortran BLAS routines from Los Alamos. The "coded BLAS" shows the results after further hand tweaking.) The Berkeley benchmarks show CLIPPER from 3.5 to 13 times as fast as the VAX 11/780 under VMS 3.7. Using the VAX 11/780 as a canonical "1 MIPS" machine, CLIPPER is about a 5-6 MIPS machine. (In actual fact, the VAX has a CISC instruction set, and thus generally runs at about .5 MIPS [Emer84]. The "canonical 1 MIPS" refers to a System/370 scientific workload running on a System/370 instruction set machine.)

8.1. Performance vs. 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, in contrast to the "one size fits all" approach that is currently popular in some quarters. The design philosophy espoused by MIPS [Henn82] and RISC [Patt85] is that all, or almost all instructions must execute in one cycle; this implementation approach was previously used by Procrustes [Bull55] in matching guests to beds.

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 (30ns) was chosen as a design goal, on the basis that the technology available at the time of chip 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 they executed correctly in the presence of traps, interrupts, and data and register dependencies.

8.2. 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.

It is important to note that (for the most part) performance improvements in scaling from one technology (e.g. 2-micron CMOS) to another (e.g. 1.25-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 permits almost halving the cycle time. Scaling of chip technology to obtain a higher performance CLIPPER implementation is underway, even though CLIPPER already has an impressively fast cycle time.

In considering the performance of CLIPPER, evidence supports comments in [Mate84], where it is noted that the factor most strictly limiting performance on a high performance microprocessor is the memory interface. As is discussed in more detail in [Cho86, Holl87], CLIPPER is most strictly limited by memory delays, despite the use of two busses (one each for addresses and data), 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, and that 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. For example, a multiplier or adder can always be made faster with the addition of more gates; thus the current multiplication time (table 3) could be reduced. Similar improvements are possible in other multicycle instructions. For comparison, we note that the Amdahl 470V/6 required 5-6 cycles per instruction, and that was roughly halved for the 580*. The DEC VAX 11/780 needed about 10 cycles per instruction [Emer84] and that was reduced to about 6 cycles for the 8600 [Foss85]; the cycle time was only reduced from 200ns to 80ns, but the total performance was improved by a factor of almost five.

Projects to improve CLIPPER performance by both technology scaling and the addition of additional (design architecture) features are underway.

9. Tradeoffs and Extensions

9.1. Instruction Set Choice

9.1.1. Why Not "Pure RISC"?

As noted above, the current research trend in computer architecture is to design machines with extremely simple instruction sets. Despite some advantages to such an approach, our feeling is that the instruction set should be made as simple as possible, but no simpler. Very simple instruction sets impose burdens on the compiler and the assembly language programmer, and increase code volume and I-cache traffic and miss ratios. Single cycle instruction execution for (almost

*Personal Knowlege

all) instructions results in less than optimal instruction overlap. By restricting ourselves to a load / store architecture, but permitting variable length instructions with variable length execution time, we believe that we've created a design which is both functional and allows efficient implementation.

In particular, the CLIPPER microprocessor architecture was designed to include string instructions (implemented in the on-chip ROM), on-chip floating point, hardware support for the TLB, hardwired pipeline interlocks, and interrupt and trap sequences (in the MIROM). We believe that given the current and likely future state of the art, these represent a good tradeoff.

9.1.2. 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, we note that 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. Other issues are discussed below.

There are a number of instructions and features that were deliberately omitted from the CLIPPER ISA, and we comment specifically on some of them here.

A natural form of computation is memory to register, register to memory, or memory to memory, but such instructions are not provided. There are three reasons for this: (a) 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 the same. (b) There is usually little savings in execution time, since the same sequence of operations must occur. (c) There is considerable additional complexity, because of the problems of 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 of this problem 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 due to 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. [Peut77, Clar82]. 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 [Cho86] 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.

There is no need for a compatibility mode 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 difficulty of implementing it on a chip so tightly constrained with regard to area. Extended precision can be obtained currently with instruction sequences, and opcodes are available to implement extending precision in the hardwired instruction set at some future time.

9.1.3. 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, when technology advances sufficiently. For example, a delayed branch has the advantage of reducing the pipeline penalty due to successful branches. The problem with a delayed branch is that of saving the state, when a trap or interrupt occurs between the time the branch is selected (the delayed branch instruction) and the time that it takes effect (one or two instructions later). The existing CLIPPER chip simply doesn't have the space on it for the necessary logic to implement this correctly. In addition to the delayed branch, a delayed load and vector instructions are under consideration.

9.2. 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 RISC [Patt85] and MIPS [Chow86], where the compiler must reorganize code and insert noops as necessary. We chose to use hardware control deliberately, as we believe that: (a) it is an unreasonable burden to require that the compiler understand the pipeline and insert noops as necessary; (b) it is an unreasonable burden on the assembly language programmer

and/or code generator to require that he overcome the lack of hardware; (c) the implications of (a and (b are that without interlocks, code will tend to be "buggy"; (d) compilers and programs become implementation dependent; instead of just depending on the instruction set architecture, they depend on the precise features of the pipeline. Object code is thus not portable between different implementations of the same instruction set architecture.

9.3. 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 [Cho86]) but we briefly note some of the points. 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. In addition, there is the problem of virtual vs. real addressing, synonyms, cache flushing, and cache consistency [Smit82]. For a future design, a 2- or 3-level cache (on chip, CAMMU chip, cache board) is a possibility.

9.4. Address Space Size

An issue which has been emphasized throughout this paper is that of address space extensibility. 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 would require few if any user programs to undergo conversion. We expect that within 10 or 15 years, both physical and virtual addresses will need more than 32 bits.

9.5. Better Multiprocessor Cache Consistency

As explained in [Cho86], 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 only need take action when the reference is marked shared. Because consistency operations involve significant performance costs, 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 [Swea86].

10. Conclusions and Overview

In this paper, we've discussed the instruction set architecture and the implementation of the Fairchild CLIPPER microprocessor. The machine was designed from scratch to provide high performance, convenient programmability and the ability to extend the architecture as technology improves and the art of computer architecture design advances. Our discussion has included both functional description (concentrating on those functions that are interesting and/or unusual) and a significant consideration of design tradeoffs and choices. We believe that CLIPPER not only represents a good set of choices, but that this paper is important in discussing and documenting those tradeoffs.

Acknowledgements

We wish to thank the entire team responsible for designing and implementing CLIPPER. We especially note Vern Brethour, James Cho, Rich Dickson, John Kellum, Kevin Kissell, David Neff, Laura Neff, Kevin Norman, and Ray Ryan, who had major roles throughout the project.

Bibliography*

[Bull55] Thomas Bullfinch, "The Age of Fable", first published 1855. Mentor Edition, (New American Library, New York), 1962.

[Cava84] Joseph Cavanagh, "Digital Computer Arithmetic - Design and Implementation", McGraw-Hill, New York, 1984.

[Cho86] James Cho, Alan Jay Smith and Howard Sachs, "The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPER", March, 1986, UC Berkeley CS Division Technical Report UCB/CSD 86/289.

[Chow86] F. Chow, M. Himmelstein, E. Killian, L. Weber, "Engineering a RISC Compiler System", Proc. IEEE Compcon, March, 1986, San Francisco, Ca., pp. 132-137.

[Clar82] Douglas Clar and Henry Levy, "Measurement and Analysis of Instruction Use in the VAX-11/780", Proc. 9'th Ann. Symposium on Computer Architecture, April, 1982, Austin, Texas, pp. 9-17. (Also Computer Architecture News, 10, 3.)

[Cody84] W. J. Cody, J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson, "A Proposed Radix- and Word- length- Independent Standard for Floating Point Arithmetic", IEEE Micro, 4, 4, pp. 86-100, August, 1984.

[Colw85] Robert P. Colwell, Charles Y. Hitchcock III, E. Douglas Jensen, H. M. Brinkley Sprung, and Charles P. Kollar, "Computers, Complexity and Controversy," IEEE Computer, September,

*The various papers marked here as "in preparation" should be completed by the time this paper is finally published.

1985, pp. 8-19.

[Curn76] H. J. Curnow and B. A. Wichman, "A Synthetic Benchmark", Computer Journal, 19, 1, February, 1976, pp. 43-49.

[Demo86] M. DeMoney, J. Moore and J. Mashey, "Operating System Support on a RISC", Proc. IEEE Compcon, March, 1986, pp. 138-143. San Francisco, Ca.

[DEC81] Digital Equipment Corp., VAX Architecture Handbook, 1981.

[Dong83] J. J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment", Computer Architecture News, 11, 5, December, 1983.

[Emer84] Joel S. Emer and Douglas W. Clark, "A Characterization of Processor Performance in the VAX-11/780", Proc. 11'th Ann. Symp. on Computer Architecture, June, 1984, pp. 301-309.

[Fair86] Fairchild, "CLIPPER 32-bit Microprocessor User's Manual", 1986, Fairchild Semiconductor Corporation, Mt. View, CA.

[Foss85] Tryggve Fossum, James McElroy, and William English, "An Overview of the VAX 8600 System", Digital Technical Journal, 1, August, 1985, pp. 8-23.

[Groc86] Edward T. Grochowski, "An Instruction Tracer for the Motorola 68010", MS Project Report, Computer Science Division, EECS Dept., University of California, Berkeley, CA., May, 1986.

[Hans82] P. M. Hansen, Linto, Mayo, Murphy, and Patterson, "A Performance Evaluation of the Intel iAPX 432", Computer Architecture News, 10, 4, June, 1982, pp. 17-26.

[Henn82] John Hennessy, Norman Jouppi, Forest Baskett, Thomas Gross and John Gill, "Hardware/Software Tradeoffs for Increased Performance", Proc. Symp. on Architectural Support for Programming Languages and Operating Systems", (Sigarch Computer Architecture News, 10, 2, March, 1982), pp. 2-11.

[Henn84] John Hennessy, "VLSI Processor Architecture", IEEETC, C-23, 12, December, 1984, pp. 1221-1246.

[Holl87] Walt Hollingsworth, Alan Jay Smith and Howard Sachs, "Analysis of Processor Performance in the Fairchild CLIPPER", in preparation.

[Huck83] Jerome Huck, "Comparative Analysis of Computer Architectures", Computer Systems Laboratory Technical Report 83-243, May, 1983, Stanford University, Stanford, CA.

[Hunt84] Colin Hunter and Erin Farquhar, "Introduction to the NS16000 Architecture", IEEE MICRO, April, 1984, pp. 26-47.

[IBM76] IBM System/370 Principles of Operation, Form Number GA22-7000-5, IBM Corporation, Poughkeepsie, New York, 1976.

[Kirr83] Hubert Kirrmann, "Data Format and Bus Compatibility in Microprocessors", IEEE MICRO, 3, 4, August, 1983, pp. 32-47.

[Lamp83] Butler W. Lampson, "Hints for Computer System Designers", Proc. Ninth ACM Symp. on Operating Systems Principles", October, 1983, Bretton Woods, NH, pp. 33-48. Also Operating Systems Review, 17, 5.

[Levy80] Henry Levy and Richard Eckhouse, "Computer Programming and Architecture: The VAX-11", Digital Press, Bedford, Mass., 1980.

[Lund74] Amund Lunde, "Evaluation of Instruction Set Processor Architecture by Program Tracing", Technical Report, Dept. of Computer Science, Carnegie Mellon University, July, 1974.

[Mate84] Richard Mateosian, "System Considerations in the NS32032 Design," Proc. NCC, 1984, pp. 77-81.

[Moto82] Motorola Corporation, "16-Bit Microprocessor User's Manual", 3'rd Edition, 1982.

[Mous86] John Moussouris, Les Crudele, Dan Freitas, Craig Hansen, Ed Hudson, Steve Przybylski, Tom Riordan and Chris Rowen, "A CMOS RISC Processor with Integrated System Functions", Proc. IEEE Compcon, March, 1986, pp. 126-131.

[Neff86a] David Neff, "C Compiler Implementation Issues on the CLIPPER Microprocessor", Proc. Compcon, March, 1986, San Francisco, Ca., pp. 196-201.

[Neff86b] Laura Neff, "CLIPPER Microprocessor Architecture Overview", Proc. Compcon, March, 1986, San Francisco, Ca., pp. 191-195.

[Neuh80] Charles J. Neuhauser, "Analysis of the PDP-11 Instruction Stream", Technical Rpt. 183, Computer Systems Laboratory, Stanford Electronics Laboratories, Stanford University, Stanford, Ca. 94305., February, 1980.

[Patt81] David A. Patterson and Carlo H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer", Proc. 8'th Ann. Symp. on Computer Architecture, 1981, pp. 443-457.

[Patt85] David Patterson, "Reduced Instruction Set Computers", CACM, 28, 1, January, 1985, 8-21.

[Peut77] Bernard Peuto and Leonard Shustek, "An Instruction Timing Model of CPU Performance", Proc. 4'th Ann. Symp. on Computer Arch., College Park, MD, March, 1977, pp. 165-178.

[Radi82] George Radin, "The 801 Minicomputer", Proc. Sigarch/Sigplan Symp. on Arch. Support for Prog. Lang. and Op. Sys., March, 1982, Palo Alto, Ca. pp. 39-47.

[Ritc74] Dennis M. Ritchie and Ken Thompson, "The UNIX Timesharing System", CACM, 17, 7, July, 1974, pp. 365-375.

[Smit81] Alan Jay Smith, "Input/Output Optimization and Disk Architecture: A Survey", Performance Evaluation, 1, 2, 1981, pp. 104-117.

[Smit82] Alan Jay Smith, "Cache Memories", Computing Surveys, 14, 3, September, 1982, pp. 473-530.

[Smit85] Alan Jay Smith, "Problems, Directions and Issues in Memory Hierarchies", Proc. 18'th Annual Hawaii International Conference on System Sciences, January 2-4, 1985, Honolulu, Hawaii, pp. 468-476. Also available as UC Berkeley CS Report UCB/CSD84/220.

[Swea86] Paul Sweazey and Alan Jay Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus", Proc. 13'th Ann. Int. Symp. on Computer Arch., Tokyo, Japan, June, 1986, pp. 414-423.

[Wiec82] Cheryl A. Wiecek, "A Case Study of VAX-11 Instruction Set Usage For Compiler Construction", Proc. Symp. on Arch. Support for Programming Languages and Operating Systems, March, 1982, Palo Alto, Ca., Computer Architecture News 10, 2, March, 1982, pp. 177-184.