Documentation for the CHIP Computer System (Version 1.1)*

Özalp Babaoğlu
Mimi Bussan
Rogerio Drummond
Fred B. Schneider

TR 83-584
December 1983
(Revised August 1986)

Department of Computer Science
Cornell University
Ithaca, NY 14853

* Partial support for this work was provided by the National Science Foundation under Grants No. MCS82-10356 and MCS81-03605 and the Government of Brazil through a Fellowship to the third author.

Documentation for the CHIP Computer System (Version 1.1) *

Özalp Babaoğlu
Mimi Bussan
Rogerio Drummond
Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

ABSTRACT

CHIP (Cornell Hypothetical Instructional Processor) is a computer system that was designed as an educational tool for teaching undergraduate courses in operating systems and machine architecture. This document constitutes the sole reference manual for the CHIP computer system. A simulator for this hypothetical system exists under the UNIX† operating system. The CHIP architecture includes dynamic memory mapping suitable for implementing virtual memory, eight interrupt priority levels, memory-mapped input/output and two modes of processor operation. The central processor of CHIP is compatible with the PDP‡-11 at the user-mode instruction level. Therefore, any non-privileged code written for the PDP-11 can be executed on CHIP. Several new user and kernel-mode instructions have been added to CHIP for increased efficiency. The CHIP simulator also supports input/output devices such as terminals, drums, disks and printers. All interactions with CHIP take place through an *operator's console* being simulated on a terminal. Users can examine/alter memory locations, set breakpoints, detect the referencing of specified memory locations, start/stop execution, etc. through a *console command language*. Program global variables and functions can be referred to by symbolic name with the mapping to absolute addresses being performed automatically by the system. The software support environment for CHIP includes a C compiler, assembler and loader.

August 19, 1986

---

* Partial support for this work was provided by the National Science Foundation under Grants No. MCS82-10356 and MCS81-03605 and the Government of Brazil through a Fellowship to the third author.

† UNIX is a Trademark of AT&T Bell Laboratories.

‡ PDP is a Trademark of Digital Equipment Corporation.

Copyright © 1983 by Özalp Babaoğlu

Chapter One
Principles of Operation

1. Introduction

CHIP (Cornell Hypothetical Instructional Processor) is based on the architecture of the DEC PDP-11 and the IBM 370. Most of the instructions are borrowed from the DEC machine, while the interrupt architecture and memory mapping mechanism have their origins in the IBM architecture. The new machine supports dynamic memory mapping, eight interrupt priorities, memory-mapped I/O devices and two modes of operation: KERNEL and USER.

The CHIP system has been implemented on a VAX under UNIX. Also simulated is a subset of the I/O devices that are to be available with the CHIP system.

This document, together with a processor handbook for any model of the PDP-11 family, forms a complete description of the CHIP machine.

Throughout this manual the following notational conventions are used:

- Words being defined are *italicized.*
- Field F of register R is denoted by R.F.
- Bits of storage units are numbered right-to-left starting with 0.
- The i-th bit of a storage unit named N is denoted by N[i].
- The contents of a storage unit named N is denoted by (N).
- Memory addresses and operation codes are given in octal.

2. Major Components

As shown in Figure 1, the CPU and I/O devices are connected through main memory. Device controllers, which act as a communication channel between devices and memory, transfer data directly to or from memory without CPU intervention. The CPU is interrupted by a device only upon the completion of an I/O operation.

3. Machine Registers

This section describes the registers internal to the CPU. Device registers, which are used to control I/O devices, are discussed in §8. Each CPU register can hold a 16-bit quantity called a *word.*

The following registers form the *Processor State*: R0, R1, R2, R3, R4, R5, SP, PC, PS1, PS2, STA and STL. They can be loaded and stored as a

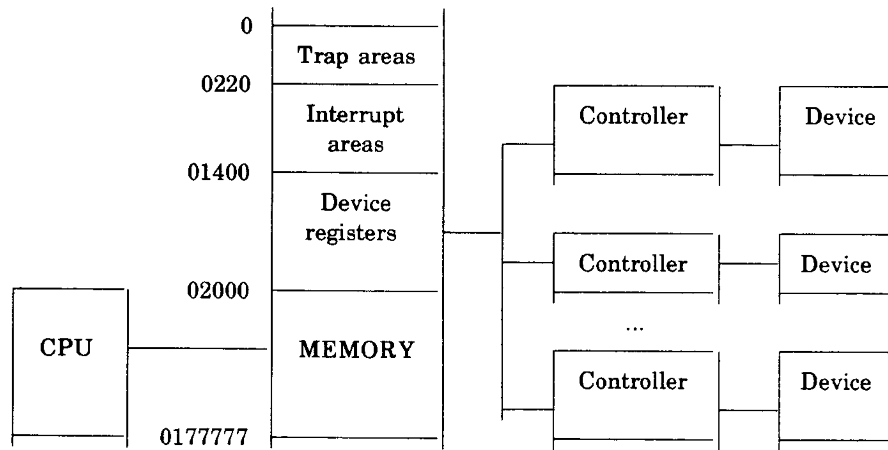
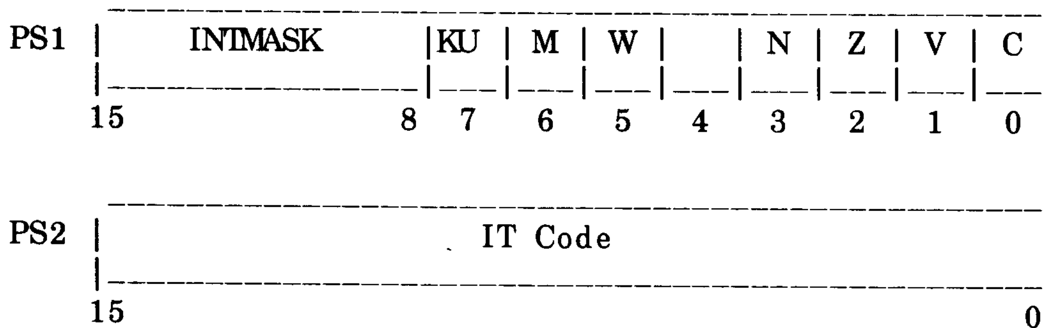


Figure 1. *Major components of CHIP*

block of twelve words in the order in which they are listed above by using the LDST (load state) and STST (store state) instructions, respectively. Registers R0-R5 are available for general purpose intermediate data storage. Register SP is the Stack Pointer and PC is the Program Counter. As in the PDP-11, the stack of CHIP grows from high towards lower memory addresses.

The processor status registers in CHIP, PS1 and PS2, are shown below:



FIELD	FUNCTION
INTMASK	if PS1.INTMASK[i]=1, the i-th priority level interrupts are disabled
KU	if 1, the processor is in KERNEL mode; USER mode otherwise
M	if 1, memory Mapping is enabled
W	if 1, the CPU ceases to execute until an interrupt occurs
N	set to 1 if last instruction executed yielded a Negative number; 0 otherwise
Z	set to 1 if last instruction executed yielded a Zero result; 0 otherwise
V	set to 1 if last instruction executed caused arithmetic overflow; 0 otherwise
C	set to 1 if last instruction executed caused a Carry; 0 otherwise
IT Code	identifies the cause of the last Interrupt or Trap

Table 1. *Processor status register fields*

The setting of the condition codes N, Z, V and C are identical to the respective codes in the PDP-11 and the reader is referred to the *PDP-11 Processor Handbook* for a detailed description.

The STA (Segment Table Address) register contains the physical address of the Segment Table currently in use. It is used only when memory mapping is in effect, i.e., PS1.M=1 (see §6 for details).

The STL (STack Limit) register contains the lowest address to which the stack can extend before causing a trap (see §7.1 for details).

TDCK (Time of Day ClocK) is a double-word register containing the elapsed time, in microseconds, since power-up. Its value can be accessed by executing the STCK (Store Time of day ClocK) instruction.

IT (Interval Timer) is also a double-word register. It is decremented by one each microsecond. When the value reaches zero, a Clock interrupt occurs. Register IT can be set by the LDIT (LoaD Interval Timer) instruction.

4. Modes of Operation

The CHIP processor can operate in two modes: USER and KERNEL. The mode of operation is determined by the setting of the PS1.KU bit. To change the processor mode, a new processor state must be loaded. This can

be done by executing LDST or as the result of a trap or interrupt.

In USER mode, execution of Privileged Instructions is illegal and causes a program trap. All instructions are executable in KERNEL mode.

4.1. USER Mode

Most of the USER-mode instructions are as in the PDP-11. However, several new ones have been implemented, and these are described below. The definitive document for the semantics of the PDP-11 instructions that have been incorporated into CHIP is the book *The Design and Analysis of Instruction Set Processors*, M. Barbacci and D. Sieworek, McGraw-Hill, 1982, which over rides any discrepancies that may arise between it and a PDP-11 processor handbook. For completeness, all of the CHIP instructions are listed in the Appendix. Note that the floating point instructions of the PDP-11 are not implemented in CHIP. In the following descriptions, the "Usage" field indicates how the instructions can be invoked through C program statements to be compiled with the **pcc** compiler (see Chapter 2 for details).

4.2. New USER Instructions

STCK STore time of day ClocK

Opcode: 107100

Operation: addr ← (SP)
(addr) ← most significant word of TDCK
(addr+2) ← least significant word of TDCK

Description: The contents of TDCK are stored in the two consecutive words pointed to by the argument on the top of the stack.

Usage: **long** time;
STCK(&time);

SYS Cause a system call trap

Opcode: 104400 to 104777

Operation: PS2 ← system call number
(000140).Old ← current processor state
processor state registers ← (000140).New

Description: Executing SYS causes a system call trap. First, PS2 is loaded with the number of the system call (bits 0-7 of the opcode). Then, the current processor state is saved in the system call trap old area and a new processor state is loaded from the system call trap new area.

Usage: SYS0(); SYS1(); SYS2(); ... SYS255();

INPRG Initialize program

Opcode: 107200

Operation: tmp ← (SP)
(SP) ← PC
PC ← tmp

Description: A restricted form of the "JSR" instruction. Usually, used only during program start up to invoke the entry point *main*.

Usage: main();
INPRG(main);

CSV Function prologue

Opcode: 107300

Operation: ↓(SP) ← R5
R5 ← SP
↓(SP) ← R4
↓(SP) ← R3
↓(SP) ← R2

Description: The contents of the general purpose registers R2-R5 are saved on the stack and R5 is established as the frame pointer. This is consistent with the C function calling convention.

Usage: CSV();

CRET Function epilogue

Opcode: 107400

Operation: SP ← R5−6
R2 ← (SP)↑
R3 ← (SP)↑
R4 ← (SP)↑
R5 ← (SP)↑
PC ← (SP)↑

Description: The registers saved by CSV are restored and control is returned to the caller.

Usage: CRET();

MOVBCK Move block of bytes
Opcode: 107000
Operation: src ← (SP)
dst ← (SP)+2
size ← (SP)+4
(dst+i) ← (src+i) for i=0, 1, 2, ..., size-1
Description: A block of *size* bytes is moved from *src* to *dst*. The maximum block size that can be moved is 512 bytes.
Usage:
```
int size;
char *src, *dst;
MOVBCK(src, dst, size);
```

4.3. KERNEL Mode

In KERNEL mode, privileged instructions as well as USER mode instructions are executable. The privileged instructions are described here.

LDST LoaD processor STate
Opcode: 007000
Operation: addr ← (SP)
processor state registers ← (addr+i), i=0,2,...,22
Description: The processor state registers are loaded from the state stored in the 12 consecutive words pointed to by the argument on the top of the stack. Loading takes place in the following order: R0-R5, SP, PC, PS1, PS2, STA and STL.
Usage:
```
state_t newstate;
LDST(&newstate);
```

STST STore processor STate
Opcode: 007100
Operation: addr ← (SP)
(addr+i) ← current processor state, i=0,2,...,22
Description: The current processor state is stored in the 12 consecutive words pointed to by the argument on the top of stack. Storing takes place in the following order: R0-R5, SP, PC, PS1, PS2, STA and STL.
Usage:
```
state_t savearea;
STST(&savearea);
```

LDIT LoaD Interval Timer
Opcode: 007200
Operation: addr ← (SP)
most significant word of IT ← (addr)
least significant word of IT ← (addr+2)
Description: Register IT is loaded with the contents of the double word pointed to by the argument on the top of stack.
Usage: **long** interval;
LDIT(&interval);

LDIM LoaD Interrupt Mask
Opcode: 007300
Operation: PS1.INTMASK ← byte(SP)
Description: PS1.INTMASK is loaded with the least significant byte of the argument on the top of stack.
Usage: **char** mask;
LDIM(mask);

LDSTL LoaD STack Limit register
Opcode: 007400
Operation: STL ← (SP)
Description: The STL register is loaded with the contents of the top of the stack.
Usage: **int** limit;
LDSTL(limit);

HALT Halt
Opcode: 000000
Description: After execution of the HALT instruction, the CPU has stopped executing, the PC is pointing to the instruction following the HALT, and the console has been given control.
Usage: HALT();

5. Memory

The CHIP memory is byte addressable and may contain up to 64K bytes. A *word* in CHIP memory is two bytes long and is divided into a *high byte* and a *low byte*. The low bytes of words are stored at even-numbered locations and the high bytes at odd-numbered locations. An even address can denote a word or a byte, depending upon the context. An even address is said to be *word aligned*, since it refers to the low byte of a word.

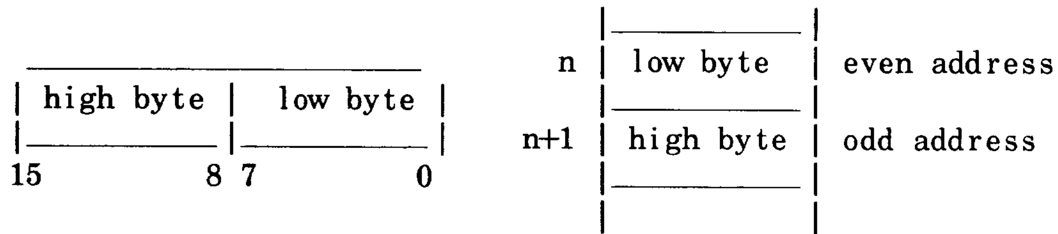


Figure 2. *A CHIP word and its representation in memory*

Instructions in memory must be word aligned. An attempt to execute an instruction starting at an odd address causes a Program Trap. Addresses of operands must also be word aligned, except for byte operands in byte instructions.

Certain memory locations are reserved for special purposes. The first 1024 bytes of physical memory are used for the Interrupt and Trap Areas as described in §7 and for the device registers as described in §8.

6. The Memory Management System

A CHIP address can be interpreted either as a physical address or as a virtual address. An address is interpreted as a physical address whenever the memory mapping mechanism is off (i.e. PS1.M=0) and interpreted as a virtual address otherwise (i.e. PS1.M=1). In CHIP, both the physical and virtual spaces are 64K bytes long with addresses ranging from 0 to 177777. The virtual address space is divided into 512-byte units called *pages* and the physical address space is divided into 512-byte units called *page frames*.

When memory mapping is in effect, a virtual address is automatically mapped into a physical address. The Segment Table Address register, STA, contains the physical address of a *Segment Table*, which is composed of four entries. Each entry is two words long and is called a *Segment Descriptor* (SD).

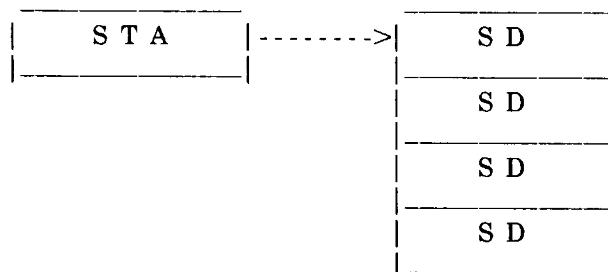
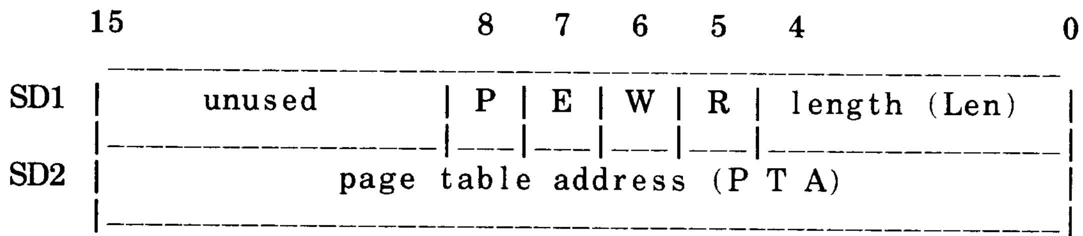


Figure 3. *Segment table*

A segment descriptor has the following format:



FIELD	MEANING
Len	Maximum valid page number within segment (one less than actual segment size)
E,W,R	Execute, Write and Read access rights for the segment, respectively; if 1, the corresponding access right is permitted
P	Presence bit; if 0, a missing segment trap is generated
STA	physical address of the corresponding page table

Table 2. *Segment descriptor fields*

A *page table* can have up to 32 entries. One less than its actual size is given by the SD1.Len field as defined above. Each entry in a page table is one word long and is called a *page descriptor* (*PD*).

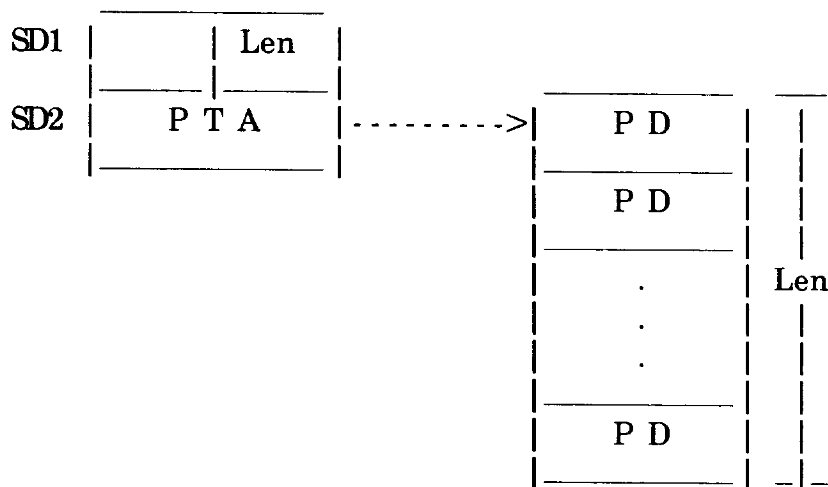
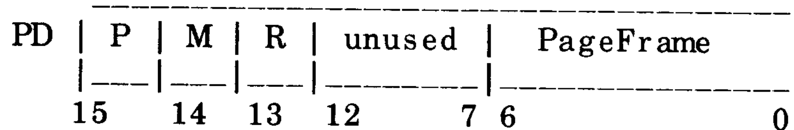


Figure 4. *Page table*

The format of a page descriptor is given below:



FIELD	MEANING
P	Presence bit; if 0, a missing page trap is generated
M	Modified bit; set to 1 by the processor whenever the page is modified
R	Reference bit; set to 1 whenever the page is read, written or executed
PageFrame	if the Presence bit is set, contains the number of the page frame to which this page is mapped; otherwise undefined

Table 3. *Page descriptor fields*

A page in CHIP must be aligned at an address that is a multiple of 512 bytes. Consequently, the real address of any page is a 16-bit number with the least significant nine bits equal zero. Therefore, the seven bits of PD.PageFrame are sufficient to uniquely identify a page frame.

6.1. Address Mapping

A *virtual address* in CHIP is composed of three fields:

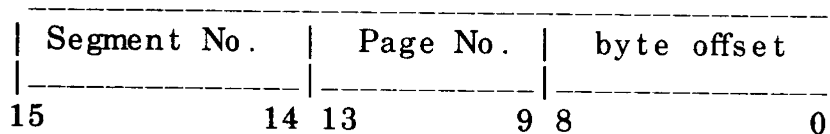


Figure 5. *Virtual address format*

Given a virtual address *va*, the corresponding physical address is calculated as follows:

1. The contents of STA (Segment Table Address register) is added to *va*.Seg, resulting in the physical address of a segment descriptor (*sd*) in the segment table.

$$sd := (\text{STA} + va.\text{Seg})$$

Step	Condition	Condition Name
2	*sd*.P=0	Segment Missing
2	*va*.Page>*sd*.Len	Invalid Page number
2	A read, write or execute access is attempted when the corresponding *sd*.R, *sd*.W or *sd*.E is equal to 0.	Access Protection Violation
3	*pd*.P=0	Page Missing

Table 4. *Sources for Memory Management traps during memory mapping*

2. If the page table for the segment *sd* is marked as present, i.e. *sd*.P=1, and if *va* is a valid page number, i.e. *va*.Page ≤ *sd*.Len, and the current access mode does not violate the access rights of the segment, then *sd*.PTA is added to *va*.Page, resulting in the physical address of a page descriptor (*pd*). Otherwise, a Memory Management Trap occurs.

$$pd := (sd.\text{PTA} + va.\text{Page})$$

3. If the page is marked as present, i.e. *pd*.P=1, the physical address is calculated by adding *pd*.PageFrame*512 to *va*.offset. If the page is not present, a Memory Management Trap occurs.

$$pa := pd.\text{PageFrame}*512 + va.\text{offset}$$

The Memory Management Traps that might occur during the mapping are summarized in Table 4 in the order in which the conditions are tested.

The mapping of a virtual address *va* into a physical address *pa* can be expressed as:

$$pa := ((\text{STA} + va.\text{Seg}).\text{PTA} + va.\text{Page}).\text{PageFrame}*512 + va.\text{offset}$$

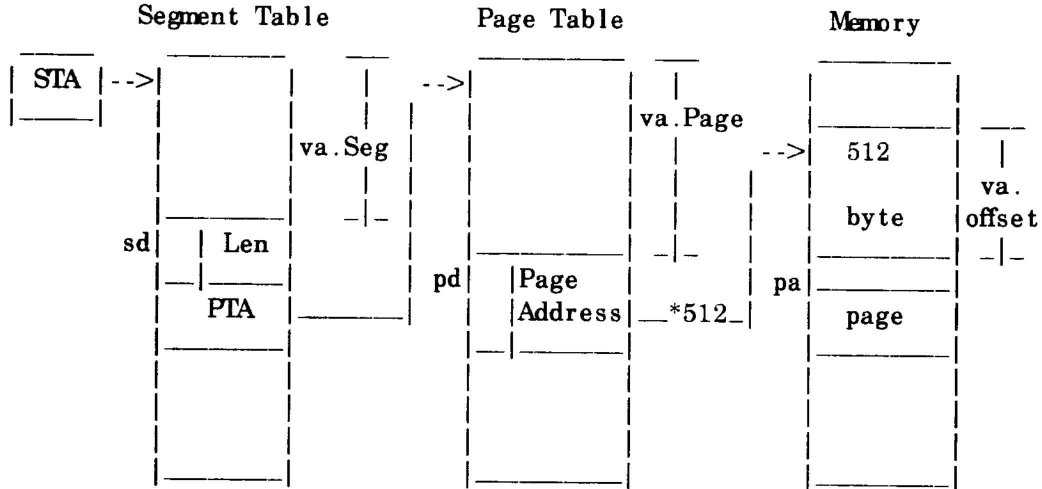


Figure 6. *Memory mapping*

7. The Interrupt/Trap System

The sequence of instructions being executed by the CPU can be altered by two types of events. *Traps* are generated by the processor as the result of instruction execution. Examples include Illegal Operation Code, Stack Overflow, Zero Divide and Page Fault. *Interrupts* are due to sources external to the processor, such as I/O devices.

7.1. Traps

Traps are grouped into three categories:

Program
Memory Management (MM)
System Call (SYS)

MM traps are generated by the memory management system as described in §6.1. SYS traps are generated by software. They occur as a consequence of executing a SYS instruction (opcodes from 104400 to 104777). *Program* traps are caused by encountering abnormal conditions during instruction execution.

MM and Program traps always occur *during* execution of an instruction, as opposed to SYS traps, which occur as a *result* of executing a SYS instruction.

SYS instructions can provide a way for a user to communicate with and request services from an operating system. For example, in order to write to

Trap Category	Trap Area Addresses	Code loaded in PS2[15..8]	PS2[7..0]	Condition
Program	000000	0	0	Illegal Opcode
			1	Un-supported Instruction
			2	Privileged Instruction
			3	Ill Instruction
			4	Odd Address
			5	Non-page Alignment
			6	Non-existent Memory
			7	Stack Limit Yellow
			8	Stack Limit Red
			9	Zero Divide
Memory Management	000060	0	segment and page number of virtual address	Access Protection Violated
		1		Page Missing
		2		Invalid Page Number
		3		Segment Missing
SYS	000140	0	System call number	Operation Codes from 104400 to 104777

Table 5. *Trap vector locations and condition codes*

a disk device a user may execute a particular SYS instruction.

Since the MM and Program traps may occur anywhere during instruction execution, there must be a mechanism for restoring the state of the processor and memory to the values they had before the instruction was executed after the trap is handled. Fortunately, CHIP does this automatically, ensuring that when the trap is signaled, the state of the machine is as it was before the offending instruction was executed.

For each of these three categories of traps, there is a reserved *trap area* in a predefined location in memory. These areas are 24 words long and logically subdivided into two consecutive regions of 12 words each. The first of these regions is called *Old* and the second *New*. Both of them contain a processor state.

When a trap occurs, the CPU services the trap by performing the following sequence of operations:

1. PS2.IT Code is loaded with a code identifying the trap as shown in Table 5.
2. The current processor state is stored into the Old region of the corresponding trap area.
3. A new processor state is loaded from the New region of the corresponding trap area.

The following is a brief description of causes for the various Program traps:

Illegal Operation Code: The operation code of the current instruction is not legal.

Unsupported Instruction: The current instruction is a valid PDP-11 instruction but is not supported in CHIP.

Privileged Instruction: The current instruction is a privileged instruction but the processor is running in USER mode.

Ill Instruction: The current instruction is ill-formed. The following can provoke this trap:

- JMP or JSR with *register* mode addressing
- DIV with an odd register number
- MOVBCK with block size greater than page size

Odd Address: The address of the current instruction, i.e. the contents of PC, is an odd number or the current instruction refers to an odd location when a word address is required.

Non-page Alignment: The current instruction refers to a location whose address is not on a page boundary (not a multiple of 512) where this is required.

Non-existent Memory: A physical memory address beyond the available physical memory is encountered.

Stack Limit Yellow: The top of the current stack is within 16 bytes of overflow. More precisely, STL+4 < SP ≤ STL+16. The offending instruction is not backed up since this trap occurs at the end of the instruction cycle.

Stack Limit Red: Execution of the current instruction would make SP ≤ STL + 4. Thus, either the current stack will exceed the stack limit or will be within two words of this limit.

Zero Divide: The current instruction is an arithmetic division in which the denominator has value zero.

7.2. Interrupts

An interrupt is signaled with an *interrupt request*. There are eight interrupt priority levels (numbered 0 through 7), each of which can be enabled or disabled by using the LDIM instruction to modify the interrupt mask field of PS1. Each device is associated with one of these levels at system configuration time. Zero or more devices can be associated with the same priority level. Among the unmasked interrupt requests, the one with the highest priority is serviced first.

For each priority level, there is an *interrupt area* starting at a predefined location in memory. These areas are similar to the trap areas presented in the previous section.

Priority	Interrupt Area Address	Interrupt Type	Code loaded in PS2[15-6]	PS2[5-3]	PS2[2-0]
0	000220	Terminal	0	0	Terminal No.
1	000300	Printer	0	1	Printer No.
2	000360	Disk	0	2	Disk No.
3	000440	Drum	0	3	Drum No.
4	000520	Not Used			
5	000600	Clock	0	5	0

Table 6. *Interrupt vector locations and interrupt codes*

In CHIP the five distinct sources of interrupts are:

Clock
Drum
Disk
Printer
Terminal

Table 6 shows their priority assignments and their effects on PS2.IT Code.

8. Input/Output Devices

The following devices are supported by CHIP:

Terminals (at most 5)
Printers (at most 2)
Disks (at most 4)
Drums (at most 4)

Associated with each device are several *Device Registers*. These registers reside in the second half of the second physical page of memory and can be accessed as ordinary memory words. The starting address of the device registers for each device is shown in Table 7.

In order to request an operation from a device, its Operation register is loaded with a code for the desired operation. This triggers the device to perform the specified operation. In carrying out the operation, the device can read and/or write to memory without interrupting the CPU. When the device finishes performing an operation, it places status information into the Status register and makes an interrupt request. Interpretation of the

Device	Starting Address	Range of i
Terminal(i)	01400 + 020*i	$0 \le i \le 4$
Printer(i)	01520 + 020*i	$0 \le i \le 1$
Disk(i)	01560 + 020*i	$0 \le i \le 3$
Drum(i)	01660 + 020*i	$0 \le i \le 3$

Table 7. *Device register locations*

various completion codes are given in Table 8.

If the Operation register is loaded while an operation is already in progress, the corresponding device behaves as if the first operation was never requested and starts the new request as described above. Other registers of the device can be read or loaded without affecting the device's current operation.

If a request is made to a device that does not exist in the current configuration, the request fails with the "Non-Existent Device" status code.

8.1. Disk

Each disk device has *UPTRACK*+1† tracks (numbered 0 to *UPTRACK*) and each track has *UPSECTOR*+1 sectors (numbered 0 to *UPSECTOR*) of 512 bytes. Disks can be read/written on a per sector basis. Each Disk has four registers. They are shown in Table 9 in order of increasing address.

The Status register is set upon completion of a Disk operation; the other registers are set by the user. Operations performed by the Disk have the codes as indicated in Table 10.

Condition	Relevant Devices	Code
Successful Completion	All	0
Hardware Failure	All	1
Invalid Operation	All	2
Invalid Buffer	All	3
Invalid Length	Printer, Terminal	4
Invalid Track No.	Disk	5
Invalid Sector No.	Disk, Drum	6
End of Input	Terminal	7
Non-Existent Device	All	8
Device Not Ready	Printer,Terminal	9

Table 8. *Status register codes*

† System configuration parameters such as the number of devices of each type and device capacities are defined by constants within the simulator and have the same names as given here. To change any one of them, it suffices to edit the configuration file and recompile the system.

Operation
Disk Address
Buffer Address
Status

Table 9. *Disk device registers positions*

Operation	Code
Disk Read	0
Disk Write	1
Disk Seek	2

Table 10. *Disk device operation codes*

A *Disk Seek* moves the disk head to the track number specified in the Disk Address register. A *Disk Read* operation reads a sector of the current track and stores it in the 512-byte buffer whose first location address is given in the Buffer Address register. Similarly, a *Disk Write* operation writes the contents of the 512-byte buffer addressed by the Buffer Address register onto a sector of the current track. For a Disk Read or Write, the sector number is always taken from the Disk Address register. The Buffer Address register always contains a *physical* address.

8.2. Drum

Drums are mass storage devices with a read/write head for each track. Each Drum has *UPDRUMSEC*+1 512-byte sectors (numbered 0 to *UPDRUMSEC*) and is read/written on a per sector basis. Drums have the same registers as Disks, but only read and write operations (with the same codes as the Disk) are allowed. The status register is loaded with the same codes as for the Disk except for the lack of the "Invalid Track Number" code.

8.3. Printer

Four device registers are associated with printers:

Operation
Length
Buffer Address
Status

Table 11. *Printer device registers*

The printer performs only one operation—*Print Line* (operation code 1)—which prints the string of bytes (up to *UPAMOUNT* bytes in length) pointed to by the Buffer Address register. A 'New Line' character is appended to the string. An attempt to write more than *UPAMOUNT* bytes in a *single* operation results in the "Invalid Length" completion code to be returned without performing any I/O. All output written to printer number *i* is collected in the file named *printer*i in the current directory. That is, output for printer number 0 will be in the file *printer0*, printer number 1 in the file *printer1*, etc. If the appropriate file cannot be created in the current directory (due to lack of permission), the operation terminates with the "Device Not Ready" status code.

The Length register contains the number of bytes, starting at the Buffer Address, to be printed. The device completion Status codes are as given in Table 8.

8.4. Terminal

Terminals can be written to or read from one line at a time. Each terminal has the same registers as a Printer. Terminal Status register codes are as given in the Table 8. A *Terminal Write* operation (code 1) is performed in exactly the same way as a Printer Print operation. All output written to terminal number *i* is collected in the file named *termout*i in the current directory. As with the printers, if the appropriate file cannot be created in the current directory, the operation terminates with the "Device Not Ready" status code.

A *Terminal Read* operation (code 0) reads a string of bytes (from the file named *termin*i in the current directory) up to a 'New Line' character (octal 012) or *UPAMOUNT* bytes, whichever occurs first, and stores them in the buffer pointed to by the Buffer Address register. The 'New Line' character is not considered to be part of the string. The actual number of bytes read is

stored in the Length register and the "Successful Completion" code is returned. Subsequent reads from the same terminal will continue to read from the point where the last read operation left off. The last read operation which exhausts the available data associated with the terminal returns the "End of Input" rather than the "Successful Completion" status code (as usual, the Length register will contain the count of characters read). Subsequent read operations will continue to return the "End of Input" status. If on a read operation from terminal *i*, the device is not ready for input (the file *termin*i does not exist in the current directory), the status register is loaded with the "Device Not Ready" code.

Chapter Two
User Interface

1. Introduction

The CHIP simulator currently runs under the UNIX operating system. To facilitate its use, a C compiler, an assembler, a loader and a console are available. This chapter describes these utility programs. It is organized as follows. Section 2 describes how to compile, generate assembler listings for and load programs that are targeted for CHIP. Sections 3, 4 and 5 describe the console, the primary means by which the user interacts with the CHIP computer. In section 3 the conditions that activate the console are described. Section 4 describes the console screen itself and Section 5 describes the *console command language*.

2. Compiling C Programs and Starting up CHIP

The C compiler for CHIP, called **pcc**, is a slightly modified PDP-11 C compiler. The command line syntax for this compiler is very similar to that of the compiler for the PDP-11. Consult the manual page for details. There are several differences between a C program running in the PDP-11 and one running in the CHIP environment:

- The load module produced by **pcc** (the *a.out* file) has the program origin at address 2000 (octal) rather than 0
- The symbolic name *start* is associated with address 2000
- The program prologue causes a transfer of control to the function *main()* through the INPRG instruction
- No library functions that request UNIX system services (e. g., *printf(), scanf(), open()*, etc.) can be used.

The symbolic names *etext* and *edata* are synonyms for the last location in the program (text) and data areas, respectively.

The CHIP simulator is invoked by typing the UNIX command line

chip [-s] [*file*]

where *file* is an optional argument naming an object file in PDP-11 format produced by the *pcc* command. If the object file argument is omitted, the *a.out* file in the current directory is taken as the default. If the **-s** flag is present, the symbol table for the object file is written to the file *file*.symbols in the current directory. As part of its initialization phase, the simulator

loads the program and initialized data contained in the object file into physical memory starting at location 2000 and initializes the processor state as follows:

- the Program Counter (PC) points to the first executable instruction, i.e., PC=2000
- the Stack Pointer (SP) points to the first word beyond the end of available physical memory
- the Stack Limit Register (STL) points to the first word of the last page of available physical memory
- Processor Status Word 1 (PS1) indicates that all interrupts are masked, memory mapping is not in effect and the machine is in KERNEL mode
- all other registers are set to 0.

3. Control Transfer

When the **chip** command is typed, the machine enters the IPL (Initial Program Load) state and the console is in *command input mode*. Thus, it is ready to accept commands. In order to begin program execution, the user types **run**.

During program execution, six conditions cause the processor to stop and the console to enter command input mode. They are:

Halt: The machine executes a HALT instruction.

Wait: The machine enters a Wait state. This occurs when the Wait bit of Processor Status Word 1 (PS1.W) is turned on and there are no unmasked interrupt requests and no possible sources of unmasked future interrupts.

Bkpt: A breakpoint is encountered. Breakpoints are marked instructions in the program. To set a breakpoint, the user types **bi** (for breakpoint insert) and a list of memory addresses, specified as byte offsets from symbolic function names, denoting the locations of the breakpoints. The processor stops just before the marked instruction is executed.

Susp: A suspect variable is about to be referenced. Suspects are marked addresses in memory. To mark a location as suspect, the user types **si** (for suspect insert) and a list of symbolic names or addresses. The

machine stops just prior to executing the instruction that will generate a reference to the suspect location.

SS: The machine is single stepping. In this case, execution halts after each instruction. This occurs when the *pace* parameter of the console is set to 0. Higher values of this parameter cause execution speed to increase. When pace has its maximum value 9, the machine runs at full speed. To assign a new value to pace, the user types **p** and an integer between 0 and 9.

Stop: Execution is manually stopped. This is done by typing a '^C' (type 'c' while holding down the 'Control' key). The machine stops upon completion of the current instruction.

4. The Console Screen

When the simulator is invoked, the user's terminal becomes an operator's console. The CHIP console screen consists of several fields and is organized as follows:

- at the top of the screen is a line of status information
- near the bottom of the screen is the command input line with ':' (colon) as the prompt character
- just below the command input line is an error message line
- between the status and command input lines is the contents of a *virtual screen.*

There are three different virtual screens:

Static Screen: Contains addresses of all active breakpoints and suspects. Breakpoints are on the left half of the window and suspects on the right half.

Dynamic Screen: Contains addresses and values of all data regions being traced. To put a region on the trace list, the user types **ti** (for *trace insert*) and a list of data regions. When execution of an instruction modifies values within a traced region, the contents of the dynamic window are updated accordingly. Traces are listed in order of insertion.

Work Screen: Contains a record of memory location read and write requests issued by the user. To read memory values, the user types a '.' (period) and the names or addresses of memory regions to be read. Contents of memory locations can be altered by supplying the new value on the right hand side of an '=' (equals) character. For example, the command .200=0 would clear the word at location 200 (octal).

The screen that has been referenced most recently is the one that is actually displayed. This reference may be explicit or it may be implicit. For example, the **ss** console command (for *static screen*) causes explicit switching to the static screen; the **ti** command causes the dynamic screen to be brought into view to carry out the trace insertion.

Upon initialization, the status line looks as follows:

B	T	S	RD	SL	P	PC		SP	Intmask	K	M	W	N	Z	V	C	PS2	SCREEN	STATE
1	1	1	0	0	9	2000	start+0	20000	11111111	1	0	0	0	0	0	0	0	Dynamic	IPL

The B, T, S, RD and SL fields are console *switches*. When a 1 appears below a switch name on the status line, that switch is on; when a 0 appears it is off. To toggle a switch, the user simply types the switch name. The meanings of the five console switches are:

B If on, all breakpoints are enabled; if off, encountering a breakpoint does not cause CHIP to halt. Toggling B off is useful when the user wants all breakpoints to be temporarily ignored.

T If on and the dynamic screen is being displayed, as modifications of traced variables occur, their values are seen on the console window. If off, changes to traced variables are not seen until the machine stops and the dynamic screen is brought into view.

S If on, all suspects defined are enabled; if off, suspects remain on the suspect list, but referencing any one does not cause the machine to stop. Toggling S off is useful for temporarily disabling the suspect mechanism.

RD If on, the contents of the machine registers appear on the dynamic screen. Whether they are updated dynamically depends upon the setting of the T switch.

SL If on, the status line is updated while the machine is running; if off, changes do not appear until the machine stops.

The values of the Pace parameter (field *P*), the Program Counter, the Stack Pointer and the Processor Status Word are also reported on the status line. The contents of the PC is given in two formats: in octal and in the form *function name* +*offset*, where *function name* is the symbolic name of the function the current instruction is in. SP and PS2 are displayed in octal, while PS1 is displayed in binary with its various fields labeled symbolically.

The field labeled SCREEN on the status line identifies the virtual screen that is being displayed. The field labeled STATE either identifies the condition that has caused the processor to stop or indicates that the processor is running.

5. The Console Command Language

Here we present the syntax and semantics of the CHIP Console Command Language. The following standard conventions in notation are used in the command language description:

NOTATION	MEANING
{S1 | S2}	one of S1 or S2 must be given
[S]	S may be given 0 or 1 times
[S]*	S may be given 0 or more times
[S]+	S may be given 1 or more times
ε	the null command

Additionally,

- Words in **boldface** are keywords
- Words in *italics* are defined in the command descriptions
- Words in normal lower case font are nonterminals of the grammar which are defined at the end of this section.

The following is a list of the CHIP Console Commands and their meanings:

ds Bring the dynamic screen into view.

ss Bring the static screen into view.

ws Bring the work screen into view.

wsc Clear the work screen.

p *int* Set the pace parameter to *int*, where *int* is a positive integer between 0 and 9. When the pace is set to 0, **run** causes the machine to single step.

si [interval [, {**r** | **w**}]]+
Insert intervals into the suspect list. Modifier **r** indicates that a suspect reference should cause the machine to stop execution only when a value is about to be read from the interval; modifier **w** indicates that a reference should cause the machine to stop only when a value is about to be written. The default is to stop execution when a location is about to be read or written. A location can appear only once on the suspect list.

sr [*] Delete intervals from the suspect list. If * is given, all entries currently on the suspect list are removed. If no argument appears, the console enters interactive *removal mode*, initially positioning the cursor at the top of the list of suspects being shown. At this point, typing **d** deletes the interval whose specifications appear to the right of the cursor. To move down to other interval specifications the user types 'j' or 'New Line'; to move up the user can type 'k' or 'Space'. If the suspect list has more entries than there are lines in the static window, the window provides only a partial view of the list; movement and deletion may result in re-appearance of other entries. To return to command input mode type 'q'.

s Toggle the console switch S.

ti [interval [/ {**d** | **b** | **a**}]]+
Insert intervals into the trace list. If **d**, **b** or **a** is given, the data

values are displayed in Decimal, Binary or ASCII, respectively. The default is Octal. In ASCII format, unprintable characters are displayed as '@'.

tr [*] Delete intervals from the trace list. If * is given, the entire list is deleted. Otherwise, the user selects intervals to be deleted through the mechanism described above for removing suspect variables.

t Toggle the console switch T.

rd Toggle the console switch RD.

bi [*fname* [offset] [^count]]+

Insert locations into the breakpoint list. The location given to identify a breakpoint is specified as a function name *fname* and optionally an offset. If a count is given, the breakpoint will not cause a break until it is encountered count times. The default for count is one. If a breakpoint insert command is issued on a location that is already in the list, then the original entry is updated.

br [*] Delete locations from the breakpoint list. Typing * causes all current breakpoint entries to be deleted. Removing individual breakpoints is similar to deleting entries from the trace and suspect lists.

b Toggle the console switch B.

.[{interval | register } [/ { **d** | **b** | **a**}]]+

Display values of intervals and registers. The display selectors **d**, **b** and **a** are as in the trace list insert command.

.[{interval | register} = { *string* | [-] nbr }]+

Write new values to registers and intervals. The values may be given as strings or as numbers. A *string* is a sequence of characters enclosed in double quote marks '"'. A double quote mark can be part of the string if it is typed as \". A back slash can be included in the string by typing \\. If the interval contains more than one byte, successive bytes are assigned characters from *string* until either *string*

or the interval is exhausted.

fin[put] *file*
Read console commands from *file* rather than from the terminal. Argument *file* may be any existing UNIX file containing console commands just as if they had been typed directly to the console. If an error is encountered in processing a command in *file*, the line number of the erroneous command is reported and no further commands are executed.

fout[put] *file*
Write the contents of the current window to the file *file*. Argument *file* may be any UNIX file name. It is not possible to write the contents of the work screen.

re[draw]
Redraw the entire screen.

ipl Bring the machine to a state identical to that right after power up without leaving the simulator.

{**r[un]** | ε}
Run the loaded program. The null command (a blank line terminated by a 'New Line') is a synonym for **run** if the pace is set to zero (single step mode).

{? | **help**}
Display the list of valid commands.

{**bye** | **q[uit]** | **end**}
Power down CHIP.

Where the nonterminals are defined as follows:

interval — loc [**:** [{loc | offset}]]
A region in the CHIP computer memory. An interval is a block of memory words (two bytes) defined by a lower and upper bound. If no upper bound is supplied, the interval is treated as consisting of a single memory word at the address given by the lower bound. When the user gives only an offset as the upper bound, the address is calculated by adding the offset to the lower bound.

loc — {*id* | nbr | (loc)} [offset]
A single memory word in CHIP. An *id* is any identifier that is a global symbol in the current program. Note that the PDP-11 C compiler truncates identifiers to seven characters. If an offset is given, its value is added to the address of the identifier or to the specified number in order to determine the location address.

offset — {+ | -} nbr
A signed memory offset.

nbr — *int* [**,** {**d** | **b** }]
A numeric value and its interpretation. *Int* is any unsigned integer and the selectors **d** and **b** indicate whether it is to be interpreted as a decimal or binary number, respectively. The default interpretation is as an octal number.

register — {**#r0** | **#r1** | **#r2** | **#r3** | **#r4** | **#r5** | **#r6** | **#r7** | **#pc** | **#sp** | **#ps1** | **#ps2** | **#sta** | **#stl** | **#tdck** | **#it**}
A register of the CHIP processor. Note that **#sp** is a synonym for **#r6** and **#pc** is a synonym for **#r7**.

Appendix
The CHIP Instruction Set

This appendix is intended to serve as a quick reference for instruction opcodes and formats. New instructions are described in §4.1 and §4.2; PDP-11 instructions are described in detail in the *PDP-11 Processor Handbook.*

The format and notation for the brief descriptions below are as follows:

- For each instruction, the symbolic name is followed by the operation code (in octal) and a brief description
- Instructions preceded by an asterisk are new instructions. Some of these have the same opcodes as their PDP-11 counterparts but function differently
- Instructions with byte as well as word versions are marked with "(b)" and the "n" field distinguishes between them
- The "ddd" field denotes the destination operand
- The "sss" field denotes the source operand
- The "rrr" field denotes the register selector
- The "ooo" field denotes the offset.

Privileged Instructions

*LDST	0	000	111	000	000	000	007000	Load process state
*STST	0	000	111	001	000	000	007100	Store process state
*LDIT	0	000	111	010	000	000	007200	Load interval timer
*LDIM	0	000	111	011	000	000	007300	Load interrupt mask
*LDSTL	0	000	111	100	000	000	007400	Load stack limit register
*HALT	0	000	000	000	000	000	000000	Halt

Single Operand Instructions

General

clr(b)	n	000	101	000	ddd	ddd	n050DD	Clear destination
dec(b)	n	000	101	011	ddd	ddd	n053DD	Decrement destination
inc(b)	n	000	101	010	ddd	ddd	n052DD	Increment destination
neg(b)	n	000	101	100	ddd	ddd	n054DD	Negate destination
tst(b)	n	000	101	111	ddd	ddd	n057DD	Test destination
com(b)	n	000	101	001	ddd	ddd	n051DD	Complement destination
*MOVBCK	1	000	111	000	000	000	107000	Move block of bytes
*STCK	1	000	111	001	000	000	107100	Store time of day clock
*INPRG	1	000	111	010	000	000	107200	Initialize Program

Shifts

asr(b)	n	000	110	010	ddd	ddd	n062DD	Arith. shift right destination

asl(b)	n	000	110	011	ddd	ddd	n063DD	Arith. shift left destination
ash	0	111	010	rrr	sss	sss	072RSS	Shift arithmetically
ashc	0	111	011	rrr	sss	sss	073RSS	Arith. shift combined

Multiple precision

adc(b)	n	000	101	101	ddd	ddd	n055DD	Add carry destination
sbc(b)	n	000	101	110	ddd	ddd	n056DD	Subtract carry destination
sxt	0	000	110	111	ddd	ddd	0067DD	Sign extended destination

Rotates

rol(b)	n	000	110	001	ddd	ddd	n061DD	Rotate left destination
ror(b)	n	000	110	000	ddd	ddd	n060DD	Rotate right destination
swab	0	000	000	011	ddd	ddd	0003DD	Swap bytes destination

Double operand instructions

General

mov(b)	n	001	sss	sss	ddd	ddd	n1SSDD	Move source to destination
add	0	110	sss	sss	ddd	ddd	06SSDD	Add source to destination
sub	1	110	sss	sss	ddd	ddd	16SSDD	Subtract source from destination
cmp(b)	n	010	sss	sss	ddd	ddd	n2SSDD	Compare source to destination
mul	0	111	000	rrr	sss	sss	070RSS	Multiply
div	0	111	001	rrr	sss	sss	071RSS	Divide
xor	0	111	100	rrr	ddd	ddd	074RDD	Exclusive or

Logical

bis(b)	n	101	sss	sss	ddd	ddd	n5SSDD	Bit set
bit(b)	n	011	sss	sss	ddd	ddd	n3SSDD	Bit test
bic(b)	n	100	sss	sss	ddd	ddd	n4SSDD	Bit clear

Program control instructions

Branches

br	0	000	000	1oo	ooo	ooo	0004offset	Unconditional branch

Simple conditional branches

beq	0	000	001	1oo	ooo	ooo	0014offset	Branch on equal
bne	0	000	001	0oo	ooo	ooo	0010offset	Branch on not equal
bmi	1	000	000	1oo	ooo	ooo	1004offset	Branch on minus
bpl	1	000	000	0oo	ooo	ooo	1000offset	Branch on plus
bcs	1	000	011	1oo	ooo	ooo	1034offset	Branch on carry set
bcc	1	000	011	0oo	ooo	ooo	1030offset	Branch on carry clear
bvs	1	000	010	1oo	ooo	ooo	1024offset	Branch on overflow set
bvc	1	000	010	0oo	ooo	ooo	1020offset	Branch on overflow clear

Signed conditional branches

blt	0	000	010	1oo	ooo	ooo	0024offset	Branch on less than
bge	0	000	010	0oo	ooo	ooo	0020offset	Branch on greater than or equal
ble	0	000	011	1oo	ooo	ooo	0034offset	Branch on less than or equal
bgt	0	000	011	0oo	ooo	ooo	0030offset	Branch on greater than

Unsigned conditional branches

bhi	1	000	001	0oo	ooo	ooo	1010offset	Branch on higher
blos	1	000	001	1oo	ooo	ooo	1014offset	Branch on lower or same
blo	1	000	011	1oo	ooo	ooo	1034offset	Branch on lower
bhis	1	000	011	0oo	ooo	ooo	1030offset	Branch on higher or same

Subroutine

jsr	0	000	100	rrr	ddd	ddd	004RDD	Jump to subroutine
mark	0	000	110	100	nnn	nnn	0064NN	Mark
rts	0	000	000	010	000	rrr	00020R	Return from subroutine
*CSV	1	000	111	011	000	000	107300	Save registers for subroutine call
*CRET	1	000	111	100	000	000	107400	Restore saved registers and return

Program control

jmp	0	000	000	001	ddd	ddd	0001DD	Jump
sob	0	111	111	rrr	ooo	ooo	077Roffset	Subtract one and branch

System calls

*SYS	1	000	100	100	000	000	104400	Generate a system call trap
				to			to	
	1	000	100	111	111	111	104777	

Condition code operators

clc	0	000	000	010	100	001	000241	Clear C
clv	0	000	000	010	100	010	000242	Clear V
clz	0	000	000	010	100	100	000244	Clear Z
cln	0	000	000	010	101	000	000250	Clear N
sec	0	000	000	010	110	001	000261	Set C
sev	0	000	000	010	110	010	000262	Set V
sez	0	000	000	010	110	100	000264	Set Z
sen	0	000	000	010	111	000	000270	Set N
scc	0	000	000	010	111	111	000277	Set all
ccc	0	000	000	010	101	111	000257	Clear all