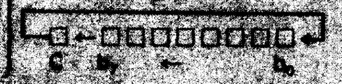
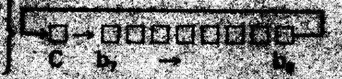
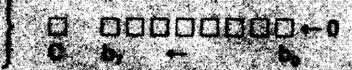
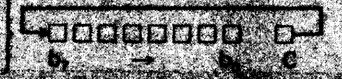
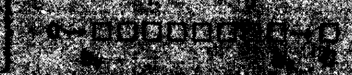
Push data	PSHA					36	A → M_{SP}, SP − 1 → SP
	PSHB					37	B → M_{SP}, SP − 1 → SP
Pull data	PULA					32	SP + 1 → SP, M_{SP} → A
	PULB					33	SP + 1 → SP, M_{SP} → B
Rotate left	ROL			69	79		M A B 
	ROLA					49	
	ROLB					59	
Rotate right	ROR			66	76		M A B 
	RORA					46	
	RORB					56	
Shift left, arithmetic	ASL			68	78		M A B 
	ASLA					48	
	ASLB					58	
Shift right, arithmetic	ASR			67	77		M A B 
	ASRA					47	
	ASRB					57	
Shift right, logic	LSR			64	74		M A B 
	LSRA					44	
	LSRB					54	
Store accumulator	STAA		97	A7	B7		A → M
	STAB		D7	E7	F7		B → M
Subtract	SUBA	80	90	A0	B0		A − M → A
	SUBB	C0	D0	E0	F0		B − M → B
Subtract accumulator	SBA					10	A − B → A
Subtract with carry	SBCA	82	92	A2	B2		A − M − C → A
	SBCB	C2	D2	E2	F2		B − M − C → B
Transfer accumulators	TAB					16	A → B
	TBA					17	B → A
Test, zero or minus	TST			6D	7D		M − 00
	TSTA					4D	A − 00
	TSTB					5D	B − 00

*Courtesy Motorola Corp.
†OP code (hexadecimal)
+ Arithmetic plus
− Arithmetic minus
· Boolean AND
M_{SP} Contents of memory location pointed to by stack pointer
+ Boolean inclusive OR

⊕ Boolean exclusive OR
$\overline{M}$ Complement of M
→ Transfer into
0 Bit = zero
00 Byte = zero

Note: Accumulator addressing mode instructions are included in the column for implied addressing.

TABLE 10.14 **INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS**

		ADDRESSING MODES					
		IMMED	DIRECT	INDEX	EXTND	IMPLIED	
POINTER OPERATIONS	MNEMONIC	OP	OP	OP	OP	OP	BOOLEAN ARITHMETIC OPERATION†
Compare index register	CPX	8C	9C	AC	BC		$X_H - M, X_L - (M + 1)$
Decrement index register	DEX					09	$X - 1 \rightarrow X$
Decrement stack pointer	DES					34	$SP - 1 \rightarrow SP$
Increment index register	INX					08	$X + 1 \rightarrow X$
Increment stack pointer	INS					31	$SP + 1 \rightarrow SP$
Load index register	LDX	CE	DE	EE	FE		$M \rightarrow X_H, (M + 1) \rightarrow X_L$
Load stack pointer	LDS	8E	9E	AE	BE		$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$
Store index register	STX		DF	EF	FF		$X_H \rightarrow M, X_L \rightarrow (M + 1)$
Store stack pointer	STS		9F	AF	BF		$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$
Index register → stack pointer	TXS					35	$X - 1 \rightarrow SP$
Stack pointer → index register	TSX					30	$SP + 1 \rightarrow X$

†See footnotes to Table 10.13

TABLE 10.15 JUMP AND BRANCH INSTRUCTIONS

		ADDRESSING MODES				
		RELATIVE	INDEX	EXTND	IMPLIED	
OPERATIONS	MNEMONIC	OP	OP	OP	OP	BRANCH TEST†
[illegible]	BRA	[illegible]				None
Branch if carry clear	BCC	[illegible]				C = 0
Branch if carry set	BCS	[illegible]				C = 1
Branch if = 0	BEQ	[illegible]				Z = 1
Branch if ≥ 0	BGE	[illegible]				N ⊕ V = 0
Branch if > 0	BGT	[illegible]				Z + (N ⊕ V) = 0
Branch if higher	BHI	[illegible]				C + Z = 0
Branch if ≤ 0	BLE	[illegible]				Z + (N ⊕ V) = 1
Branch if lower or same	BLS	[illegible]				C + Z = 1
Branch if < 0	BLT	[illegible]				N ⊕ V = 1
Branch if minus	BMI	[illegible]				N = 1
Branch if not equal	BNE	[illegible]				Z = 0
Branch if overflow clear	BVC	[illegible]				V = 0
Branch if overflow set	BVS	[illegible]				V = 1
Branch if plus	[illegible]	[illegible]				N = 0
[illegible]	[illegible]	[illegible]				
[illegible]	[illegible]		[illegible]	[illegible]		
[illegible]	[illegible]		[illegible]	[illegible]		
[illegible]	NOP				[illegible]	Advances program counter only
[illegible]	RTI				[illegible]	
[illegible]	[illegible]				[illegible]	
[illegible]	[illegible]				[illegible]	
[illegible]	[illegible]				[illegible]	

†See footnotes to Table 10.13.

TABLE 10.16 CONDITION CODE REGISTER BITS

Condition code register. The condition code register indicates the results of an ALU operation: negative (*N*), zero (*Z*), overflow (*V*), carry from bit 7 (*C*), and half-carry from bit 3 (*H*). These bits of the condition code register are used as testable conditions for the conditional branch instructions. Bit 4 is the interrupt mask bit (*I*). The unused bits of the condition code register (b_6 and b_7) are 1s.

The programmer wrote the columns from Label to the right. The assembler generated the leftmost two columns.

The first instruction, CLRA, simply clears accumulator *A*. The LDAB instruction gets the number of bytes in the table from location 50_{16} in the memory and stores that number in register *B*. Notice that in 6800 assembly language a hexadecimal number is designated by placing a $ in front of the number. Also notice that the 50 occurs in the second memory address of the instruction word, and the addressing mode is immediate.

The LDX #$01 loads the value 1 in the index register. The symbol # tells the assembler to use this as an actual number, not as an address. The resulting immediate address operand requires 2 bytes since the index register contains 16 bits. Also note that the least significant byte is the last byte in the 3-byte instruction word.

The ADDA $50, X is an addition instruction in *indexed* addressing mode. In the 6800 the indexed mode is indicated by the X in the statement. The $50 (for hexadecimal 50) gives the offset. The actual address used is formed by adding the offset to the contents of the index register. In this case, the first time through the loop, the address will be the offset 50_{16} plus 1, or 51_{16}. Notice that the offset is loaded in the memory following the OP code and is a single byte. (A check of the OP codes will indicate that AB is the OP code for an indexed-mode addition.) After this instruction is executed, accumulator *A* will contain the number at location 51.

The INX adds 1 to the index register, which will now contain 2. The DECB decrements register *B* and also sets the status bits. In particular, if *B* becomes 0, the *Z* status bit will be set to a 1, and this will indicate that the entire table has been processed.

The BNE LOOP instruction tests the *Z* bit and branches if *Z* is *not* a 1. When *Z* becomes 1, the program control "falls through" the BNE to the STAA instruction.

TABLE 10.17 CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

OPERATIONS	MNEMONIC	OP CODE	BOOLEAN ARITHMETIC OPERATION
Clear carry	CLC	0C	$0 \rightarrow C$
Clear interrupt mask	CLI	0E	$0 \rightarrow I$
Clear overflow	CLV	0A	$0 \rightarrow V$
Set carry	SEC	0D	$1 \rightarrow C$
Set interrupt mask	SEI	0F	$1 \rightarrow I$
Set overflow	SEV	0B	$1 \rightarrow V$
Accumulator $A \rightarrow$ CCR	TAP	06	$A \rightarrow CCR$
CCR $\rightarrow$ accumulator A	TPA	07	$CCR \rightarrow A$

TABLE 10.18 A 6800 PROGRAM

MEMORY ADDRESS	CONTENTS	LABEL	OP CODE	OPERAND	COMMENTS
0000	4F		CLRA		CLEAR A
0001	D6		LDAB	$50	GET NO. OF ENTRIES
0002	50				
0003	CE		LDX	#$01	LOAD INDEX REGISTER
0004	00				
0005	01				
0006	AB	LOOP	ADDA	$50, X	
0007	50				
0008	08		INX		INCREMENT IR
0009	5A		DECB		DECREMENT B
000A	26		BNE	LOOP	
000B	FA				
000C	97		STAA	$0F	
000D	0F				

When the program control loops back the first time, the index register plus the offset now equals 52, so the number at that address will be added into accumulator A by the ADDA $50, X instruction. This process continues with numbers at successive locations being added into A until the table end is reached. Then the STAA $0F instruction stores the sum at location F in the memory.

Subroutine calls for the 6800 are made by a JSR (jump subroutine) instruction which pushes the program counter's contents (2 bytes)[12] on top of the stack (also adjusting the stack pointer). When an RTS (return from subroutine) instruction is given, the address (2 bytes) on top of the stack is placed in the program counter, causing return to the instruction after the initial JSR.

An example of a subroutine is shown in Table 10.19. The ORG $30 is an *assembler directive* which tells the assembler to place the subroutine starting at location 30_{16} in the memory. The purpose of this subroutine is to find where in a table in the memory a character lies. The parameters are passed[13] as follows: (1) the address of the end of the table must be in the index register before the subroutine is entered, (2) the number of entries in the table is placed in accumulator B, (3) the character to be searched for must be in accumulator A.

[12]After the program counter has already been updated to point to the next instruction.
[13]See description of the 8080 for a discussion of parameter passing.

TABLE 10.19 6800 SUBROUTINE FOR TABLE LOOKUP

LABEL	OP CODE	OPERAND	COMMENTS
	ORG	$30	SET ORIGIN
SRCH	CMPA	0,X	CHAR = TABLE ENTRY?
	BEQ	FINIS	YES QUIT
	DEX		INCREMENT IR
	DECB		DECREMENT B
	BNE	SRCH	TEST FOR END
FINIS	RTS		RETURN TO CALLER

TABLE 10.20 CALLING 6800 SUBROUTINE

LABEL	OP CODE	OPERAND	COMMENTS
	LDX	#ENDTA	LOAD IR WITH TABLE END
	LDAB	#20	LOAD B WITH NO. OF ENTRIES
	LDAA	CHAR	LOAD A WITH CHAR
	JSR	SRCH	
	STAA	MABEL	

The subroutine is entered at SRCH, where the CMPA 0, X instruction causes the byte at the memory address given by the index register (notice that the offset is 0) to be compared with accumulator A. If they are equal, the Z status bit will be set to 1, and the BEQ FINIS instruction will test this instruction and branch to FINIS. Otherwise, the index register will be decremented so that it points to the next lowest entry in the table. Accumulator B will then be decremented by the DECB, and if this sets the Z flag to 1, indicating a 0 in B, the search will be ended. Otherwise, the return to SRCH will cause the next entry in the table to be compared with the character in accumulator A. This will be repeated until all table entries have been examined.

The RTS will cause a return to the calling program with accumulator B containing the number in the table at which the matched character lies.

A possible calling program segment is shown in Table 10.20. LDX loads the index register with the address of the end of the table, which is assumed to be at ENDTA. The number of table entries is assumed to be 20_{10}, and LDAB loads B with that value. (No $ symbol means decimal.) JSR causes a jump to the subroutine, and the jump back from the subroutine using RTS will cause the STAA instruction to be executed.

When a set of chips for a microprocessor of this kind is used with a fixed program, such as in an industrial controller, the program is generally developed by using system software which is provided by the chips' manufacturer and software vendors and placed in a ROM memory. A ROM memory can be addressed and used just like a RAM memory when it is connected to a microprocessor CPU (except, of course, that we cannot write into a ROM). It is common practice to write the program for the microprocessor and assemble this program using another computer. The program is sometimes tested using this larger computer, which runs it on a simulator. The larger computer then produces a tape which is used to set up the ROM in which the program will be stored.

Considerable effort is made by the manufacturers of the microprocessor chips to facilitate programming the microprocessor, preparing the ROMs, and loading the RAMs, when required. Sometimes higher-level languages are provided, enabling programs to be written in Fortran, PL/M, or other compiler languages, which are then translated into the program for the microcomputer.

PDP-11

10.13 The PDP-11 series includes minicomputers and microcomputers. These computers have 16-bit words, each containing two 8-bit bytes (see Fig. 10.19). Notice, however, that each address in memory contains 1 byte. The eight general registers are 16 bits each, and a computer word normally has 16 bits.

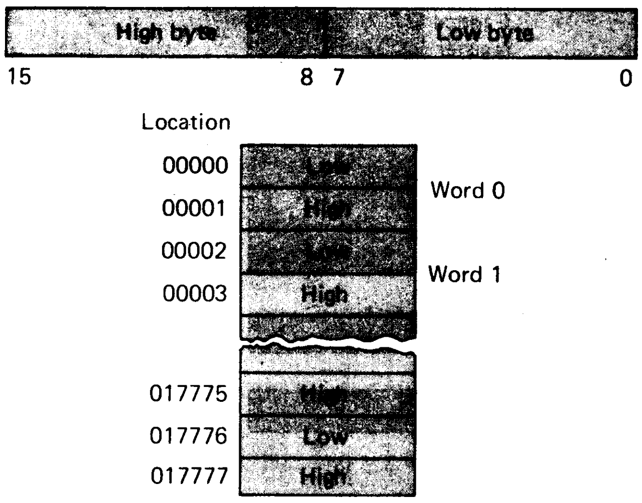


FIGURE 10.19

Memory organization of PDP-11.

The PDP-11 reads from and writes into external input-output devices in the same manner that it reads from and writes into high-speed memory. Each input-output device is simply given an address in memory. To read from an address, and in turn an input-output device, the computer uses not a special input-output instruction, but a MOVE instruction, an ADD instruction, or whatever is desired. This means that status registers must be used by the CPU (as in Chap. 8) to determine whether a device can be written into, has something to read, etc.

There is a complex interrupt structure in the PDP-11 where the CPU continually puts a status number on three wires of its bus. Each external device has a status number, and if that status number is greater than the CPU status number, it has the right to interrupt. Setting a CPU's status number to its maximum stops all interrupts. The CPU's status number is set under program control and can be changed as the program operates.

Interrupts are "vectored" (see Chap. 8 and the description of the 8080, Sec. 10.11) in that an interrupting device places data (an interrupt vector) on the bus lines which enable the CPU to transfer control directly to a service program for the interrupting device.

The general registers of the PDP-11 are shown in Fig. 10.20(*a*). Notice that register R_6 is a stack pointer and R_7 is the program counter. These can be used and addressed just as the other general-purpose registers, making for interesting instruction variations.

The CPU in the PDP-11 includes a status register, as shown in Fig. 10.20(*b*). This status register contains the priority number just discussed, which is placed on the bus in bits 5 to 7. Bits 11 to 15 in Fig. 10.20(*b*) are used by the operating system to control program operations in the 11/45, one of the larger PDP-11 models, and are not discussed here. (Details are given in the manuals listed in the Bibliography.)

The *N, Z, V, C* bits in the status word are set and reset as instructions are operated. For example, the *N* bit indicates when a result is negative. If an ADD instruction is performed and the result is negative, the *N* bit will be set to a 1;

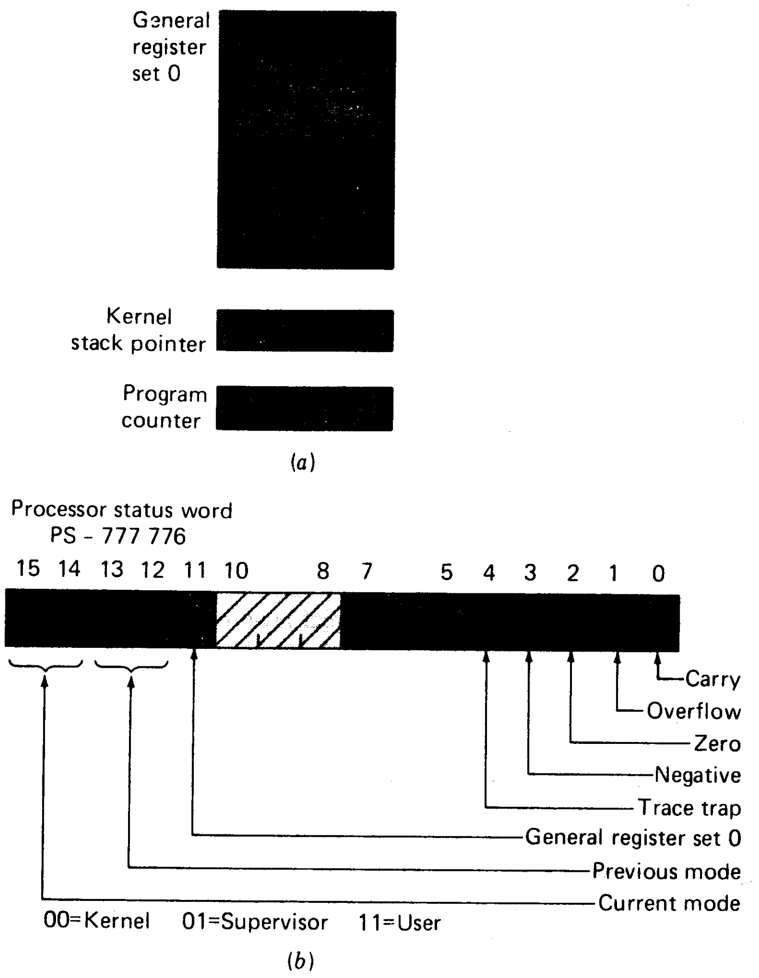


FIGURE 10.20

PDP-11 organization. (*a*) General registers. (*b*) Processor status word.

otherwise, it will be a 0. Similarly, the *Z* bit indicates a zero result and will be set on if an instruction's result is zero. (*V* is for overflow and *C* is for carry.)

The conditional jump or branch instructions in the PDP-11 use these bits to determine whether a jump is to be taken. For instance, BNE (branch on negative) will cause a branch only if the *N* bit is a 1. As another example, BEQ (branch on equal) causes a branch only if the *Z* bit is a 1.

Table 10.21 lists the addressing modes for the PDP-11. The addressing mode number is placed before the register number in an instruction word and indicates how the designated register is to be used. Table 10.22 gives the instructions for the computer.

Table 10.23 shows a sample section of a program for a PDP-11. This is a subroutine, or subprogram, which reads from a teletypewriter keyboard. There is a status byte (interface register) at address 177030 in the memory which tells when the keyboard has a new character. The subprogram places characters in a table until a period is typed, at which time control is transferred to another subprogram.

TABLE 10.21 ADDRESSING MODES FOR PDP-11 MINICOMPUTER

ADDRESS	MODE	NAME	SYMBOLIC	DESCRIPTION
General register	0	Register	R	(R) is operand [e.g., R_2 = %2]
Mode *R*	1	Register deferred	(R)	(R) is address
	2	Auto increment	(R)+	(R) is address; (R) + (1 or 2)
	3	Auto increment deferred	@(R)+	(R) is address of address; (R) + 2
	4	Auto decrement	−(R)	(R) − (1 or 2); (R) is address
	5	Auto decrement deferred	@−(R)	(R) − 2; (R) is address of address
	6	Index	X(R)	(R) + X is address
	7	Index deferred	@X(R)	(R) + X is address of address
Program counter, Reg = 7	2	Immediate	#n	Operand *n* follows instruction
Mode 7	3	Absolute	@#A	Address *A* follows instruction
	6	Relative	A	Instruction address + 4 + X is address
	7	Relative deferred	@A	Instruction address + 4 + X is [illegible]

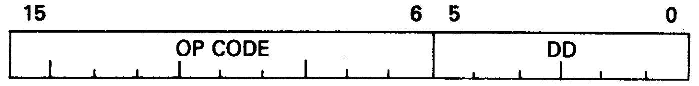
PDP-11

TABLE 10.22 PDP-11 INSTRUCTION REPERTOIRE

LEGEND

OP CODES	OPERATIONS	BOOLEAN	CONDITION CODES
[illegible]	[illegible]	[illegible]	[illegible]

SINGLE OPERAND: OPR dst



MNEMONIC	OP CODE	INSTRUCTION	dst RESULT	*N*	*Z*	*V*	*C*
General							
CLR(B)	■050DD	Clear	0	0	1	0	0
COM(B)	■051DD	Complement (1s)	~d	•	•	0	1
INC(B)	■052DD	Increment	d + 1	•	•	•	[illegible]
DEC(B)	■053DD	Decrement	d − 1	•	•	•	[illegible]
NEG(B)	■054DD	Negate (2s complement)	−d	•	•	•	•
TST(B)	■057DD	Test	d	•	•	0	0
Rotate and Shift							
ROR(B)	■060DD	Rotate right	→ C, d	•	•	•	•
ROL(B)	■061DD	Rotate left	C, d ←	•	•	•	•
ASR(B)	[illegible]	Arithmetic shift right	[illegible]	•	•	•	•
ASL(B)	[illegible]	Arithmetic shift left	[illegible]	•	•	•	•
[illegible]	[illegible]	[illegible]		•	•	•	[illegible]
[illegible]							
[illegible]	[illegible]	[illegible]	[illegible]	•	•	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	•	•	[illegible]	[illegible]

DOUBLE OPERAND: OPR src, dst OPR src, R or OPR R, dst

15–12	11–6	5–0
OP CODE	SS	DD

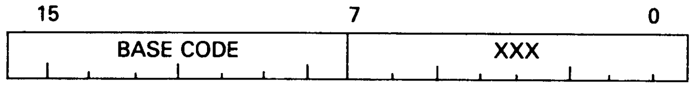
15–9	8–6	5–0
OP CODE	R	SS OR DD

MNEMONIC	OP CODE	INSTRUCTION	OPERATION	*N*	*Z*	*V*	*C*
[illegible]							
[illegible]	[illegible]	[illegible]	d←s	●	●	0	—
[illegible]	[illegible]	[illegible]	[illegible]	●	●	●	●
[illegible]	[illegible]	[illegible]	d←s + d	●	●	●	●
[illegible]	[illegible]	[illegible]	d←d − s	●	●	●	●
[illegible]							
[illegible]	[illegible]	[illegible]	s∧d	●	●	0	—
[illegible]	[illegible]	[illegible]	d←(~s)∧d	●	●	0	—
[illegible]	[illegible]	[illegible]	d←s∨d	●	●	0	—
[illegible]							
[illegible]	[illegible]	[illegible]	[illegible]	●	●	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	●	●	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	●	●	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	●	●	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	d←r∨d	●	●	0	[illegible]

TABLE 10.22 PDP-11 INSTRUCTION REPERTOIRE (*continued*)

LEGEND

BRANCH: B—location
If condition is satisfied:
Branch to location,
$$\text{New PC} \leftarrow \underbrace{\text{updated PC}}_{\text{address of branch instruction } + 2} + (2 \times \text{offset})$$



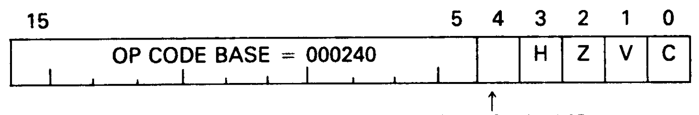
OP code = base code + XXX

MNEMONIC	BASE CODE	INSTRUCTION	BRANCH CONDITION	
[illegible]				
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	Z = 0
[illegible]	[illegible]	[illegible]	[illegible]	Z = 1
[illegible]	[illegible]	[illegible]	[illegible]	N = 0
[illegible]	[illegible]	[illegible]	[illegible]	N = 1
[illegible]	[illegible]	[illegible]	[illegible]	V = 0
[illegible]	[illegible]	[illegible]	[illegible]	V = 1
[illegible]	[illegible]	[illegible]	[illegible]	C = 0
[illegible]	[illegible]	[illegible]	[illegible]	C = 1
[illegible]				
[illegible]	[illegible]	[illegible]	[illegible]	$N \veebar V = 0$
[illegible]	[illegible]	Branch if less than (0)	<0	$N \veebar V = 1$
[illegible]	[illegible]	Branch if greater than (0)	>0	$Z \vee (N \veebar V) = 0$
BLE	[illegible]	Branch if less than or equal to (0)	≤0	$Z \vee (N \veebar V) = 1$
Unsigned Conditional Branches				
[illegible]	101000	Branch if [illegible]	>	$C \vee Z = 0$
[illegible]	101400	Branch if [illegible]	≤	$C \vee Z = 1$
[illegible]	[illegible]	Branch if [illegible]	[illegible]	C = 0
[illegible]	[illegible]	Branch if [illegible]	[illegible]	C = 1

JUMP AND SUBROUTINE MNEMONIC	OP CODE	INSTRUCTION	NOTES
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]

TRAP AND INTERRUPT MNEMONIC	OP CODE	INSTRUCTION	NOTES
EMT	104000 to 104377	Emulator trap (not for general use)	PC at 30, PS at 32
TRAP	104400 to 104777	Trap	PC at 34, PS at 36
BPT	000003	Breakpoint trap	PC at 14, PS at 16
IOT	000004	Input-output trap	PC at 20, PS at 22
RTI	000002	Return from interrupt	
▲RTT	000006	Return from interrupt	Inhibit T bit trap

CONDITION CODE OPERATIONS



0 = CLEAR SELECTED CONDITION CODE BITS
1 = SET SELECTED CONDITION CODE BITS

MNEMONIC	OP CODE	INSTRUCTION	*N*	*Z*	*V*	*C*
CLC	000241	Clear *C*	–	–	–	0
CLV	000242	Clear *V*	–	–	0	–
CLZ	000244	Clear *Z*	–	0	–	–
CLN	000250	Clear *N*	0	–	–	–
CCC	000257	Clear all condition code bits	0	0	0	0
SEC	000261	Set *C*	–	–	–	1
SEV	000262	Set *V*	–	–	1	–
SEZ	000264	Set *Z*	–	1	–	–
SEN	000270	Set *N*	1	–	–	–
SCC	000277	Set all condition code bits	1	1	1	1

MISCELLANEOUS

MNEMONIC	OP CODE	INSTRUCTION
HALT	000000	[illegible]
WAIT	000001	[illegible]
RESET	000005	[illegible]
NOP	000240	[illegible]
●SPL	00023N	[illegible]
▲MFPI	0065SS	[illegible]
▲MTPI	0066DD	[illegible]
●MFPD	1065SS	[illegible]
●MTPD	1066DD	[illegible]

Note: ▲ Applies to 11/35, 11/40, 11/45 computers. ● Applies to 11/45 computer.

TABLE 10.23 A PDP-11 PROGRAM SEGMENT

MEMORY ADDRESS	CONTENTS	LABEL	OP CODE	ADDRESS PART	COMMENTS
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

The section shown in Table 10.23 is from an actual assembler listing for a PDP-11, and all numbers are in octal. The programmer writes all text from the Label column to the right. Semicolons indicate comments, and everything to the right of a semicolon is a comment and is ignored by the assembler.

The listing was prepared by the programmer who wrote the assembly-language program and then fed it into the assembler program which generated this listing.

The leftmost column lists locations in the memory, and the next column the contents of these locations. For instance, TSTB (test byte) has OP code 105767, and the assembler has read the programmer's TSTB instruction and converted it into octal value.

The statement TSTB tests the byte at the address given. If the value there is negative, it places a 1 in the N bit; if it is zero, a 1 is placed in the Z bit. KSR designates the address in memory, 177030, where the status byte for the keyboard is located. The programmer has (in an earlier section of the program) told the assembler the value of KSR. If the keyboard has a character ready, the sign bit of the KSR byte will be a 1, causing the N bit to go on.

The next instruction, BPL READ, says branch to READ if $N = 0$. This means that if no character is available, the program goes back to READ and looks again. This continues until a character is ready and $N = 1$.

The BPL has an OP code of 100. The next byte contains the displacement, or offset, for the branch in 2s complement form. The address for the branch is equal to two times the offset byte's value (375) added to the address of the next instruction. In this case the offset value is negative, and a branch would go back to location 524.

When $N = 1$, the instruction word at location 532 will be executed. This is a MOVB (move byte) instruction which causes a byte to be moved from KSB, which is 177024 (the address of the keyboard's buffer, the value of which the program has already given to the assembler), to the value pointed to by R_0. This is an example of indirect addressing, where R_0 is used to point to the actual address. Prior to this section of the program the programmer has loaded R_0 with the starting location of the table in the memory where the input characters are to be stored.

The program now checks to see whether the input character is a period, which has octal code 256, by comparing it with the character just loaded in the memory. Notice that indirect addressing is again used. The plus sign causes the value in R_0 to be incremented. Only if the character pointed to is equal to 256 will the Z bit be set to 1.

The BNE (branch on not equal) instruction checks this. If the character is a period, it transfers control to another program; otherwise, control is transferred back to the READ, where another character is then read from the keyboard.

The variety and complexities of the PDP-11's instruction repertoire can be appreciated only through a study of the manuals for this computer. The preceding example should point out the kind of efficient programs which can be written for this computer.

8086 AND 8088 MICROPROCESSORS

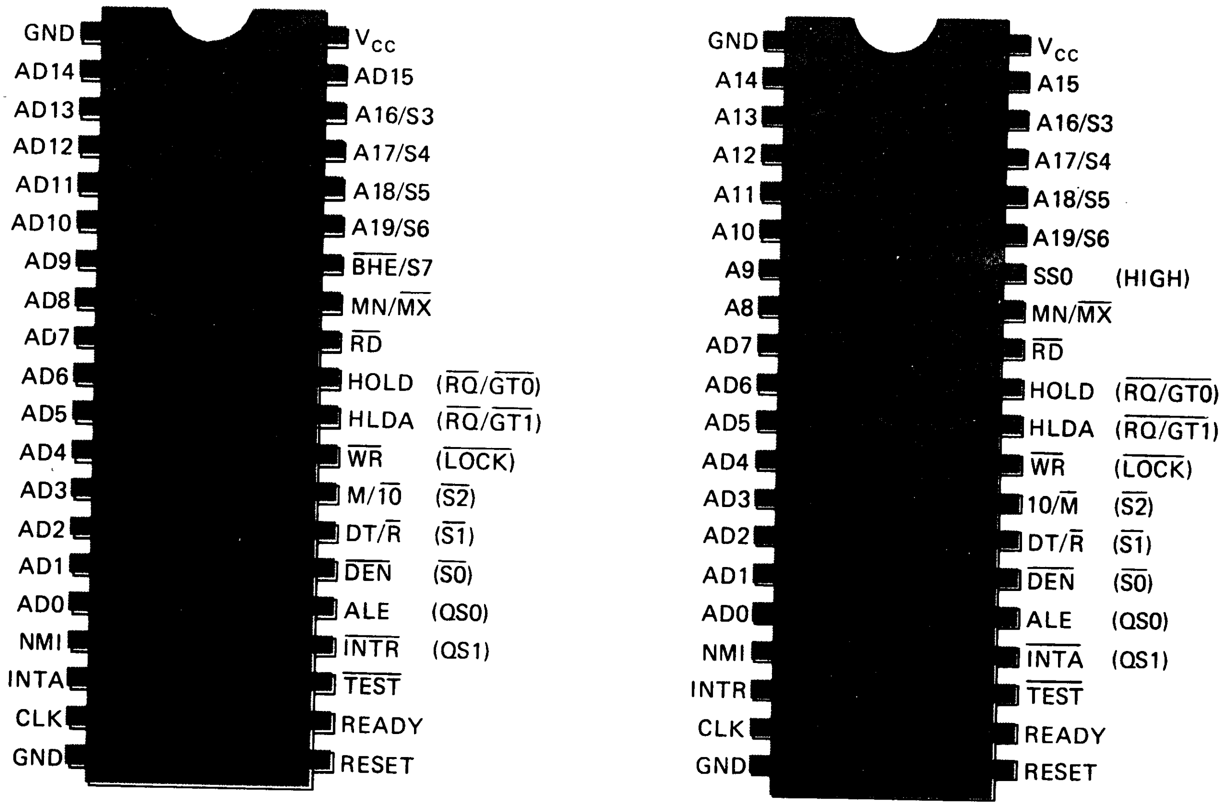
10.14 The 8086 and 8088 microprocessors are extensions of Intel's earlier 8080 microprocessor series. There are a number of changes in the 8086/8088, the most obvious being the fact that computations can be performed using 16-bit data versus 8-bit data for the 8080. There are a number of other advantages, however, including multiplication and division instructions, instruction queuing to improve operation speed, the ability to address a million bytes of memory, more general registers, and more instructions and addressing modes.

The 8086 and 8088 chips are part of a series which include the clock generator and interface chips shown in Fig. 8.22 and a floating-point arithmetic chip (the 8089).

The pin-outs for the 8086 and 8088 are shown in Fig. 10.21. As explained in Chap. 8, the address and data lines are shared by using time-division multiplexing. The principal difference between the 8086 and 8088 lies in the number of data lines output to the bus. The 8086 has 16 data lines on its bus, and the 8088 has only 8. This is shown by the number of AD (address/data) lines versus A (address) lines in the pin-out for each chip. The 8086 use 16 of the address lines for data also, so that $AD0$ to $AD15$ are used for address and data while the 8088 has only $AD0$ to $AD7$ for data and uses $A8$ to $A19$ for addresses. The internal data paths on the chips are the same, however; and each can add, subtract, multiply, or divide 16-bit binary numbers.

The result of the sharing of output lines is that several chips are required to demultiplex the address data and control lines. One possible configuration is shown in Fig. 10.22. The output from these extra chips forms the actual bus for the 8086 or 8088. The control lines from the chips are used to strobe the data address and control signals into the bus interface chips.

An important feature of the 8086 and 8088 microprocessor is the instruction queue used in each. The 8086/8088 chips read instructions in order from the memory in advance of their operation, and the instructions are placed in a queue consisting of a set of flip-flop registers. This speeds up operation because the processor can continue executing a time-consuming instruction (multiplication, for example) and at the same time read instructions from the memory of the processor. Then the processor can execute fast instructions (shift or test instructions, for example) from the queue at a speed faster than memory cycle times. Logic is supplied so that if the computer branches (jumps), the instructions in the queue are discarded if necessary.



GND
AD14
AD13
AD12
AD11
AD10
AD9
AD8
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
NMI
INTA
CLK
GND
V_{CC}
AD15
A16/S3
A17/S4
A18/S5
A19/S6
$\overline{BHE}$/S7
MN/$\overline{MX}$
$\overline{RD}$
HOLD ($\overline{RQ}/\overline{GT0}$)
HLDA ($\overline{RQ}/\overline{GT1}$)
$\overline{WR}$ ($\overline{LOCK}$)
M/$\overline{IO}$ ($\overline{S2}$)
DT/$\overline{R}$ ($\overline{S1}$)
$\overline{DEN}$ ($\overline{S0}$)
ALE (QS0)
$\overline{INTR}$ (QS1)
$\overline{TEST}$
READY
RESET
GND
A14
A13
A12
A11
A10
A9
A8
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
NMI
INTR
CLK
GND
V_{CC}
A15
A16/S3
A17/S4
A18/S5
A19/S6
SS0 (HIGH)
MN/$\overline{MX}$
$\overline{RD}$
HOLD ($\overline{RQ}/\overline{GT0}$)
HLDA ($\overline{RQ}/\overline{GT1}$)
$\overline{WR}$ ($\overline{LOCK}$)
IO/$\overline{M}$ ($\overline{S2}$)
DT/$\overline{R}$ ($\overline{S1}$)
$\overline{DEN}$ ($\overline{S0}$)
ALE (QS0)
$\overline{INTA}$ (QS1)
$\overline{TEST}$
READY
RESET

FIGURE 10.21

Pin-outs for 8086 and 8088 processors.

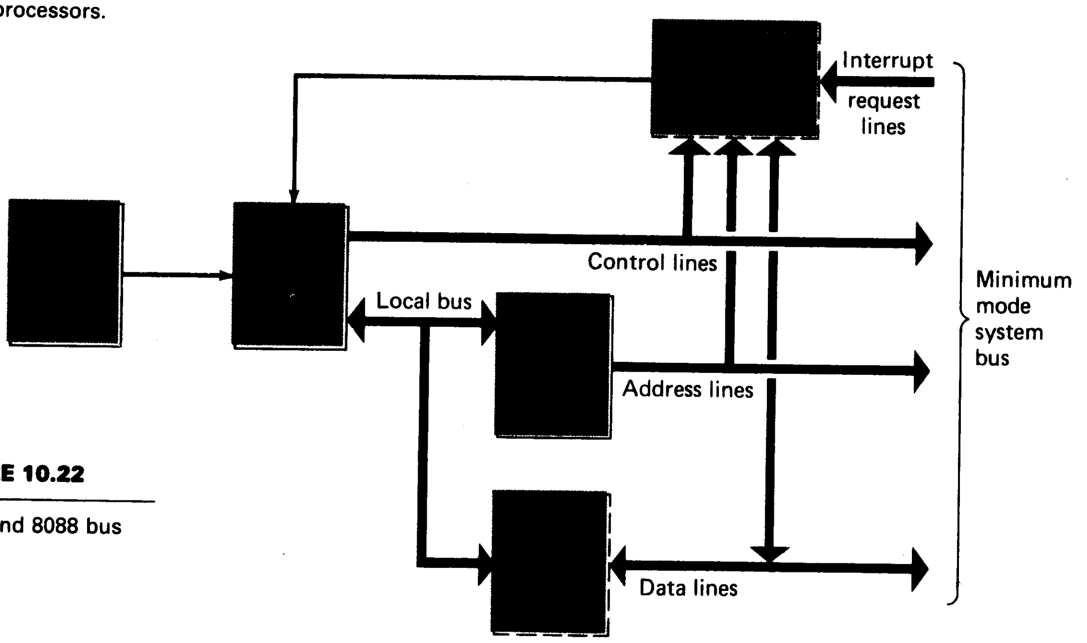


FIGURE 10.22

8086 and 8088 bus setup.

The 8086/8088 pair each have a special output pin, the *MN/MX* pin. When this pin is connected to TSV, the processor is placed in a minimum mode; when it is connected to OV, the processor is placed in a maximum mode. When in the minimum mode, the processor is used in single processor systems. In the maximum mode, several processors can be used with an 8288 bus controller which provides a special multibus architecture for multiprocessor systems. The maximum mode is for large arrays of memory, processors, and I/O devices.

A block diagram of the registers of the 8086 and 8088 is shown in Fig. 10.23. Notice eight general registers.

The 8086/8088 processors have a number of addressing modes. Addresses are 20 bits in length. Each address is formed in two sections which are then added: a *segment* address and an offset. The segment address is a full 20 bits, and the offset address is 16 bits.

There are four segment registers, *CS, DS, SS,* and *ES,* each containing 16 bits. These registers must be loaded by the program to starting values because the contents of one of these registers are ***automatically*** added to each address as it is generated. The contents of the 16-bit segment registers are first shifted left four binary places, however. (This is the equivalent of multiplying the contents of the registers by 16.) Loading the segment registers with 0s would simply place the program and stacks in the first 2^{16} words in memory and would effectively remove this feature for simple programs.

When a program is operated, the content of the program counter is automatically added to *CS* to form each instruction address. Data offsets are automatically added to *DS* (or *ES* in special cases), and stack offsets are automatically added to *SS*. Setting the *CS, DS,* and *SS* registers to addresses in different parts of a large memory would cause the instructions, data, and stacks to be in different parts of the memory. Setting the *CS, SS,* and *DS* registers to the same number would place everything in the same part of memory. Once the segment registers are set, the processor simply generates 16-bit offset addresses in a conventional manner from the instruction words while adding the segment register to each address to form the final 20-bit address. If a program really needed 2^{20} addresses, it would be necessary to change the segment registers from time to time to utilize the entire memory.

In effect, the 8086 and 8088 generate conventional 16-bit (offset) addresses by using instruction words and then add the contents of a 20-bit number to each of these offsets to form a 20-bit final address.

Quite a number of addressing modes are used to form the offsets in the 8086/8088 chips. Operands can be in general registers, memory, or I/O ports, and immediate addressing is provided. When 20-bit addresses are generated, the second byte in an instruction word contains the information as to how the 16-bit offset or effective address part of the address is to be calculated. (The first 3 and last 2 bits in this byte provide that information.) In general, this section of the address is formed by summing the contents of a displacement (part of the instruction word), an index register, and a base register. Any combination of these three can be used. And this implements, for example, direct addressing, register indirect addressing, and based indexed addressing (summing the base register, index register, and displacement).

The segment registers make it possible to address a 2^{20}-word memory while

68000 MICROPROCESSOR

10.15 The 68000 microprocessor is a semiconductor chip with a number of support chips such as I/O processors, a floating-point arithmetic chip, and bus handler chips. The 68000 has 16-bit data paths on its system bus and performs 32-bit arithmetic and logic operations internally. The 68000 microprocessor can directly address 16 Mbytes of memory, having a 24-bit address bus. There are 14 addressing modes and 56 types of instructions. The I/O is memory-mapped.

Chapter 8 showed a drawing of the 68000 bus and the timing signals for reads and writes on the bus. The bus is asynchronous in order to accommodate both slow and fast memory and I/O devices.

The basic registers in the 68000 are shown in Fig. 10.25. The registers are 32 bits, and there are eight data registers along with seven address registers and a program counter. There are actually two stack pointers. A status bit determines whether the 68000 is in the *supervisor* (operating system) *mode* or *user mode;* this bit also determines which of the two stack pointers are in use. The status register is shown in Fig. 10.25 and contains 5 bits for condition codes.

The 68000 *supervisor* and *user* modes are an important feature. There are privileged instructions which can be executed in supervisor mode, but not in user mode. When the supervisor-user mode select bit is a 1, the 68000 uses the supervisor stack pointer and the privileged instructions are available. When the select bit is a 0, the user stack pointer is employed, and certain instructions will not execute.

Figure 10.26 shows that data are organized into bits, bytes, words, and long words and shows how these are placed in the memory which has 8 bits (1 byte) at each address. Instruction words can be from one word (16 bits) to four words in length.

Stacks in the 68000 go from high memory to low memory. So the stack pointer is decremented when data are pushed into a stack and incremented when data are popped from a stack.

Let us examine a particular instruction to understand how the addressing operates. The ANDI (for AND immediate) has the following instruction word format.

15	14	13	12	11	10	9	8	7 6	5 4 3	2 1 0
0	0	0	0	0	0	1	0	Size	Mode	Address register
Word data (16)								Byte data (8)		
Long word (32 bits)										

The OP-code part of this instruction is 02_{16}. This tells the microprocessor that the instruction is ANDI. The function of this instruction is to AND the immediate data which follow the first 16 bits of the instruction (word), which is called the source, with the destination operand and to place the result in the destination.

The number of bits in an ANDI instruction word depends on the 2 bits in the *size* section. If these are 00, the operation is a byte operation and the instruction

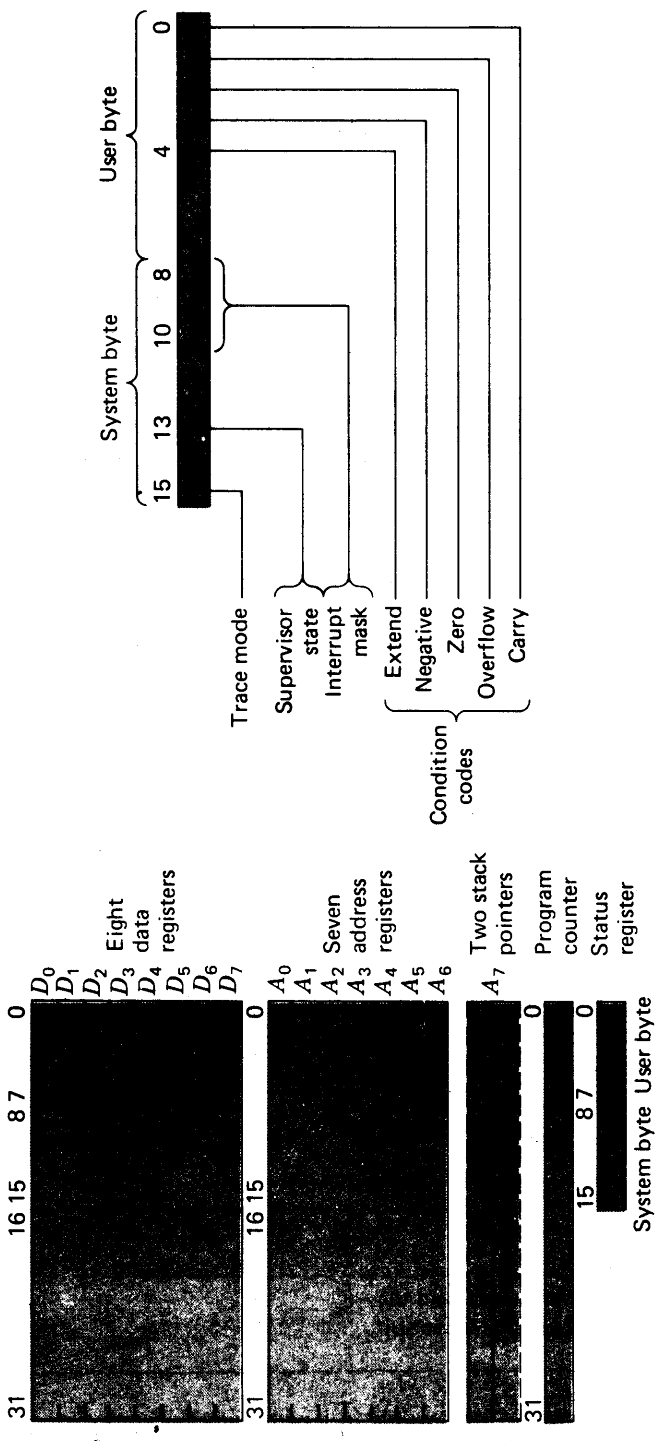


FIGURE 10.25

68000 programmable registers.

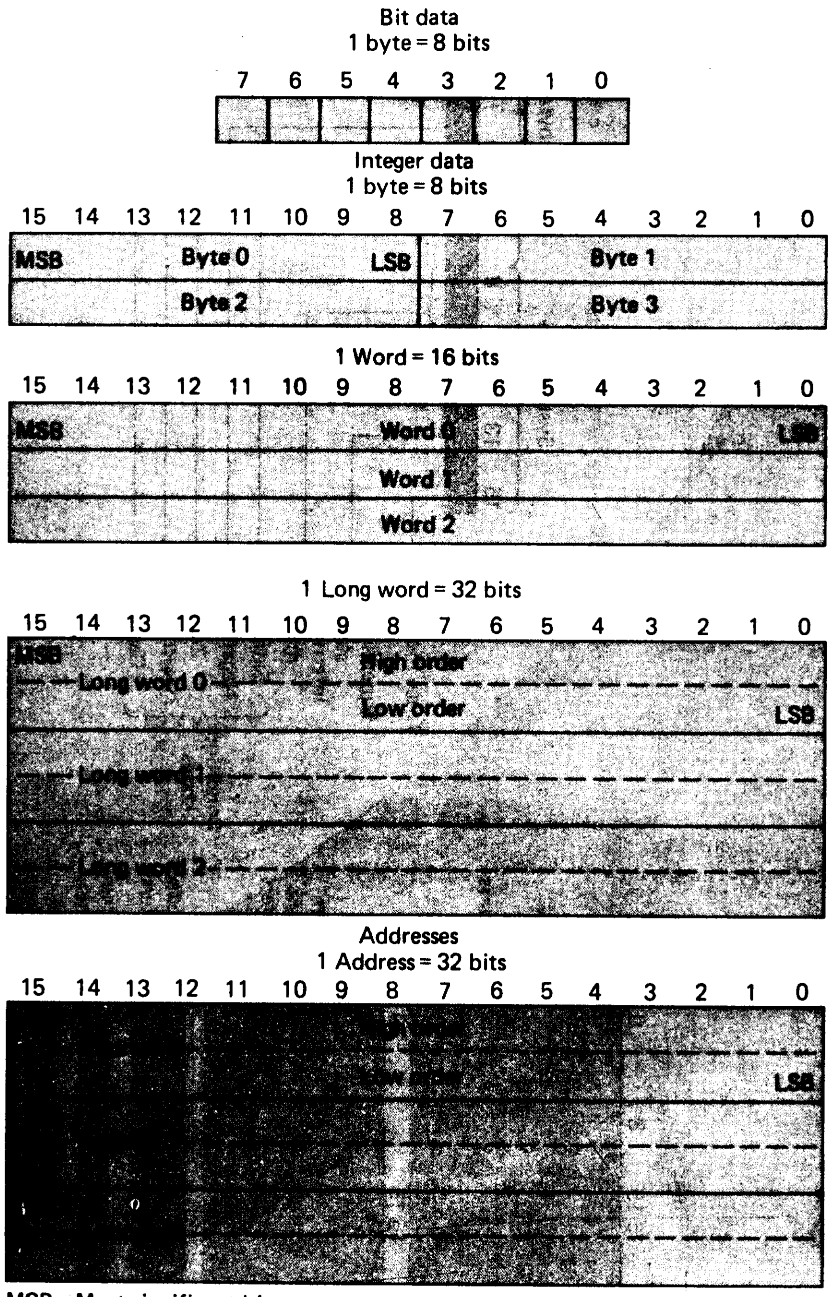


FIGURE 10.20

Organization of data in memory for 68000.

word is two 16-bit words in length with the immediate data in bits 7 to 0 of the second word. If the size bits are 01, the instruction word is again 32 bits in length, but the immediate operand is the entire 16 bits in the second word. If the size bits are 10, then ANDI has a 32-bit-long word for its immediate data and the instruction word is 48 bits in length.

Mode	Generation
Register direct addressing	
Data Register Direct	EA = Dn
Address Register Direct	EA = An
Absolute data addressing	
Absolute Short	EA = (Next Word)
Absolute Long	EA = (Next Two Words)
Program counter relative addressing	
Relative with Offset	EA = (PC) + d_{16}
Relative with Index and Offset	EA = (PC) + (Xn) + d_8
Register indirect addressing	
Register Indirect	EA = (An)
Postincrement Register Indirect	EA = (An), An ← An + N
Predecrement Register Indirect	An ← An − N, EA = (An)
Register Indirect with Offset	EA = (An) + d_{16}
Indexed Register Indirect with Offset	EA = (An) + (Xn) + d_8
Immediate data addressing	
Immediate	DATA = Next Word(s)
Quick Immediate	Inherent Data
Implied addressing	
Implied Register	EA = SR, USP, SP, PC

Notes:
EA = Effective Address
An = Address Register
Dn = Data Register
Xn = Address or Data Register used as Index Register
SR = Status Register
PC = Program Counter
d_8 = 8-bit Offset (displacement)
d_{16} = 16-bit Offset (displacement)
N = 1 for Byte, 2 for Words, and 4 for Long Words.
If An is the stack pointer and the operand size byte, N = 2 to keep the stack pointer on a word boundary.
( ) = Contents of
← = Replaces

FIGURE 10.27

Addressing modes for 68000.

The destination operand is determined by the mode bits and address register bits. If the mode bits are 000, then the destination is the register given by the address register bits. If these are 010, for example, then data register 2 (refer to Fig. 10.25) will be the destination register and the part of that register used (byte word or long word) will be determined by the size section. If, for example, the data register used is 010 and the size bits are 00, bits 7 to 0 in data register 2 will be ANDed with bits 7 to 0 in the second word in the instruction, and the result is placed in bits 7 to 0 of data register 2.

If the mode bits are 010, then the address register given by the address register bits in the instruction word will contain the address of the (destination) data in memory. For example, if the mode bits are 010, the address register bits are 011; then address register 3 will contain the address of the operand. So if address register 3 contains 0143_{16}, then the operand will be at that location in memory. If the size bits are 00, then bits 7 to 0 of that location in memory will be used for the AND, and the result placed in these bits. If the size bits are 01, then the entire word in location 0143_{16} will be ANDed with the 16 bits in the second word of the instruction, and the result placed in memory location 0143_{16}.

The instruction repertoire is described in what Motorola calls its register transfer language (as in Chap. 9). In their system, (A_n) means, "The contents of A_n gives the location in memory of the operand." (The n in A_n is given by the address register bits. The register A_n is frequently called the *pointer* because it points to the operand.) The notation $A_n@+$ is called "address register indirect with post-increment" and $A_n-@$ is called "address register indirect with predecrement."

The notation (A_n-) means, "Decrement A_n and then use the result as the address of the operand." The notation $(A_n)+$ means, "Use the contents of A_n to determine the location in memory and then increment." For $(A_n)+$ and (A_n-), the number in A_n is incremented or decremented by 1, 2, or 4 depending on whether the instruction is a byte, word, or long word instruction.

The notation $A_n@$ is also used to mean, "A_n contains the address of the operand." This is the same as (A_n) but leads to the notation $(A_n)@$ which means, "Take the address in A_n, go to that address in memory, and at that address find

The following register transfer language definitions are used for the operation description in the details of the instruction set.

OPERANDS

An	address register	**SSP**	supervisor stack pointer
Dn	data register	**USP**	user stack pointer
Rn	any data or address register	**SP**	active stack pointer (equivalent to A7)
PC	program counter	**X**	extend operand (from condition codes)
SR	status register	**Z**	zero condition code
CCR	condition codes (low order byte of status register)	**V**	overflow condition code

Immediate Data	— immediate data from the instruction		
d	— address displacement	Destination	destination location
Source	— source location	Vector	location of exception vector

SUBFIELDS AND QUALIFIERS

< **bit** > **OF** < **operand** >	selects a single bit of the operand
< **operand** >[< **bit number** >:< **bit number** >]	selects a subfield of an operand
(< **operand** >)	the contents of the referenced location
< **operand** > **101**	the operand is binary coded decimal; operations are to be performed in decimal.
< **operand** > @ < **mode** >	the register indirect operator which indicates that the operand register points to the memory location of the instruction operand. The optional mode qualifiers are −, +, (d) and (d, ix); these are explained in Chapter 2.

OPERATIONS

Operations are grouped into binary, unary, and other.

Binary These operations are written < operand > < op > < operand > where < op > is one of the following:

→	the left operand is moved to the location specified by the right operand
↔	the contents of the two operands are exchanged
+	the operands are added
−	the right operand is subtracted from the left operand
*	the operands are multiplied
/	the first operand is divided by the second operand
Λ	the operands are logically ANDed
v	the operands are logically ORed
⊕	the operands are logically exclusively ORed
<	relational test, true if left operand is less than right operand
>	relational test, true if left operand is not equal to right operand
shifted by **rotated by**	the left operand is shifted or rotated by the number of positions specified by the right operand

UNARY

~ < **operand** >	the operand is logically complemented
< **operand** > **sign-extended**	the operand is sign extended, all bits of the upper half are made equal to high order bit of the lower half
< **operand** > **tested**	the operand is compared to 0, the results are used to set the condition codes

MULS Signed Multiply

Operation: (Source)*(Destination) → Destination

Assembler syntax: MULS < ea >, Dn

Attributes: Size = (Word)

Description: Multiply two signed 16-bit operands yielding a 32-bit signed result. The operation is performed using signed arithmetic. A register operand is taken from the low order word; the upper word is unused. All 32 bits of the product are saved in the destination data register.

Condition codes

X	N	Z	V	C
—	*	*	0	0

N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared
X Not affected.

Instruction format

15	14	13	12	11 10 9	8	7	6	5 4 3	2 1 0
1	1	0	0	Register	1	1	1	Effective Mode	Address Register

Instruction fields
Register field Specifies one of the data registers. This field always specifies the destination.
Effective address field Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	register number	d(An, Xi)	110	register number
An	—	—	Abs.W	111	000
(An)	010	register number	Abs.L	111	001
(An)+	011	register number	d(PC)	111	010
-(An)	100	register number	d(PC, Xi)	111	011
d(An)	101	register number	Imm	111	100

ANDI AND Immediate

Operation: Immediate Data Λ (Destination) → Destination

Assembler syntax: ANDI # < data >, < ea >

Attributes: Size = (Byte, Word, Long)

Description: AND the immediate data to the destination operand and store the result in the destination location. The size of the operation may be specified to be byte, word, or long. The size of the immediate data matches the operation size.

Condition codes

X	N	Z	V	C
—	*	*	0	0

N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.
X Not affected.

Instruction format

15	14	13	12	11	10	9	8	7 6	5 4 3	2 1 0
0	0	0	0	0	0	1	0	Size	Effective Mode	Address Register
Word Data (16 bits)								Byte Data (8 bits)		
Long Data (32 bits, including previous word)										

Instruction fields
Size field Specifies the size of the operation:
00 byte operation.
01 word operation.
10 long operation.
Effective Address field Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	register number	d(An, Xi)	110	register number
An			Abs.W	111	000
(An)	010	register number	Abs.L	111	001
(An)+	011	register number	d(PC)	—	—
-(An)	100	register number	d(PC, Xi)	—	—
d(An)	101	register number	Imm	—	—

Immediate field (Data immediately following the instruction):
If size = 00, then the data is the low order byte of the immediate word.
If size = 01, then the data is the entire immediate word.
If size = 10, then the data is the next two immediate words.

Bcc Branch Conditionally

Operation: If (condition true) then PC + d → PC

Assembler syntax: Bcc < label >

Attributes: Size = (Byte, Word)

Description: If the specified condition is met, program execution continues at location (PC) + displacement. Displacement is a twos complement integer which counts the relative distance in bytes. The value in PC is the current instruction location plus two. If the 8-bit displacement in the instruction word is zero, then the 16-bit displacement (word immediately following the instruction) is used. "cc" may specify the following conditions:

CC	carry clear	0100	$\overline{C}$	LS	low or same	0011	$C + Z$
CC	carry set	0101	C	LT	less than	1101	$N\cdot\overline{V}+\overline{N}\cdot V$
EQ	equal	0111	Z	MI	minus	1011	N
GE	greater or equal	1100	$N\cdot V+\overline{N}\cdot\overline{V}$	NE	not equal	0110	$\overline{Z}$
GT	greater than	1110	$N\cdot V\cdot\overline{Z}+\overline{N}\cdot\overline{V}\cdot\overline{Z}$	PL	plus	1010	$\overline{N}$
HI	high	0010	$\overline{C}\cdot\overline{Z}$	VC	overflow clear	1000	$\overline{V}$
LE	less or equal	1111	$Z+N\cdot\overline{V}+\overline{N}\cdot V$	VS	overflow set	1001	V

Condition codes: Not affected.

Instruction format

15	14	13	12	11 10 9 8	7 6 5 4 3 2 1 0
0	1	1	0	Condition	8-bit Displacement
16-bit Displacement if 8-bit Displacement = 0					

Instruction fields
Condition field One of fourteen conditions discussed in description.
8-bit displacement field Twos complement integer specifying the relative distance (in bytes) between the branch instruction and the next instruction to be executed if the condition is met.
16-bit displacement field Allows a larger displacement than 8 bits. Used only if the displacement is equal to zero.

Note: A short branch to the immediately following instruction cannot be done because it would result in a zero offset which forces a word branch instruction definition.

CMP — Compare

Operation: (Destination) − (Source)

Assembler syntax: CMP < ea >, Dn

Attributes: Size = (Byte, Word, Long)

Description: Subtract the source operand from the destination operand and set the condition codes according to the result; the destination location is not changed. The size of the operation may be specified to be byte, word, or long.

Condition codes

X	N	Z	V	C
—	*	*	*	*

N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.
X Not affected.

Instruction format

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	1	1	Register	Op-Mode	Effective Mode	Address Register

Instruction fields
Register field Specifies the destination data register.
Op-Mode field

Byte	Word	Long	Operation
000	001	010	(< Dn >) − (< ea >)

Effective address field Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	register number	d(An, Xi)	110	register number
An*	001	register number	Abs.W	111	000
(An)	010	register number	Abs.L	111	001
(An) +	011	register number	d(PC)	111	010
− (An)	100	register number	d(PC, Xi)	111	011
d(An)	101	register number	Imm	111	100

*Word and Long only.

Note: CMPA is used when the destination is an address register. CMPI is used when the source is immediate data. CMPM is used for memory to memory compares. Most assemblers automatically make this distinction.

DIVS — Signed Divide

Operation: (Destination)/(Source) → Destination

Assembler syntax: DIVS < ea >, Dn

Attributes: Size = (Word)

Description: Divide the destination operand by the source operand and store the result in the destination. The destination operand is a long operand (32 bits) and the source operand is a word operand (16 bits). The operation is performed using signed arithmetic. The result is a 32-bit result such that:
1. The quotient is in the lower word (least significant 16-bits).
2. The remainder is in the upper word (most significant 16-bits).
The sign of the remainder is always the same as the dividend unless the remainder is equal to zero. Two special conditions may arise:
1. Division by zero causes a trap.
2. Overflow may be detected and set before completion of the instruction. If overflow is detected, the condition is flagged but the operands are unaffected.

Condition codes

X	N	Z	V	C
—	*	*	*	0

N Set if the quotient is negative. Cleared otherwise. Undefined if overflow.
Z Set if the quotient is zero. Cleared otherwise. Undefined if overflow.
V Set if division overflow is detected. Cleared otherwise.
C Always cleared.
X Not affected.

Instruction format

15	14	13	12	11 10 9	8	7	6	5 4 3	2 1 0
1	0	0	0	Register	1	1	1	Effective Mode	Address Register

Instruction fields
Register field Specified any of the eight data registers. This field always specifies the destination operand.
Effective Address field Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addresing Mode	Mode	Register
Dn	000	register number	d(An, Xi)	110	register number
An	—	—	Abs.W	111	000
(An)	010	register number	Abs.L	111	001
(An) +	011	register number	d(PC)	111	010
− (An)	100	register number	d(PC, Xi)	111	011
d(An)	101	register number	Imm	111	100

Note: Overflow occurs if the quotient is larger than a 16-bit signed integer.

JSR — Jump to Subroutine

Operation: PC → − (SP); Destination → PC

Assembler syntax: JSR < ea >

Attributes: Unsized

Description: The long word address of the instruction immediately following the JSR instruction is pushed onto the system stack. Program execution then continues at the address specified in the instruction.

Condition codes: Not affected.

Instruction format

15	14	13	12	11	10	9	8	7	6	5 4 3	2 1 0
0	1	0	0	1	1	1	0	1	0	Effective Mode	Address Register

Instruction fields
Effective address field Specifies the address of the next instruction. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	d(An, Xi)	110	register number
An	—	—	Abs.W	111	000
(An)	010	register number	Abs.L	111	001
(An) +	—	—	d(PC)	111	010
− (An)	—	—	d(PC, Xi)	111	011
d(An)	101	register number	Imm		

RTS — Return from Subroutine

Operation: (SP) + → PC

Assembler syntax: RTS

Attributes: Unsized

Description: The program counter is pulled from the stack. The previous program counter is lost.

Condition codes: Not affected.

Instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

FIGURE 10.28

Selected instructions for 68000.

SUB Subtract Binary

Operation: (Destination) − (Source) → Destination

Assembler: SUB < ea >, Dn
Syntax: SUB Dn, < ea >

Attributes: Size = (Byte, Word, Long)

Description: Subtract the source operand from the destination operand and store the result in the destination. The size of the operation may be specified to be byte, word, or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition codes

X	N	Z	V	C
*	*	*	*	*

N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.
X Set the same as the carry bit.

Instruction format

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	0	1	Register	Op-Mode	Effective Mode	Address Register

Instruction fields
Register field Specifies any of the eight data registers.
Op-Mode field

Byte	Word	Long	Operation
000	001	010	(< Dn >) − (< ea >) → < Dn >
100	101	110	(< ea >) − (< Dn >) → < ea >

Effective address field Determines addressing mode:
(*a*) If the location specified is a source operand, then all addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	register number	d(An, Xi)	110	register number
An*	001	register number	Abs.W	111	000
(An)	010	register number	Abs.L	111	001
(An)+	011	register number	d(PC)	111	010
−(An)	100	register number	d(PC, Xi)	111	011
d(An)	101	register number	Imm	111	100

If the location specified is a destination operand, then only alterable memory addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	d(An, Xi)	110	register number
An	—	—	Abs.W	111	000
(An)	010	register number	Abs.L	111	001
(An)+	011	register number	d(PC)	—	—
−(An)	100	register number	d(PC, Xi)	—	—
d(An)	101	register number	Imm	—	—

Notes: 1. If the destination is a data register, then it cannot be specified by using the destination < ea > mode, but must use the destination Dn mode instead.
2. SUBA is used when the destination is an address register. SUBI and SUBQ are used when the source is immediate data. Most assemblers automatically make this distinction.
3. For byte size data register direct is not allowed.

FIGURE 10.28 (Cont.)

ADD Add Binary

Operation: (Source) + (Destination) → Destination

Assembler: ADD < ea >, Dn
Syntax: ADD Dn, < ea >

Attributes: Size = (Byte, Word, Long)

Description: Add the source operand to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word, or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition codes

X	N	Z	V	C
*	*	*	*	*

N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.
X Set the same as the carry bit.

Instruction format

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	1	Register	Op-Mode	Effective Mode	Address Register

Instruction fields
Register field Specifies any of the eight data registers.
Op-Mode field

Byte	Word	Long	Operation
000	001	010	(< Dn >) + (< ea >) → < Dn >
100	101	110	(< ea >) + (< Dn >) → < ea >

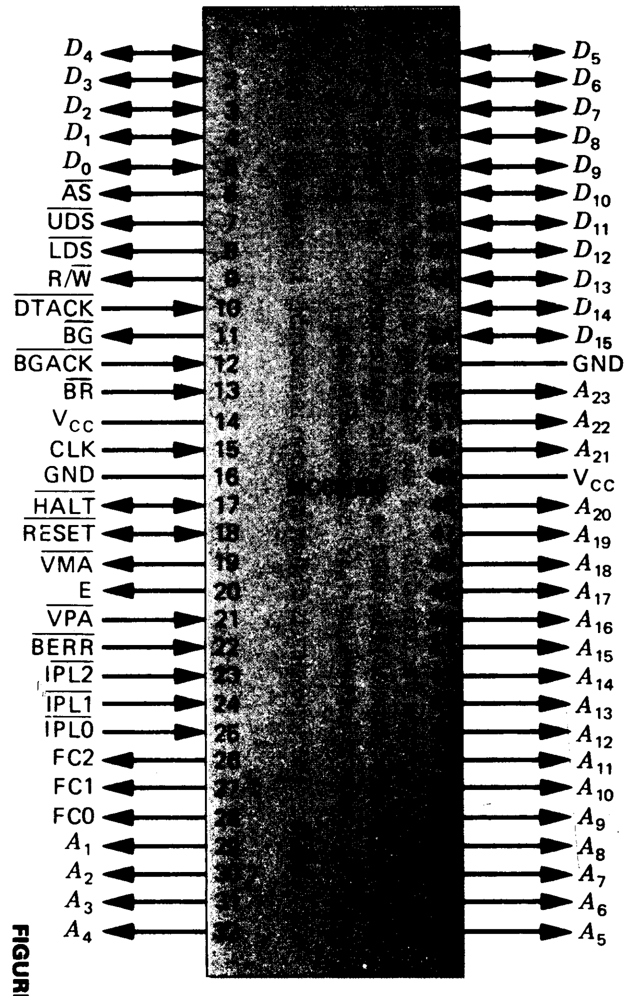
Effective address field Determines addressing mode:
(*a*) If the location specified is a source operand, then all addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	register number	d(An, Xi)	110	register number
An*	001	register number	Abs.W	111	000
(An)	010	register number	Abs.L	111	001
(An)+	011	register number	d(PC)	111	010
−(An)	100	register number	d(PC, Xi)	111	011
d(An)	101	register number	Imm	111	100

(*b*) If the location specified is a destination operand, then only alterable memory addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	d(An, Xi)	110	register number
An	—	—	Abs.W	111	000
(An)	010	register number	Abs.L	111	001
(An)+	011	register number	d(PC)	—	—
−(An)	100	register number	d(PC, Xi)	—	—
d(An)	101	register number	Imm	—	—

Notes: 1. If the destination is a data register, then it cannot be specified by using the destination < ea > mode, but must use the destination Dn mode instead.
2. ADDA is used when the destination is an address register. ADDI and ADDQ are used when the source is immediate data. Most assemblers automatically make this distinction.
3. Word and Long only.



Pin Name	Description
D0-D15	Data bus
A1-A23	Address bus
$\overline{AS}$	Address strobe
$R/\overline{W}$	Read/Write control
$\overline{UDS}$, $\overline{LDS}$	Upper, lower data strobes
$\overline{DTACK}$	Data transfer acknowledge
FC0, FC1, FC2	Function code (status) outputs
$\overline{IPL0}$, $\overline{IPL1}$, $\overline{IPL2}$	Interrupt requests
$\overline{BERR}$	Bus error
$\overline{HALT}$	Halt processor
$\overline{RESET}$	Reset processor or reset external devices
CLK	Clock
$\overline{BR}$	Bus request
$\overline{BG}$	Bus grant
$\overline{BGACK}$	Bus grant acknowledge
E	Enable (clock) output
$\overline{VMA}$	Valid memory address
$\overline{VPA}$	Valid peripheral address
V_{CC}, GND	Power (+5 V) and Ground

FIGURE 10.29

Pin-out for 68000.

]

the address of the operand." In effect, for $(A_n)@$, the number in A_n is the address of the address of the operand.[14]

Figure 10.27 shows the addressing modes for the 68000. These cover most of the conventional modes for addressing and offer considerable options to the programmer.

Several selected instructions from the 68000's large instruction set are shown in Fig. 10.28. A large number of instructions are available, including most conventional instructions.

Notice that addresses are generated in 32-bit registers and are complete addresses. The pin-out in Fig. 10.29 shows that all 24 address lines and 16-bit data lines are externally available. And there is no multiplexing of these lines because the 68000 has a 64-pin package which provides for the necessary connections.

The 68000 microprocessor chip and its support chips provide a powerful instruction repertoire and high-speed operation of programs. They are widely used in everything from personal computers to communication and control systems.

QUESTIONS

10.1 Discuss the advantages and disadvantages of the following addressing strategies in a microcomputer: (*a*) Paging, (*b*) indirect addressing, (*c*) index registers.

10.2 Describe some advantages and disadvantages of multiple-accumulator (general-purpose registers) versus single-accumulator computer architecture. Include effects on instruction word length, convenience in programming, etc.

10.3 (*a*) The 6100 series uses the original paging scheme for addressing. The PDP-11 and other computers use a relative address or sliding page. Discuss the advantages and disadvantages of these two techniques.

(*b*) What is the obvious problem in indirect addressing of 8K or larger memories which arises in the 6100 but does not occur for 16-bit-word computers such as the NOVA, PDP-11, or Varian?

10.4 Discuss the desirability of the following computer architectural features for a microcomputer to be used as a traffic light controller: (*a*) Paging of memory, (*b*) floating-point arithmetic, (*c*) indirect addressing.

10.5 The following is a short program in assembly language for the 6100:

```
7ØØ
         CLA    CLL     /CLEARS AC AND LINK
         TAD    DAT1
GO,      ISZ    DAT2
         JMP    GO
         RAL            /RIGHT SHIFT AC AND LINK
         JMP    GO
DAT1,    ØØ77
DAT2,    Ø
$
```

[14]Alternate ways to write $(A_n)@$ would be $A_n@@$ or $((A_n))$.

After this program has been assembled and loaded, it appears as follows in memory (all digits are octal), except that the contents of two addresses in memory need to be filled in:

ADDRESS NUMBER	CONTENTS
Ø7ØØ	
Ø7Ø1	
Ø7Ø2	23Ø7
Ø7Ø3	53Ø2
Ø7Ø4	7ØØ4
Ø7Ø5	53Ø2
Ø7Ø6	ØØ77
Ø7Ø7	ØØØØ

Supply the contents of (octal) locations 700 and 701 in memory.

10.6 The program in Question 10.5 rotates a sequence of 0s and 1s through the accumulator link.

(*a*) How many 0s and how many 1s? In what order?

(*b*) How many instructions must be executed for a complete cycle of a given 0 and 1 pattern?

10.7 A microcomputer has a bus with a single INTERRUPT line which an external device is to raise when it wishes to be serviced. The bus is controlled by the microcomputer's CPU chip. Explain the CPU's problem in determining which input-output device(s) generated an interrupt, and discuss two possible solutions.

10.8 The following is a short program for the 6800 which was written to compute $Y = 32(9 - 7)$ and store it. The programmer then converted the program into hexadecimal and is now prepared to enter it into the computer. There are several mistakes in the program. Find as many as possible and explain each.

#	ADDRESS	OP	OPER	LABEL	MNEMONICS	OPERAND	COMMENTS
01	0200	4F		START	CLRA		; CLEAR REGISTER A
02	0201	86	OE		LDA	X	; LOAD X INTO REGISTER A
03	0203	43			COMA		; COMPLEMENT X
04	0204	8B	09		ADDA(IM)	#09	; ADD 9
05	0206	CE	05		LDX(IM)	#05	; LOAD INDEX REGISTER WITH 5
06	0208	49		LOOP	ROLA		; ROTATE LEFT 1 BIT (MULTIPLY BY 2)
07	0209	09			DEX		; DECREMENT INDEX REGISTER
08	020A	26	FD		BNE	LOOP	; ROTATE AGAIN IF INDEX REGISTER $\neq$ 0
09	020C	97	OF		STAA	Y	; AFTER MULTIPLYING BY $2^5 = 32$, STORE THE RESULT IN Y
10	020E	07		X	DATA	1 BYTE	
11	020F	00		Y	DATA	1 BYTE	; DATA IN THIS LOCATION WILL BE REPLACED BY VALUE OF Y

10.9 A short program has been written for the 6800 to determine the number of bytes in a table which have 1s in their sign bits. The number of elements in the table is stored at location 50, and the table begins in location 60. The number of bytes with 1 in the sign bit is to be stored in location 51. Modify this program so that the number of nonzero bytes with a 0 in the sign bit is stored in location 55.

```
        LDX     #$60     /LOAD I REGISTER
        CLRB
LOOKN   LDAA    X        /CHECK FOR NEGATIVES
        BPL     HOUS
        INCB
HOUS    INX
        DEC     $50
        BNE     LOOKN    /DONE?
        STAB    $51
```

10.10 A two-address computer has a large IC memory with a 0.5-μs memory cycle time and a small high-speed memory with a 0.25-μs memory cycle time. An addition instruction word looks like this:

ADD	1st address	2d address

The first address refers to the small high-speed memory and the second address to the large memory. The sum is placed in the high-speed memory at the first address. How long will it take to perform an addition instruction? Why?

10.11 Using the index instructions given in Sec. 10.9, write a program that adds 40 numbers located in the memory, starting at address 200 and storing the sum in register 300.

10.12 If we use three binary digits in the instruction word to indicate which index register is used, or if one is to be used, then how many index registers can be used in the machine?

10.13 Modify the program in Sec. 10.9 so that the numbers located at memory addresses 353 through 546 are added and stored at address 600.

10.14 Modify the program in Sec. 10.9 so that the numbers located at addresses 300 through 305 are multiplied and the product is stored at address 310.

10.15 The use of paging enables the relocation of programs in the memory without extensive modification of the addresses in the program. Explain why.

10.16 Explain how paging and indirect addressing can be useful in relocating subprograms when a program is rewritten. What are some disadvantages of paging and indirect addressing?

10.17 Discuss the architecture of the 6800 versus the 8080 microprocessors.

10.18 Compare the PDP-11 addressing to the 8080 addressing modes.

10.19 Explain how paging as an addressing technique can be useful in relocating programs. Then explain how small pages can sometimes force programmers to "think in segments." Are the above characteristics desirable or undesirable?

10.20 Explain the addressing of pages in the 6100 series and the displacement-plus-instruction-location addressing technique used by the PDP-11. Why do you think systems architects have elected to use these systems?

10.21 The following is a section of program and memory contents from an assembly listing for a 6100. The programmer who wrote this contends that after the instruction at location 255_8 is executed, the location 256_8 in memory will contain the difference $A - B$ of the numbers A and B at locations 260_8 and 2053_8. Is the programmer correct? Give the reason for your answer, explaining the program operation.

```
                          25Ø

          Ø25Ø     73ØØ              CLA      CLL
          Ø251     1657              TAD      I         F
          Ø252     7Ø4Ø              CMA
          Ø253     126Ø              TAD      A
          Ø254     3256              DCA      C
          Ø255     74Ø2              HLT
          Ø256     ØØØ4              C,       Y
          Ø257     2Ø53              F,       2Ø53
          Ø26Ø     ØØØ5              A,       5

          2Ø53     ØØØ6     2Ø53     B,       6
          \___/    \___/             \____________/

         memory   contents          program written
         address     of               in assembly
                   memory              language
```

10.22 Sketch, describe, and discuss the merits of one of the machine architectures that have been presented, or of any other machine with which you are familiar (or with which you would like to be familiar—including any of your own "ideal" designs).

10.23 A real-time system for manufacturing control is to be constructed using a computer. The system is to perform two functions:

(*a*) The computer has to automatically test cameras as they are manufactured. This involves, among other things, reading 1000 values each second from several A-to-D converters and checking to see whether the values are within prescribed limits.

(*b*) The computer has to service four terminals which run inquiries against the data base maintained on the cameras, and it also has to run some Fortran and Cobol programs.

Give an architecture for the computer to be used.

10.24 Show how a recursive subroutine call can erase the return location planted at the beginning of a subroutine when the 6100 JMS instruction is used.

10.25 Show how the 6800 microprocessor subroutine call will not destroy the return address in a recursive subroutine call because of the use of the stack.

10.26 Show how the 8080 jump to a subroutine will not destroy the return location in a recursive subroutine call because of the stack.

10.27 Write a program in assembly language for the 6800 microprocessor which will multiply Y by 16 and then subtract 15 from the result. (Ignore overflows.)

10.28 Write a program in assembly language for the 8080 microprocessor which will multiply a number X by 32 and then subtract 14 from the result. (Ignore overflows.)

10.29 Write a subroutine for the 8080 microprocessor which will double the number in the accumulator and then subtract 5. (Ignore overflows.)

10.30 Write a subroutine for the 6800 which will add accumulator A to accumulator B, store the result in accumulator A, and then subtract 12 from this result, storing that in accumulator B. (Ignore overflows.)

10.31 Write a subroutine call for the 6800 which will utilize the subroutine written in Question 10.30. Before this subroutine call, place 13 in accumulator A and 23 in accumulator B.

10.32 Discuss PDP-11 addressing [where the addressing mode (or modes) is carried in the address section] versus placing that information in the OP code.

10.33 Show the mode information in the two 3-bit fields in a PDP-11 instruction word which uses both source and destination registers in a direct addressing mode.

10.34 Write a program for the PDP-11 which will add the number in R_1 to the number in R_3 and then store this number at location 63_8 in the memory.

10.35 Why is it not a good idea to store data at location 0 in the 6100 memory?

10.36 Explain why placing often used data in the first 256 words of the memory in a 6800 will shorten some instruction words.

10.37 Explain the following sentence: The 6800 has one index register, the PDP-11 can use any general register as an index register, and the 8080 has no index register.

10.38 Explain how the auto-increment and auto-decrement instruction modes in the PDP-11 can be useful in processing tables.

10.39 In the 8080 an interrupt is serviced as follows. The device which is being serviced places on the data lines of the 8080 bus an instruction word which is a special instruction, called RST. Three bits of this instruction give the address of the next instruction to be executed. The interrupting device places the correct 3 bits in the section of the RST instruction on the bus, and the 8080 then takes the next instruction from that location. (See OP-code description of RST.) In that location is a jump to the subroutine which actually services the device generating the interrupt. Discuss the advantages and disadvantages of this procedure.

10.40 The IBM series of computers uses a priority delegation scheme where interrupt devices are interconnected as shown below. When an interrupt service is issued, the leftmost point of the "daisy-chain" wire is raised. If a device wishes to be serviced, it does not forward this 1 to the device on the right. If it does not wish to be serviced, it forwards this 1 to the device on the right. Each device in turn makes this decision, either passing the 1 to the right or passing a 0. (A 0 on