10

MOTOROLA MC68000

This chapter describes the basic features of Motorola's MC68000 (16-bit microprocessor). The addressing modes, instruction set, I/O, and system design concepts of the MC68000 are covered in detail.

Motorola's original MC68000 was designed using HMOS technology. Motorola's MC68000 is replaced it by a lower power MC68HC000, which is designed using HCMOS technology. The MC68HC000 is equivalent to the MC68000 in all aspects except that the MC68HC000 is designed using HCMOS whereas the MC68000 was designed using HMOS technology. This means that unlike the MC68000, the unused inputs of the MC68HC000 should not be kept floating, they should be connected to +5 V, ground, or outputs of other chips as appropriate. Also, note that an HCMOS output can drive 10 LSTTL inputs. However, an LSTTL output is not guaranteed to provide HCMOS input voltage. Hence, the HCT gates may be required when driving HC inputs. The MC 68HC000 has the same registers, addressing modes, instruction set, pins and signals, and I/O capabilities as the MC68000. The term "MC68000" will be used interchangeably with the term "MC68HC000" throughout this chapter.

The MC68HC000, implemented in HCMOS, is applicable to designs for which the following considerations are relevant:

- The MC68HC000 completely satisfies the input/output drive requirements of HCMOS logic devices.
- The MC68HC000 provides an order of magnitude reduction in power dissipation when compared to the HMOS MC68000.
- The minimum operating frequency of the MC68HC000 is 4 MHz.

Although the MC68HC000 is implemented with input protection diodes, care should be exercised to ensure that the maximum input voltage specification (-0.3 V to +6.5 V) is not exceeded.

10.1 Introduction

The MC68000 is Motorola's first 16-bit microprocessor. Its address and data registers are all 32 bits wide, and its ALU is 16 bits wide. The 68000 requires a single 5-V supply. The processor can be operated from a maximum internal clock frequency of 25 MHz. The 68000 is available in several frequencies, including 4, 6, 8, 10, 12.5, 16.67, and 25 MHz. The 68000 does not have on-chip clock circuitry and therefore, requires an external crystal oscillator or clock generator/driver circuit to generate the clock.

The 68000 has several different versions, which include the 68008, 68010, and 68012. The 68000 and 68010 are packaged either in a 64-pin DIP (dual in-line package)

with all pins assigned or in a 68-pin quad pack or PGA (pin grid array) with some unused pins. The 68000 is also packaged in 68-terminal chip carrier. The 68008 is packed in a 48-pin dual in-line package, whereas the 68012 is packed in an 84-pin grid array. The 68008 provides the basic 68000 capabilities with inexpensive packaging. It has an 8-bit data bus, which facilitates the interfacing of this chip to inexpensive 8-bit peripheral chips. The 68010 provides hardware-based virtual memory support and efficient looping instructions. Like the 68000, it has a 16-bit data bus and a 24-bit address bus. The 68012 includes all the 68010 features with a 31-bit address bus. The clock frequencies of the 68008, 68010, and 68012 are the same as those of the 68000. The following table summarizes the basic differences among the 68000 family members:

	68000	68008	68010	68012
Data size (bits)	16	8	16	16
Address bus size (bits)	24	20	24	31
Virtual memory	No	No	Yes	Yes
Control registers	None	None	3	3
Directly addressable memory (bytes)	16 MB	1 MB	16 MB	2 GB

To implement operating systems and protection features, the 68000 can be operated in two modes: supervisor and user. The supervisor mode is also called the "operating system mode." In this mode, the 68000 can execute all instructions. The 68000 operates in one of these modes based on the S bit of the status register. When the S bit is 1, the 68000 operates in the supervisor mode; when the S bit is 0, the 68000 operates in the user mode.

Table 10.1 lists the basic differences between the 68000 user and supervisor modes. From Table 10.1, it can be seen that the 68000 executing a program in the supervisor mode can enter the user mode by modifying the S bit of the status register to 0 via an instruction. Instructions such as `MOVE` to SR, `ANDI` to SR, and `EORI` to SR can be used to accomplish this. On the other hand, the 68000 executing a program in the user mode can enter the supervisor mode only via recognition of a trap, reset, or interrupt. Note that, upon hardware reset, the 68000 operates in the supervisor mode and can execute all instructions. An attempt to execute *privileged instructions* (instructions that can only be executed in the supervisor mode) in the user mode will automatically generate an internal interrupt (trap) by the 68000.

The logical level in the 68000 function code pin (FC2) indicates to the external devices whether the 68000 is currently operating in the user or supervisor mode. The 68000 has three function code pins (FC2, FC1, and FC0), which indicate to the external devices whether the 68000 is accessing supervisor program/data or user program/data or performing an interrupt acknowledge cycle.

The 68000 can operate on five different data types: bits, 4-bit binary-coded decimal (BCD) digits, bytes, 16-bit words, and 32-bit long words. The 68000 instruction set includes 56 basic instruction types. With 14 addressing modes, 56 instructions, and 5 data types, the 68000 contains over 1000 op-codes. The fastest instruction is one that copies the contents of one register into another register. It is executed in 500 ns at an 8-MHz clock rate. The slowest instruction is 32-bit by 16-bit divide, which in executed in 21.25 μs at 8 MHz. The 68000 has no I/O instructions. Thus, the I/O is memory mapped.

TABLE 10.1 68000 User and Supervisor Modes

	Supervisor Mode	*User Mode*
Enter mode by	Recognition of a trap, reset, or interrupt	Clearing status bit S
System stack pointer	Supervisor stack pointer	User stack pointer
Other stack pointers	User stack pointer and registers A0-A6	registers, A0-A6

Instructions available	All including:
	`STOP`
	`RESET`
	`MOVE` to/from SR
	`ANDI` to/from SR
	`ORI` to/from SR
	`EORI` to/from SR
	`MOVE` USP to (An)
	`MOVE` to USP
	`RTE`
		All except those listed under Supervisor mode

Function code pin FC2	1	0

Hence, `MOVE` instructions between a register and a memory address are also used as I/O instructions. The MC68000 is a general-purpose register-based microprocessor. Although the 68000 PC is 32 bits wide, only the low-order 24 bits are used. Because this is a byte-

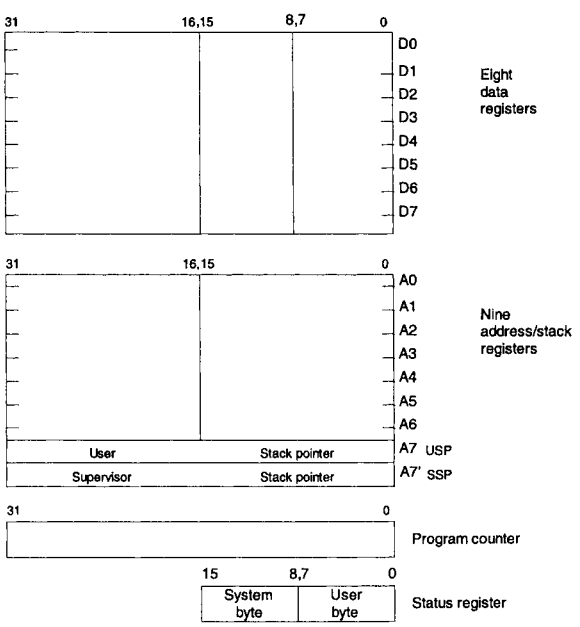


FIGURE 10.1 MC68000 programming model

addressable machine, it follows that the 68000 microprocessor can directly address 16 MB of memory. Note that symbol [ ] is used in the examples throughout this chapter to indicate the contents of a 68000 register or a memory location.

10.2 68000 Registers

Figure 10.1 shows the 68000 registers. This microprocessor includes eight 32-bit data registers (D0–D7) and nine 32-bit address registers (A0–A7 plus A7'). Data registers normally hold data items such as 8-bit bytes, 16-bit words, and 32-bit long words. An address register usually holds the memory address of an operand; A0-A6 can be used as 16- or 32-bit. Because the 68000 uses 24-bit addresses, it discards the uppermost 8 bits (bits 24–31) while using the address registers to hold memory addresses. The 68000 uses A7 or A7' as the user or supervisor stack pointer (USP or SSP), respectively, depending on the mode of operation.

The 68000 status register is composed of two bytes: a user byte and a system byte (Figure 10.2). The user byte includes typical condition codes such as C, V, N, Z, and X. The meaning of the C, V, N, and Z flags is obvious. Let us explain the meaning of the X bit. Note that the 68000 does not have any `ADDC` or `SUBC` instructions; rather, it has `ADDX` and `SUBX` instructions.

Because the flags C and X are usually affected in an identical manner, one can use `ADDX` or `SUBX` to reflect the carries or borrows in multiprecision arithmetic. The contents of the system byte include a 3-bit interrupt mask (I2, I1, I0), a supervisor flag (S), and a trace flag (T). When the supervisor flag is 1, then the system operates in the supervisor mode; otherwise, the user mode of operation is assumed. When the trace flag is set to 1, the processor generates a trap (internal interrupt) after executing each instruction. A debugging routine can be written at the interrupt address vector to display registers and/or memory after execution of each instruction. Thus, this will provide a single-stepping facility. Note that the trace flag can be set to one in the supervisor mode by executing the instruction ORI# $8000, SR.

The interrupt mask bits (I2, I1, I0) provide the status of the 68000 interrupt pins $\overline{IPL2}$, $\overline{IPL1}$ and $\overline{IPL0}$. I2 I1 I0 = 000 indicates that all interrupts are enabled. I2 I1 I0 = 111 indicates that all maskable interrupts except the nonmaskable interrupt (Level 7) are disabled. The other combinations of I2, I1, and I0 provide the maskable interrupt levels. Note that the signals on the $\overline{IPL2}$, $\overline{IPL1}$ and $\overline{IPL0}$ pins are inverted internally and then compared with I2, I1, and I0, respectively.

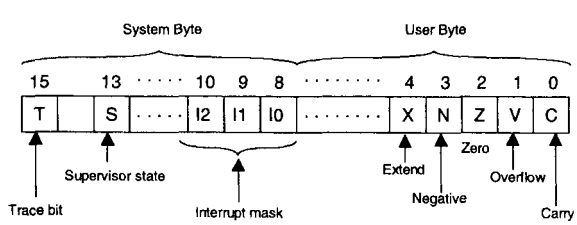


FIGURE 10.2 68000 status register

	15 8	7 0	
Address = N	Byte 0	Byte 1	$N+1$
$N+2$	Byte 2	Byte 3	$N+3$

(a) 68000 Words Stored in Bytes (4 Bytes)

	15 0	
Address = N	Word 0	$N+1$
$N+2$	Word 1	$N+3$
$N+4$	Word 2	$N+5$

(b) 68000 Word Structure (3 Words)

	15 0	
Address = N	Long word 0 (H)	$N+1$
$N+2$	Long word 0 (L)	$N+3$
$N+4$	Long word 1 (H)	$N+5$
$N+6$	Long word 1 (L)	$N+7$

(c) 68000 Long Word Structure (2 Long Words)

FIGURE 10.3 68000 addressing structure (N is an even number)

10.3 68000 Memory Addressing

The MC68000 supports bytes (8 bits), words (16 bits), and long words (32 bits) as shown in Figure 10.3. Byte addressing includes both odd and even addresses (0, 1, 2, 3, ...), word addressing includes only even addresses in increments of 2 (0, 2, 4, ...), and long word addressing contains even addresses in increments of 4 (0, 4, 8, ...). As an example of 68000 addressing structure, consider `MOVE.L D0,$506080`. If [D0] = $07F12481, then after this `MOVE`, [$506080] = $07, [$506081] = $F1, [$506082] = $24, and [$506083] = $81. In the 68000, all instructions must be located at even addresses for byte, word, and long word instructions; otherwise, the 68000 generates an internal interrupt. The size of each 68000 instruction is even multiples of a byte. This means that once the programmer writes a program starting at an even address, all instructions are located at even addresses after assembling the program. For byte instructions, data can be located at even or odd addresses. On the other hand, data for word and long word instruction must be located at even addresses; otherwise the 68000 generates an internal interrupt.

Note that in 68000 for word and long word data, the low-order address stores the high-order byte of a number. This is called *Big-endian byte ordering.*

10.4 68000 Addressing Modes

The 14 addressing modes of the 68000 shown in Table 10.2 can be divided into 6 basic groups: register direct, address register indirect, absolute, program counter relative, immediate, and implied.

As mentioned, the 68000 has three types of instructions: no operand, single operand, and double operand. The single-operand instructions contain the effective address (EA) in the operand field. The EA for these instructions is calculated by the 68000 using the addressing mode used for this operand. In the case of two-operand instructions, one of the operands usually contains the EA and the other operand is usually a register or memory location. The EA in these instructions is calculated by the 68000 based on the addressing

TABLE 10.2 68000 Addressing Modes

Addressing Mode	*Generation*	*Assembler Syntax*
• Register direct addressing		
Data register direct	EA = Dn	Dn
Address register direct	EA = An	An
• Address register indirect addressing		
Register indirect	EA = (An)	(An)
Postincrement register indirect	EA = (An), An ← An + N	(An)+
Predecrement register indirect	An ← An - N, EA = (An)	-(An)
Register indirect with offset	EA = (An) + d_{16}	d(An)
Indexed register indirect with offset	EA = (An) + (Ri) + d_8	d(An, Ri)
• Absolute data addressing		
Absolute short	EA = (Next word)	xxxx
Absolute long	EA = (Next two words)	xxxxxxxx
• Program counter relative addressing		
Relative with offset	EA = (PC) + d_{16}	d
Relative with index and offset	EA = (PC) + (Ri) + d_8	d(Ri)
• Immediate data addressing		
Immediate	DATA = Next word(s)	#xxxx
Quick immediate	Inherent data	#xx
• Implied addressing		
Implied register	EA = SR, USP, SP, PC	

Notes:

EA	= effective address	USP	= user stack pointer
An	= address register	d_8	= 8-bit signed offset (displacement)
Dn	= data register	d_{16}	= 16-bit signed offset (displacement)
Ri	= address or data register used as index register	N	= 1 for byte, 2 for words, and 4 for long words
SR	= status register	( )	= contents of
PC	= program counter	←	= replaces
SP	= active system stack pointer		

mode used for the EA.

Some two-operand instructions have the EA in both operands. This means that the operands in these instructions use two addressing modes. Note that the 68000 address registers do not support byte-sized operands. Therefore, when an address register is used as a source operand, either the low-order word or the entire long word operand is used, depending on the operation size. When an address register is used as the destination operand, the entire register is affected regardless of operation size. If the operation size is a word, an address register in the destination operand is sign-extended to 32 bits after the operation is performed. Data registers, on the other hand, support data operands of byte,

word, or long word size.

To identify the operand size of an instruction, the following notation is placed after a 68000 mnemonic: .B for byte, .W or none (default) for word, and .L for long word. For example,

```
ADD.B D0, D1  ;    [D1]low byte     ← [D0]low byte     + [D1]low byte
ADD.W D0, D1  ;    [D1]low 16 bit   ← [D0]low 16 bit   + [D1]low 16 bit
ADD.L D0, D1  ;    [D1]32 bits      ← [D1]32 bits      + [D0]32 bits
```

10.4.1 Register Direct Addressing

In this mode, the eight data registers (D0–D7) or seven address registers (A0–A6) contain the data operand. For example, consider `ADD.W $005000,D0`. The destination operand of this instruction is in data register direct mode. Now, if [005000] = 0002_{16} and [D0.W] = 0003_{16}, then after execution of `ADD $005000,D0`, the contents of D0.W = 0002 + 0003 = 0005. Note that in this instruction, the $ symbol is used by Motorola to represent hexadecimal numbers. Also note that instructions are not available for byte operations using address registers.

10.4.2 Address Register Indirect Addressing

There are five different types of address register indirect mode. In this mode, an address register contains the effective address. For example, consider `CLR.W(A1)`. If [A1.L]=$00003000, then, after execution of `CLR.W(A1)`, the 16-bit contents of memory location $003000 will be cleared to zero.

The postincrement address register indirect mode increments an address register by 1 for byte, 2 for word, and 4 for long word after it is used. For example, consider `CLR.L (A0)+`. If [A0] = 00005000_{16}, then after execution of `CLR.L (A0)+`, the 16-bit contents of each of the memory locations 005000_{16} and 005002_{16} are cleared to zero and [A0] = 00005000 + 4 = 00005004. The postincrement mode is typically used with memory arrays stored from LOW to HIGH memory locations. For example, to clear 1000_{16} words starting at memory location 003000_{16} and above, the following instruction sequence can be used:

```
        MOVE.W     #$1000,D0        ;  Load length of data into D0
        MOVEA.L    #$00003000,A0    ;  Load starting address into A0
REPEAT  CLR.W      (A0)+            ;  Clear a location pointed to
                                    ;  by A0 and increment A0 by 2
        SUBQ.W     #1,D0            ;  Decrement D0 by 1
        BNE.B      REPEAT           ;  Branch to REPEAT if Z = 0;
        ...                         ;  otherwise, go to next instruction
```

Note that the symbol # in the above is used by the Motorola assember to indicate the immediate mode. This will be discussed later in this section. Also, note that `CLR.W (A0)+` automatically points to the next location by incrementing A0 by 2 after clearing a memory location.

The predecrement address register indirect mode, on the other hand, decrements an address register by 1 for byte, 2 for word, and 4 for long word before using a register. For example, consider `CLR.W -(A0)`. If [A0] = $00002004, then the content of A0 is first decremented by 2—that is, [A0] = 00002002_{16}. The content of memory location 002002 is then cleared to zero. The predecrement mode is used with arrays stored from HIGH to LOW memory locations. For example, to clear 1000_{16} words starting at memory location 004000_{16} and below, the following instruction sequence can be used:

```
        MOVE.W     #$1000,D0        ;  Load length of data into D0
```

```
        MOVEA.L     #$00004002,A0  ;   Load starting address plus 2 into A0
REPEAT  CLR.W       -(A0)          ;   Decrement A0 by 2 and clear memory
                                   ;   location addressed by A0
        SUBQ.W      #1,D0          ;   Decrement D0 by 1
        BNE.B       REPEAT         ;  If Z = 0, branch to REPEAT
        ...                        ;   otherwise, go to next instruction
```

In this instruction sequence, `CLR.W -(A0)` first decrements A0 by 2 and then clears the location. Because the starting address is 004000_{16}, A0 must initially be loaded with 00004002_{16}. It should be pointed out that the predecrement and postincrement modes can be combined in a single instruction. A typical example is `MOVE.W (A5)+,-(A3)`.

The two other address register modes provide accessing of the tables by allowing offsets and indexes to be included with an indirect address pointer. The *address register indirect with offset mode* determines the effective address by adding a 16-bit signed integer to the contents of an address register. For example, consider `MOVE.W $10(A5),D3` in which the source operand is in address register indirect with offset mode. If [A5] = 00002000_{16} and $[002010]_{16} = 0014_{16}$, then, after execution of `MOVE.W $10(A5),D3`, register D3.W will contain 0014_{16}.

The *indexed register indirect with offset mode* determines the effective address by adding an 8-bit signed integer and the contents of a register (data or address register) to the contents of an address (base) register. This mode is usually used when the offset from the base address register needs to be varied during program execution. The size of the index register can be a signed 16-bit integer or an unsigned 32-bit value. As an example, consider `MOVE.W $10(A4,D3.W),D4` in which the source is in the indexed register indirect with offset mode. Note that in this instruction A4 is the base register and D3.W is the 16-bit index register (sign-extended to 32 bits). This register can be specified as 32 bits by using D3.L in the instruction, and 10_{16} is the 8-bit offset that is sign-extended to 32 bits. If [A4] = 00003000_{16}, [D3.W] = 0200_{16}, and $[003210_{16}] = 0024_{16}$, then this `MOVE` instruction will load 0024_{16} into the low 16 bits of register D4.

The address register indirect with offset mode can be used to access a single table. The offset (maximum 16 bits) can be the starting address of the table (fixed number), and the address register can hold the index number in the table to be accessed. Note that the starting address plus the index number provides the address of the element to be accessed in the table. For example, consider `MOVE.W $3400(A5),D1`. If A5 contains 04, then this `MOVE` instruction transfers the contents of 3404 (i.e. the fifth element, 0 being the first element) into the low 16 bits of D1. The indexed register indirect with offset mode, on the other hand, can be used to access multiple tables. Here, the offset (maximum 8 bits) can be the element number to be accessed. The address register pointer can be used to hold the starting address of the table containing the lowest starting address, and the index register can be used to hold the difference between the starting address of the table being accessed and the table with the lowest starting address. For example, consider three tables, with table 1 starting at 002000_{16}, table 2 at 003000_{16}, and table 3 at 004000_{16}. To transfer the seventh element (0 being the first element) in table 2 to the low 16 bits of register D0, the instruction `MOVE.W $06(A2, D1.W),D0` can be used, where [A2] = the starting address of the table with the lowest address (= 002000_{16} in this case) and $[D1]_{\text{low 16 bits}}$ = the difference between the starting address of the table being accessed and the starting address of the table with the lowest address = $003000_{16} - 002000_{16} = 1000_{16}$. Therefore, this `MOVE` instruction will transfer the contents of address 003006_{16} (the seventh element in table 2) to register D0. The indexed register indirect with offset mode can also be used to access two-dimensional arrays such as matrices.

10.4.3 Absolute Addressing

In this mode, the effective address is part of the instruction. The 68000 has two modes: absolute short addressing, in which a 16-bit address is used (the address is sign-extended to 24 bits before use), and absolute long addressing, in which a 24-bit address is used. For example, consider `ADD $2000,D2` as an example of the absolute short mode. If [$002000] = 0012_{16} and [D2.W] = 0010_{16}, then, after executing `ADD $2000,D2` , register D2.W will contain 0022_{16}. The absolute long addressing mode is used when the address size is more than 16 bits. For example, `MOVE.W $240000,D5` loads the 16-bit contents of memory location 240000_{16} into the low 16 bits of D5. The absolute short mode includes an address ADDR in the range of 0 ≤ ADDR ≤ $7FFF or $FF8000 ≤ ADDR ≤ $FFFFFF. Note that a single instruction may use both short and long absolute modes, depending on whether the source or destination address is less than, equal to, or greater than the 16-bit address. A typical example is `MOVE.W $500002,$1000`. Also, note that the absolute long mode must be used for `MOVE` to or from address $008000. For example, `MOVE.W` $8000,D1 will move the 16-bit contents of location $FF8000 to D1 while `MOVE.W` $008000,D1 will transfer the 16-bit contents of address $008000 to D1.

10.4.4 Program Counter Relative Addressing

The 68000 has two program counter relative addressing modes: relative with offset and relative with index and offset. In the relative with offset mode, the effective address is obtained by adding the contents of the current PC with a signed 16-bit displacement. This mode can be used when the displacement needs to be fixed during program execution. Typical branch instructions such as `BEQ`, `BRA`, and `BLE` use the relative with offset mode. This mode can also be used by some other instructions. For example, consider `ADD $30(PC),D5`, in which the source operand is in the relative with offset mode. Now suppose that the current PC contents is $002000, the content of 002030_{16} is 0005, and the low 16 bits of D5 contain 0010_{16}. Then, after execution of this `ADD` instruction, D5 will contain 0015_{16}.

In the relative with index and offset mode, the effective address is obtained by adding the contents of the current PC, a signed 8-bit displacement (sign-extended to 32 bits), and the contents of an index register (address or data register). The size of the index register can be 16 or 32 bits wide. For example, consider `ADD.W $4(PC,D0.W),D2`. If [D2] = 00000012_{16}, [PC] = 002000_{16}, $[D0]_{\text{low 16 bits}} = 0010_{16}$, and [002014] = 0002_{16}, then, after this `ADD`, $[D2]_{\text{low 16 bits}} = 0014_{16}$. This mode is used when the displacement needs to be changed during program execution by modifying the content of the Index register.

An advantage of the relative mode is that the destination address is specified relative to the address of the instruction after the instruction. Since the 68000 instructions with relative mode do not contain an absolute address, the program can be placed anywhere in memory which can still be excuted properly by the 68000. A program which can be placed anywhere in memory, and can still run correctly is called a "relocatable" program. It is a good practice to write relocatable programs.

10.4.5 Immediate Data Addressing

Two immediate modes are available with the 68000: immediate and quick immediate modes. In immediate mode, the operand data is constant data, which is part of the instruction. For example, consider `ADDI.W #$0005,D0`. If [D0.W] = 0002_{16}, then, after this `ADDI` instruction, [D0.W] = $0002_{16} + 0005_{16} = 0007_{16}$. Note that the # symbol is used by Motorola to indicate the immediate mode. Quick immediate (ADD or SUBTRACT) mode allows

TABLE 10.3 68000 Addressing Modes – Functional Categories

Addressing Modes	*Addressing Category*			
	Data	*Memory*	*Control*	*Alterable*
Data register direct	X	-	-	X
Address register direct	-	-	-	X
Address register indirect	X	X	X	X
Address register indirect with postincrement	X	X	-	X
Address regisiter indirect with predecrement	X	X	-	X
Address register indirect with displacement	X	X	X	X
Address register indirect with index	X	X	X	X
Absolute short	X	X	X	X
Absolute long	X	X	X	X
Program counter with displacement	X	X	X	-
Program counter with index	X	X	X	-
Immediate	X	X	-	-

one to increment or decrement a register or a memory location (.B, .W, .L) by a number from 0 to 7. For example, `ADDQ.B #1,D0` increments the low 8-bit contents of D0 by 1. Note that immediate data, 1 is inherent in the instruction. That is, data 0 to 7 is contained in the three bits of the instruction. Note that ADDQ.B #0,Dn is similar to NOP instruction.

10.4.6 Implied Addressing

The instructions using implied addressing mode do not require any operand, and registers such as PC, SP, or SR are referenced in these instructions. For example, `RTS` returns to the main program from a subroutine by placing the return address into PC using the PC implicitly.

It should be pointed out that in the 68000 the first operand of a two-operand instruction is the source and the second operand is the destination. Recall that in the case of the 8086, the first operand is the destination and the second operand is the source.

10.5 Functional Categories Of 68000 Addressing Modes

All of the 68000 addressing modes in Table 10.2 can be further divided into four functional categories as shown in Table 10.3.

- ***Data Addressing Mode.*** An addressing mode is said to be a data addressing mode if it references data objects. For example, all 68000 addressing modes except the address register direct mode fall into this category.
- ***Memory Addressing Mode.*** An addressing mode capable of accessing a data item stored in memory is classified as a memory addressing mode. For example, the data and address register direct addressing modes cannot satisfy this definition.
- ***Control Addressing Mode.*** This refers to an addressing mode that has the ability to access a data item stored in memory without the need to specify its size. For example, all 68000 addressing modes except the following are classified as control addressing

TABLE 10.4 Some of the 68000 Instructions affecting Conditional codes.

Instruction	X	N	Z	V	C
ABCD	✓	U	✓	U	–
ADD, ADDI, ADDQ, ADDX	✓	✓	✓	✓	✓
AND, ANDI	–	✓	✓	0	0
ASL, ASR	✓	✓	✓	✓	✓
BCHG, BCLR, BSET, BTST	–	–	✓	–	–
CHK	–	✓	U	U	U
CLR	–	0	1	0	0
CMP, CMPA, CMPI, CMPM	–	✓	✓	✓	✓
DIVS, DIVU	–	✓	✓	✓	0
EOR, EORI	–	✓	✓	0	0
EXT	–	✓	✓	0	0
LSL, LSR	✓	✓	✓	0	✓
MOVE (ea), (ea)	--	✓	✓	0	0
MOVE TO CCR	✓	✓	✓	✓	✓
MOVE TO SR	✓	✓	✓	✓	✓
MOVEQ	–	✓	✓	0	0
MULS, MULU	–	✓	✓	0	0
NBCD	✓	U	✓	U	✓
NEG, NEGX	✓	✓	✓	✓	✓
NOT	–	✓	✓	0	0
OR, ORI	–	✓	✓	0	0
ROL, ROR	–	✓	✓	0	✓
ROXL, ROXR	✓	✓	✓	0	✓
RTE, RTR	✓	✓	✓	✓	✓
SBCD	✓	U	✓	U	✓
STOP	✓	✓	✓	✓	✓
SUB, SUBI, SUBQ, SUBX	✓	✓	✓	✓	✓
SWAP	–	✓	✓	0	0
TAS	–	✓	✓	0	0
TST	–	✓	✓	0	0

✓ Affected, – Not Affected, U Undefined

Note: ADDA, B_{cc}, and RTS do not affect flags.

modes: data register direct, address register direct, address register indirect with postincrement, address register indirect with predecrement, and immediate.

- ***Alterable Addressing Mode.*** If the effective address of an addressing mode is written into, then that mode is an alterable addressing mode. For example, the immediate and the program counter relative addressing modes will not satisfy this definition.

10.6 68000 Instruction Set

The 68000 instruction set contains 56 basic instructions. Table 10.4 lists some of the instructions affecting the condition codes. Appendices D and G provide the 68000 instruction execution times and the instruction set (alphabetical order), respectively.
The 68000 instructions can be classified into eight groups as follows:

TABLE 10.5 68000 Data Movement Instructions

Instruction	*Size*	*Comment*
EXG Rx, Ry	L	Exchange the contents of two registers. Rx or Ry can be any address or data register. No flags are affected.
LEA (EA), An	L	The effective address (EA) is calculated using the particular addressing mode used and then loaded into the address register. (EA) specifies the actual data to be loaded into An.
LINK An, #-displacement	Unsized	The current contents of the specified address register are pushed onto the stack. After the push, the address register is loaded from the updated SP. Finally, the 16-bit sign-extended displacement is added to the SP. A negative displacement is specified to allocate stack.
MOVE (EA), (EA)	B, W,L	(EA)s are calculated by the 68000 using the specific addressing mode used. (EA)s can be register or memory location. Therefore, data transfer can take place between registers, between a register and a memory location, and between different memory locations. Flags are affected. For byte-size operation, address register direct is not allowed. An is not allowed in the destination (EA). The source (EA) can be An for word or long word transfers.
MOVEM reg list, (EA) or (EA), reg list	W, L	Specified registers are transferred to or from consecutive memory locations starting at the location specified by the effective address.
MOVEP Dn, d (Ay) or d (Ay), Dn	W, L	Two (W) or four (L) bytes of data are transferred between a data register and alternate bytes of memory, starting at the location specified and incrementing by 2. The high-order byte of data is transferred first, and the low-order byte is transferred last. This instruction has the address register indirect with displacement only mode.
MOVEQ # data, Dn	L	This instruction moves the 8-bit inherent data into the specified data register. The data is then sign-extended to 32 bits.
PEA (EA)	L	Computes an effective address and then pushes the 32-bit address onto the stack.
SWAP Dn	W	Exchanges 16-bit halves of a data register.
UNLK An	Unsized	An → SP; (SP) + → An

- (EA) in LEA (EA), An can use all addressing modes except Dn, An, (An) +, – (An), and immediate.
- Destination (EA) in MOVE (EA), (EA) can use all modes except An, relative, and immediate.
- Source (EA) in MOVE (EA), (EA) can use all modes.
- Destination (EA) in MOVEM reg list, (EA) can use all modes except, An, (An)+, relative, and immediate.
- Source (EA) in MOVEM (EA), reg list can use all modes except Dn, An,– (An), and immediate.
- (EA) in PEA (EA) can use all modes except, An, (An)+, – (An), and immediate.

1. Data movement instructions
2. Arithmetic instructions
3. Logical instructions
4. Shift and rotate instructions
5. Bit manipulation instructions
6. Binary-coded decimal instructions
7. Program control instructions
8. System control instructions

10.6.1 Data Movement Instructions

These instructions allow data transfers from register to register, register to memory, memory to register, and memory to memory. In addition, there are also special data movement instructions such as MOVEM (move multiple registers). Typically, byte, word, or long word data can be transferred. A list of the 68000 data movement instructions is given in Table 10.5. Let us now explain the data movement instructions.

MOVE Instructions

The format for the basic MOVE instruction is MOVE.S (EA), (EA), where S = L, W, or B. (EA) can be a register or memory location, depending on the addressing mode used. Consider MOVE.B D3,D1, which uses the data register direct mode for both the source and destination. If [D3.B] = 05_{16} and [D1.B] = 01_{16}, then, after execution of this MOVE instruction, [D1.B] = 05_{16} and [D3.B] = 05_{16}.

There are several variations of the MOVE instruction. For example MOVE.W CCR, (EA) moves the contents of the low-order byte of SR (i.e., CCR) to the low-order byte of the destination operand; the upper byte of SR is considered to be zero. The source operand is a word. Similarly, MOVE.W (EA), CCR moves an 8-bit immediate number, or low-order 8-bit data, from a memory location or register into the condition code register; the upper byte is ignored. The source operand is a word. Data can also be transferred between (EA) and SR or USP (A7) using the following privileged instructions:

```
MOVE.W (EA), SR
MOVE.W SR, (EA)
MOVE.L A7, An
MOVE.L An, A7
```

MOVEA.W or.L (EA), An can be used to load an address into an address register. Word-size source operands are sign-extended to 32 bits. Note that (EA) is obtained by using an addressing mode. As an example, MOVEA.W #$2000, A5 moves the 16-bit word 2000_{16} into the low 16 bits of A5 and then sign-extends 2000_{16} to the 32-bit number 00002000_{16}. Note that sign extension means extending bit 15 of 2000_{16} from bit 16 through bit 31. As mentioned before, sign extension is required when an arithmetic operation between two signed binary numbers of different sizes is performed. The (EA) in MOVEA can use all addressing modes.

The MOVEM instruction can be used to push or pop multiple registers to or from the stack. For example, MOVEM.L D0-D7/A0-A6,-(SP) saves the contents of all eight data registers and seven address registers in the stack. This instruction stores address registers in the order A6–A0 first, followed by data registers in the order D7–D0, regardless of the order in the register list. MOVEM.L (SP)+,D0-D7/A0-A6 restores the contents of the registers in the order D0–D7, A0–A6, regardless of the order in the register list.

The MOVEM instruction can also be used to save a set of registers in memory. In

addition to the preceding predecrement and postincrement modes for the effective address, the MOVEM instruction allows all the control modes. If the effective address is in one of the control modes, such as absolute short, then the registers are transferred starting at the specified address and up through higher addresses. The order of transfer is from D0 to D7 and then from A0 to A6. For example, `MOVEM.W A5/D1/D3/A1-A3,$2000` transfers the low 16-bit contents of D1, D3, A1, A2, A3, and A5 to locations $2000, $2002, $2004, $2006, $2008, and $200A, respectively.

The `MOVEQ.L #$d8, Dn` instruction moves the immediate 8-bit data into the low byte of D*n*. The 8-bit data is then sign-extended to 32 bits. This is a one-word instruction. For example, `MOVEQ.L #$8F,D5` moves $FFFFFF8F into D5.

To transfer data between the 68000 data registers and 6800 (8-bit) peripherals, the MOVEP instruction can be used. This instruction transfers 2 or 4 bytes of data between a data register and alternate byte locations in memory, starting at the location specified and incrementing by 2. Register indirect with displacement is the only addressing mode used with this instruction. If the address is even, all transfers are made on the high-order half of the data bus; if the address is odd, all transfers are made on the low-order half of the data bus. The high-order byte to/from the register is transferred first, and the low-order byte is transferred last. For example, consider `MOVEP.L $0020(A2),D1`. If [A2] = $00002000, $[002020_{16}] = 02$, $[002022_{16}] = 05$, $[002024_{16}] = 01$, and $[002026_{16}] = 04$, then, after execution of this MOVEP instruction, D1 will contain 02050104_{16}.

EXG and SWAP Instructions

The `EXG.L Rx, Ry` instruction exchanges the 32-bit contents of R*x* with that of R*y*. The exchange is between two data registers, two address registers, or an address register and a data register. The EXG instruction exchanges only 32-bit-long words. The data size (L) does not have to be specified after the EXG instruction because this instruction has only one data size (L) and it is assumed that the default is this single data size. No flags are affected. The `SWAP.W Dn` instruction, on the other hand, exchanges the low 16 bits of D*n* with the high 16 bits of D*n*. All condition codes are affected.

LEA and PEA Instructions

The `LEA.L (EA), An` instruction moves an effective address (EA) into the specified address register. The (EA) can be calculated based on the addressing mode of the source. For example, `LEA $00256022,A5` moves $00256022 into A5. This instruction is equivalent to `MOVEA.L #$00256022,A5`. Note that $00256022 is contained in PC. It should be pointed out that the LEA instruction is very useful when address calculation is desired during program execution. The (EA) in LEA specifies the actual data to be loaded into An, whereas the (EA) in MOVEA specifies the address of actual data. For example, consider `LEA $04(A5, D2.W),A3`. If [A5] = 00002000_{16} and [D2] = 0028_{16}, then the LEA instruction moves $0000202C_{16}$ into A3. On the other hand, `MOVEA $04(A5, D2.W), A3` moves the contents of $00202C_{16}$ into A3. Therefore, it is obvious that if address calculation is required, the instruction LEA is very useful.

The PEA.L (EA) computes an effective address and then pushes it on to the Supervisor stack (S=1) or User stack (S=0). This instruction can be used when the 16-bit address in absolute short mode is required to be pushed onto the stack. For example, consider PEA.L $9000 in the user mode. If [A7]=$00003006, then $9000 is sign-extended to 32 bits ($FFFF9000). The low-order 16 bits ($9000) are pushed at $003004, and the high order 16 bits ($FFFF) are pushed at $003002.

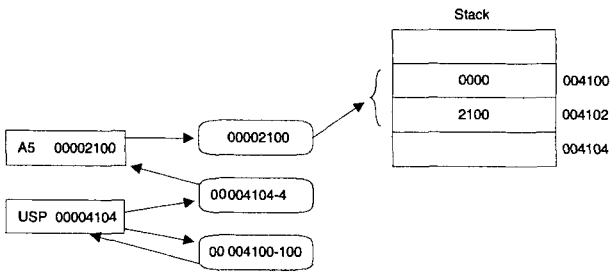


FIGURE 10.4 Execution of the LINK instruction

LINK and UNLK Instructions

Before calling a subroutine, the main program quite often transfers the values of certain parameters to the subroutine. It is convenient to save these variables onto the stack before calling the subroutine. These variables can then be read from the stack and used by the subroutine for computations. The 68000 LINK and UNLK instructions are used for this purpose. In addition, the 68000 LINK instruction allows one to reserve temporary storage for the local variables of a subroutine. This storage can be accessed as needed by the subroutine and can be released using UNLK before returning to the main program. The LINK instruction is usually used at the beginning of a subroutine to allocate stack space for storing local variables and parameters for nested subroutine calls. The UNLK instruction is usually used at the end of a subroutine before the RETURN instruction to release the local area and restore the stack pointer contents so that it points to the return address.

The LINK An, #- displacement instruction causes the current contents of the specified An to be pushed onto the system stack. The updated SP contents are then loaded into An. Finally, a sign-extended twos complement displacement value is added to the SP. No flags are affected. For example, consider `LINK A5,#-$100`. If [A5] = 00002100_{16} and [USP] = 00004104_{16}, then after execution of the LINK instruction, the situation shown in Figure 10.4 occurs. This means that after the LINK instruction, [A5] = $00002100 is pushed onto the stack and the [updated USP] = $004100 is loaded into A5. USP is then loaded with $004000 and therefore 100_{16} locations are allocated to the subroutine at the beginning of which this particular LINK instruction can be used. Note that A5 cannot be used in the subroutine.

The UNLK instruction at the end of this subroutine before the RETURN instruction releases the 100_{16} locations and restores the contents of A5 and USP to those prior to using the LINK instruction. For example, `UNLK A5` will load [A5] = $00004100 into USP and the two stack words $00002100 into A5. USP is then incremented by 4 to contain $00004104. Therefore, the contents of A5 and USP prior to using the LINK instruction are restored.

In this example, after execution of the LINK, addresses $0003FF and below can be used as the system stack. One hundred (Hex) locations starting at $004000 and above can be reserved for storing the local variables of the subroutine. These variables can then be accessed with an address register such as A5 as a base pointer using the address register indirect with displacement mode. `MOVE.W d(A5),D1` for read and `MOVE.W D1,d(A5)` for write are typical examples.

The use of LINK and UNLK can be illustrated by the following subroutine structure:

```
SUBR LINK A2, #-50     ;       Allocate 50 bytes
```

```
.
.
UNLK A2          ;       Restore original values
RTS              ;       Return to subroutine
```

The LINK instruction is used in this case to allocate 50 bytes for local variables. At the end of the subroutine, UNLK A2 is used before RTS to restore the original values of the registers and the stack. RTS returns program execution in the main program.

10.6.2 Arithmetic Instructions

These instructions allow:

- 8-, 16-, or 32-bit additions and subtractions.
- 16-bit by 16-bit multiplication (both signed and unsigned) and 32-bit by 16-bit division (both signed and unsigned)
- Compare, clear, and negate instructions.
- Extended arithmetic instruction for performing multiprecision arithmetic.
- Test (TST) instruction for comparing the operand with zero.
- Test and set (TAS) instruction, which can be used for synchronization in a multiprocessor system.

The 68000 arithmetic instructions are summarized in Table 10.6. Let us now explain the arithmetic instructions.

TABLE 10.6 68000 Arithmetic Instructions

Instruction	*Size*	*Operation*
		Addition and Subtraction Instructions
ADD (EA), (EA)	B, W, L	(EA) + (EA) → (EA)
ADDI #Data, (EA)	B, W, L	(EA) + data → (EA)
ADDQ #d_8, (EA)	B, W, L	(EA) + d_8 → (EA) d_8 can be an integer from 0 to 7
ADDA (EA), An	W, L	An + (EA) → An
SUB (EA), (EA)	B, W, L	(EA) – (EA) → (EA)
SUBI # data, (EA)	B, W, L	(EA) – data → EA
SUBQ #d_8, (EA)	B, W, L	(EA) – d_8 → EA d_8 can be an integer from 0 to 7
SUBA (EA), An	W, L	An – (EA) → An
		Multiplication and Division Instructions
MULS (EA), Dn	W	$(Dn)_{16}$ * $(EA)_{16}$ → $(Dn)_{32}$ (signed multiplication)
MULU (EA), Dn	W	$(Dn_{)16}$ * $(EA)_{16}$ → $(Dn)_{32}$ (unsigned multiplication)
DIVS (EA), Dn	W	$(Dn)_{32}$ / $(EA)_{16}$ → $(Dn)_{32}$

		(signed division, high word of Dn contains remainder and low word of Dn contains the quotient)
DIVU (EA), Dn	W	$(Dn)_{32} / (EA)_{16} \rightarrow (Dn)_{32}$ (unsigned division, remainder is in high word of Dn and quotient is in low word of Dn)
	Compare, Clear, and Negate Instructions	
CMP (EA), Dn	B, W, L	Dn - (EA) → No result. Affects flags.
CMPA (EA), An	W, L	An - (EA) → No result. Affects flags.
CMPI # data, (EA)	B, W, L	(EA) – data → No result. Affects flags.
CMPM (Ay) +, (Ax) +	B, W, L	(Ax)+ – (Ay)+ → No result. Affects flags.
CLR (EA)	B,W,L	0 → (EA)
NEG (EA)	B,W,L	0 – (EA) →(EA)
	Extended Arithmetic Instructions	
ADDX Dy,Dx	B, W, L	Dx + Dy + X → Dx
ADDX – (Ay), – (Ax)	B, W, L	– (AX) + – (Ay) + X → (Ax)
EXT Dn	W, L	If size is W, then sign extend low byte of Dn to 16 bits. If size is L, then sign extend low 16 bits of Dn to 32 bits.
NEGX (EA)	B, W, L	0 – (EA) – X → (EA)
SUBX Dy,Dx	B, W, L	Dx – Dy – X → Dx
SUBX – (Ay), – (Ax)'	B, W, L	– (Ax) – – (Ay) – X → (Ax)
	Test Instruction	
TST (EA)	B, W, L	(EA) - 0 ⇒ Flags are affected.
	Test and Set Instruction	
TAS (EA)	B	If (EA) = 0, then set Z = 1; else Z = 0, N = 1 and then always set bit 7 of (EA) to 1.

NOTE: If source (EA) in the ADDA or SUBA instruction is an address register, the operand length is WORD or LONG WORD.
(EA) in any instruction is calculated using the addressing mode used.
All instructions except ADDA and SUBA affect condition codes.

- Source (EA) in the above ADD, ADDA, SUB, and SUBA can use all modes. Destination (EA) in the above ADD and SUB instructions can use all modes except An. relative, and immediate.
- Destination (EA) in ADDI and SUBI can use all modes except An. relative, and

immediate.

- Destination (EA) in `ADDQ` and `SUBQ` can use all modes except relative and immediate.
- (EA) in all multiplication and division instructions can use all modes except An.
- Source (EA) in `CMP` and `CMPA` instructions can use all modes.
- Destination (EA) in `CMPI` can use all modes except An, relative, and immediate.
- (EA) in `CLR` and `NEG` can use all modes except An, relative, and immediate.
- (EA) in `NEGX` can use all modes except An, relative and immediate.
- (EA) in TST can use all modes except An, relative, and immediate.
- (EA) in TAS can use all modes except An, relative, and immediate.

Addition and Subtraction Instructions

- Consider `ADD.W $122000,D0`. If $[122000_{16}] = 0012_{16}$ and $[D0] = 0002_{16}$, then, after execution of this `ADD`, the low 16 bits of D0 will contain 0014_{16}. C = 0 (No Carry), X = 0 (Same as C), V=0 (No Overflow since previous Carry and the final Carry are the same), N = 0 (Most Significant Bit of the result is 0), Z = 0 (Nonzero result).
- The `ADDI` instruction can be used to add immediate data to a register or memory location. The immediate data follows the instruction word. For example, consider `ADDI.W #$0012,$100200`. If $[100200_{16}] = 0002_{16}$, then, after execution of this `ADDI`, memory location 100200_{16} will contain 0014_{16}.
- `ADDQ` adds a number from 0 to 7 to the register or memory location in the destination operand. This instruction occupies 16 bits, and the immediate data 0 to 7 is specified by 3 bits in the instruction word. For example, consider `ADDQ.B #2,D1`. If $[D1]_{low\ byte} = 20_{16}$, then, after execution of this `ADDQ`, the low byte of register D1 will contain 22_{16}.
- All subtraction instructions subtract the source from the destination. For example, consider `SUB.W D2,$122200`. If $[D2]_{low\ word} = 0003_{16}$ and $[122200_{16}] = 0007_{16}$, then, after execution of this SUB, memory location 122200_{16} will contain 0004_{16}.
- SUBX.B D1,D2 subtracts the source byte (D1.B) plus the X-bit (same as the Carry flag) from the destination byte (D2.B); the result is stored in the destination byte, no other bytes of the destination register are affected. All condition codes are affected. For example, if $[D2.L] = 2AB10003_{16}$, $[D1.L] = A2345602_{16}$, and X = C = 1, then, after SUBX.B D1,D2, the contents of D2.B = 03 - 02 - 1 = 00_{16}. $[D2.L] = 2AB10000_{16}$.

  1111 1111← Intermediate Carries
  Using two's complement subtraction, [D2.B] = 0000 0011 (+3)
  Add two's complement of 3 (D1.B plus Carry) = +1111 1101 (-3)

  Final Carry →1 0000 0000

  Final carry is one's complemented after subtraction to reflect the correct borrow. Hence, C = 0.
  Also, X = 0 (Same as C), Z = 1 (Zero Result), N = 0 (Most Significant of the result is zero), and $V = C_f \oplus C_p = 1 \oplus 1 = 0$.
- Consider `SUBI.W #3,D0`. If $[D0]_{low\ word} = 0014_{16}$, then, after execution of this `SUBI`, D0 will contain 0011_{16}. Note that the same result can be obtained by using a `SUBQ.W #3,D0`. However in this case, the data item 3 is inherent in the instruction word.

Multiplication and Division Instructions

The 68000 instruction set includes both signed and unsigned multiplication of integer numbers.

- `MULS` (EA), D*n* multiplies two 16-bit signed numbers and provides a 32-bit result. For example, consider `MULS #-2, D5`. If [D5.W] = 0003_{16}, then, after this `MULS`, D5 will contain the 32-bit result $FFFFFFFA_{16}$, which is -6 in decimal.
- `MULU` (EA), D*n* performs unsigned multiplication. Consider `MULU (A0), D1`. If [A0] = 00102000_{16}, [102000_{16}] = 0300_{16}, and [D1.W] = 0200_{16}, then, after this `MULU`, D1 will contain the 32-bit result 00060000_{16}.
- Consider `DIVS #2, D1`. If [D1] = -5_{10} = $FFFFFFFB_{16}$, then, after this `DIVS`, register D1 will contain

D1	FFFF	FFFE
	16-bit remainder = -1_{10}	16-bit quotient = -2_{10}

Note that in the 68000, after `DIVS`, the sign of remainder is always the same as the dividend unless the remainder is equal to zero. Therefore, in this example, because the dividend is negative (-5_{10}), the remainder is negative (-1_{10}). Also, division by zero causes an internal interrupt automatically. A service routine can be written by the user to indicate an error. N = 1 if the quotient is negative, and V = 1 if there is an overflow.

- `DIVU` is the same as the `DIVS` instruction except that the division is unsigned. For example, consider `DIVU #4, D5`. If [D5] = 14_{10} = $00000000E_{16}$, then after this `DIVU`, register D5 will contain

D5	0002	0003
	16-bit remainder	16-bit quotient

As with the `DIVS` instruction, division by zero using `DIVU` causes a trap (internal interrupt).

Compare, Clear, and Negate Instructions

- The Compare (CMP) instruction subtracts source from destination providing no result of subtraction; all condition codes are affected based on the result. Note that the SUBTRACT instruction provides the result and also affects the Condition Codes. Consider `CMP.B D3,D0`. If prior to execution of the instruction, [D0.B] = $40 and [D3.B] = $30 then after execution of `CMP.B D3, D0`, the condition codes are as follows: C = 0, X = 0, Z = 0, N = 0, and V = 0. Suppose it is desired to find the number of matches for an 8-bit number in a 68000 register such as D5.B in a data array (stored from low to high memory) of 50 bytes in memory pointed to by A0. The following instruction sequence with `CMP.B (A0)+,D5` rather than `SUB.B (A0)+,D5` can be used :

```
        CLR.B   D0          ; Clear D0.B to 0, D0.B to hold number of matches
        MOVE.B  #50,D1      ; Initialize array count
START   CMP.B   (A0)+,D5    ; Compare the number to be matched in D5
        BNE     DECR        ; with a data byte in the array. If there
        ADDQ.B  #1,D0       ; is a match, Z=1 and increment D0.
DECR    SUBQ.B  #1,D1       ; Decrement D1 by 1, go back to START if
        BNE     START       ; Z = 0.If Z = 1, go to the next
```

```
                ; instruction
                ; D0.B contains the number of matches
```

In the above, if `SUB.B (A0)+,D5` were used instead of `CMP.B (A0)+,D5`, the number to be matched needs to be loaded after each subtraction because the contents of D5.B would have been lost after each SUB. Since we are only interested in the match rather than the result, `CMP.B (A0)+,D5` instead of `SUB.B (A0)+,D5` should be used in the above.

- The 68000 instruction set includes a memory to memory COMPARE instruction. For example, `CMPM.W (A0)+, (A1)+`. If [A0] = 00100000_{16}, [A1] = 00200000_{16}, [100000_{16}] = 0005_{16}, and [200000_{16}] = 0006_{16}, then, after this `CMPM` instruction, N = 0, C = 0, X = 0, V = 0, Z = 0, [A0] = 00100002_{16}, and [A1] = 00200002_{16}.
- `CLR.L D5` clears all 32 bits of D5 to zero.
- Consider `NEG.W (A0)`. If [A0] = 00200000_{16} and [200000] = 5_{10}, then after this `NEG` instruction, the low 16 bits of location 200000_{16} will contain $FFFB_{16}$.

Extended Arithmetic Instructions

- The `ADDX` and `SUBX` instruction can be used in performing multiprecision arithmetic because there are no `ADDC` (add with carry) or `SUBC` (subtract with borrow) instructions. For example, in order to perform a 64-bit addition, the following two instructions can be used:

```
ADD.L D0,D5     ;Add low 32 bits of data and store in D5.
ADDX.L D1,D6    ;Add high 32 bits of data along with any carry from
                ;the low 32-bit addition and store result in D6.
```

  Note that in this example, D1D0 contain one 64-bit number and D6D5 contain the other 64-bit number. The 64-bit result is stored in D6D5.
- Consider `EXT.W D2`. If $[D2]_{\text{low byte}} = F3_{16}$, then, after the `EXT`, $[D2]_{\text{low word}} = FFF3_{16}$.
- An example of sign extension is that, to multiply a signed 8-bit number by a signed 16-bit number, one must first sign-extend the signed 8-bit into a signed 16-bit number and then the instruction `IMUL` can be used for 16 × 16 signed multiplication. For unsigned multiplication of a 16-bit number by an 8-bit number, the 8-bit number must be zero extended to 16 bits using logical instruction such as AND before using the `MUL` instruction.

Test Instruction

Consider `TST.W (A0)`. If [A0] = 00300000_{16} and [300000_{16}] = $FFFF_{16}$, then, after the `TST.W (A0)`, the operation $FFFF_{16} - 0000_{16}$ is performed internally by the 68000, Z is cleared to 0, and N is set to 1. The V and C flags are always cleared to 0.

Test and Set Instruction

`TAS.B (EA)` is usually used to synchronize two processors in multiprocessor data transfers. For example, consider the two 68000-based microcomputers with shared RAM as shown in Figure 10.5.

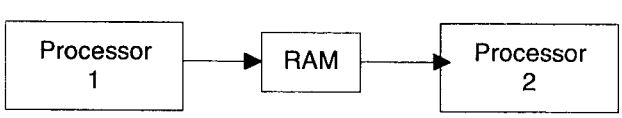


FIGURE 10.5 Two 68000s interfaced via shared RAM using TAS instruction

Suppose that it is desired to transfer the low byte of D0 from processor 1 to the low byte of D2 in processor 2. A memory location, namely, TRDATA, can be used to accomplish this. First, processor 1 can execute the TAS instruction to test the byte in the shared RAM with address TEST for zero value. If it is, processor 1 can be programmed to move the low byte of D0 into location TRDATA in the shared RAM. Processor 2 can then execute an instruction sequence to move the contents of TRDATA from the shared RAM into the low byte of D2. The following instruction sequence will accomplish this:

Processor 1 Routine			*Processor 2 Routine*		
Proc_1	TAS.B	TEST	Proc_2	TAS.B	TEST
	BNE	Proc_1		BNE	Proc_2
	MOVE.B	D0,TRDATA		MOVE.B	TRDATA,D2
	CLR.B	TEST		CLR.B	TEST
	—			—	
	—			—	
	—			—	

Note that in these instruction sequences, `TAS.B TEST` checks the byte addressed by TEST for zero. If [TEST] = 0, then Z is set to 1; otherwise, Z = 0 and N = 1. After this, bit 7 of [TEST] is set to 1. Note that a zero value of [TEST] indicates that the shared RAM is free for use, and the Z bit indicates this after the TAS is executed. In each of the instruction sequences, after a data transfer using the `MOVE` instruction, [TEST] is cleared to zero so that the shared RAM is free for use by the other processor. To avoid testing the TEST byte simultaneously by two processors, the `TAS` is executed in a read-modify-write cycle. This means that once the operand is addressed by the 68000 executing the `TAS`, the system bus is not available to the other 68000 until the `TAS` is completed.

10.6.3 Logical Instructions

These instructions include logical `OR`, `EOR`, `AND`, and `NOT` as shown in Table 10.7.

- Consider `AND.B #$8F,D0`. If prior to execution of this instruction, [D0.B] = $72, then after execution of `AND.B #$8F,D0`, the following result is obtained:

```
[D0.B]  = $72 =0111 0010
  AND  $8F =  1000 1111
              ----------
[D0.B] =  0000 0010
```

Z = 0 (Result is nonzero) and N = 0 (Most Significant Bit of the result is 0). C and V are always cleared to 0 after logic operation. The condition codes are similarly affected after execution of other logical instructions such as OR, EOR, and NOT.

The AND instruction can be used to perform a masking operation. If the bit value in a particular bit position is desired in a word, the word can be logically ANDed with appropriate data to accomplish this. For example, the bit value at bit 2 of an 8-bit number 0100 1Y10 (where unknown bit value of Y is to be determined) can be obtained as follows:

```
          0 1 0 0 1 Y 1 0  -- 8-bit number
AND       0 0 0 0 0 1 0 0  -- Masking data
          -----------------
          0 0 0 0 0 Y 0 0  -- Result
```

If the bit value Y at bit 2 is 1, then the result is nonzero (Flag Z=0); otherwise, the result is zero (Z=1). The Z flag can be tested using typical conditional JUMP instructions such as BEQ (Branch if Z=1) or BNE (Branch if Z=0) to determine

TABLE 10.7 68000 Logical Instructions

Instruction	*Size*	*Operation*
AND (EA), (EA)	B, W, L	(EA) AND (EA) → (EA); (EA) cannot be address register
ANDI # data, (EA)	B, W, L	(EA) AND # data → (EA); (EA) cannot be address register
ANDI # $data_8$, CCR	B	CCR AND # data → CCR
ANDI # $data_{16}$, SR	W	SR AND# data → SR
EOR Dn, (EA)	B, W, L	Dn ⊕ (EA)→ (EA); (EA) cannot be address register
EORI # data, (EA)	B, W, L	(EA) ⊕ # data → (EA); (EA) cannot be address register
NOT (EA)	B, W, L	One's complement of (EA) → (EA);
OR (EA), (EA)	B, W, L	(EA) OR (EA) → (EA); (EA) cannot be address register
ORI # data, (EA)	B, W, L	(EA) OR # data → (EA); (EA) cannot be address register
ORI # $data_8$, CCR	B	CCR OR # $data_8$ → CCR
ORI # $data_{16}$, SR	W	SR OR # data → SR

whether Y is 0 or 1. This is called masking operation. The AND instruction can also be used to determine whether a binary number is ODD or EVEN by checking the Least Significant bit (LSB) of the number (LSB=0 for even and LSB=1 for odd).

- Consider `AND.W D1,D5`. If [D1.W] = 0001_{16} and [D5.W] = $FFFF_{16}$, then, after execution of this `AND`, the low 16 bits of both D1 and D5 will contain 0001_{16}.
- Consider `ANDI.B #$00,CCR`. If [CCR] = 01_{16}, then, after this `ANDI`, register CCR will contain 00_{16}.
- Source (EA) in `AND` and `OR` can use all modes except An.
- Destination (EA) in `AND` or `OR` or `EOR` can use all modes except An, relative, and immediate.
- Destination (EA) in `ANDI`, `ORI`, and `EORI` can use all modes except An, relative, and immediate.
- (EA) in `NOT` can use all modes except An, relative, and immediate.
- Consider `EOR.W #2,D5`. If prior to execution of this instruction, `[D5.W] = $2342`, then after execution of `EOR.W #2,D5`, low 16-bit contents of D5 will be `$2340`. All condition codes are affected in the same manner as the AND instruction. The Exclusive-OR instruction can be used to find the ones complement of a binary number by XORing the number with all 1's as follows:

```
      0 1 0 1 1 1 0 0 -- 8-bit number
XOR   1 1 1 1 1 1 1 1 -- data
      -----------------------
      1 0 1 0 0 0 1 1 -- Result ( Ones Complement of the 8-bit number
                                  0 1 0 1 1 1 0 0 )
```

- Consider `EOR.W D1,D2`. If [D1.W] = $FFFF_{16}$ and [D2.W] = $FFFF_{16}$, then, after

execution of this EOR, register D2.W will contain 0000_{16}, and D1 will remain unchanged at $FFFF_{16}$.

- Consider NOT.B D5. If [D5.B] = 02_{16}, then, after execution of this NOT, the low byte of D5 will contain FD_{16}.
- Consider OR.B D2,D3. If prior to execution of this instruction, [D2.B] = $A2_{16}$ and [D3.B] = $5D_{16}$, then after exection of OR.B D2,D3, the contents of D3.B are FFH. All flags are affected similar to the AND instruction. The OR instruction can typically be used to insert a 1 in a particular bit position of a binary number without changing the values of the other bits. For example, a 1 can be inserted using the OR instruction at bit number 3 of the 8-bit binary number 0 1 1 1 0 0 1 1 without changing the values of the other bits as follows:

```
    0 1 1 1 0 0 1 1 -- 8-bit number
OR  0 0 0 0 1 0 0 0 -- data for inserting a 1 at bit number 3
    -----------------
    0 1 1 1 1 0 1 1 -- Result
```

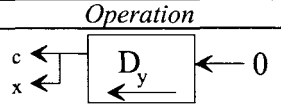
- Consider ORI #$1002,SR. If [SR] = $111D_{16}$, then after execution of this ORI, register SR will contain $111F_{16}$. Note that this is a privileged instruction because the high byte of SR containing the control bits is changed and therefore, can be executed only in the supervisor mode.

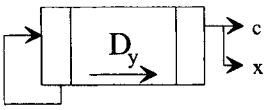
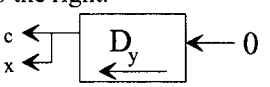
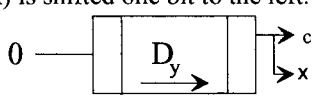
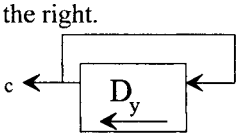
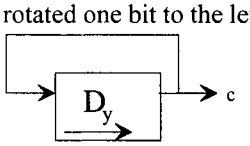
10.6.4 Shift and Rotate Instructions

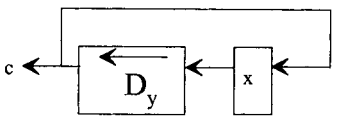
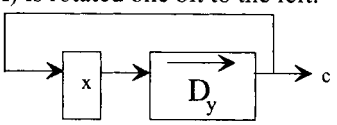
The 68000 shift and rotate instruction are listed in Table 10.8.

- All the instructions in Table 10.8 affect N and Z flags according to the result. V is reset to zero except for ASL.
- Note that in the 68000 there is no true arithmetic shift left instruction. In true arithmetic shifts, the sign bit of the number being shifted is retained. In the 68000, the instruction ASL does not retain the sign bit, whereas the instruction ASR retains the sign bit after performing the arithmetic shift operation.

TABLE 10.8 68000 Shift and Rotate Instructions

Instruction	*Size*	*Operation*
ASL Dx, Dy	B, W, L	 Shift [Dy] by the number of times to left specified in Dx; the low 6 bits of Dx specify the number of shifts from 0 to 63.
ASL # data, Dn	B, W, L	Same as ASL Dx, Dy, except that the number of shifts is specified by immediate data from 0 to 7.
ASL (EA)	B, W, L	(EA) is shifted one bit to the left; the most significant bit of (EA) goes to x and c, and zero moves into the least significant bit.

ASR Dx, Dy	B, W, L	 Arithmetically shift [Dy] to the right by retaining the sign bit; the low 6 bits of Dx specify the number of shifts from 0 to 63.
ASR # data, Dn	B, W, L	Same as above except the number of shifts is from 0 to 7.
ASR (EA)	B, W, L	Same as above except (EA) is shifted once to the right.
LSL Dx, Dy	B, W, L	 Low 6 bits of Dx specify the number of shifts from 0 to 63.
LSL # data, Dn	B, W, L	Same as above except that the number of shifts is specified by immediate data from 0 to 7.
LSL (EA)	B, W, L	(EA) is shifted one bit to the left.
LSR Dx, Dy	B, W, L	 Same as LSL Dx, Dy, except shift is to the right.
LSR # data, Dn	B, W, L	Same as above except shift is to the right by immediate data from 0 to 7.
LSR (EA)	B, W, L	Same as LSL (EA) except shift is once to the right.
ROL Dx, Dy	B, W, L	 Low 6 bits of Dx specify the number of times [Dy] to be rotated.
ROL # data, Dn	B, W, L	Same as above except that the immediate data specifies that [Dn] to be rotated from 0 to 7.
ROL (EA)	B, W, L	(EA) is rotated one bit to the left.
ROR Dx, Dy	B, W, L	
ROR # data, Dn	B, W, L	Same as above except the rotate is to the right by immediate data from 0 to 7.
ROR (EA)	B, W, L	(EA) is rotated one bit to the right.

ROXL Dx, Dy	B, W, L	 Low 6 bits of Dx contain the number of rotates from 0 to 63.
ROXL # data, Dn	B, W, L	Same as above except that the immediate data specifies number of rotates from 0 to 7.
ROXL (EA)	B, W, L	(EA) is rotated one bit to the left.
ROXR Dx, Dy	B, W, L	 Low 6 bits of Dx contain the number of rotates from 0 to 63.
ROXR # data, Dn	B, W, L	Same as above except the rotate is to the right by immediate data from 0 to 7.
ROXR (EA)	B, W, L	Same as above except the rotate is once to the right.

- (EA) in ASL, ASR, LSL, LSR, ROL, ROR, ROXL, and ROXR can use all modes except Dn, An, relative, and immediate.
- Consider ASL.W D1, D5. If $[D1]_{\text{low 16 bits}} = 0002_{16}$ and $[D5]_{\text{low 16 bits}} = 9FF0_{16}$, then, after this ASL instruction, $[D5]_{\text{low 16 bits}} = 7FC0_{16}$, C = 0, and X = 0. Note that the sign of the contents of D5 is changed from 1 to 0 and, therefore, the overflow is set. The sign bit of D5 is changed after shifting [D5] twice. For ASL, the overflow flag is set to one if the sign bit changes during or after shifting. The contents of D5 are not updated after each shift. The ASL instruction can be used to multiply a signed number by 2^n by shifting the number n times to the left; the result is correct if V = 0 while the result is incorrect if V = 1. Since execution time of the multiplication instruction is longer, multiplication by shifting may be more efficient when multiplication of a signed number by 2^n is desired.
- ASR retains the sign bit. For example, consider ASR.W #2, D1. If [D1.W] = $FFE2_{16}$, then, after this ASR, the low 16 bits of [D1] = $FFF8_{16}$, C = 1, and X = 1. Note that the sign bit is retained.
- ASL (EA) or ASR (EA) shifts (EA) 1 bit to left or right, respectively. For example, consider ASL.W (A0). If [A0] = 00002000_{16} and $[002000_{16}] = 9001_{16}$, then, after execution of this ASL, $[002000_{16}] = 2002_{16}$, X = 1, and C = 1. On the other hand, after ASR.W (A0), memory location 002000_{16} will contain $C800_{16}$, C = 1, and X = 1.
- The LSL and ASL instructions are the same in the 68000 except that with the ASL, V is set to 1 if the sign of the result is changed from the sign of the original value during or after shifting. This will allow one to multiply a signed number by 2^n by shifting the number n times to left; the result is correct if V = 0 while the result is incorrect if V = 1. Since execution time of the multiplication instruction is longer, multiplication by shifting may be more efficient when multiplication of a signed number by 2^n is desired.

TABLE 10.9 Bit Manipulation Instructions

Instruction	*Size*	*Operation*
`BCHG Dn, (EA)` `BCHG # data, (EA)`	B,L	A bit in (EA) specified by Dn or immediate data is tested: the 1's complement of the bit is reflected in both the Z flag and the specified bit position.
`BCLR Dn, (EA)` `BCLR # data, (EA)`	B,L	A bit in (EA) specified by Dn or immediate data is tested and the 1's complement of the bit is reflected in the Z flag: the specified bit is cleared to zero.
`BSET Dn, (EA)` `BSET # data, (EA)`	B,L	A bit in (EA) specified by Dn or immediate data is tested and the 1's complement of the bit is reflected in the Z flag: the specified bit is then set to one.
`BTST Dn, (EA)` `BTST # data, (EA)`	B,L	A bit in (EA) specified by Dn or immediate data is tested. The 1's complement of the specified bit is reflected in the Z flag.

- (EA) in the above instructions can use all modes except An, relative, and immediate.
- If (EA) is memory location then data size is byte: if (EA) is Dn then data size is long word.

- Consider `LSR.W #3,D1`. If [D1.W] = 8000_{16}, then after this `LSR`, [D1.W] = 1000_{16}, X = 0, and C = 0.
- Consider `ROL.B #2,D2`. If [D2.B] = $B1_{16}$ and C = 1, then, after this `ROL`, the low byte of [D2] = $C6_{16}$ and C = 0. On the other hand, with [D2.B] = $B1_{16}$ and C = 1, consider `ROR.B #2,D2`. After this `ROR`, low byte of register D2 will contain $6C_{16}$ and C = 0.
- Consider `ROXL.W D2,D1`. If [D2.W] = 0003_{16}, [D1.W] = $F201_{16}$, C = 0, and X = 1 then after execution of this `ROXL`, [D1.W] = $900F_{16}$, C = 1, and X = 1.

10.6.5 Bit Manipulation Instructions

The 68000 has four bit manipulation instructions, and these are listed in Table 10.9.

- In all of the instructions in Table 10.9, the ones complement of the specified bit is reflected in the Z flag. The specified bit is ones complemented, cleared to 0, set to 1, or unchanged by `BCHG`, `BCLR`, `BSET`, or `BTST`, respectively. In all the instructions in Table 10.9, if (EA) is D*n*, then the length of D*n* is 32 bits; otherwise, the length of the destination is one byte memory.
- Consider `BCHG.B #2,$003000`. If [003000_{16}] = 05_{16}, then, after execution of this `BCHG`, Z = 0 and [003000_{16}] = 01_{16}.
- Consider `BCLR.L #3,D1`. If [D1] = $F210E128_{16}$, then after execution of this `BCLR`, register D1 will contain $F210E120_{16}$ and Z = 0.
- Consider `BSET.B #0,(A1)`. If [A1] = 00003000_{16} and [003000_{16}] = 00_{16}, then, after execution of this `BSET`, memory location 003000_{16} will contain 01_{16} and Z = 1.
- Consider `BTST.B #2,$002000`. If [002000_{16}] = 02_{16}, then, after execution of this `BTST`, Z = 1, and [002000_{16}] = 02_{16}; no other flags are affected.

10.6.6 Binary-Coded-Decimal Instructions

The 68000 instruction set contains three BCD instructions, namely, `ABCD` for adding, `SBCD` for subtracting, and `NBCD` for negating. They operate on packed BCD byte(s) and provide the result containing one packed BCD byte. These instructions always include the

TABLE 10.10 68000 Binary Coded Decimal Instructions

Instruction	*Operand Size*	*Operation*
ABCD Dy, Dx	B	$[Dx]_{10} + [Dy]_{10} + X \rightarrow [Dx]$
ABCD - (Ay), -(Ax)	B	$-(Ax)_{10} + -(Ay)_{10} + X \rightarrow (Ax)$
SBCD Dy, Dx	B	$[Dx]_{10} - [Dy]_{10} - X \rightarrow [Dx]$
SBCD - (Ay), - (Ax)	B	$-(Ax)_{10} - -(Ay)_{10} - X \rightarrow (Ax)$
NBCD (EA)	B	$0 - (EA)_{10} - X \rightarrow (EA)_{10}$

• (EA) in NBCD can use all modes except An, relative, and immediate.

extend (X) bit in the operation. The BCD instructions are listed in Table 10.10.

- Consider `ABCD.B D1,D2`. If [D1.B] = 25_{10}, [D2.B] = 15_{10}, and X = 0, then, after execution of this ABCD instruction, [D2.B] = 40_{10}, X = 0, and Z = 0.
- Consider `SBCD.B -(A2),-(A3)`. If [A2] = 00002004_{16}, [A3] = 00003003_{16}, [002003_{16}] = 05_{10}, [003002_{16}] = 06_{10}, and X = 1, then after execution of this SBCD instruction, [003002_{16}] = 00_{10}, X = 0, and Z = 1.
- Consider `NBCD.B (A1)`. If [A1] = [00003000_{16}], [003000_{16}] =05_{10}, and X = 1, then, after execution of this NBCD instruction, [003000_{16}] = -6_{10}.

Note that packed BCD subtraction used in the instructions SBCD and NBCD can be obtained by using the concepts discussed in Chapter 2 (Section 2.5.2).

10.6.7 Program Control Instructions

These instructions include branches, jumps, and subroutine calls as listed in Table 10.11.

Consider Bcc *d*. There are 14 branch conditions. This means that the *cc* in B*cc* can be replaced by 14 conditions providing 14 instructions: BCC, BCS, BEQ, BGE, BGT, BHI, BLE, BLS, BLT, BMI, BNE, BPL, BVC, and BVS. It should be mentioned that some of these instructions are applicable to both signed and unsigned numbers, some can be used with only signed numbers, and some instructions are applicable to only unsigned numbers.

After signed arithmetic operations, instructions such as BEQ, BNE, BVS, BVC, BMI, and BPL can be used. On the other hand, after unsigned arithmetic operations, instructions such as BCC, BCS, BEQ, and BNE can be used. It should be pointed out that if V = 0, BPL and BGE have the same meaning, Likewise, if V = 0, BMI and BLT perform the same function.

The conditional branch instruction can be used after typical arithmetic instructions such as subtraction to branch to a location if *cc* is true. For example, consider `SUB.W D1, D2`. Now if [D1] and [D2] are unsigned numbers, then

BCC *d* can be used if [D2] > [D1]
BCS *d* can be used if [D2] ≤ [D1]
BEQ *d* can be used if [D2] = [D1]
BNE *d* can be used if [D2] ≠ [D1]
BHI *d* can be used if [D2] < [D1]
BLS *d* can be used if [D2] ≤ [D1]

On the other hand, if [D1] and [D2] are signed numbers, the after `SUB.W D1, D2`, the following branch instruction can be used:

BEQ *d* can be used if [D2] = [D1]
BNE *d* can be used if [D2] ≠ [D1]
BLT *d* can be used if [D2] < [D1]

TABLE 10.11 68000 Program Control Instructions

Instruction	*Size*	*Operation*
Bcc d	B,W	If condition code cc is true, then PC + d → PC. The PC value is current instruction location plus 2. d can be 8- or 16-bit signed displacement. If 8-bit displacement is used, then the instruction size is 16 bits with the 8-bit displacement as the low byte of the instruction word. If 16-bit displacement is used, then the instruction size is two words with 8-bit displacement field (low byte) in the instruction word as zero and the second word following the instruction word as the 16-bit displacement. There are 14 conditions such as BCC (Branch if Carry Clear), BEQ (Branch if result equal to zero, i.e., Z = 1), and BNE (Branch if not equal, i.e., Z = 0). Note that the PC contents will always be even since the instruction length is either one word or two words depending on the displacement widths.
BRA d	B,W	Branch always to PC + d where PC value is current instruction location plus 2. As with Bcc, d can be signed 8 or 16 bits. This is an unconditional branching instruction with relative mode. Note that the PC contents are even since the instruction is either one word or two words.
BSR d	B,W	PC → – (SP) PC + d → PC The address of the next instruction following PC is pushed onto the stack. PC is then loaded with PC + d. As before, d can be signed 8 or 16 bits. This is a subroutine call instruction using relative mode.
DB*cc* Dn, d	W	If *cc* is false, then Dn – 1 → Dn, and if Dn = – 1, then PC + 2 → PC If Dn ≠ – 1 , then PC + d → PC; else PC + 2 → PC.
JMP (EA)	unsized	(EA) → PC This is an unconditional jump instruction which uses control addressing mode.
JSR (EA)	unsized	PC → – (SP) (EA) → PC This is a subroutine call instruction which uses control addressing mode
RTR	unsized	(SP) + → CCR (SP) + → PC Return and restore condition codes
RTS	unsized	Return from subroutine (SP) + → PC
S*cc* (EA)	B	If *cc* is true, then the byte specified by (EA) is set to all ones; otherwise the byte is cleared to zero.

•(EA) in JMP and JSR can use all modes except Dn, An, (An) +, – (An), and immediate.

•(EA) in Scc can use all modes except An, relative, and immediate.

BLE *d* can be used if [D2] ≤ [D1]
BGT *d* can be used if [D2] > [D1]
BGE *d* can be used if [D2] ≥ [D1]

Now as a specific example, consider BEQ BEGIN. If current [PC] = 000200_{16}, and BEGIN=$20 then, after execution of this BEQ, program execution starts at 000220_{16} if Z = 1; if Z = 0, program execution continues at 000200_{16}. The instructions BRA and JMP are unconditional jump instructions. BRA uses the relative addressing mode, whereas JMP uses only control addressing mode. For example, consider BRA.B START. If [PC] = 000200_{16}, and START=$40 then, after execution of this BRA, program execution starts at 000240_{16}. Now, consider JMP (A1) . If [A1] = 00000220_{16}, then, after execution of this JMP, program execution starts at 000220_{16}.

- The instructions BSR and JSR are subroutine call instructions. BSR uses the relative mode, whereas JSR uses the control addressing mode. Consider the following program segment: Assume that the main program uses all registers; the subroutine stores the result in memory.

```
Main Program                    Subroutine
                    SUB   MOVEM.L D0-D7/A0-A6, - (SP)
     —                    —
     —                    —
     —                    —
     JSR  SUB             —          Main body of
START —                   —          subroutine
     —                    —
     —                    —
                          MOVEM.L (SP)+, D0-D7/A0-A6
                          RTS
```

Here, the JSR SUB instruction calls the subroutine SUB. In response to JSR, the 68000 pushes the current PC contents called START onto the stack and loads the starting address SUB of the subroutine into PC. The first MOVEM in the SUB pushes all registers onto the stack and, after the subroutine is executed, the second MOVEM instruction pops all the registers back. Finally, RTS pops the address START from the stack into PC, and program control in returned to the main program. Note that BSR SUB could have been used instead of JSR SUB in the main program. In that case, the 68000 assembler would have considered the SUB with BSR as a displacement rather than as an address with the JSR instruction.

- DB*cc* D*n*, *d* tests the condition codes and the value in a data register. DB*cc* first checks if *cc* (NE, EQ, GT, etc.) is satisfied. If *cc* is satisfied, the next instruction is executed. If *cc* is not satisfied, the specified data register is decremented by 1; if [D*n*] = -1, then the next instruction is executed; on the other hand, if D*n* ≠ -1, then branch to PC + d is performed. For example, consider DBNE.W D5, BACK with [D5] = 00003002_{16}, BACK= -4 and [PC] = 002006_{16}. If Z = 1, then [D5] = 00003001_{16}. Because [D5] ≠ -1, program execution starts at 002002_{16}. It should be pointed out that there is a false condition in the DB*cc* instruction and that this instruction is the DBF (some assemblers use DBRA for this). In this case, the condition is always false. This means that, after execution of this instruction, D*n* is decremented by 1 and if [D*n*] = –1, then the next instruction is executed. If [D*n*] ≠ –1, then branch to PC + *d*.

TABLE 10.12 68000 System Control Instructions

Instruction	*Size*	*Operation*
RESET	Unsized	If supervisor state, then assert reset line; else TRAP
RTE	Unsized	If supervisor state, then restore SR and PC; else TRAP
STOP # data	Unsized	If supervisor state, then load immediate data to SR and then STOP; else TRAP
ORI to SR MOVE USP ANDI to SR EORI to SR MOVE (EA) to SR		These instructions were discussed earlier
Trap and Check Instructions		
TRAP # vector	Unsized	PC → - (SP) SR → - (SP) Vector address → PC
TRAPV	Unsized	TRAP if V = 1
CHK (EA), Dn	W	If Dn < 0 or Dn > (EA), then TRAP; else, go to the next instruction.
Status Register		
ANDI to CCR EORI to CCR MOVE (EA) to/from CCR ORI to CCR MOVE SR to (EA)		Already explained earlier

•(EA) in CHK can use all modes except An.

- Consider `SPL.B(A5)`. If [A5] = 00200020_{16} and N = 0, then, after execution of this `SPL`, memory location 200020_{16} will contain 11111111_2.

10.6.8 System Control Instructions

The 68000 system control instructions contain certain privileged instructions including `RESET`, `RTE`, `STOP` and instructions that use or modify SR. Note that the privileged instructions can be executed only in the supervisor mode. The system control instructions are listed in Table 10.12.

- The `RESET` instruction when executed in the supervisor mode outputs a low signal on the reset pin of the 68000 in order to initialize the external peripheral chips. The 68000 reset pin is bidirectional. The 68000 can be reset by asserting the reset pin using hardware, whereas the peripheral chips can be reset using the software RESET instruction.
- `MOVE.L A7,An` or `MOVE.L An,A7` can be used to save, restore, or change the contents of the `A7` in supervisor mode. `A7` must be loaded in supervisor mode because

MOVE A7 is a privileged instruction. For example, A7 can be initialized to $005000 in supervisor mode using

```
MOVEA.L #$00005000,A1
MOVE.L A1,A7
```

- Consider TRAP #*n*. There are 16 TRAP instructions with *n* ranging from 0 to 15. The hexadecimal vector address is calculated using the equation: Hexadecimal vector address = 80 + 4 × *n*. The TRAP instruction first pushes the contents of the PC and then the SR onto the stack. The hexadecimal vector address is then loaded into PC. TRAP is basically a software interrupt. The TRAP instruction can be used for service calls to the operating system. For application programs running in the user mode, TRAP can be used to transfer control to a supervisor utility program. RTE at the end of the TRAP routine can be used to return to the application program by placing the saved SR from the stack, thus causing the 68000 to return to the user mode.

  There are other traps that occur due to certain arithmetic errors. For example, division by zero automatically traps to location 14_{16}. On the other hand, an overflow condition (i.e., if V = 1) will trap to address $1C_{16}$ if the instruction TRAPV is executed.

- The CHK.W (EA), D*n* instruction compares [D*n*] with (EA). If $[Dn]_{\text{low 16 bits}} < 0$ or if $[Dn]_{\text{low 16 bits}} >$ (EA), then a trap to location 0018_{16} is generated. Also, N is set to 1 if $[Dn]_{\text{low 16 bits}} < 0$, and N is reset to 0 if $[Dn]_{\text{low 16 bits}} >$ (EA). (EA) is treated as a 16-bit twos complement integer. Note that program execution continues if $[Dn]_{\text{low 16 bits}}$ lies between 0 and (EA).

  Consider CHK.W (A5),D2. If $[D2]_{\text{low 16 bits}} = 0200_{16}$, $[A5] = 00003000_{16}$, and $[003000_{16}] = 0100_{16}$, then, after execution of this CHK, the 68000 will trap because $[D2] = 0200_{16}$ is greater than $[003000] = 0100_{16}$.

  The purpose of the CHK instruction is to provide boundary checking by testing if the content of a data register is in the range from zero to an upper limit. The upper limit used in the instruction can be set equal to the length of the array. Then, every time the array is accessed, the CHK instruction can be executed to make sure that the array bounds have not been violated.

  The CHK instruction is usually placed after the computation of an index value to ensure that the index value is not violated. This permits a check of whether or not the address of an array being accessed is within array boundaries when address register indirect with index mode is used to access an array element. For example, the following instruction sequence permits accessing of an array with base address in A2 and array length of 50_{10} bytes:

```
—
—
CHK.W #49,D2
MOVE.B 0(A2,D2*W),D3
—
—
```

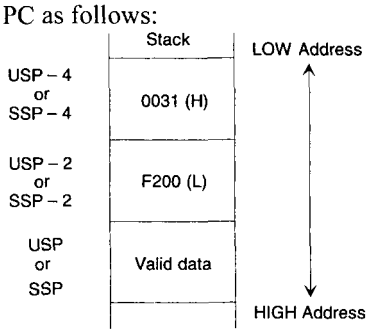
Here, if the low 16 bits of D2 are less than 0 or greater than 49, the 68000 will trap to location 0018_{16}. It is assumed that D2 is computed prior to execution of the CHK instruction.

10.6.9 68000 Stack

The 68000 supports stacks with the address register indirect postincrement and predecrement addressing modes. In addition to two system stack pointers (A7 and A7'), all seven address

registers (A0–A6) can be used as user stack pointers by using appropriate addressing modes. Subroutine calls, traps, and interrupts automatically use the system stack pointers: USP (A7) when S = 0 and SSP (A7') when S = 1. Subroutine calls push the PC onto the system stack; `RTS` pops the PC from the stack. Traps and interrupts push both PC and SR onto the system stack; `RTE` pops PC and SR from the stack.

The 68000 accesses the system stack from the top for operations such as subroutine calls or interrupts. This means that stack operations such as subroutine calls or interrupts access the system stack automatically from HIGH to LOW memory. Therefore, the system SP is decremented by 2 for word or 4 for long word after a push and incremented by 2 for word or 4 for long word after a pop. As an example, suppose that a 68000-`CALL` instruction (JSR or BSR) is executed when PC = $0031F200; then, after execution of the subroutine call, the stack will push the PC as follows:

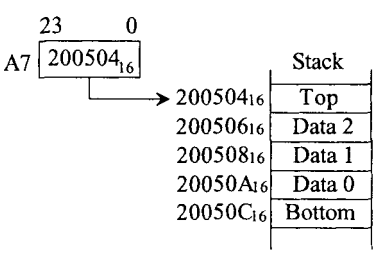


Note that the 68000 SP always points to valid data.

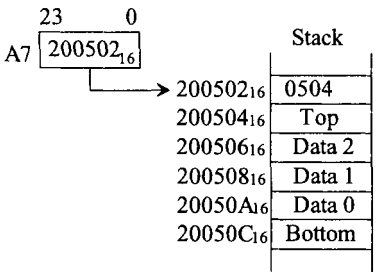
In 68000, stacks can be created by using address register indirect with postincrement or predecrement modes. Typical 68000 memory instructions such as `MOVE` to/from can be used to access the stack. Also, by using one of the seven address registers (A0–A6) and system stack pointers (A7,A7'), stacks can be filled from either HIGH to LOW memory or vice versa:

1. Filling a stack from HIGH to LOW memory (Top of the stack) is implemented with predecrement mode for push and postincrement mode for pop.
2. Filling a stack from LOW to HIGH (Bottom of the stack) memory is implemented with postincrement for push and predecrement for pop.

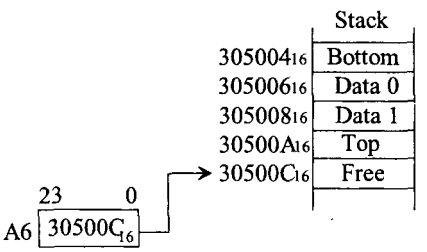
For example, consider the following stack growing from HIGH to LOW memory addresses in which A7 is used as the stack pointer:



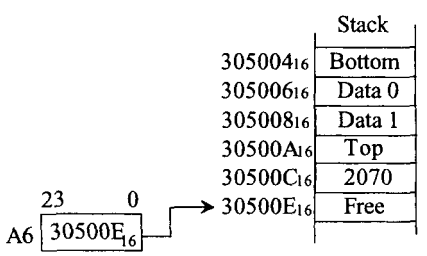
To push the 16-bit contents 0504_{16} of memory location 305016_{16}, the instruction `MOVE.W $305016,-(A7)` can be used as follows:



The 16-bit data item 0504_{16} can be popped from the stack into the low 16 bits of D0 by using `MOVE.W (A7)+,D0`. Register A7 will contain 200504_{16} after the pop. Note that, in this case, the stack pointer A7 points to valid data. Next, consider the stack growing from LOW to HIGH memory addresses in which the user utilizes A6 as the stack pointer:



To push the 16-bit contents 2070_{16} of the low 16 bits of D5, the instruction `MOVE.W D5, (A6)+` can be used as follows. The 16-bit data item 2070_{16} can be popped from the stack into the 16-bit contents of memory location 417024_{16} by using `MOVE.W -(A6), $417024`. Note that, in this case, the stack pointer A6 points to the free location above the valid data.



10.7 68000 Delay Routine

Typical 68000 software delay loops can be written using MOVE and DBF instructions. For example, the following instruction sequence can be used for a delay loop of 2 millisecond:

```
        MOVE.W      #count,D0
DELAY   DBF.W       D0,DELAY
```

Note that DBF.W in the above decrements D0.W by one, and if D0.W ≠ -1 branches to DELAY; if D0.W = -1, the 68000 executes the next instruction. Since DBF.W checks for D0.W for -1, the value of "count" must be one less than the required loop count. The initial loop counter value of "count" can be calculated using the cycles (Appendix D)

required to execute the following 68000 instructions:

```
MOVE.W #n,D0          (8 cycles)
DBF.W  D0,DELAY       (10/14 cycles)
```

Note that the 68000 DBF.W instruction requires two different execution times. DBF.W requires 10 cycles when the 68000 branches if the content of D0.W is not equal to -1after autodecrementing D0.W by 1. However, the 68000 goes to the next instruction and does not branch when [D0.W] = -1 after autodecrementing D0.W by 1, and this requires 14 cycles. This means that the DELAY loop will require 10 cycles for "count" times, and the last iteration will take 14 cycles.

Assuming 4-MHz 68000 clock, each cycle is 250ns. For 2 millisecond delay, total cycles = $\frac{2\,m\,\text{sec}}{250\,n\,\text{sec}}$ = 8,000. The loop will require 10 cycles for "count" times when D0.W $\neq$ -1 and the last iteration will take 14 cycles when no branch is taken (D0.W = -1). Thus, total cycles including the `MOVE.W` = 8 + 10 × (count) + 14 = 8,000. Hence, count $\cong 798_{10} = 031E_{16}$. Therefore, D0.W must be loaded with 798_{10} or $031E_{16}$.

Now, in order to obtain delay of two seconds, the above DELAY loop of 2 millisecond can be used with an external counter. Counter value = $\frac{2\,\text{sec}}{2\,m\,\text{sec}}$ = 1000. The following instruction sequence will provide an approximate delay of two seconds:

```
        MOVE.W  #1000,D1        ;Initialize counter for
                                ;2 second delay
BACK    MOVE.W  #798,D0
DELAY   DBF.W   D0,DELAY        ;20msec delay
        SUBQ.W  #1,D1
        BNE.B   BACK
```

Next, the delay time provided by the above instruction sequence can be calculated. From Appendix D, the cycles required to execute the following 68000 instructions:

```
MOVE.W  #n,D1  (8 cycles)
SUBQ.W #n, D1  (4 cycles)
BNE.B          (10/8 cycles)
```

As before, assuming 4-MHz 68000 clock, each cycle is 250ns. Total time from the above instruction sequence for two-second delay = Execution time for MOVE.W + 1000 * (2 msec delay) + 1000 * (Execution time for SUBQ.W) + 999* (Execution time for BNE.B for Z = 0 when D1 $\neq$ 0) + (Execution time for BNE.B for Z = 1 when D1 = 0 for last iteration) = 8 * 250ns + 1000 * 2msec + 1000 * 4 * 250ns + 999 * 10 * 250ns + 8 * 250ns $\cong$ 2.0035 seconds which is approximately 2 seconds discarding the execution times of MOVE.W, SUBQ.W, and BNE.B.

Example 10.1

Determine the effect of each of the following 68000 instructions:

- `CLR D0`
- `MOVE.L D1, D0`
- `CLR.L (A0) +`
- `MOVE -(A0), D0`
- `MOVE 20(A0), D0`
- `MOVEQ.L #$D7, D0`
- `MOVE 21(A0, A1.L), D0`

Assume the following initial configuration before each instruction is executed; also assume

all numbers in hex:

[D0] = 22224444, [D1] = 55556666
[A0] = 00002224, [A1] = 00003333
[002220] = 8888, [002222] = 7777
[002224] = 6666, [002226] = 5555
[002238] = AAAA, [00556C] = FFFF

Instruction	*Effective Address*	*Net Effect (Hex)*
`CLR D0`	Destination EA = D0	D0 ← 22220000
`MOVE.L D1,D0`	Destination EA = D0	D0 ← 55556666
`CLR.L (A0)+`	Destination EA = [A0]	[002224] ← 0000 [002226] ← 0000 A0 ← 00002228
`MOVE -(A0),D0`	Source EA = [A0] – 2 Destination EA = D0	A0 ← 00002222 D0 ← 22227777
`MOVE 20(A0),D0`	Source EA = [A0] + 20_{10} (or 14_{16}) = 002238 Destination EA = D0	D0 ← 2222AAAA
`MOVEQ.L #$0D7,D0`	Source data = $D7_{16}$ Destination EA = D0	D0 ← FFFFFFD7
`MOVE 21(A0, A1.L),D0`	Source EA = [A0] + [A1] + 21_{10} = $00556C Destination EA = D0	D0 ← 2222FFFF

Example 10.2

Write a 68000 assembly language program that implements each of the following C language program segments:

i)

(a)
```
if    (x >= y)
      x = x + 10;
else y = y - 12;
```

where x is the address of a 16-bit signed integer and, y is the address of a 16-bit signed integer.

(b)
```
sum  = 0;
for  (i = 0; i <= 9; i= i + 1)
sum  = sum + a[i];
```

where sum is the address of the 16-bit result of addition.

ii) Write a 68000 assembly language program to find (X^2) / (32765_{10}) where X is a 16-bit signed number stored in D0.W. Store the 32-bit result (quotient and remainder) onto the user stack.

iii) What are the remainder, quotient, and register containing them after execution of the following 68000 instruction sequence?

```
MOVE.W #0FFFFH, D1
DIVS.W #2, D1
```

Solution

i)

(a)
```
x       EQU    100
y       EQU    200
        LEA.L  x,A0          ; Initialize A0
        LEA.L  y,A1          ; Initialize A1
        MOVE.W (A0),D0       ; Move [x] into D0
        CMP.W  (A1),D0       ; Compare [x] with [y]
        BGE.B  THPRT
        SUBI.W #12,(A1)      ; Execute else part
        BRA.B  STAY
THPRT   ADDI.W #10,(A0)      ; Execute then part
STAY    JMP    STAY          ; Halt
```
(b) Assume register A0 holds the address of the first element of the array.
```
SUM     EQU    300           ;  Initialize SUM to 300 for result
        LEA.L  200,A0        ;  Point A0 to a[0]
        CLR.W  D0            ;  Clear the sum to zero
        MOVE.W #9,D1         ;  Initialize D1 with loop limit
LOOP    ADD.W (A0)+,D0       ;  Perform the iterative summation
        DBF.W  D1,LOOP
        MOVE.W D0,SUM        ;  Store 16-bit result in address SUM
FINISH  JMP   FINISH         ;  Halt
```

Note that, in the above condition F in DBF is always false. Hence, the program exits from the LOOP when D1= -1. Therefore, the addition process is performed 10 times.

ii)
```
        MULS.  D0,D0         ;  Compute X² and store in D0.L
        DIVU.W #32765,D0     ;  Since X² and 32765 are both
                             ;  positve, use
        MOVE.L D0,-(A7)      ;  unsigned division.
                             ;  Remainder in high word
FINISH  JMP    FINISH        ;  of D0 and quotient in low word
                             ;  of D0. Push
                             ;  D0.L to stack
```

iii)
```
MOVE.W #0FFFFH, D1           ; D1 = FFFFH = -1
DIVS.W #2, D1                ; D1/2 = -1/2
```

High D1.W	Low D1.W
FFFFH	0000H
16-bit remainder = -1_{10}	16-bit quotient = 0

Example 10.3

Write a 68000 assembly program at address $002000 to clear 100_{10} consecutive bytes (from low to high addresses) to zero starting at location $003000.

Solution
```
00002000                        1         ORG $2000
00002000   207C 00003000        2         MOVEA.L #$3000,A0 ;LOAD A0 WITH $3000
00002006   303C 0063            3         MOVE.W #99,D0     ;MOVE 99 INTO  D0
0000200A   4218                 4 LOOP    CLR.B (A0)+       ;CLEAR[3000H]+
0000200C   51C8 FFFC            5         DBF.W D0,LOOP     ;DECREMENT AND
                                                            ;BRANCH
00002010   4EF8 2010            6 FINISH  JMP FINISH        ;HALT
No errors detected
```

```
No warnings generated
```

Note that the 68000 has no HALT instruction.. Therefore, the unconditional jump to the same location such as `FINISH JMP FINISH` is normally used at the end of the program. Because `DBF` is a word instruction and considers D0's low 16-bit word as the loop count, one should be careful about initializing D0 using `MOVEQ.L #d8,Dn` since this instruction sign extends low byte of Dn to 32 bits.

Example 10.4

Write a 68000 assembly language program at address $001000 to compute $\sum_{i=1}^{N} X_i Y_i$, where X_i and Y_i are signed 16-bit numbers and $N = 100$. Store the 32-bit result in D1. Assume that the starting addresses of X_i and Y_i are 100_{16} and 200_{16} respectively.

Solution

```
00000000  =00000100     1  P       EQU  $100
00000000  =00000200     2  Q       EQU  $200
00001000                3          ORG  $1000
00001000  303C 0063     4          MOVE.W #99,D0     ;MOVE 99 INTO DO
00001004  41F8 0100     5          LEA.L P,A0        ;LOAD ADDRESS P INTO A0
00001008  43F80200      6          LEA.L Q,A1        ;LOAD ADDRESS Q INTO A1
0000100C  4281          7          CLR.L D1          ;INITIALIZE D1 TO ZERO
0000100E  3418          8LOOP      MOVE.W (A0)+,D2   ;MOVE [X] TO D2
00001010  C5D9          9          MULS.W (A1)+,D2   ;D2 <--[X]*[Y]
00001012  D282          10         ADD.L D2,D1       ;D1 <-- SUM XiYi
00001014  51C8 FFF8     11         DBF.W D0,LOOP     ;DECREMENT AND BRANCH
00001018  4EF8 1018     12FINISH   JMP FINISH        ;HALT
0000101C                13
No errors detected
No warnings generated
```

Note: In order to execute the above program, values for X_i and Y_i must be stored in memory using assembler directive, DC.W.

Example 10.5

Write a 68000 subroutine to compute $Y = \sum_{i=1}^{N} X_i^2 / N$. Assume the X_i's are 16-bit signed integers and $N = 100$. The numbers are stored in consecutive locations. Assume A0 points to the X_i's and A7 is already initialized in the main program. Store 32-bit result in D1 (16-bit remainder in high word of D1 and 16-bit quotient in the low word of D1). Assume user mode.

Solution

```
00000000  48E7 3080     1 SQR   MOVEM.L D2/D3/A0,-(A7);SAVE REGISTERS
00000004  4281          2       CLR.L D1              ;CLEAR SUM
00000006  343C 0063     3       MOVE.W #99,D2         ;INITIALIZE LOOP COUNT
0000000A  3618          4 BACK  MOVE.W (A0)+,D3       ;MOVE Xi's INTO D3
0000000C  C7C3          5       MULS.W D3,D3          ;COMPUTE Xi**2 USING
                                                      ;MULS
0000000E  D283          6       ADD.L D3,D1           ;SINCE Xi**2 IS
                                                      ;ALWAYS +VE
00000010  51CA FFF8     7       DBF.W D2,BACK         ;COMPUTE
00000014  82FC 0064     8       DIVU.W #100,D1        ;SUM OF Xi**2/N
                                                      ;USING DIVU
00000018  4CDF 0004     9       MOVEM.L(A7)+,D2/D3/A0 ;RESTORE REGISTERS
0000001C  4E75          10      RTS
No errors detected
No warnings generated
```

In the above program, DIVU is used for computing $\sum X_i^2/N$ since both SUM (X_i**2) and N=100 are unsigned (positive). Note that in order to execute the above program, values for X_i must be stored in memory using assembler directive, DC.W.

Example 10.6

Write a 68000 assembly language program at address 0 to move a block of 16-bit data of length 100_{10} from the source block starting at location 002000_{16} to the destination block starting at location 003000_{16} from low to high addresses.

Solution

```
00000000  387C 2000   1        MOVEA.W #$2000,A4   ;LOAD A4 WITH SOURCE ADDR
00000004  3A7C 3000   2        MOVEA.W #$3000,A5   ;LOAD A5 WITH DEST ADDR
00000008  303C 0063   3        MOVE.W #99,D0       ;LOAD D0 WITH COUNT -1=99
0000000C  3ADC        4 START  MOVE.W (A4)+,(A5)+  ;MOVE SOURCE DATA TO DEST
0000000E  51C8 FFFC   5        DBF.W D0,START      ;BRANCH IF D0≠-1
00000012  4EF8 0012   6 STAY   JMP STAY            ;HALT
No errors detected
No warnings generated
```

Note: Typical assemblers assemble a program starting at address 0 if assembler directive ORG is not used at the beginning of the program.

Example 10.7

Write a 68000 assembly language program at address 0 to add two words, each containing two ASCII digits. The first word is stored in two consecutive locations (from LOW to HIGH) with the low byte pointed to by A0 at address 000300_{16}, and the second word is stored in two consecutive locations (from LOW to HIGH) with the low byte pointed to by A1 at 000700_{16}. Store the packed BCD result in D5.

Solution

```
00000000  7401       1       MOVEQ.L #1,D2
00000002  307C 0300   2       MOVEA.W #$0300,A0  ;INITIALIZE A0
00000006  327C 0700   3       MOVEA.W #$0700,A1  ;INITIALIZE A1
0000000A  0218 000F  4START   ANDI.B #$0F,(A0)+  ;CONVERT IST # TO UNPAC.BCD
0000000E  0219 000F   5       ANDI.B #$0F,(A1)+  ;CONVERT 2ND # TO UNPAC.BCD
00000012  51CA FFF6   6       DBF.W D2,START
00000016  1C20        7       MOVE.B -(A0),D6    ;GET HIGH UNPAC.BYTE OF IST#
00000018  1E20        8       MOVE.B -(A0),D7    ;GET LOW UNPAC. BYTE OF IST#
0000001A  E90E        9       LSL.B #4,D6        ;SHIFT IST# HIGH BYTE 4
                                                 ;TIMES
0000001C  8C07        10      OR.B D7,D6         ;D6=PACKED BCD BYTE OF IST#
0000001E  1A21        11      MOVE.B -(A1),D5    ;GET HIGH UNPAC. BYTE OF
                                                 ;2ND#
00000020  1821        12      MOVE.B -(A1),D4    ;GET LOW UNPAC. BYTE OF 2ND#
00000022  E90D        13      LSL.B #4,D5        ;SHIFT 2ND # HIGH BYTE 4
                                                 ;TIMES
00000024  8A04        14      OR.B D4,D5         ;D5 HAS PACKED BCD BYTE OF
                                                 ;2ND#
00000026  0600 0000  15       ADDI.B #0,D0       ;CLEAR X-BIT
0000002A  CB06       16       ABCD.B D6,D5       ;D5.B =PACKED BCD RESULT
0000002C  4EF8 002C  17       FINISH JMP FINISH
No errors detected
No warnings generated
```

Example 10.8

Write a 68000 assembly language program that will perform : $5 \times X + 6 \times Y + [Y/8] \rightarrow$[D1.L] where X is an unsigned 8-bit number stored in the lowest byte of D0 and Y is a 16-bit signed number stored in the upper 16 bits of D1. Neglect the remainder of $Y/8$.

Solution

```
00000000   0240 00FF   1  ANDI.W #$00FF,D0     ;CONVERT X TO UNSIGNED 16-BIT
00000004   C0FC 0005   2  MULU.W #5,D0         ;COMPUTE UNSIGNED 5*X IN D0.L
00000008   4841        3  SWAP.W D1           ;MOVE Y TO LOW 16 BITS IN D1
0000000A   3401        4  MOVE.W D1,D2        ;SAVE Y TO LOW 16 BITS OF D2
0000000C   C3FC 0006   5  MULS.W #6,D1        ;COMPUTE SIGNED 6*Y IN D1.L
00000010   D280        6  ADD.L D0,D1         ;ADD 5*X WITH 6*Y
00000012   48C2        7  EXT.L D2            ;SIGN EXTEND
00000014   E682        8  ASR.L #3,D2         ;PERFORM Y/8;DISCARD REMAINDER
00000016   D282        9  ADD.L D2,D1         ;PERFORM 5*X+6*Y +Y/8
00000018   4EF8 0018   10 FINISH JMP FINISH
No errors detected
No warnings generated
```

Example 10.9

Write a 68000 assembly language program to convert temperature from Fahrenheit to Celsius using the following equation: $C = [(F - 32)/9] \times 5$; assume that the low byte of D0 contains the temperature in Fahrenheit. The temperature can be positive or negative. Store result in D0.

Solution

```
00000000   4880        1         EXT.W D0      ;SIGN EXTEND (F) LOW BYTE OF D0
00000002   0440 0020   2         SUBI.W #32,D0 ;PERFORM F-32
00000006   C1FC 0005   3         MULS.W #5,D0  ;PERFORM 5* (F-32)/9 AND STORE
0000000A   81FC 0009   4         DIVS.W #9,D0  ;REMAINDER IN HIGH WORD OF D0
0000000E   4EF8 000E   5 FINISH  JMP FINISH;AND QUOTIENT IN LOW WORD OF D0
No errors detected
No warnings generated
```

Example 10.10

Write a 68000 assembly language program at address $4000 to add four 32-bit numbers stored in consecutive locations starting at address $3000. Store the 32-bit result onto the user stack. Assume that no carry is generated due to addition of two consecutive 32-bit numbers and A7 is already initialized.

Solution

```
00003000                            1          ORG      $3000
00003000   00000001 00000002        2          DC.L     1,2,3,4
00003002   00000003 00000004
00004000                            3          ORG      $4000
00004000   7003                     4          MOVEQ.L  #3,D0
00004002   207C 00003000            5          MOVEA.L  #$3000,A0
00004008   4281                     6          CLR.L    D1
0000400A   D298                     7 START    ADD.L    (A0)+,D1
0000400C   51C8 FFFC                8          DBF.W    D0,START
00004010   2F01                     9          MOVE.L   D1,-(A7)
00004012   4EF8 4012                10 FINISH  JMP      FINISH

No errors detected
No warnings generated
```

Example 10.11

Write a subroutine in 68000 assembly language to implement the C language assignment statement: p = p + q; where addresses p and q hold two 16-digit (64-bit) packed BCD numbers (N1 and N2). The main program will initialize addresses p and q to $002000 and $003000 respectively. Address $002007 will hold the lowest byte of N1 with the highest byte at address $002000 while Address $003007 will contain the lowest byte of N2 with

the highest byte at address $003000. Also, write the main program at address $004000 which will perform all initializations including address p (pointer A0 to $002000), address q (pointer A1 to $003000), loop count (D1 to 7), and then call the subroutine at $008000 and stop. The subroutine will accomplish the task with the initialized values of A0, A1, and D1 in the main program. Use ABCD.B for BCD addition with predecrement mode. Assume supervisor mode. Note that the 68000 supervisor stack pointer is initialized upon hardware reset.

Solution

```
00004000                        1          ORG $004000
00004000  307C 2000             2          MOVEA.W #$2000,A0
00004004  327C 3000             3          MOVEA.W #$3000,A1
00004008  323C 0007             4          MOVE.W #7,D1
0000400C  4EB9 00008000         5          JSR BCDADD
00004012  4EF8 4012             6  STAY    JMP STAY
00004016                        7
00008000                        8          ORG $008000
00008000  41F0 1001             9  BCDADD  LEA.L 1(A0,D1.W),A0      ;UPDATE A0
00008004  43F1 1001             10         LEA.L 1(A1,D1.W),A1      ;AND A1
00008008  0600 0000             11         ADDI.B #0,D0             ;X-BIT =0
0000800C  C109                  12 ALOOP   ABCD.B -(A1),-(A0)       ;ADD
0000800E  51C9 FFFC             13         DBF.W D1,ALOOP
00008012  4E75                  14         RTS
No errors detected
No warnings generated
```

Example 10.12

Write a 68000 assembly program to multiply an 8-bit signed number in the low byte of D1 by a 16-bit signed number in the high word of D5. Store the result in D3.

Solution

```
00000000  4881          1     EXT.W D1      ;SIGN EXTENDS LOW BYTE OF D1
00000002  4845          2     SWAP.W D5     ;SWAP LOW WORD WITH HIGH
                              ;WORD OF D5
00000004  CBC1          3     MULS.W D1,D5  ;MULTIPLY D1 WITH D5,
                              ;STORE RESULT
00000006  2605          4     MOVE.L D5,D3  ;COPY RESULT IN D3
00000008  4EF8 0008     5 FINISH JMP FINISH
No errors detected
No warnings generated
```

Example 10.13

Write a 68000 assembly language program at address $2000 to add ten 32-bit numbers stored in consecutive locations starting at address $502040. Initialize A6 to $00200504 and use the low 24 bits of A6 as the stack pointer to push the 32-bit result. Use only ADDX instruction for adding two 32-bit numbers each time through the loop. Assume that no carry is generated due to the addition of two consecutive 32-bit numbers; this will provide the 32-bit result. This example illustrates use of the 68000 ADDX instruction.

Solution

```
00001000                                  1           ORG     $1000
00000002  00000002 00000003 00000007 ... 2           DC.L    2,3,7,5,1,9,6,4,6,1
00001028  =00001000                       3  START_ADR EQU     $1000
00002000                                  4           ORG     $2000
00002000  =00000009                       5  COUNT    EQU     9
00002000  207C 00001000                   6           MOVEA.L #START_ADR,A0 ;LOAD STARTING
                                                                ;ADDRESS IN A0
00002006  103C 0009                       7           MOVE.B  #COUNT,D0     ;USE D0 AS A
                                                                ;COUNTER
0000200A  2C7C 00200504                   8           MOVEA.L #$00200504,A6 ;USE A6 AS THE
```

```
                                                                ;SP
00002010  4281                          9           CLR.L   D1          ;CLEAR D1
                                                                ;REGISTER
00002012  0606 0000                     10          ADDI.B  #0,D6       ;CLEAR X BIT
00002016  2618                          11  AGAIN   MOVE.L  (A0)+,D3    ;MOVE A 32 BIT
                                                                ;NUMBER
                                                                ;IN D3
00002018  D383                          12          ADDX.L  D3,D1       ;ADD NUMBERS
                                                                ;USING
                                                                ;ADDX
0000201A  51C8 FFFA                     13          DBF.W   D0,AGAIN    ;REPEAT UNTIL
                                                                ;D0=-1
0000201E  2D01                          14          MOVE.L  D1,-(A6)    ;PUSH 32-bit
                                                                ;RESULT
                                                                ;ONTO STACK
00002020  4EF8 2020                     15  FINISH  JMP     FINISH

No errors detected
No warnings generated
```

Note that ADDX adds the contents of two data registers or the contents of two memory locations using predecrement modes.

Example 10.14

Write a 68000 assembly language program at address $2000 to subtract two 32-bit packed BCD numbers. The BCD number 1 is stored at the locations starting from $003000 through $003003, with the least significant byte at $003003 and the most significant byte at $003000. Similarly, the BCD number 2 is stored at the locations starting from $004000 through $004003, with the least significant byte at $004003 and the most significant byte at $004000. The BCD number 2 is to be subtracted from BCD number 1. Store the packed BCD result at addresses $005000 (Lowest byte of the result) through $005003 (Highest byte of the result). In the program, first initialize loop counter D7 to 4, source pointer A0 to $003000, source pointer A1 to $004000, destination pointer A3 to $005000, and then write the program to accomplish the above using these initialized values.

Solution

```
00003000                1          ORG     $003000
00003000  99221133      2          DC.L    $99221133
00004000                3          ORG     $004000
00004000  33552211      4          DC.L    $33552211
00002000                5          ORG     $2000
00002000  3E3C 0004     6          MOVE.W  #4,D7       ;NUMBER OF BYTES TO BE SUBTRACTED
00002004  307C 3000      7         MOVEA.W #$3000,A0   ;STARTING ADDRESS FOR FIRST NUMBER
00002008  327C 4000      8         MOVEA.W #$4000,A1   ;STARTING ADDRESS FOR SECOND NUMBER
0000200C  D0C7           9         ADDA.W  D7,A0       ;MOVE ADDRESS POINTERS TO THE END
0000200E  D2C7          10         ADDA.W  D7,A1       ;OF EACH 32 BIT PACKED BCD NUMBER
00002010  367C 5000     11         MOVEA.W #$5000,A3   ;LOAD POINTER FOR DESTINATION ADDR
00002014  5347          12         SUBQ.W  #1,D7       ;SUBTRACT D7 by 1 for DBF
00002016  0607 0000     13         ADDI.B  #0,D7       ;CLEAR X-BIT
0000201A  1020          14 LOOP    MOVE.B  -(A0),D0    ;GET A BYTE FROM FIRST NUMBER
0000201C  1221          15         MOVE.B  -(A1),D1    ;GET A BYTE FROM SECOND NUMBER
0000201E  8101          16         SBCD.B  D1,D0       ;BCD SUBTRACTION, RESULT IN D0
00002020  16C0          17         MOVE.B  D0,(A3)+    ;STORE RESULT IN DESTINATION ADDR
00002022  51CF FFF6     18         DBF     D7,LOOP     ;CONTINUE UNTIL COUNTER HAS EXPIRED
00002026  4EF8 2026     19 FINISH  JMP     FINISH

No errors detected
No warnings generated
```

Note that SBCD subtracts the contents of two data registers or the contents of two memory locations using predecrement modes.

Example 10.15

Write a 68000 assembly program at address $1000 which is equivalent to the following C language segment:

sum = 0;
for (i = 0; i <= 9; i = i + 1)
sum = sum + x[i] * y[i];

Assume that the arrays, x[i] and y[i] contain unsigned 16-bit numbers already stored in memory starting at addresses $3000 and $4000 respectively. Store the 32-bit result at address $5000.

Solution

```
00001000                1           ORG $1000
00001000  =00003000     2  x        EQU $3000
00001000  =00004000     3  y        EQU $4000
00001000  =00005000     4 sum       EQU $5000
00001000                5
00001000  303C 0009     6           MOVE.W #9,D0  ;USE D0 AS A LOOP COUNTER
00001004  41F8 3000     7           LEA.L x,A0    ;INITIALIZE A0 WITH x
00001008  43F8 4000     8           LEA.L y,A1    ;INITIALIZE A1 WITH y
0000101C  45F8 5000     9           LEA.L sum,A2  ;INITIALIZE A2 WITH SUM
00001010  4285          10          CLR.L D5      ;CLEAR SUM TO 0
00001012  3418          11 LOOP     MOVE.W (A0)+,D2;MOVE X[i] INTO D2
00001014  C4D9          12          MULU.W (A1)+,D2;COMPUTE X[i] *y[i]
00001016  DA82          13          ADD.L D2,D5    ;UPDATE SUM
00001018  51C8 FFF8     14          DBF.W D0,LOOP  ;REPEAT UNTIL D0=-1
0000101C  2485          15          MOVE.L D5,(A2) ;STORE SUM IN MEMORY
0000101E  4EF8 101E     16 FINISH   JMP FINISH
No errors detected
No warnings generated
```

10.8 68000 Pins And Signals

The 68000 is usually packaged in one of the following:

a) 64-pin dual in-line package (DIP)
b) 68-pin quad pack
c) 68-terminal chip carrier
d) 68-pin grid array (PGA)

Figure 10.6 shows the 68000 pin diagram for the DIP. Appendix C provides data sheets for the 68000 and support chips.

The 68000 is provided with two V_{cc} (+5 V) and two ground pins. Power is thus distributed in order to reduce noise problems at high frequencies. Also, to build a prototype to demonstrate that the paper design for the 68000-based microcomputer is correct, one must use either wire-wrap or solder for the actual construction. Prototype board must not be used because, at high frequencies (above 4 MHz), there will be noise problems due to stray capacitances. The 68000 consumes about 1.5 W of power.

D_0–D_{15} are the 16 data bus pins. All transfers to and from memory and I/O devices are conducted over the 8-bit (LOW or HIGH) or 16-bit data bus depending on the size of the device. A_1–A_{23} are the 23 address lines. A_0 is obtained by encoding the $\overline{UDS}$ (upper data strobe) and $\overline{LDS}$ (lower data strobe) lines.

The 68000 operates on a single-phase TTL-level clock at 4, 6, 8, 10, 12.5, 16.67, or 25 MHz. The clock signal must be generated externally and applied to the 68000 clock input line. An external crystal oscillator chip is required to generate the clock. Figure 10.7 shows the 68000 CLK waveform and clock timing specifications. The clock is at TTL-compatible voltage. The clock timing specifications provide data for three different clock frequencies: 8 MHz, 10 MHz, and 12.5 MHz The 68000 CLK input can be provided by an external crystal oscillator or by designing an external circuit.

The 68000 signals can be divided into five functional categories:

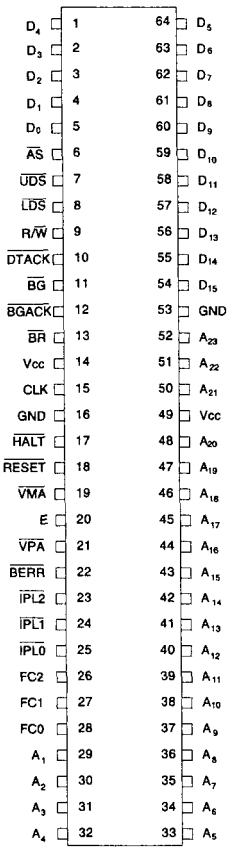


FIGURE 10.6 68000 pins and signals

Characteristic	Symbol	8 MHz		10 MHz		12.5 MHz		Unit
		Min	Max	Min	Max	Min	Max	
Frequency of operation	f	4.0	8.0	4.0	10.0	4.0	12.5	MHz
Cycle time	t_{CYC}	125	250	100	250	80	250	ns
Clock pulse width	t_{CL}	55	125	45	125	35	125	ns
	t_{CH}	55	125	45	125	35	125	
Rise and fall times	t_{Cr}	—	10	—	10	—	5	ns
	t_{Cf}	—	10	—	10	—	5	

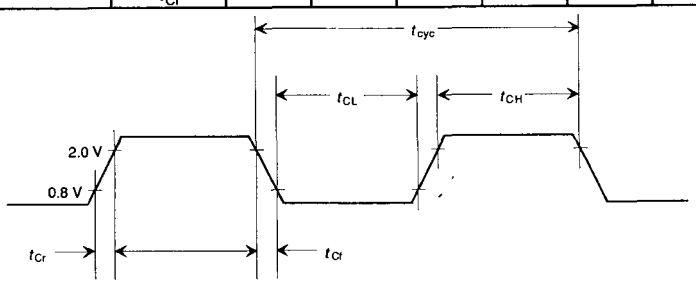


FIGURE 10.7 68000 clock input timing diagram and AC electrical specifications

1. Synchronous and asynchronous control lines
2. System control lines
3. Interrupt control lines
4. DMA control lines
5. Status lines

10.8.1 Synchronous and Asynchronous Control Lines

The 68000 bus control is asynchronous. This means that once a bus cycle is initiated, the external device must send a signal back to complete it. The 68000 also contains three synchronous control lines that facilitate interfacing to synchronous peripheral devices such as Motorola's inexpensive MC6800 family.

Synchronous operation means that bus control is synchronized or clocked using a common system clock signal. In 6800 family peripherals, this common clock is the E clock signal depending on the particular chip used. With synchronous control, all READ and WRITE operations must be synchronized with the common clock. However, this may create problems when interfacing with slow peripheral devices. This problem does not arise with asynchronous bus control.

Asynchronous operation is not dependent on a common clock signal. The 68000 utilizes the asynchronous control lines to transfer data between the 68000 and peripheral devices via handshaking. Using asynchronous operation, the 68000 can be interfaced to any peripheral chip regardless of the speed.

The 68000 has three control lines to transfer data over its bus in a synchronous manner: E (enable), $\overline{\text{VPA}}$ (valid peripheral address), and $\overline{\text{VMA}}$ (valid memory address). The E clock corresponds to the clock of the 6800. The E clock is output at a frequency that is one tenth of the 68000 input clock. $\overline{\text{VPA}}$ is an input and tells the 68000 that a 6800 device is being addressed and therefore the data transfer must be synchronized with the E clock. $\overline{\text{VMA}}$ is the processor's response to $\overline{\text{VPA}}$. $\overline{\text{VMA}}$ is asserted when the memory address is valid. This also tells the external device that the next data transfer over the data bus will be synchronized with the E clock.

$\overline{\text{VPA}}$ can be generated by decoding the address pins and address strobe ($\overline{\text{AS}}$). Note that the 68000 asserts $\overline{\text{AS}}$ LOW when the address on the address bus is valid. $\overline{\text{VMA}}$ is typically used as the chip select of the 6800 peripheral. This ensures that the 6800 peripherals are selected and deselected at the correct time. The 6800 peripheral interfacing sequence is as follows:

1. The 68000 initiates a cycle by starting a normal read or write cycle.
2. The 6800 peripheral defines the 68000 cycle by asserting the 68000 $\overline{\text{VPA}}$ input. If $\overline{\text{VPA}}$ is asserted as soon as possible after assertion of $\overline{\text{AS}}$, then $\overline{\text{VPA}}$ will be recognized as being asserted after three cycles. If $\overline{\text{VPA}}$ is not asserted after three cycles, the 68000 inserts wait states until $\overline{\text{VPA}}$ is recognized by the 68000 as asserted. $\overline{\text{DTACK}}$ should not be asserted while $\overline{\text{VPA}}$ is asserted. The 6800 peripheral must remove $\overline{\text{VPA}}$ within 1 clock period after $\overline{\text{AS}}$ is negated.
3. The 68000 monitors enable (E) until it is LOW. The 68000 then synchronizes all READ and WRITE operations with the E clock. The $\overline{\text{VMA}}$ output pin is asserted LOW by the 68000.
4. The 6800 peripheral waits until E is active (HIGH) and then transfers the data.
5. The 68000 waits until E goes to LOW (on a read cycle, the data is latched as E goes to LOW internally). The 68000 then negates $\overline{\text{VMA}}$, $\overline{\text{AS}}$, $\overline{\text{UDS}}$, and $\overline{\text{LDS}}$. The

68000 thus terminates the cycle and starts the next cycle.

The 68000 utilizes five lines to control address and data transfers asynchronously: $\overline{AS}$(address strobe), R/$\overline{W}$ (read/write), $\overline{DTACK}$ (data acknowledge), $\overline{UDS}$ (upper data strobe), and $\overline{LDS}$ (lower data strobe).

The 68000 outputs to notify the peripheral device when data is to be transferred. $\overline{AS}$ is active LOW when the 68000 provides a valid address on the address bus. The R/$\overline{W}$ output line indicates whether the 68000 is reading data from or writing data into a peripheral device. R/$\overline{W}$ is HIGH for read and LOW for write. $\overline{DTACK}$ is used to tell the 68000 that a transfer is to be performed. When the 68000 wants to transfer data asynchronously, it first activates the $\overline{AS}$ line and at the same time generates the required address on the address lines to select the peripheral device.

Because the $\overline{AS}$ line tells the peripheral chip when to transfer data, the $\overline{AS}$ line should be part of the address decoding scheme. After enabling $\overline{AS}$, the 68000 enters the wait state until it receives $\overline{DTACK}$ from the selected peripheral device. On receipt of $\overline{DTACK}$, the 68000 knows that the peripheral device is ready for data transfer. The 68000 then utilizes the R/$\overline{W}$ and data lines to transfer data. $\overline{UDS}$ and $\overline{LDS}$ are defined as follows:

$\overline{UDS}$	$\overline{LDS}$	Data Transfer Occurs Via:	Address
1	0	D_0–D_7 pins for byte	Odd
0	1	D_8–D_{15} pins for byte	Even
0	0	D_0–D_{15} pins for word or long word	Even

A_0 is encoded from $\overline{UDS}$ and $\overline{LDS}$. When $\overline{UDS}$ is asserted, the contents of even addresses are transferred on the high-order eight lines of the data bus, D_8–D_{15}. The 68000 internally shifts this data to the low byte of the specified register. When $\overline{LDS}$ is asserted, the contents of odd addresses are transferred on the low-order eight lines of the data bus, D_0–D_7. During word and long word transfers, both $\overline{UDS}$ and $\overline{LDS}$ are asserted and information is transferred on all 16 data lines, D_0–D_{15} pins. Note that during byte memory transfers, A_0 corresponds to $\overline{UDS}$ for even addresses ($A_0 = 0$) and to $\overline{LDS}$ for odd addresses ($A_0 = 1$). The circuit in Figure 10.8 shows how even and odd addresses are interfaced to the 68000.

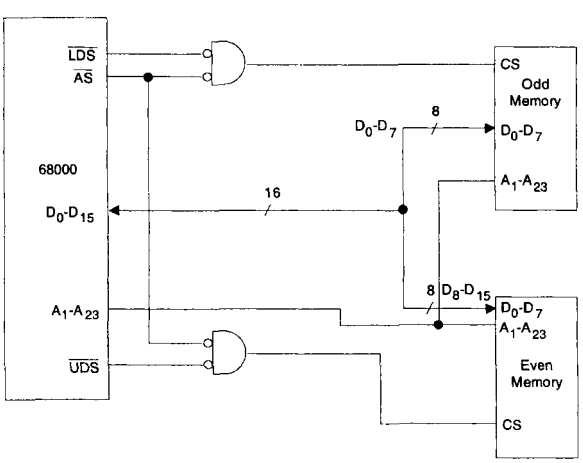


FIGURE 10.8 Interfacing of the 68000 to even and odd addresses

10.8.2 System Control Lines

The 68000 has three control lines, $\overline{\text{BERR}}$ (bus error), $\overline{\text{HALT}}$, and $\overline{\text{RESET}}$, which are used to control system-related functions. $\overline{\text{BERR}}$ is an input to the 68000 and is used to inform the processor that there is a problem with the instruction cycle currently being executed. With asynchronous operation, this problem may arise if the 68000 does not receive $\overline{\text{DTACK}}$ from a peripheral device. An external timer can be used to activate the $\overline{\text{BERR}}$ pin if the external device does not send $\overline{\text{DTACK}}$ within a certain period of time. On receipt of $\overline{\text{BERR}}$, the 68000 does one of the following:

- Reruns the instruction cycle that caused the error.
- Executes an error service routine.

The troubled instruction cycle is rerun by the 68000 if it receives a $\overline{\text{HALT}}$ signal along with the $\overline{\text{BERR}}$ signal. On receipt of LOW on both the $\overline{\text{HALT}}$ and $\overline{\text{BERR}}$ pins, the 68000 completes the current instruction cycle and then goes into the high-impedance state. On removal of both $\overline{\text{HALT}}$ and $\overline{\text{BERR}}$ (that is, when both $\overline{\text{HALT}}$ and $\overline{\text{BERR}}$ are HIGH), the 68000 reruns the troubled instruction cycle. The cycle can be rerun repeatedly if both $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ are enabled/disabled continually.

On the other hand, an error service routine is executed only if the $\overline{\text{BERR}}$ signal is received without $\overline{\text{HALT}}$. In this case, the 68000 will branch to a bus error vector address where the user can write a service routine. If two simultaneous bus errors are received via the $\overline{\text{BERR}}$ pin without $\overline{\text{HALT}}$, the 68000 automatically goes into the halt state until it is reset.

The $\overline{\text{HALT}}$ line can also be used by itself to perform single stepping or to provide DMA. When the $\overline{\text{HALT}}$ input is activated, the 68000 completes the current instruction and goes into a high-impedance state until $\overline{\text{HALT}}$ is returned to HIGH. By enabling/disabling the $\overline{\text{HALT}}$ line continually, the single-stepping debugging can be accomplished. However, because most 68000 instructions consist of more than one clock cycle, single stepping using $\overline{\text{HALT}}$ is not normally used. Rather, the trace bit in the status register is used to single-step the complete instruction.

One can also use $\overline{\text{HALT}}$ to perform microprocessor-halt DMA. Because the 68000 has separate DMA control lines, DMA using the $\overline{\text{HALT}}$ line will not normally be used. The $\overline{\text{HALT}}$ pin can also be used as an output signal. The 68000 will assert the $\overline{\text{HALT}}$ pin LOW when it goes into a halt state as a result of a catastrophic failure. The double bus error (activation of $\overline{\text{BERR}}$ twice) is an example of this type of error. When this occurs, the 68000 goes into a high-impedance state until it is reset. The $\overline{\text{HALT}}$ line informs the peripheral devices of the catastrophic failure.

The $\overline{\text{RESET}}$ line of the 68000 is also bidirectional. To reset the 68000, both the $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ pins must be LOW for 10 clock cycles at the same time except when Vcc is initially applied to the 68000. In this case, an external reset must be applied for at least 100 ms. The 68000 executes a reset service routine automatically for loading the PC with the starting address of the program.

The 68000 $\overline{\text{RESET}}$ pin can also be used as an output line. A LOW can be sent to this output line by executing the RESET instruction in the supervisor mode in order to reset external devices connected to the 68000 $\overline{\text{RESET}}$ pin. Upon execution of the RESET instruction, the 68000 drives the $\overline{\text{RESET}}$ pin LOW for 124 clock periods and does not affect any data, address, or status registers. Therefore, the RESET instruction can be placed anywhere in the program whenever the external devices need to be reset.

Upon hardware reset, the 68000 sets the S-bit in SR to 1, and then loads the supervisor stack pointer from location $000000 (high 16 bits) and $000002 (low 16 bits)

and loads the PC from $000004 (high 16 bits) and $000006 (low 16 bits); but the low 24 bits are used. In addition, the 68000 clears the trace bit in SR to 0 and sets bits I2 I1 I0 in SR to 111. All other registers are unaffected.

10.8.3 Interrupt Control Lines

$\overline{IPL0}$, $\overline{IPL1}$, and $\overline{IPL2}$ are the three interrupt control lines These lines provide for seven interrupt priority levels ($\overline{IPL2}$, $\overline{IPL1}$, $\overline{IPL0}$ = 111 means no interrupt, and $\overline{IPL2}$, $\overline{IPL1}$, $\overline{IPL0}$ = 000 means nonmaskable interrupt with the highest priority). The 68000 interrupts will be discussed later in this chapter.

10.8.4 DMA Control Lines

The $\overline{BR}$ (bus request), $\overline{BG}$ (bus grant), and $\overline{BGACK}$ (bus grant acknowledge) lines are used for DMA purposes. The 68000 DMA will be discussed later in this chapter.

10.8.5 Status Lines

The 68000 has the three output lines called function code pins (output lines) FC2, FC1, and FC0. These lines tell external devices whether user data/program or supervisor data/program is being addressed. These lines can be decoded to provide user or supervisor programs/data and interrupt acknowledge as shown in Table 10.13.

The FC2, FC1, and FC0 pins can be used to partition memory into four functional areas: user data memory, user program memory, supervisor data memory, and supervisor program memory. Each memory partition can directly access up to 16 megabytes, and thus the 68000 can be made to directly address up to 64 megabytes of memory. This is shown in Figure 10.9.

10.9 68000 Clock and Reset Signals

This section covers generation of 68000 clock and reset signals in detail because the clock signal and the reset pins are two important signals of any microprocessor.

10.9.1 68000 Clock Signals

As mentioned before, the 68000 does not include an on-chip clock generation circuitry. This means that an external crystal oscillator chip is required to generate the clock. The 68000 CLK input can be provided by a crystal oscillator or by designing an external circuit. Figure 10.10 shows a simple oscillator to generate the 68000 CLK input.

This circuit uses two inverters connected in series. Inverter 1 is biased in its

TABLE 10.13 Function Code Lines

FC2	*FC1*	*FC0*	*Operation*
0	0	0	Unassigned
0	0	1	User data
0	1	0	User program
0	1	1	Unassigned
1	0	0	Unassigned
1	0	1	Supervisor data
1	1	0	Supervisor program
1	1	1	Interrupt acknowledge

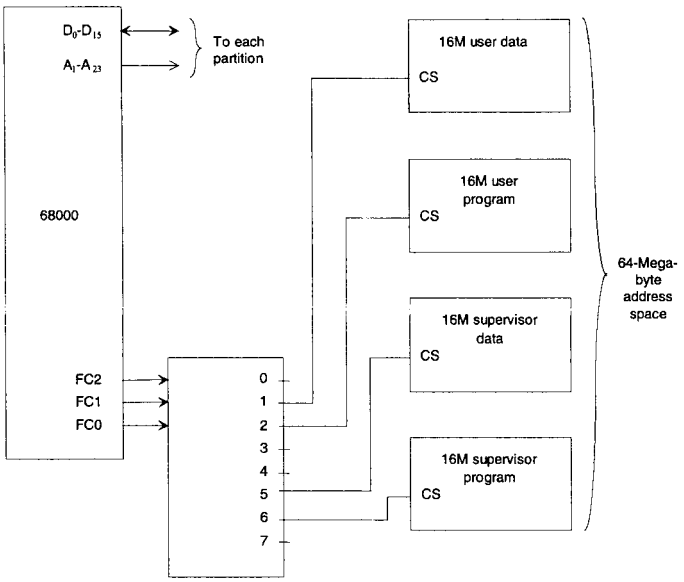


FIGURE 10.9 Partitioning 68000 address space using FC2, FC1, and FC0 pins

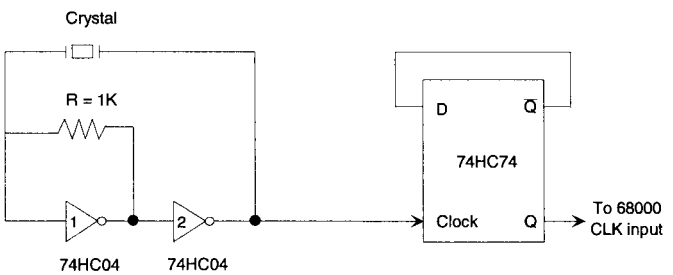


FIGURE 10.10 External clock circuitry

transition region by the resistor *R*. Inverter 1 inputs the crystal output (sinusoidal) to provide a logic pulse train at the output of inverter 1. Inverter 2 sharpens the wave and drives the crystal. For this circuit to work, HCMOS logic for the inverters must be used. Therefore, the 74HC04 inverter chip is used. The 74HC04 has high noise immunity and the ability to drive 10 LS-TTL loads. A coupling capacitor should be connected across the supply terminals to reduce the ringing effect during high-frequency switching of the HCMOS devices. Note that the ringing occurs when a circuit oscillates for a short time due to the presence of stray inductance and capacitance. In addition, the output of this oscillator is fed to the CLK input of a D flip-flop (74HC74) to further reduce the ringing. A clock signal of 50% duty cycle at a frequency of ½ the crystal frequency is generated. This means that this circuit with a 16-MHz crystal will generate an 8-MHz clock for the 68000.

10.9.2 68000 Reset Circuit

When designing the microprocessor's reset circuit, two types of reset must be considered: power-up and manual. These reset circuits must be designed using the parameters

specified by the manufacturer. Therefore, a microprocessor must be reset when its Vcc pin is connected to power. This is called "power-up reset." After some time during normal operation, the microprocessor can be reset by the designer upon activation of a manual switch such as a pushbutton. A reset circuit, therefore, needs to be designed following the timing parameters associated typically with the microprocessor's reset input pin specified by the manufacturer. The reset circuit, once designed, is typically connected to the microprocessor's reset pin.

Upon hardware reset, the 68000 sets the S-bit in SR to 1 and performs the following:

1. The 68000 loads the supervisor stack pointer from addresses $000000 (high 16 bits) and $000002 (low 16 bits) and loads the PC from $000004 (high 16 bits) and $000006 (low 16 bits). Typical 68000 assembler directives such as DC.L can be used for this purpose. For example, to load $200128 into supervisor SP and $3F1420 into PC, the following instruction sequence can be used:

```
ORG     $00000000
DC.L    $00200128
DC.L    $003F1420
```

2. The 68000 clears the trace bit in SR to 0 and sets the interrupt mask bits I2 I1 I0 in SR to 111. All other registers are unaffected.

To cause a power-up reset, Motorola specifies that both the $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ pins of the 68000 must be held LOW for at least 100 ms. This means that an external circuit needs to be designed that will generate a negative pulse with a width of at least 100 ms for both $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$. The manual $\overline{\text{RESET}}$ requires both the $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ pins to be LOW for at least 10 cycles(1.25 microseconds for 8MHz). In general, it is safer to assert $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ for much longer than the minimum requirements. Figure 10.11 shows a typical 68000 reset circuit that asserts $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ LOW for approximately 200 ms. The 555 timer is used in the circuit.

The reset circuit in the figure utilizes the 555 timer chip and provides for both power-up and manual resets by asserting the 68000 $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ pins for at least 200 ms. The computer designer does not have to know about the details of the 555 chip. Instead, the designer should know how to use the 555 chip to generate the 68000 RESET signal.

The 555 is a linear 8-pin chip. The TRIGGER pin is the input signal. When the voltage at the TRIGGER input pin is less than or equal to 1/3 V_{cc}, the OUTPUT pin is HIGH. The DISCHARGE and THRESHOLD pins are tied together to R_A and C. Note that the values of R_A and C determine the output pulse width. The CONTROL input pin controls the THRESHOLD input voltage. According to the manufacturer's data sheets, the control input should be connected to a 0.01-μF capacitor whose other lead should be grounded. Also, from the manufacturer's data sheets, the output pulse width, $t_{pw} = 1.1\ R_AC$ seconds. The values of R_A and C can be chosen for stretching out the pulse width. An RC circuit is connected at the 555 TRIGGER pin. A slow pulse obtained by charging and discharging the capacitor C_1 is applied at the 555 TRIGGER input pin. The 555 will generate a clean and fast pulse at the output. Capacitor C_1 is at zero voltage upon power-up. This is obviously lower than 1/3 V_{cc} with V_{cc} = 5 V. Thus, the 555 will generate a HIGH at the OUTPUT pin. The OUTPUT pin is connected through a 7404 inverter to provide a LOW at the 68000 $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ pins. The 7404 output is buffered via two 7407's (noninverting buffers) to ensure adequate currents for the 68000 $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ pins. Note that the 7407 provides an open collector output. Therefore, a 1-Kohm pull-up is used

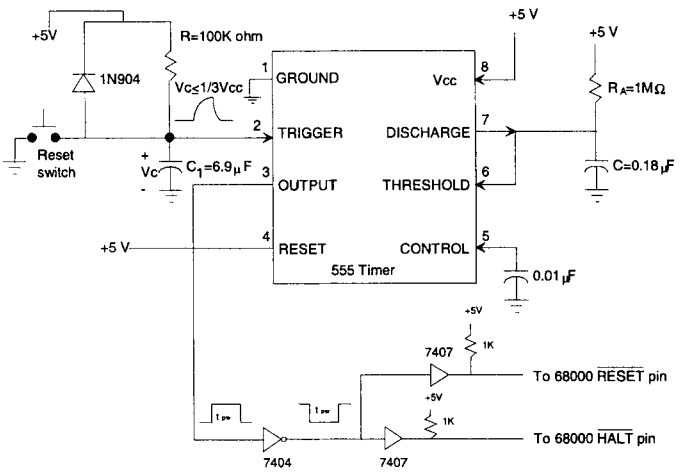


FIGURE 10.11 68000 RESET circuit

for each 7407. Now, let us explain how the timing requirements for the 68000 RESET are satisfied.

As mentioned before, capacitor C_1 is initially at zero voltage upon power-up. C_1 then charges to V_{cc} after a definite time determined by the time constant, RC_1. The charging voltage across the capacitor is

$$Vc(t) = Vcc[1 - e^{-\frac{t}{RC_1}}]$$

$V_c(t)$ must be less than or equal to $V_{cc}/3$ volts (1.7 V). To be on the safe side, let us assume that $V_c = V_{cc}/4 = 5/4 = 1.25$ V.

$$\frac{Vc(t)}{Vcc(t)} = 1 - e^{-\frac{t}{RC_1}}$$

$$\text{Hence, } \frac{1}{4} = 1 - e^{-\frac{t}{RC_1}}$$

$$e^{-\frac{t}{RC_1}} = 0.75$$

$$-\frac{t}{RC_1} = \ln(0.75)$$

$$-\frac{t}{RC_1} = -0.29$$

$$\text{Therefore, } RC_1 = \frac{t}{0.29}$$

As mentioned earlier, it is desired to provide 200 ms (arbitrarily chosen; satisfying the minimum requirements specified by Motorola) reset time for both power-up and manual reset.

$$RC_1 = \frac{200 \text{ ms}}{0.29} = 689.65 \text{ ms}$$

$$\text{Hence, } RC_1 \cong 0.69 \text{ s}$$

If R is arbitrarily chosen as 100 KΩ, then $C_1 = 6.9$ μF.

The 555 output pulse width can be determined using the equation, $t_{pw} = 1.1\ R_A\ C$. Since $t_{pw} = 200$ msec, hence $R_A\ C = 0.18$ seconds. If $R_A = 1$ MΩ (arbitrarily chosen) then $C = 0.18 / 10^6 = 0.18$ μF.

The reverse-biased diode (1N904 or equivalent) connected at the 555 TRIGGER input circuit is used to hold the capacitor (C_1 charged to 1.25 V) voltage at 1.25 V in case V_{cc} (obtained using a power supply from AC voltage) drops below 5V to a level such that the capacitor C_1 may discharge through the 100-KΩ resistor. In such a situation, the diode will be forward biased essentially shorting out the 100-Kohm resistor, thus maintaining the capacitor voltage at 1.25 V.

In Figure 10.11, upon power-up, the capacitor C_1 charges to approximately 1.25 V. After some time, if the reset switch is depressed, the capacitor is short-circuited to ground. The capacitor, therefore, discharges to zero. This logic 0 at the 555 TRIGGER input pin will provide 200 ms LOW at the 68000 $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ input pins. This will satisfy the minimum requirement of 10 clock cycles(1.25 microseconds for 8MHz clock) at the 68000 $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ pins for manual reset. The values of R and C_1 at the 555 trigger input should be recalculated for other 68000 clock frequencies for manual reset. Note that the 68000 power-up reset time is fixed with a timing requirement of at least 100 ms whereas the manual reset time depends on the 68000 clock frequency and must be at least 10 clock cycles.

Another way of generating the power-up and manual resets is by using a Schmitt-trigger inverter such as the 7414 chip. Figure 10.12 shows a typical circuit. The purpose of the Schmitt trigger in a microprocessor reset circuit has already been explained in Chapter 9 for 8086 reset using the 8284 chip. The operation of the 68000 power-up and manual resets using the RC circuit in Figure 10.12 has already been described in this section. The purpose of the two 7414 Schmitt-trigger inverters is primarily to shape up a slow pulse generated by the RC circuit to obtain a fast and clean negative pulse. Two 7407 open-collector noninverting buffers are used to amplify currents for the 68000 $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ pins. Let us now determine the values of R and C.

When the input of the 7414 Schmitt-trigger inverter is low (0 V for example), the output will be HIGH, typically at about 3.7 V. For input voltage from 0 to about 1.7 V, the output of the 7414 will be HIGH. Let us arbitrarily choose V_c = 1.5V to provide a low at the input of the first 7414 in the figure. As before,

$$Vc = Vcc[1 - e^{-\frac{t}{RC}}]$$

$$\text{Hence, } 1 - e^{-\frac{t}{RC}} = \frac{1.5}{5}$$

$$e^{-\frac{t}{RC}} = 0.7$$

Let us design the reset circuit to provide 200 ms reset time. Therefore, t = 200 ms.

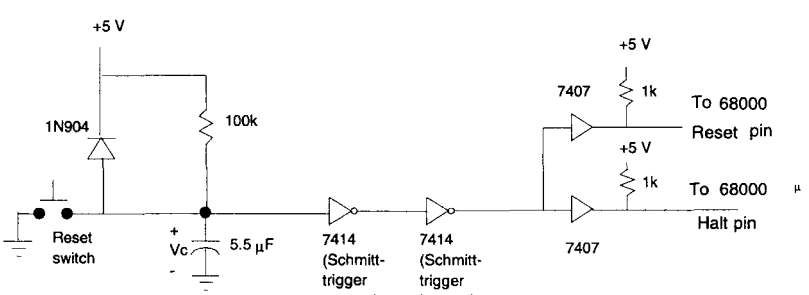


FIGURE 10.12 68000 Reset circuit using a Schmitt trigger

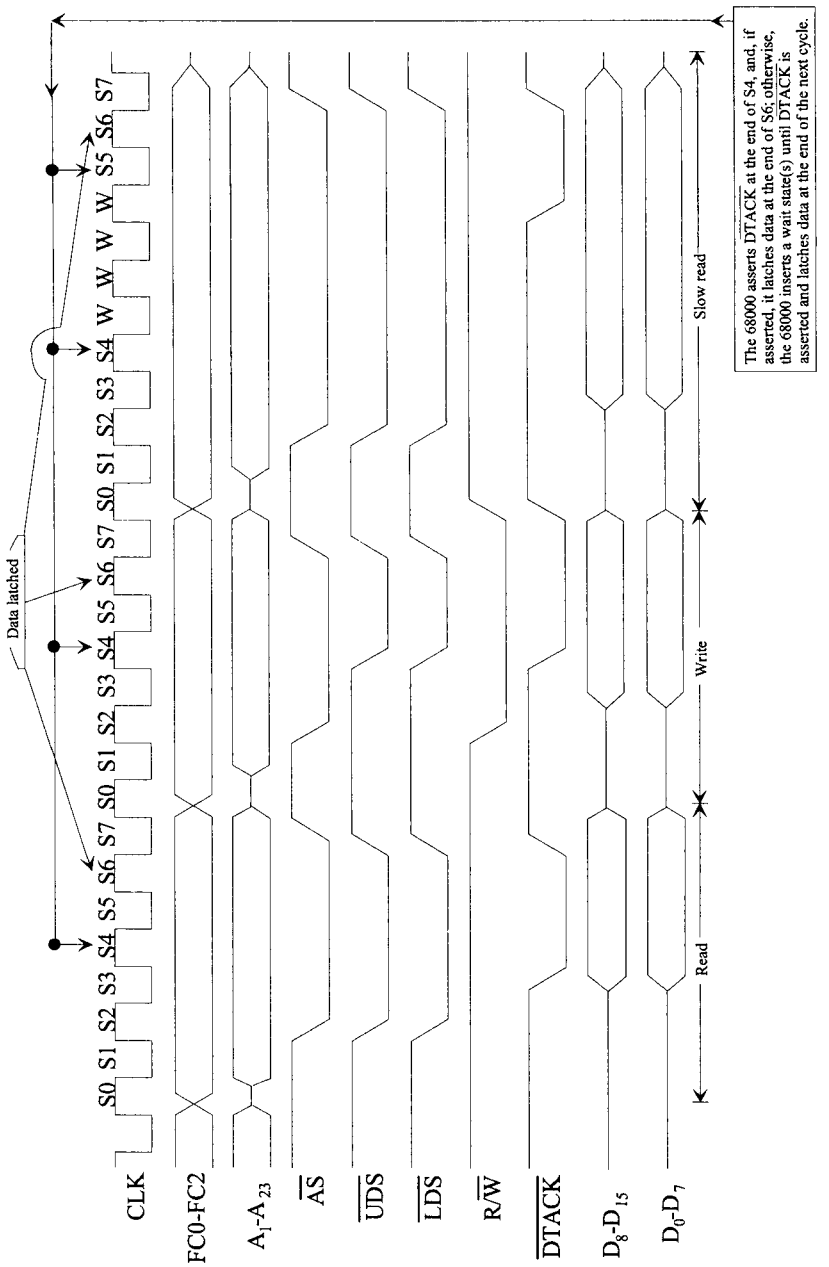


FIGURE 10.13 6800 Read and Write cycle Timing Diagrams

$$-\frac{0.2}{RC} = \ln(0.7)$$

$$-\frac{0.2}{RC} = -0.36$$

Therefore, $RC = 0.55$ seconds

If R is arbitrarily chosen as 100 KΩ, then $C = 5.5$ μF.

10.10 68000 Read and Write Cycle Timing Diagrams

The 68000 family of processors (68000, 68008, 68010, and 68012) uses a handshaking mechanism to transfer data between the processors and peripheral devices. This means that all these processors can transfer data asynchronously to and from peripherals of varying speeds.

During the read cycle, the 68000 obtains data from a memory location or an I/O port. If the instruction specifies a word (such as `MOVE.W $020504,D1`) or a long word (such as `MOVE.L $030808, D0`), the 68000 reads both upper and lower bytes at the same time by asserting the $\overline{UDS}$ and $\overline{LDS}$ pins. When the instruction is for a byte operation, the 68000 utilizes an internal bit to find which byte to read and then outputs the data strobe required for that byte.

For byte operations, when the address is even ($A_0 = 0$), the 68000 asserts $\overline{UDS}$ and reads data via the D_8–D_{15} pins into the low byte of the specified data register. On the other hand, when the address is odd ($A_0 = 1$), the 68000 outputs a LOW on $\overline{LDS}$ and reads data via the D_0–D_7 pins to the low byte of the specified data register. For example, consider `MOVE.B $507144, D5`. The 68000 outputs a LOW on $\overline{UDS}$ (because $A_0 = 0$) and a HIGH on $\overline{LDS}$. The memory chip's eight data lines must be connected to the 68000 D_8–D_{15} pins. The 68000 reads the data byte via the D_8–D_{15} pins into the low byte of D5. Note that, for reading a byte from an odd address, the data lines of the memory chip must be connected to the 68000 D_0–D_7 pins. In this case, the 68000 outputs a LOW on $\overline{LDS}$ (because $A_0 = 1$) and a HIGH on $\overline{UDS}$, and then reads the data byte into the low byte of the data register.

Figure 10.13 shows the read/write timing diagrams. During S0, address and data signals are in the high-impedance state. At the start of S1, the 68000 outputs the address on its address pins (A_1–A_{23}). During S0, the 68000 outputs FC2–FC0 signals. $\overline{AS}$ is asserted at the start of S2 to indicate a valid address on the bus. $\overline{AS}$ can be used at this point to latch the signals on the address pins. The 68000 asserts the $\overline{UDS}$, $\overline{LDS}$, and R/$\overline{W}$ = 1 to indicate a READ operation. The 68000 now waits for the peripheral device to assert $\overline{DTACK}$. Upon placing data on the data bus, the peripheral device asserts $\overline{DTACK}$. The 68000 samples the $\overline{DTACK}$ signal at the end of S4. If $\overline{DTACK}$ is not asserted by the peripheral device, the processor automatically inserts a wait state(s) (W).

However, upon assertion of $\overline{DTACK}$, the 68000 negates the $\overline{AS}$, $\overline{UDS}$, and $\overline{LDS}$ signals, and latches the data from the data bus into an internal register at the end of the next cycle. Once the selected peripheral device senses that the 68000 has obtained data from the data bus (by recognizing the negation of $\overline{AS}$, $\overline{UDS}$, or $\overline{LDS}$), the peripheral device must negate $\overline{DTACK}$ immediately so that it does not interfere with the start of the next cycle.

If $\overline{DTACK}$ is not asserted by the peripheral at the end of S4 (Figure 10.13, SLOW READ), the 68000 inserts wait states. The 68000 outputs valid addresses on the address pins and keeps asserting $\overline{AS}$, $\overline{UDS}$, and $\overline{LDS}$ until the peripheral asserts $\overline{DTACK}$. The 68000 always inserts an even number of wait states if $\overline{DTACK}$ is not asserted by the peripheral because all 68000 operations are performed using the clock with two states per clock cycle. Note in Figure 10.13 that the 68000 inserts 4 wait states or 2 cycles.

As an example of word read, consider that the 68000 is ready to execute the `MOVE.W $602122,D0` instruction. The 68000 performs as follows:

1. At the end of S0 the 68000 places the upper 23 bits of the address 602122_{16} on A_1–A_{23}.
2. At the end of S1, the 68000 asserts $\overline{AS}$, $\overline{UDS}$, and $\overline{LDS}$.

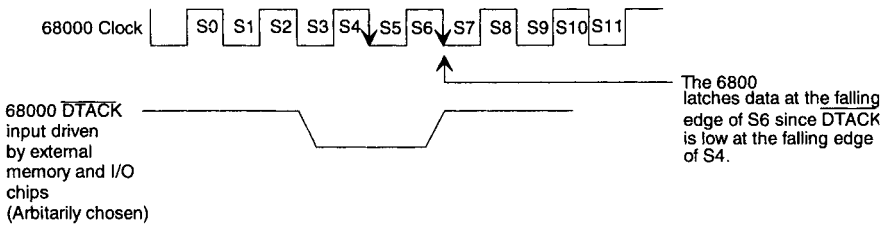


FIGURE 10.14 68000 CLK and $\overline{\text{DTACK}}$ signals

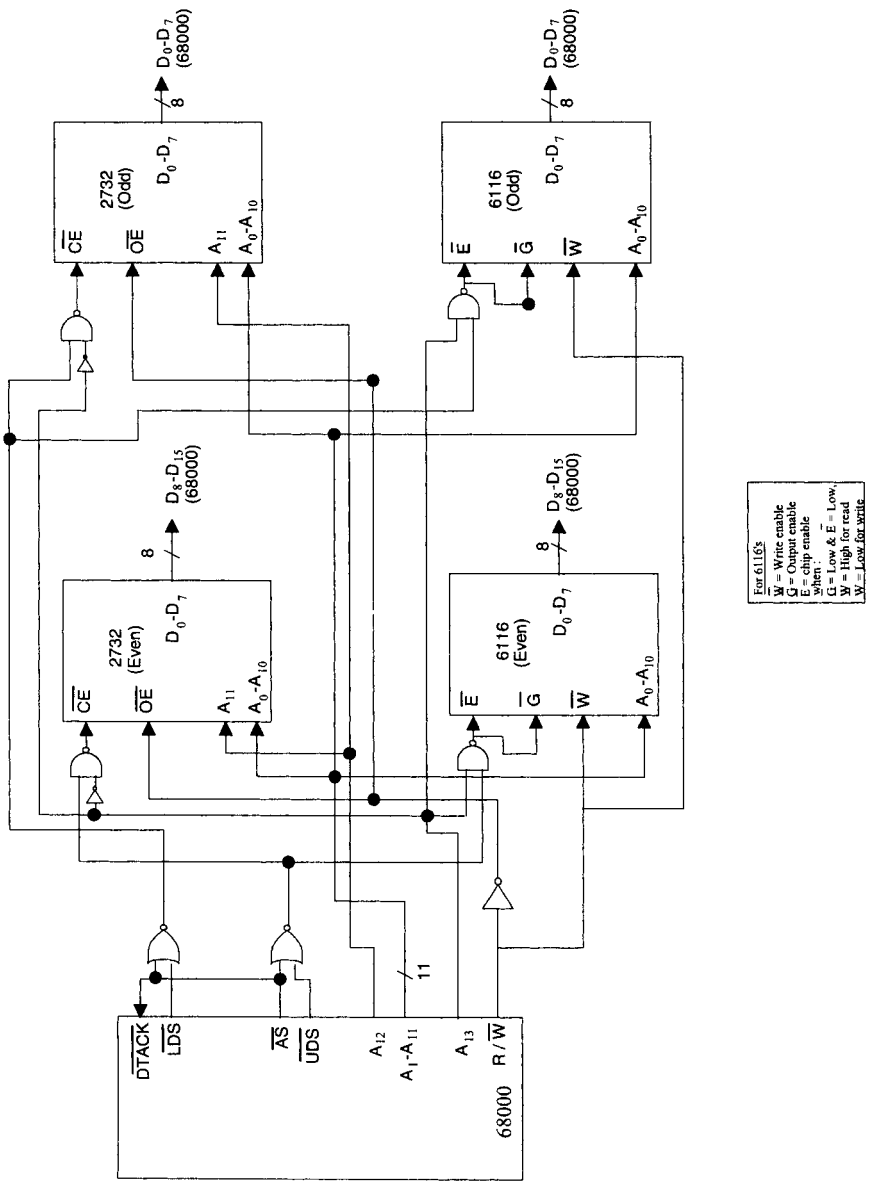


FIGURE 10.15 68000 interface to 2732 / 6116

3. The 68000 continues to output a HIGH on the R/$\overline{W}$ pin from the beginning of the read cycle to indicate a READ operation.
4. At the end of S0, the 68000 places appropriate outputs on the FC2–FC0 pins to indicate either supervisor or user read.
5. If the peripheral asserts $\overline{DTACK}$ at the end of S4, the 68000 reads the contents of 602122_{16} and 602123_{16} via the D_8–D_{15} and D_0–D_7 pins, respectively, into the high and low bytes of D0.W at the end of S6. If the peripheral does not assert $\overline{DTACK}$ at the end of S4, the 68000 continues to insert wait states.

Figure 10.14 shows a simplified timing diagram illustrating the use of $\overline{DTACK}$ for interfacing external memory and I/O chips to the 68000. As mentioned before, the 68000 checks the $\overline{DTACK}$ input pin at the falling edge of S4 (three cycles), the external memory, or I/O in this case, drives 68000 $\overline{DTACK}$ input to LOW, and the 68000 waits for one cycle and latches data at the end of S6. However, if the 68000 does not find $\overline{DTACK}$ LOW at the falling edge of S4, it waits for one clock cycle and then again checks $\overline{DTACK}$ for LOW. If $\overline{DTACK}$ is LOW, the 68000 latches data after one cycle (falling edge of S8). If the 68000 does not find $\overline{DTACK}$ LOW at the falling edge of S6, it checks for $\overline{DTACK}$ LOW at the falling edge of S8 and the process continues. Note that the minimum time to latch data is four cycles. This means that in the preceding example, if the 68000 clock frequency is 8 MHz, data will be latched after 500 ns because the $\overline{DTACK}$ is asserted LOW at the end of S4 (375 ns).

10.11 68000 Memory Interface

One of the advantages of the 68000 is that it can easily be interfaced to memory chips with various speeds because it goes into a wait state if $\overline{DTACK}$ is not asserted (LOW) by the memory devices at the end of S4. A simplified schematic showing an interface of a 68000 to two 2732's and two 6116's is given in Figure 10.15. As mentioned in Chapter 9, the 2732 is a 4K × 8 EPROM and the 6116 is a 2K × 8 static RAM. The pin diagrams of the 6116 and 2732 are provided in Appendices C and E respectively. For a 4-MHz clock, each cycle is 250 ns. Because the 68000 samples data at the falling edge of S4 (750 ns) and latches data at the falling edge of S6 (1000 ns), $\overline{AS}$ can be used to assert $\overline{DTACK}$. From the 68000 timing diagram of Figure 10.13, $\overline{AS}$ goes to LOW after approximately two cycles (500 ns). The time delay between $\overline{AS}$ going LOW and the falling edge of S6 is 500 ns. Note that $\overline{LDS}$ and $\overline{UDS}$ must be used as chip selects as in Figure 10.15. They must not be connected to A0 of the memory chips. Because in that case half of the memory in each memory chip would be wasted. Note that $\overline{LDS}$ and $\overline{UDS}$ also go to LOW after about two cycles (500 ns).

In Figure 10.15, a delay circuit for $\overline{DTACK}$ is not required because the 2732 and 6116 both place data on the bus lines before the 68000 latches data. This is because the 68000 clock frequency is 4 MHz in this case. Thus, each clock cycle is 250 ns. The access times of the 2732 and 6116 are 200 ns and 120 ns respectively. Because $\overline{DTACK}$ is sampled after 3 clock cycles (3 × 250 ns = 750 ns), both the 2732 and 6116 will have adequate time to place data on the bus for the 68000 to latch.

For example, consider the even 2732 EPROM of Figure 10.16. $\overline{UDS}$ and $\overline{AS}$ are NORed and then NANDed with inverted A_{13} to select this chip. With the 200-ns access time of the 2732 (Used to be 450ns), data will be placed on the 68000 D_8–D_{15} pins after approximately 720 nanoseconds (500 ns for $\overline{AS}$ or $\overline{UDS}$ + 10 ns for the NOR gate + 10 ns for the NAND gate + 200 ns for the 2732). Therefore, no delay circuit for the 68000 $\overline{DTACK}$

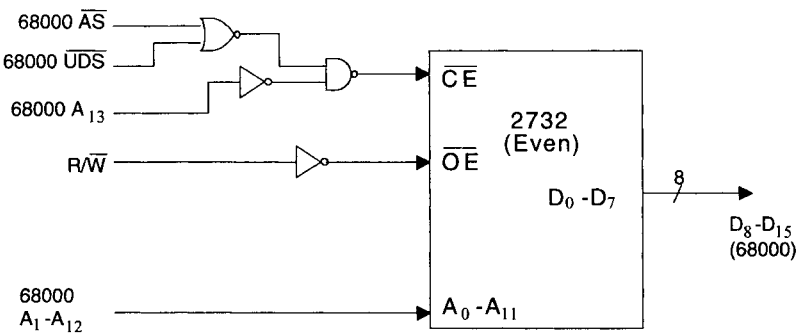


FIGURE 10.16 68000 interface to even 2732

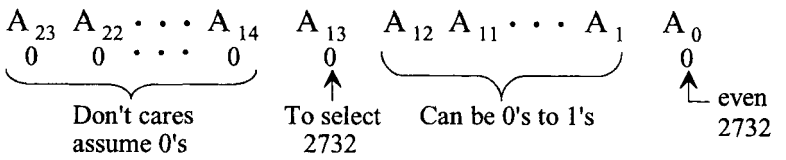
TABLE 10.14 68000-2732 Timing Example

Case	68000 Frequency	Clock Cycle	Time before first $\overline{DTACK}$ is sampled	Comment
1	12.5 MHz	80 ns	3(80) = 240 ns	Not enough time for 2732 to place data on bus; needs delay circuit for $\overline{DTACK}$
2	16.67 MHz	60 ns	3(60) = 180 ns	Same as case 1
3	25 MHz	40 ns	3(40) = 120 ns	Same as case 1

is required because the 68000 latches data from the D_8–D_{15} pins after 4 cycles (1000 ns in this case). The timing parameters of the 68000-2732 with various 68000 frequencies are shown in Table 10.14.

Next, consider odd 6116 static RAM (SRAM) with a 4-MHz 68000. Note that the 6116 signals, $\overline{W}$ (Write enable), $\overline{G}$ (Output enable), and $\overline{E}$ (Chip enable) are decoded as follows: when $\overline{G} = 0$ and $\overline{E} = 0$, then $\overline{W} = 1$ for read and $\overline{W} = 0$ for write. In this case, $\overline{LDS}$ and $\overline{AS}$ are NORed and NANDed with A13 to select this chip. With the 120-ns access time of the 6116 RAM, data will be placed on the 68000 D_0–D_7 pins after approximately 640 ns. Because the 68000 latches data after four cycles (1000 ns in this case), no delay circuit for $\overline{DTACK}$ is required. The requirements for $\overline{DTACK}$ for 68000/6116 for various 68000 clock frequencies can similarly be determined.

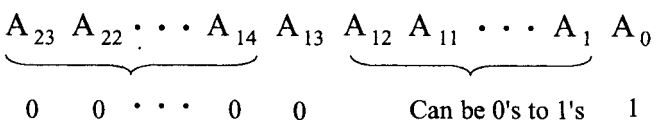
In case a delay circuit for $\overline{DTACK}$ is required, a ring counter with D flip-flops can be used. Let us now determine the memory maps. Figure 10.16 shows the 68000 interface to even 2732 obtained from Figure 10.15. When $A_{13} = 0$, $\overline{UDS} = 0$, $\overline{AS} = 0$, and $R/\overline{W} = 1$, the 2732 will be selected by the 68000 to read data from the 68000 D_8–D_{15} pins. The 68000 address pins A_{23}–A_{14} are don't cares (assume 0). The memory map for the even 2732 can be determined as follows:



Address range: $000000, $000002, ... , $001FFE

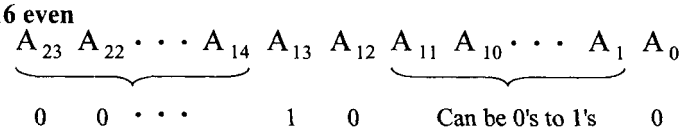
Similarly, the memory for the odd 2732, even 6116, and odd 6116 can be determined as follows:

- **2732 odd**



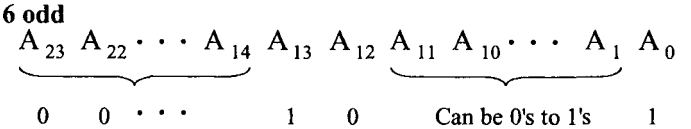
Address range: $000001, $000003, ... , $001FFF

- **6116 even**



Address range: $002000, $002002, ... , $002FFE

- **6116 odd**



Address range: $002001, $002003, ... , $002FFF

In the above, for 6116's, A_{12} and A_{14} - A_{23} are don't cares (assume 0's). Static RAMs such as 6116 are used for small memory system. Note that RAMs are needed when subroutines and interrupts requiring stack are desired in an application. Microprocessors requiring larger RAMs use dynamic RAMs (DRAMs). Concepts associated with interfacing DRAMs to 68000 will be discussed next.

DRAMs are typically used when memory requirements are 16k words or larger. DRAM is addressed via row and column addressing. For example, one megabit DRAM requiring 20 address bits is addressed using 10 address lines and two control lines, $\overline{RAS}$ (Row Address Strobe) and $\overline{CAS}$ (Column Address Strobe). To provide a 20-bit address into the DRAM, a LOW is applied to $\overline{RAS}$ and 10 bits of the address are latched. The other 10 bits of the address are applied next and $\overline{CAS}$ is then held LOW.

The addressing capability of the DRAM can be increased by a factor of 4 by adding one more bit to the address line. This is because one additional address bit results into one additional row bit and one additional column bit. This is why DRAMs can be expanded to larger memory very rapidly with inclusion of additional address bits. External logic is required to generate the $\overline{RAS}$ and $\overline{CAS}$ signals, and to output the current address bits to the DRAM.

DRAM controller chips take care of refreshing and timing requirements needed by the DRAMs. DRAMs typically require 4 millisecond refresh time. The DRAM controller performs its task independent of the microprocessor. The DRAM controller sends a wait

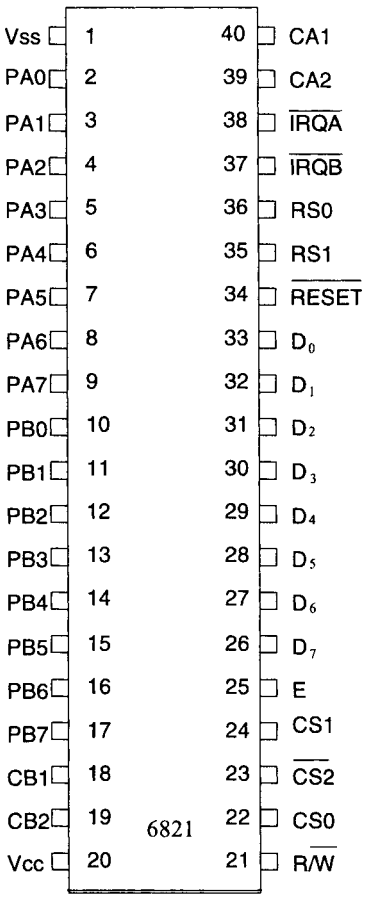


FIGURE 10.17 6821 pin diagram

signal to the microprocessor if the microprocessor tries to access memory during a refresh cycle.

Because of large memory, the address lines should be buffered using 74LS244 or 74HC244 (Unidirectional buffer), and data lines should be buffered using 74LS245 or 74HC245 (Bidirectional buffer) to increase the drive capability. Also, typical multiplexers such as 74LS157 or 74HC157 can be used to multiplex the microprocessor's address lines into separate row and column addresses.

10.12 68000 I/O

This section covers the I/O techniques associated with the Motorola 68000.

10.12.1 68000 Programmed I/O

As mentioned before, the 68000 uses memory-mapped I/O. Data transfer using I/O ports (programmed I/O) can be achieved in the 68000 in one of the following ways:

- By interfacing the 68000 with an inexpensive slow 6800 I/O chip such as the MC6821.
- By interfacing the 68000 with its own family of I/O chips such as the MC68230.

TABLE 10.15 6821 Register Definition

		Control Register Bits 2		
RS1	*RS0*	*CRA-2*	*CRB-2*	*Register Selected*
0	0	1	X	I/O port A
0	0	0	X	Data direction register A
0	1	X	X	Control register A
1	0	X	1	I/O port B
1	0	X	0	Data direction register B
1	1	X	X	Control register B

X = Don't care

68000/6821 Interface

The Motorola 6821 is a 40-pin peripheral interface adapter (PIA) chip. It is provided with an 8-bit bidirectional data bus (D_0–D_7), two register select lines (RS0, RS1), read/write (R/$\overline{W}$) and reset ($\overline{RESET}$) lines, an enable line (E), two 8-bit I/O ports (PA0–PA7), and (PB0–PB7), and other pins. Figure 10.17 shows the pin diagram of the 6821. There are six 6821 registers. These include two 8-bit ports (ports A and B), two data direction registers, and two control registers. Selection of these registers is controlled by the RS0 and RS1 inputs together with bit 2 of the control register. Table 10.15 shows how the registers are selected. In Table 10.15, bit 2 in each control register (CRA-2 and CRB-2) determines selection of either an I/O port or the corresponding data direction register when the proper register select signals are applied to RS0 and RS1. A 1 in bit 2 in CRA or CRB allows access of I/O ports; a 0 in bit 2 of CRA or CRB selects the data direction registers.

Each I/O port bit can be configured to act as an input or output. This is accomplished by sending a 1 in the corresponding data direction register bit for those bits that are to be output and a 0 for those bits that are to be inputs. A LOW on the $\overline{RESET}$ pin clears all PIA registers to 0. This has the effect of configuring PA0–PA7 and PB0–PB7 as inputs.

Three built-in signals in the 68000 provide the interface with the 6821: enable (E), valid memory address ($\overline{VMA}$), and valid peripheral address ($\overline{VPA}$). The enable signal (E) is an output from the 68000. It corresponds to the E signal of the 6821. This signal is the clock used by the 6821 to synchronize data transfer. The frequency of the E signal is one tenth of the 68000 clock frequency. This allows one to interface the 68000 (which operates much faster than the 6821) with the 6821. The valid memory address ($\overline{VMA}$) signal is output by the 68000 to indicate to the 6800 peripherals that there is a valid address on the address bus. The valid peripheral address ($\overline{VPA}$) is an input to the 68000. This signal is used to indicate that the device addressed by the 68000 is a 6800 peripheral. This tells the 68000 to synchronize data transfer with the enable signal (E).

Let us now discuss how the 68000 instructions can be used to configure the 6821 ports. As an example, bit 7 and bits 0–6 of port A can be configured, respectively, as input and outputs using the following instruction sequence:

```
BCLR.B #$2,CRA      ;       Address DDRA
MOVE.B #$7F,DDRA    ;       Configure port A
BSET.B #$2,CRA      ;       Address port A
```

Once the ports are configured to the designer's specification, the 6821 can be used to transfer data from an input device to the 68000 or from the 68000 to an output device by using the MOVE.B instruction as follows:

```
MOVE.B (EA), Dn     ;  Transfer 8-bit data from an input port
                    ;  to the specified data register Dn
MOVE.B Dn, (EA)     ;  Transfer 8-bit data from the specified
```

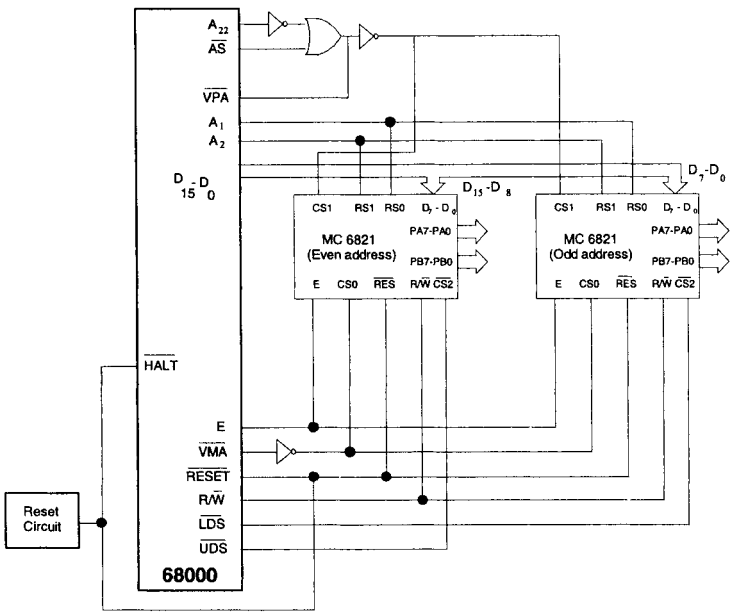


FIGURE 10.18 68000/6821 Interface

```
;  data register Dn to an output port
```

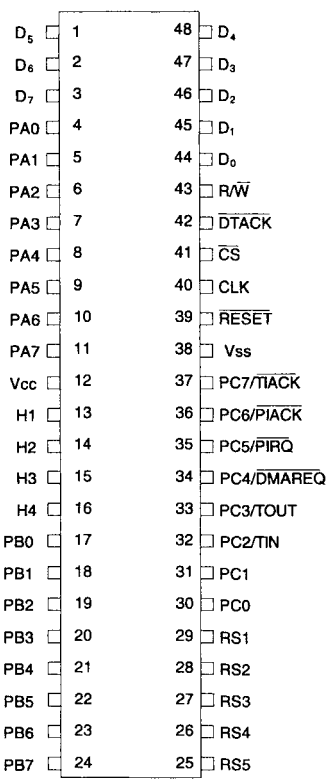


FIGURE 10.19 68230 pin diagram

Figure 10.18 shows a block diagram of how two 6821's are interfaced to the 68000 in order to obtain four 8-bit I/O ports. Note that the least significant bit, A_0, of the 68000 address pin is internally encoded to generate two signals, the upper data strobe ($\overline{UDS}$) and lower data strobe ($\overline{LDS}$). For byte transfers, $\overline{UDS}$ is asserted if an even-numbered byte is being transferred and $\overline{LDS}$ is asserted for an odd-numbered byte. In Figure 10.18, I/O port addresses can be obtained as follows: When $A_{22} = 1$ and $\overline{AS} = 0$, the OR gate output will be LOW. This OR gate output is used to assert $\overline{VPA}$. The inverted OR gate output, in turn, makes CS1 HIGH on both 6821's. Note that A_{22} is arbitrarily chosen. A_{22} is chosen to be HIGH to enable CS1 so that the addresses for the ports and the reset vector are not the same. Assuming that the don't care address lines A_{23} and A_{21}–A_3 are 0's, the addresses for the I/O ports, control registers, and data direction registers for the even 6821 ($A_0 = 0$) can be obtained as shown; similarly, the addresses for the ports, control registers, and data direction registers for the odd 6821 ($A_0 = 1$) can be determined as follows:

	Port A or DDRA	CRA	Port B or DDRB	CRB
6821(even)	$400000	$400002	$400004	$400006
6821(odd)	$400001	$400003	$400005	$400007

68000/68230 Interface

The 68230 is a 48-pin I/O chip designed for the 68000 family of microprocessors. The 68230 offers various functions such as programmed I/O, an on-chip timer, and a DMA request pin for connection to a DMA controller. Figure 10.19 shows the 68230 pin diagram. The 68230 can be configured in two modes of operation: unidirectional and bidirectional. In the unidirectional mode, data direction registers configure the corresponding ports as inputs or outputs. This is the programmed I/O mode of operation. Both 8-bit and 16-bit ports can be used. In the bidirectional mode, the 68230 provides data transfer between the 68000 and external devices via exchange of control signals (known as handshaking). This section will only cover the programmed I/O feature of the 68230.

This 68230 ports can be configured in either unidirectional or bidirectional mode by using bits 7 and 6 of the port general control register, PGCR (R0) as follows:

PGCR Bits			
7	6	Mode	
0	0	0	(unidirectional 8-bit)
0	1	1	(unidirectional 16-bit)
1	0	2	(bidirectional 8-bit)
1	1	3	(bidirectional 16-bit)

The other bits of the PGCR are defined for handshaking.

Modes 0 and 2 configure ports A and B as unidirectional or bidirectional 8-bit ports. Modes 1 and 3, on the other hand, combine ports A and B together to form a 16-

TABLE 10.16 Some of the 68230 Registers

Register Select Bits					
RS5	*RS4*	*RS3*	*RS2*	*RS1*	*Register Selected*
0	0	0	0	0	PGCR, Port General Control Register (R0)
0	0	0	1	0	PADDR, Port A Data Direrction Register (R2)
0	0	0	1	1	PBDDR, Port B Data Direction Register (R3)
0	0	1	1	0	PACR, Port A Control Register (R6)
0	0	1	1	1	PBCR, Port B Control Register (R7)
0	1	0	0	0	PADR, Port A Data Register (R8)
0	1	0	0	1	PBDR, Port B Data Register (R9)

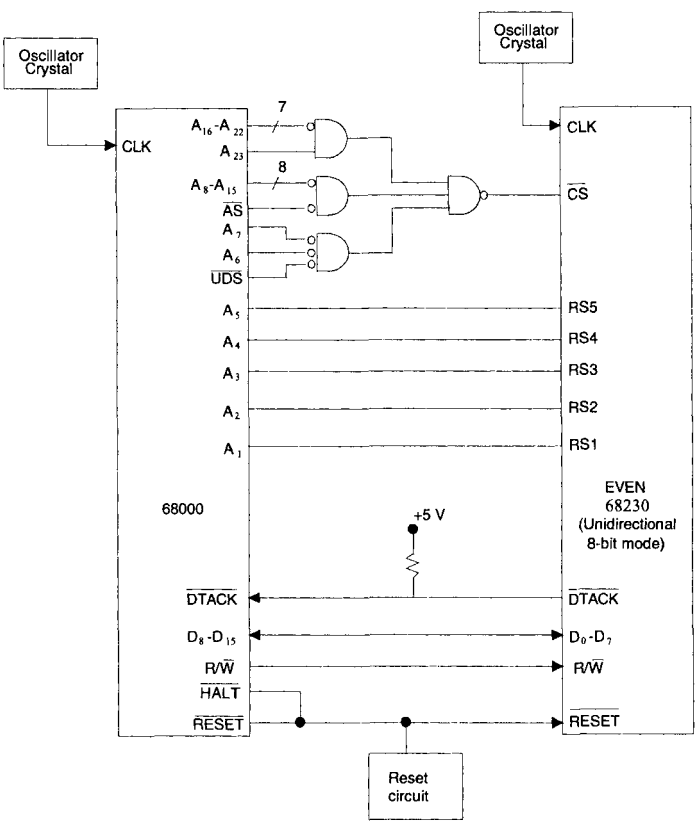


FIGURE 10.20 68000/68230 interface

bit unidirectional or bidirectional port. Ports configured as unidirectional 8-bit must be programmed further as submodes of operation using bits 7 and 6 of PACR (R6) and PBCR (R7) as follows:

Submode	*Bit 7 of PACR or PBCR*	*Bit 6 of PACR or PBCR*	*Comment*
00	0	0	Pin-definable double-buffered input or single-buffered output
01	0	1	Pin-definable double-buffered output or nonlatched input
1X	1	X	Bit I/O (pin-definable single-buffered output or nonlatched input)

Note that X means don't care. Nonlatched inputs are latched internally, but the values are not latched externally by the 68230 at the port. Bit I/O is used for programmed I/O.

The submodes define the ports as parallel input ports, parallel output ports, or bit-configurable I/O ports. In addition to these, the submodes further define the ports as latched input ports, interrupt-driven ports, DMA ports, and ports with various I/O handshake operations. Table 10.16 lists some of the 68230 registers. The registers required for programmed I/O are considered in the following discussion. Note that the 68230 register select pins (RS5–RS1) are used to select the 68230 registers. Figure 10.20 illustrates how to obtain specific addresses for the 68230 I/O ports.

The hardware schematic for the 68000/68230 interface shown in Figure 10.20 is connected in such a way that each 68230 I/O port has a unique address. A_{23} is chosen to be HIGH to select the 68230 chips so that the port addresses are different from the 68000 reset vector addresses 000000_{16}–000006_{16}. The configuration in the figure will provide even port addresses because $\overline{UDS}$ is used for enabling the 68230 $\overline{CS}$. The 68230 $\overline{DTACK}$ is an open-drain output. Hence, a pull-up resistor is required.

From the figure, addresses for registers PGCR (R0), PADDR (R2), PBDDR (R3), PACR (R6), PBCR (R7), PADR (R8), and PBDR (R9) can be obtained. Consider PGCR as follows:

A_{23} A_{22} A_{21} A_{20} $\cdots$ A_6 A_5 A_4 A_3 A_2 A_1 A_0
1 0 0 0 $\cdots$ 0 0 0 0 0 0 0 = \$800000
RS5 - RS1
$\overline{UDS}$

Therefore, Address for PGCR = \$800000
Similarly, Address for PADDR = \$800004, Address for PBDDR = \$800006
Address for PACR = \$80000C, Address for PBCR = \$80000E
Address for PADR = \$800010, Address for PBDR = \$800012

As an example, the following instruction sequence will select mode 0, submode 1X and configure bits 0–5 of Port A as outputs, bits 6 and 7 of Port A as inputs, and port B as an input port:

```
PGCR    EQU     $800000
PADDR   EQU     $800004
PBDDR   EQU     $800006
PACR    EQU     $80000C
PBCR    EQU     $80000E
        ANDI.B  #$3F,PGCR    ;  Select mode 0
        BSET.B  #7,PACR      ;  Port A bit I/O submode
```

```
        BSET.B   #7,PBCR      ;  Port B bit I/O submode
        MOVE.B   #$3F,PADDR   ;  Configure port A bits 0-5 as
                              ;  outputs and bits 6 and 7 as inputs
        MOVE.B   #$00,PBDDR   ;  Configure port B as an input port
```

Example 10.16

A 68000/68230-based microcomputer is required to drive an LED connected at bit 7 of port A based on two switch inputs connected at bits 6 and 7 of port B. If both switches are equal (either HIGH or LOW), turn the LED ON; otherwise turn it OFF. Assume that a HIGH will turn the LED ON and a LOW will turn it OFF. Write a 68000 assembly program to accomplish this.

Solution

```
PGCR    EQU     $800000
PACR    EQU     $80000C
PBCR    EQU     $80000E
PADDR   EQU     $800004
PBDDR   EQU     $800006
PADR    EQU     $800010
PBDR    EQU     $800012
        ANDI.B  #$3F,PGCR  ;   Select mode 0
        BSET.B  #7,PACR    ;   Port A bit I/o submode
        BSET.B  #7,PBCR    ;   Port B bit I/o submode
        MOVE.B  #$80,PADDR ;   Configure port A bit 7 as output
        MOVE.B  #0,PBDDR   ;   Configure port B bits 6 and 7 as
inputs
        MOVE.B  PBDR,D0    ;   Input port B
        ANDI.B  #$0C0,D0   ;   Retain bits 6 and 7
        BEQ     LEDON      ;   If both switches LOW, turn LED ON
        CMPI.B  #$0C0,D0   ;   If both switches HIGH, turn LED ON
        BEQ     LEDON
        MOVE.B  #$00,PADR  ;   Turn LED OFF
        JMP     FINISH
LEDON   MOVE.B  #$80,PADR  ;   Turn LED ON
FINISH  JMP     FINISH
```

Example 10.17

Write a 68000 assembly language program to drive an LED connected to bit 7 of Port A based on a switch input at bit 0 of Port A. If the switch is HIGH, turn the LED ON; otherwise turn the LED OFF. Assume a 68000/2732/6116/6821 microcomputer. Also, write a C++ program to accomplish the same task. Use port addresses of your choice.

Solution

The 68000 assembly language program and the C++ program follow.

- ***68000/6821 Microcomputer Assembly Code for Switch and LED***

```
PORTA       EQU     $001001
DDRA        EQU     $001001
CRA         EQU     $001003
            BCLR.B  #2,CRA  ;   address DDRA
            MOVE.B  #$80,DDRA   ;   Configure PORT A
            BSET.B  #2,CRA      ;   Address PORT A
START       MOVE.B  PORTA,D0    ;   Read switch
            ROR.B       #1,D0   ;   Rotate switch status
            MOVE.B  D0,PORTA    ;   Output to LED
            JMP     START       ;   Repeat
```

- ***68000/6821 Microcomputer C++ program for Switch and LED***

```
main()
{
    char *porta, *ddra, *cra;
    porta=0x1001;
    ddra=0x1001;
    cra=0x1003;
    *cra=0;                          /* Address DDRA */
    *ddra=0x80;                      /* Configure Port A */
    *cra=4;                          /* Address Port A */
    while (1)
        *porta=*porta <<7;           /* Read switch and send to LED */
}
```

The C++ compiler will generate more machine codes for the above program compared to the equivalent assembly program. Note that the C++ program is not 100% portable while using I/O. However, it is easier to write programs using C++ than using assembly language.

10.12.2 68000 Interrupt System

The 68000 interrupt I/O can be divided into two types: external interrupts and internal interrupts.

External Interrupts

The 68000 provides seven levels of external interrupts, 1 through 7. The external hardware provides an interrupt level using the pins $\overline{IPL0}$, $\overline{IPL1}$, and $\overline{IPL2}$. Like other microprocessors, the 68000 checks for and accepts interrupts only between instructions. It compares the value of inverted $\overline{IPL0}$–$\overline{IPL2}$ with the current interrupt mask contained in the bits 10, 9, and 8 of the status register.

If the value of the inverted $\overline{IPL0}$–$\overline{IPL2}$ is greater than the value of the current interrupt mask, then the 68000 acknowledges the interrupt and initiates interrupt processing. Otherwise, the 68000 continues with the current interrupt. Interrupt request level 0 ($\overline{IPL0}$–$\overline{IPL2}$ all HIGH) indicates that no interrupt service is requested. An inverted $\overline{IPL2}$, $\overline{IPL1}$, $\overline{IPL0}$ of 7 is always acknowledged. Therefore, interrupt level 7 is "nonmaskable." Note that the interrupt level is indicated by the interrupt mask bits (inverted $\overline{IPL2}$, $\overline{IPL1}$, $\overline{IPL0}$).

To ensure that an interrupt will be recognized, the following interrupting rules should be considered:

1. The incoming interrupt request level must have a higher priority level than the mask level set in the interrupt mask bits (except for level 7, which is always recognized).
2. The $\overline{IPL2}$–$\overline{IPL0}$ pins must be held at the interrupt request level until the 68000 acknowledges the interrupt by initiating an interrupt acknowledge ($\overline{IACK}$) bus cycle

Interrupt level 7 is edge-triggered. On the other hand, interrupt levels 1–6 are level sensitive. However, as soon as one of them is acknowledged, the processor updates its interrupt mask at the same level.

The 68000 does not have any EI (enable interrupt) or DI (disable interrupt) instructions. Instead, the level indicated by I2 I1 I0 in the SR disables all interrupts below or equal to this value and enables all interrupts above. For example, if I2 I1 I0 = 100, then interrupt levels 1–4 are disabled and 5–7 are enabled. Note that I2 I1 I0 = 000 enables all interrupts and I2 I1 I0 = 111 disables all interrupts except level 7 (nonmaskable).

Once the 68000 has decided to acknowledge an interrupt, it performs several steps:

1. Makes an internal copy of the current status register.
2. Updates the priority mask and address lines A_3–A_1 with the level of the interrupt

recognized (inverted $\overline{IPL}$ pins) and then asserts $\overline{AS}$ to inform the external devices that A_1–A_3 has the interrupt level.
3. Enters the supervisor state by setting the S bit in SR to 1.
4. Clears the T bit in SR to inhibit tracing.
5. Pushes the program counter (PC) onto the supervisor stack.
6. Pushes the internal copy of the old SR onto the supervisor stack.
7. Runs an $\overline{IACK}$ bus cycle for vector number acquisition (to provide the address of the service routine).
8. Multiplies the 8-bit interrupt vector by 4. This points to the location that contains the starting address of the interrupt service routine.
9. Jumps to the interrupt service routine.
10. The last instruction of the service routine should be RTE, which restores the original status word and program counter by popping them from the supervisor stack.

External logic can respond to the interrupt acknowledge in one of three ways: by requesting automatic vectoring (autovector), by placing a vector number on the data bus (nonautovector), or by indicating that no device is responding (spurious interrupt).

Autovector (address vectors predefined by Motorola)

If the hardware asserts $\overline{VPA}$ to terminate the $\overline{IACK}$ bus cycle, the 68000 directs itself automatically to the proper interrupt vector corresponding to the current interrupt level. No external hardware is inquired for providing the interrupt address vector. The seven levels of autovector interrupt are listed below:

	I2	*I1*	*I0*
Level 1 ← Interrupt vector $19 for	0	0	1
Level 2 ← Interrupt vector $1A for	0	1	0
Level 3 ← Interrupt vector $1B for	0	1	1
Level 4 ← Interrupt vector $1C for	1	0	0
Level 5 ← Interrupt vector $1D for	1	0	1
Level 6 ← Interrupt vector $1E for	1	1	0
Level 7 ← Interrupt vector $1F for	1	1	1

Nonautovector (user-definable address vectors via external hardware)

The interrupting device uses external hardware to place a vector number on data lines D_0–D_7 and then performs a $\overline{DTACK}$ handshake to terminate the $\overline{IACK}$ bus cycle. The vector numbers allowed are $40 to $FF, but Motorola has not implemented a protection on the first 64 entries so that user-interrupt may overlap at the discretion of the system designer.

Vector Address		Vector Number
$60, $62	Spurious interrupt	$18
$64, $66	Autovector 1	$19
$68, $6A	Autovector 2	$1A
$6C, $6E	Autovector 3	$1B
$70, $72	Autovector 4	$1C
$74, $76	Autovector 5	$1D
$78, $7A	Autovector 6	$1E
$7C, $7E	Autovector 7	$1F
$80 to $BC	TRAP instructions	$20 to $2F
$C0 to $FC	Unassigned	$30 to $3F
$100 to $3FC	User interrupts (nonautovector)	$40 to $FF

FIGURE 10.21 68000 interrupt map

Spurious Interrupt

Another way to terminate an interrupt acknowledge bus cycle is with the $\overline{\text{BERR}}$ (bus error) signal. Even though the interrupt control pins are synchronized to enhance noise immunity, it is possible that external system interrupt circuitry may initiate an $\overline{\text{IACK}}$ bus cycle as a result of noise. Because no device is requesting interrupt service, neither $\overline{\text{DTACK}}$ nor $\overline{\text{VPA}}$ will be asserted to signal the end of the nonexisting $\overline{\text{IACK}}$ bus cycle. When there is no response to an $\overline{\text{IACK}}$ bus cycle after a specified period of time (monitored by the user using an external timer), $\overline{\text{BERR}}$ can be asserted by an external timer. This indicates to the processor that it has recognized a spurious interrupt. The 68000 provides 18H as the vector to fetch for the starting address of this exception-handling routine.

It should be pointed out that the spurious interrupt and bus error interrupt due to a troubled instruction cycle (when no $\overline{\text{DTACK}}$ is received by the 68000) have two different interrupt vectors. Spurious interrupt occurs when the $\overline{\text{BERR}}$ pin is asserted during interrupt processing.

Internal Interrupts

The internal interrupt is a software interrupt. This interrupt is generated when the 68000 executes a software interrupt instruction (TRAP) or by some undesirable events such as division by zero or execution of an illegal instruction.

68000 Interrupt Map

The 68000 uses an 8-bit vector n to obtain the interrupt address vector. The 68000 reads the long-word located at memory 4* n. This long word is the starting address of the service routine. Figure 10.21 shows an interrupt map of the 68000. Vector addresses $00 through $2E (not shown in the figure) include vector addresses for reset, bus error, trace, divide by 0, and so on, and addresses $30 through $5C are unassigned. The RESET vector requires four words (addresses 0, 2, 4, and 6); the other vectors require only two words.

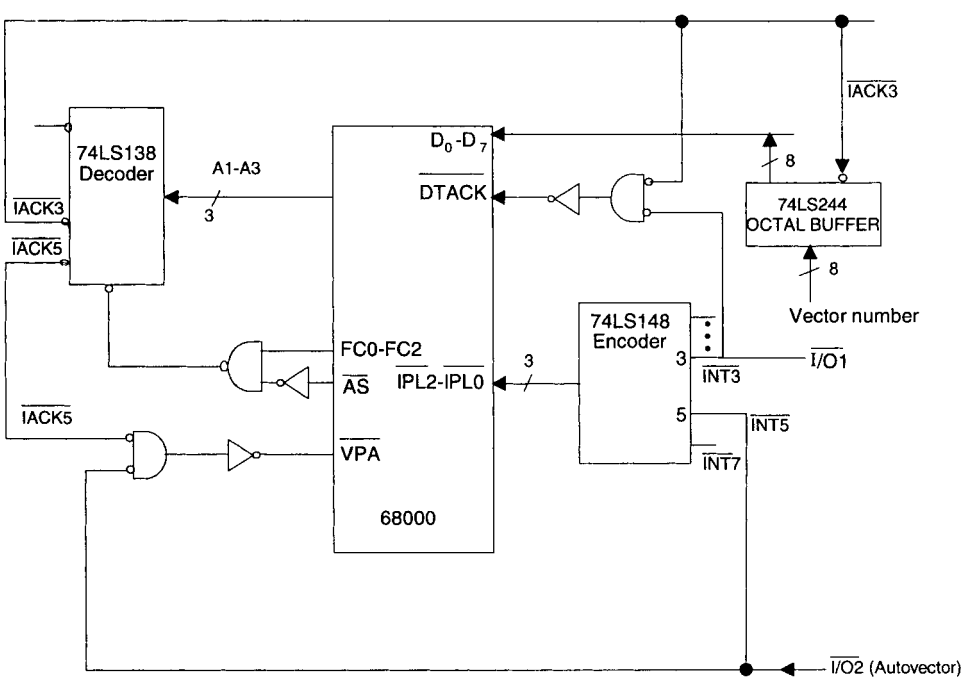


FIGURE 10.22 Autovector and nonautovector interrupts

After hardware reset, the 68000 loads the supervisor SP high and low words, respectively, from addresses 000000_{16} and 000002_{16}, and the PC high and low words, respectively, from 000004_{16} and 000006_{16}. The typical assembler directive DC (define constant) can be used to load the PC and Supervisor SP. For example, the following will load A7′ with $16F128 and PC with $781624:

```
ORG     $000000
DC.L    $0016F128
DC.L    $00781624
```

68000 Interrupt Address Vector

Suppose that the user decides to write a service routine starting at location $123456 using autovector 1. Because the autovector 1 address is $000064 and $000066, the numbers $0012 and $3456 must be stored in locations $000064 and $000066, respectively. Note that from Figure 10.21, n = $19 for autovector 1. Hence, the starting address of the service routine is obtained from the contents of the address 4 x $19 = $000064.

An Example of Autovector and Nonautovector Interrupts

As an example to illustrate the concept of autovector and nonautovector interrupts, consider Figure 10.22. In this figure, I/O device 1 uses nonautovector and I/O device 2 uses autovector interrupts. The system is capable of handling interrupts from seven devices ($\overline{IPL2}$ $\overline{IPL1}$ $\overline{IPL0}$ pins = 111 means no interrupt) because an 8-to-3 priority encoder such as the 74LS148 is used. The 74LS148 provides an inverted three-bit output with input 7 as the highest priority and input 0 as the lowest priority. Hence, if all eight inputs of the 74LS148 are low simultaneously, the three-bit output will be 000 (inverted 111) indicating a LOW

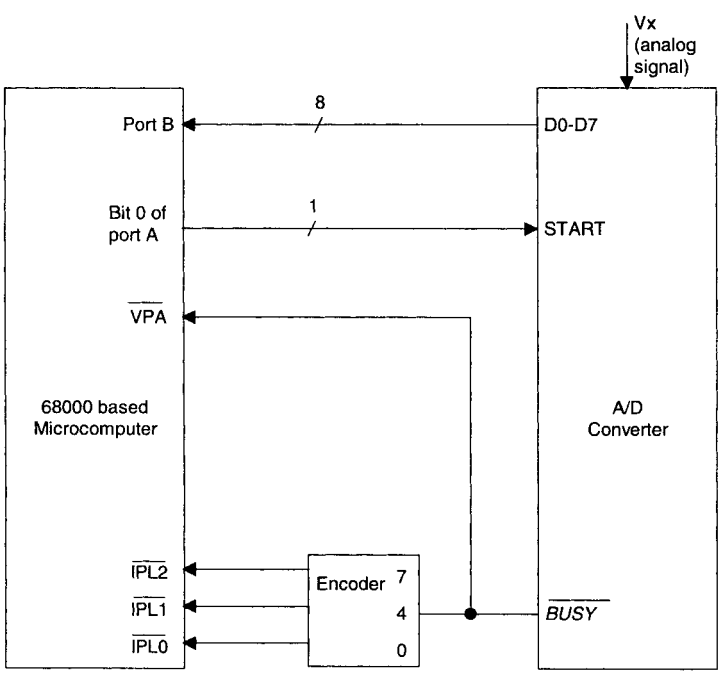


FIGURE 10.23 Interfacing of a typical 8-bit A/D converter to 68000-based microcomputer using autovector interrupt

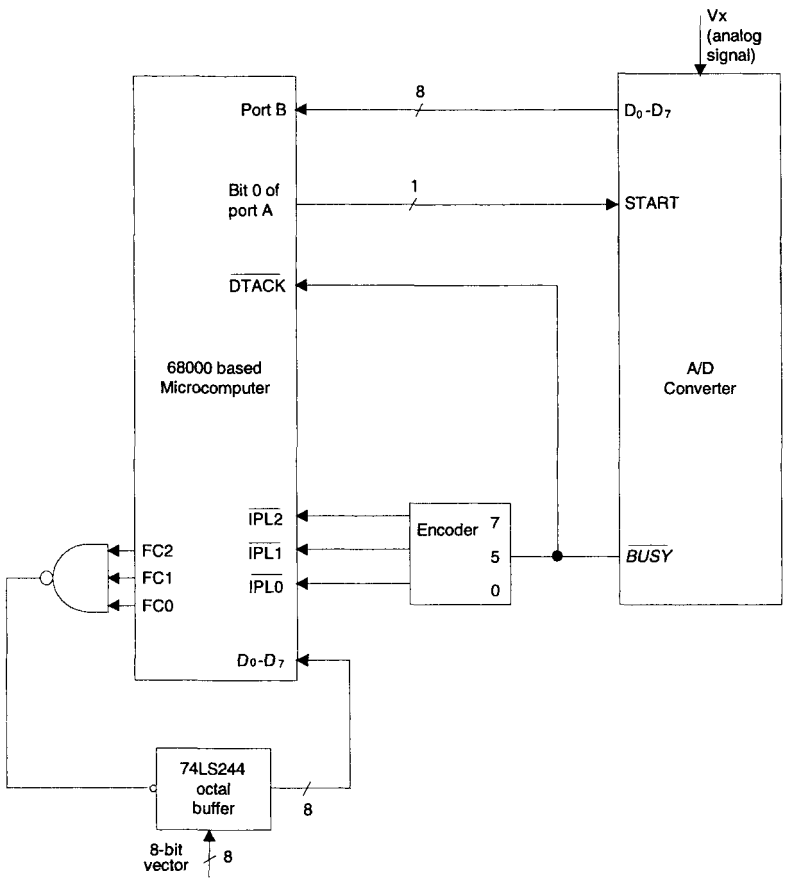


FIGURE 10.24 Interfacing of a typical 8-bit A/D converter to 68000-based microcomputer using nonautovector interrupt

on input 7. In figure 10.22, $\overline{I/O1}$ and $\overline{I/O2}$ from the interrupting devices are connected to inputs 3 and 5 of the 74LS148 encoder respectively. This means that the device with $\overline{I/O2}$ as the interrupting signal will generate level 5 autovectored interrupt while the device with $\overline{I/O1}$ as the interrupting signal will generate the nonautovectored interrupt.

Suppose that I/0 device 2 drives $\overline{I/O2}$ LOW in order to activate line 5 of the 74LS148. This, in turn, will generate a LOW on input 5 of the 74LS148. This will provide 010 (inverted 101) on $\overline{IPL2}$ $\overline{IPL1}$ $\overline{IPL0}$ pins of the 68000 generating a level 5 autovectored interrupt. When the 68000 decides to acknowledge the interrupt, it drives FC0–FC2 HIGH. The interrupt level is reflected on A_1–A_3 when $\overline{AS}$ is activated by the 68000. The $\overline{IACK5}$ and $\overline{I/O2}$ signals are used to generate $\overline{VPA}$. Once $\overline{VPA}$ is asserted, the 68000 obtains the interrupt vector address using autovectoring.

In the case of $\overline{I/O1}$, line 3 of the priority encoder is activated to initiate the nonautovectored interrupt. By using appropriate logic, $\overline{DTACK}$ is asserted using $\overline{IACK3}$ and $\overline{I/O1}$. The vector number is placed on D_0–D_7 by enabling an octal buffer such as the 74LS244 using $\overline{IACK3}$. The 68000 inputs this vector number and multiplies it by 4 to obtain the interrupt address vector.

Interfacing a Typical A/D Converter to the 68000 Using Autovector and Nonautovector Interrupts

Figure 10.23 shows the interfacing of a typical A/D converter to the 68000-based microcomputer using the autovector interrupt. In the figure, the A/D converter can be started by sending a START pulse. The signal can be connected to line 4 (for example) of the encoder.

Note that line 4 is 100_2 for $\overline{IPL2}$, $\overline{IPL1}$, $\overline{IPL0}$, which is a level 3 (inverted 100_2) interrupt. $\overline{BUSY}$ can be used to assert $\overline{VPA}$ so that, after acknowledgment of the interrupt, the 68000 will service the interrupt as a level 3 autovector interrupt. Note that the encoder in Figure 10.23 is used for illustrative purposes. This encoder is not required for a single device such as the A/D converter in the example.

Figure 10.24 shows the interfacing of a typical A/D converter to the 68000-based microcomputer using the nonautovector interrupt. In the figure, the 68000 starts the A/D converter as before. Also, the $\overline{BUSY}$ signal is used to interrupt the microcomputer using line 5 ($\overline{IPL2}$, $\overline{IPL1}$, $\overline{IPL0}$= 101, which is a level 2 interrupt) of the encoder. $\overline{BUSY}$ can be used to assert $\overline{DTACK}$ so that, after acknowledgment of the interrupt, FC2, FC1, FC0 will become 111_2, which can be NANDed to enable an octal buffer such as the 74LS244 in order to transfer an 8-bit vector from the input of the buffer to the D_0–D_7 lines of the 68000. The 68000 can then multiply this vector by 4 to determine the interrupt address vector. As before, the encoder in Figure 10.24 is not required for the single A/D converter.

10.12.3 68000 DMA

Three DMA control lines are provided with the 68000. These are $\overline{BR}$ (bus request), $\overline{BG}$ (bus grant), and $\overline{BGACK}$ (bus grant acknowledge). The $\overline{BR}$ line is an input to the 68000. The external device activates this line to tell the 68000 to release the system bus. At least one clock period after receiving $\overline{BR}$, the 68000 will enable its $\overline{BG}$ output line to acknowledge the DMA request. However, the 68000 will not relinquish the bus until it has completed the current instruction cycle. The external device must check the $\overline{AS}$ (address strobe) line to determine the completion of the instruction cycle by the 68000. When $\overline{AS}$ becomes HIGH, the 68000 will tristate its address and data lines and will give up the bus to the external device. After taking over the bus, the external device must enable the $\overline{BGACK}$ line. The $\overline{BGACK}$ line tells the 68000 and other devices connected to the bus that the bus is being used. The 68000 stays in a tristate condition until $\overline{BGACK}$ becomes HIGH.

10.13 68000 Exception Handling

A 16-bit microcomputer is usually capable of handling unusual or exceptional conditions. These conditions include situations such as execution of illegal instruction or division by zero. In this section, the exception-handling capabilities of the 68000 are described.

The 68000 exceptions can be divided into three groups, namely, groups 0, 1, and 2. Group 0 has the highest priority, and group 2 has the lowest priority. Within each group, there are additional priority levels. A list of 68000 exceptions along with individual priorities is as follows:

Group 0 Reset (highest level in this group), address error (next level), and bus error (lowest level)

Group 1 Trace (highest level), interrupt (next level), illegal op-code (next level), and privilege violation (lowest level)

Group 2 TRAP, TRAPV, CHK, and ZERO DIVIDE (no individual priorities assigned in group 2)

Exceptions from group 0 always override an active exception from group 1 or group 2.

Group 0 exception processing begins at the completion of the current bus cycle (2 clock cycles). Note that the number of cycles required for a READ or WRITE operation is called a "bus cycle." This means that during an instruction fetch if there is a group 0 interrupt, the 68000 will complete the instruction fetch and then service the interrupt. Group 1 exception processing begins at the completion of the current instruction. Group 2 exceptions are initiated through execution of an instruction. Therefore, there are no individual priority levels within group 2. Exception processing occurs when a group 2 interrupt is encountered, provided there are no group 0 or group 1 interrupts.

When an exception occurs, the 68000 saves the contents of the program counter and status register onto the stack and then executes a new program whose address is provided by the exception vectors. Once this program is executed, the 68000 returns to the main program using the stored values of program counter and status register.

Exceptions can be of two types: internal or external. The internal exceptions are generated by situations such as division by zero, execution of illegal or unimplemented instructions, and address error. As mentioned before, internal interrupts are called "traps." The external exceptions are generated by bus error, reset, or interrupt instructions. The basic concepts associated with interrupts, relating them to the 68000, have already been described. In this section, we will discuss the other exceptions.

In response to an exceptional condition, the processor executes a user-written program. In some microcomputers, one common program is provided for all exceptions. The beginning section of the program determines the cause of the exception and then branches to the appropriate routine. The 68000 utilizes a more general approach. Each exception can be handled by a separate program.

As mentioned before, the 68000 has two modes of operation: user state and supervisor state. The operating system runs in supervisor mode, and all other programs are executed in user mode. The supervisor state is therefore more privileged. Several privileged instructions such as `MOVE` to SR can be executed only in supervisor mode. Any attempt to execute them in user mode causes a trap.

We will now discuss how the 68000 handles exceptions caused by external resets, trap instructions, bus and address errors, tracing , execution of privileged instructions in user mode, and execution of illegal/unimplemented instructions:

- The reset exception is generated externally. In response to this exception, the 68000 automatically loads the initial starting address into the processor.
- The 68000 has a `TRAP` instruction, which always causes an exception. The operand for this instruction varies from 0 to 15. This means that there are 16 `TRAP` instructions. Each `TRAP` instruction has an exception vector. `TRAP` instructions are normally used to call subroutines in an operating system. Note that this automatically places the 68000 in supervisor state. TRAPs can also be used for inserting breakpoints in a program. Two other 68000 instructions cause traps if a particular condition is true: `TRAPV` and `CHK`. `TRAPV` generates an exception if the overflow flag is set. The `TRAPV` instruction can be inserted after every arithmetic operation in a program in order to cause a trap whenever there is the possibility of an overflow. A routine can be written at the vector address for the `TRAPV` to indicate to the user that an overflow has occurred. The `CHK` instruction is designed to ensure that access to an array in memory is within the range specified by the

user. If there is a violation of this range, the 68000 generates an exception.

- A bus error occurs when the 68000 tries to access an address that does not belong to the devices connected to the bus. This error can be detected by asserting the $\overline{BERR}$ pin on the 68000 chip by an external timer when no $\overline{DTACK}$ is received from the device after a certain period of time. In response to this, the 68000 executes a user-written routine located at an address obtained from the exception vectors. An address error, on the other hand, occurs when the 68000 tries to read or write a word (16 bits) or long word (32 bits) in an odd address. This address error has a different exception vector from the bus error.
- The trace exception in the 68000 can be generated by setting the trace bit in the status register. In response to the trace exception, the 68000 causes an internal exception after execution of every instruction. The user can write a routine at the exception vectors for the trace instruction to display register and memory contents. The trace exception provides the 68000 with the single-stepping

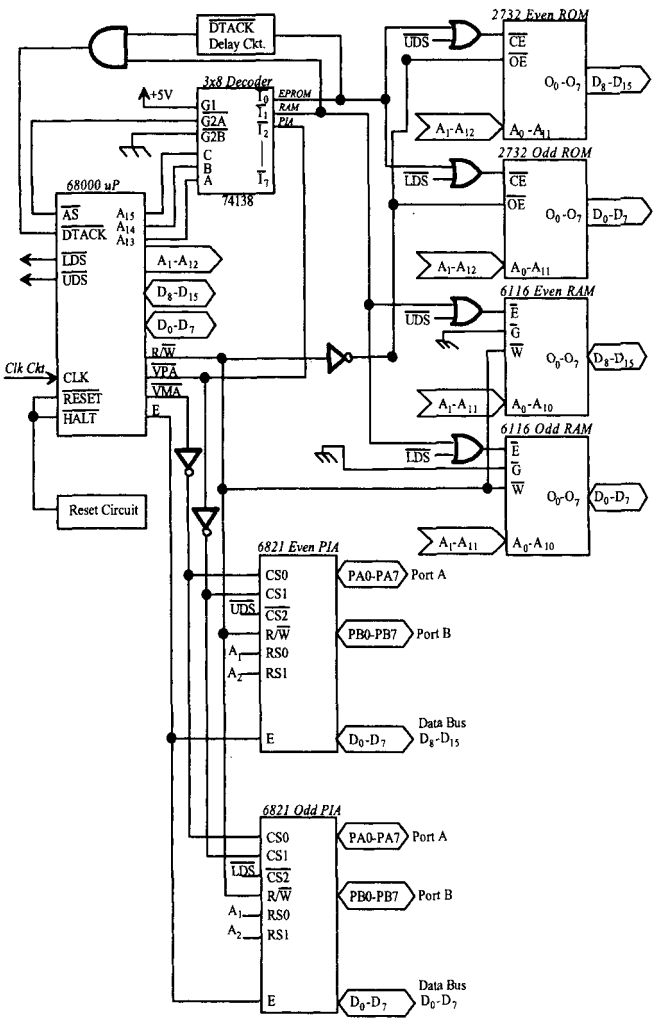


FIGURE 10.25 68000-based microcomputer

debugging feature.

- As mentioned before, the 68000 has privileged instructions, which must be executed in supervisor mode. An attempt to execute these instructions causes privilege violation.
- Finally, the 68000 causes an exception when it tries to execute an illegal or unimplemented instruction.

10.14 68000/2732/6116/6821-Based Microcomputer

Figure 10.25 shows the schematic of a 68000-based microcomputer with a 4K EPROM, a 4K static RAM, and four 8-bit I/O ports. Let us explain the various sections of the hardware schematic. Two 2732 and two 6116 chips are required to obtain the 4K EPROM and 4K RAM. The $\overline{LDS}$ and $\overline{UDS}$ pins are ORed with the memory select signal to enable the chip selects for the EPROMs and the RAMs. Address decoding is accomplished by using a 3 × 8 decoder. The decoder enables the memory or the I/O chips depending on the status of address lines A_{12}–A_{14} and the $\overline{AS}$ line of the 68000. $\overline{AS}$ is used to enable the decoder. $\overline{I_0}$ selects the EPROMs, $\overline{I_1}$ selects the RAMs, and $\overline{I_2}$ selects the I/O ports.

When addressing memory chips, the $\overline{DTACK}$ input of the 68000 must be asserted for data acknowledge. The 68000 clock in the hardware schematic is 10 MHz. Therefore, each clock cycle is 100 ns. In Figure 10.25, $\overline{AS}$ is used to enable the 3 × 8 decoder. The outputs of the decoder are gated to assert 68000 $\overline{DTACK}$. This means that $\overline{AS}$ is indirectly

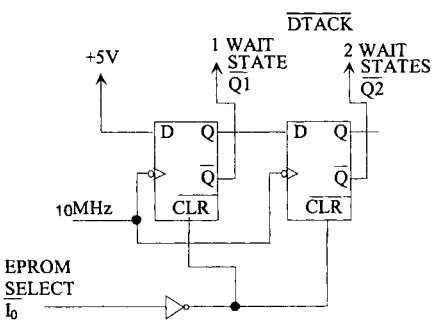


FIGURE 10.26 Delay circuit for $\overline{DTACK}$

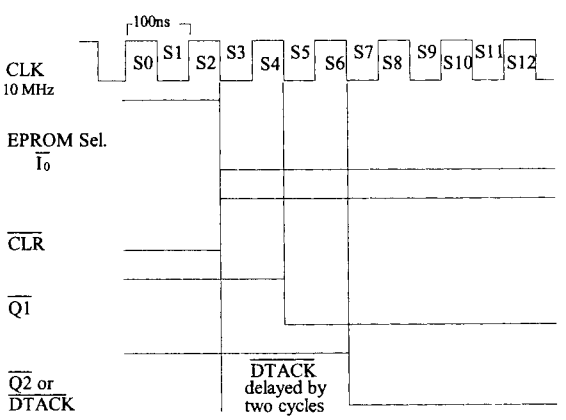


FIGURE 10.27 Timing diagram for the $\overline{DTACK}$ delay circuit

used to assert $\overline{DTACK}$. From the 68000 read timing diagram, $\overline{AS}$ goes to LOW after approximately 2 cycles (200 ns for the 10-MHz clock) from the beginning of the bus cycle. With no wait states, the 68000 samples $\overline{DTACK}$ at the falling edge of S4 (300 ns) and, if $\overline{DTACK}$ is recognized, the 68000 latches data at the falling edge of S6 (400 ns). If $\overline{DTACK}$ is not recognized at the falling edge of S4, the 68000 inserts a 1-cycle (100 ns in this case) wait state, samples $\overline{DTACK}$ at the end of S6, and, if $\overline{DTACK}$ is recognized, latches data at the end of S8 (500 ns), and the process continues. Because the access time of the 2732 is 200 ns (Used to be 450ns), data will not be available at the output pins of the 2732's until after approximately 400 ns. To be on the safe side, $\overline{DTACK}$ recognition by the 68000 at the falling edge of S6 (400 ns) and latching of data at the falling edge of S8 (500 ns) will definitely satisfy the timing requirement. This means that the decoder output $\overline{I_0}$ for EPROM select should go to LOW at the end of S6. Therefore, 200ns delay (Two cycles) for $\overline{DTACK}$ is assumed.

A delay circuit, as shown in Figure 10.26, is designed using two D flip-flops. EPPOM select activates the delay circuit. The input is then shifted right 2 bits to obtain a 2-cycle wait state to allow sufficient time for data transfer. $\overline{DTACK}$ assertion and recognition are delayed by 2 cycles during data transfer with EPROMs. Figure 10.27 shows the timing diagram for the $\overline{DTACK}$ delay circuit. Note that $\overline{DTACK}$ goes to Low after about 2 cycles if asserted by $\overline{AS}$ providing erronous result. Therefore, $\overline{DTACK}$ must be delayed.

When the EPROM is not selected by the decoder, the clear pin is asserted (output of inverter), so Q is forced LOW and $\overline{Q}$ is HIGH. Therefore, $\overline{DTACK}$ is not asserted. When the processor selects the EPROMs, the output of the inverter is HIGH, so the clear pin is not asserted. The D flip-flop will accept a high at the input, and Q2 will be HIGH and $\overline{Q2}$ will be LOW. Now that $\overline{Q2}$ is LOW, it can assert $\overline{DTACK}$. $\overline{Q1}$ will provide one wait cycle and $\overline{Q2}$ will provide two wait cycles. Because the 2732 EPROM has a 200-ns access time and the microprocessor is operating at 10 MHz (100-ns clock cycle), two wait cycles are inserted before asserting $\overline{DTACK}$ (2 × 100 = 200 ns). Therefore, $\overline{Q2}$ can be connected to the $\overline{DTACK}$ pin through an AND gate. No wait state is required for RAMs because the access time for the RAMs is only 120 nanoseconds.

Four 8-bit I/O ports are obtained by using two 6821 chips. When the I/O ports are selected, the $\overline{VPA}$ pin is asserted instead of $\overline{DTACK}$. This will acknowledge to the 68000 that it is addressing a 6800-type peripheral. In response, the 68000 will synchronize all data transfer with the E clock.

The memory and I/O maps for the schematic are as follows:

- *Memory Maps (all numbers in hex)* . A_{23} - A_{16} are don't cares and assumed to be 0's.

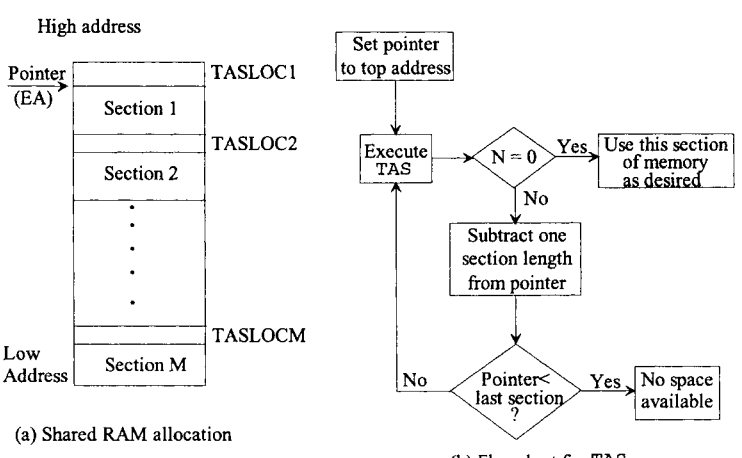
					$\overbrace{\overline{LDS} \text{ or } \overline{UDS}}$	
A_{23}–A_{16}	A_{15}	A_{14}	A_{13}	A_{12}–A_1	A_0	
0–0	0	0	0	0–0	0	EPROM(even) = 4K
				:		
0–0	0	0	0	1–1	0	$000000, $000002, $000004, ... , $001FFE
				:		
0–0	0	0	0	0–0	1	EPROM(odd) = 4K
				:		
0–0	0	0	0	1–1	1	$000001, $000003, $000005, ... , $001FFF

A_{23}–A_{10}	A_{15}	A_{14}	A_{13}	A_{11}–A_1	A_0	A_{12} is don't care for RAM (assume 0)
0–0	0	0	1	0–0	0	RAM(even) = 2K
				:		
0–0	0	0	1	1–1	0	$002000, $002002, ... , $002FFE
				:		
0–0	0	0	1	0–0	1	RAM(odd) = 2K
				:		
0–0	0	0	1	1–1	1	$002001, $002003, ... , $002FFF

Note that, upon hardware reset, the 68000 loads the supervisor SP high and low words, respectively, from addresses $000000 and $000002 and the PC high and low words, respectively, from locations $000004 and $000006. The memory map contains these reset vector addresses in the even and odd 2732 chips.

- *Memory Mapped I/O (all numbers in hex).* A_{23}–A_{16} and A_{12}–A_3 are don't cares and assumed to be 0's.

						RS1	RS0	$\overline{UDS}$ or $\overline{LDS}$	
A_{23}–A_{16}	A_{15}	A_{14}	A_{13}	A_{12}–A_3	A_2	A_1		A_0	Register Selected (Address) — Even
0–0	0	1	0	0–0	0	0		0	Port A or DDRA = $004000
0–0	0	1	0	0–0	0	1		0	CRA = $004002
0–0	0	1	0	0–0	1	0		0	Port B or DDRB = $004004
0–0	0	1	0	0–0	1	1		0	CRB = $004006
									Register Selected (Address) — Odd
0–0	0	1	0	0–0	0	0		1	Port A or DDRA = $004001
0–0	0	1	0	0–0	0	1		1	CRA = $004003
0–0	0	1	0	0–0	1	0		1	Port B or DDRB = $004005
0–0	0	1	0	0–0	1	1		1	CRB = $004007



(a) Shared RAM allocation

(b) Flowchart for TAS

FIGURE 10.28 Memory allocation using TAS

For both memory and I/O chips, $\overline{AS}$, $\overline{UDS}$ and $\overline{LDS}$ must be used in chip select logic. Note that:

1. For memory, both even and odd chips are required. However, for I/O chips, an odd-addressed I/O chip, an even-addressed I/O chip, or both can be used, depending on the number of ports required in an application. $\overline{UDS}$ and/or $\overline{LDS}$ must be used in I/O chip select logic depending on the number of I/O chips used. The same chip select logic must be used for both the even and its corresponding odd memory chip.
2. $\overline{DTACK}$ must be connected to an external input (typically a signal from the address decoding logic) to satisfy the timing requirements. In many instances, $\overline{AS}$ is directly connected to $\overline{DTACK}$.
3. The 68000 must be connected to ROMs / EPROMs / E^2PROMs in such a way that the 68000 RESET vector address is included as part of the memory map.

10.15 Multiprocessing with the 68000 Using the TAS Instruction and the $\overline{AS}$ Signal

Earlier, the 68000 TAS instruction was discussed. The TAS instruction supports the software aspects of interfacing two or more 68000's via shared RAM. When TAS is executed, the 68000 $\overline{AS}$ pin stays low. During both the read and write portions of the cycle, $\overline{AS}$ remains LOW and the cycle starts as the normal read cycle. However, in the normal read, $\overline{AS}$ going inactive indicates the end of the read. During execution of TAS, $\overline{AS}$ stays LOW throughout the cycle, so $\overline{AS}$ can be used in the design as a bus-locking circuit. Due to the bus locking, only one processor at a time can perform a TAS operation in a multiprocessor system.The TAS instruction supports multiprocessor operations (globally shared resources) by checking a resource for availability and reserving or locking it for use by a single processor.

The TAS instruction can, therefore, be used to allocate free memory spaces . The TAS instruction execution flowchart for allocating memory is shown in Figure 10.28. The shared RAM of the Figure 10.28 is divided into *M* sections. The first byte of each section will be pointed to by (EA) of the TAS (EA) instruction. In the flowchart of Figure 10.28, (EA) first points to the first byte of section 1. The instruction TAS (EA) is the executed. The TAS instruction checks the most significant bit (N bit) in (EA). N = 0 indicates that section 1 is free; N = 1 means that section 1 is busy. If N = 0, then section 1 will be allocated for use. If N = 1 (section 1 is busy), then a program will be written to subtract one section length from (EA) to check the next section for availability. Also, (EA) must be checked with the value TASLOCM. If (EA) < TASLOCM, then no space is available for allocation. However, if (EA) > TASLOCM, then TAS is executed and the availability of that section is determined.

In a multiprocessor environment, the TAS instruction provides software support for interfacing two or more 68000's via shared RAM. The $\overline{AS}$ signal can be used to provide the bus-locking mechanism.

Example 10.18

Assume that the 68000/2732/6116/6821 microcomputer shown in Figure 10.29 is required to perform the following:

(a) If Vx > Vy , turn the LED ON if the switch is open; otherwise turn the LED OFF. Write a 68000 assembly language program starting at address $000300 to accomplish the above by inputting the comparator output via bit 0 of Port B. Use Port A address = $002000, Port B address = $002004, CRA = $002002, CRB = $002006. Assume the

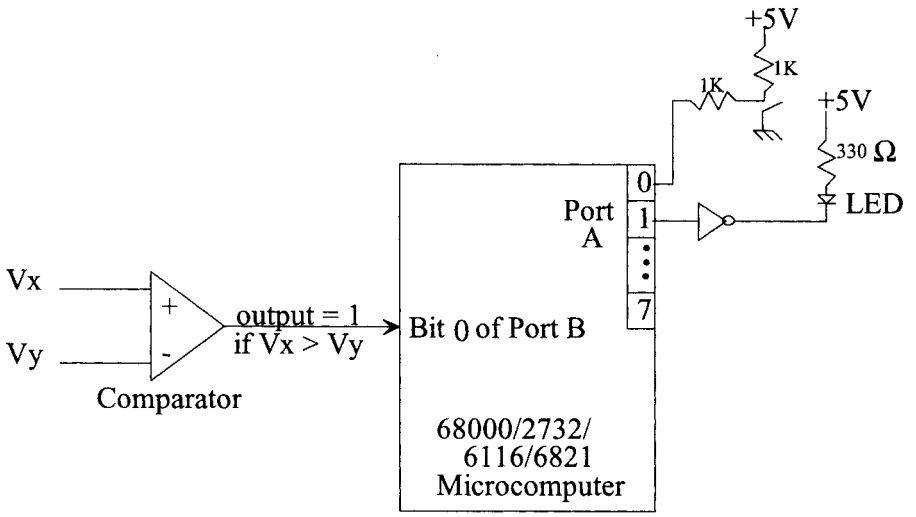


FIGURE 10.29 Figure for Example 10.18

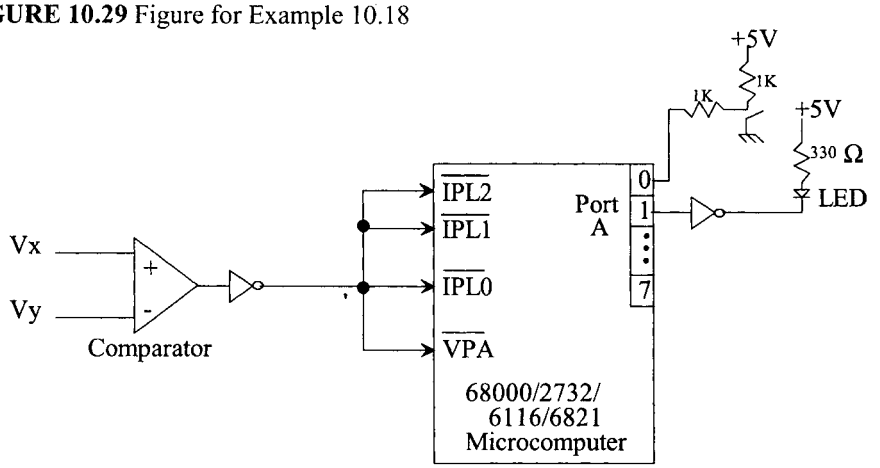


FIGURE 10.30 Example 10.18 using autovectors

LED is OFF initially.

(b) Repeat part (a) using autovector level 7 and nonautovector (Vector $40). Use Port A (address $002000) for LED and switch as above with CRA=$002002. Assume supervisor mode. Write the main program and service routine in 68000 assembly language starting at addresses $000300 and $000A00 respectively. Also, initialize the supervisor stack pointer at $001200.

Solution

(a) ***Using Programmed I/O***

From figure 10.29, the following 68000 assembly language program can be written:

```
CRA     EQU     $002002
CRB     EQU     $002006
PORTA   EQU     $002000
DDRA    EQU     PORTA
PORTB   EQU     $002004
DDRB    EQU     PORTB
        ORG     $000300
```

```
        BCLR.B  #2,CRA      ;   Address DDRA
        MOVE.B  #2,DDRA     ;   Configure PORTA
        BSET.B  #2,CRA      ;   Address PORTA
        BCLR.B  #2,CRB      ;   Address DDRB
        MOVE.B  #0,DDRB     ;   Configure PORTB
        BSET.B  #2,CRB      ;   Address PORTB
COMP    MOVE.B  PORTB,D0    ;   Input PORTB
        LSR.B   #1,D0       ;   Check
        BCC.B   COMP        ;   Comparator
        MOVE.B  PORTA,D1    ;   Input switch
        LSL.B   #1,D1       ;   Align LED data
        MOVE.B  D1,PORTA    ;   Output to LED
LED     JMP     LED
```

(b) ***Using Autovector Level 7 (nonmaskable interrupt)***

Figure 10.30 shows the pertinent connections for Autovector Level 7 interrupt.

Main Program

```
CRA     EQU     $002002
PORTA   EQU     $002000
DDRA    EQU     PORTA
        ORG     $000300
        BCLR.B  #2,CRA      ;   Address DDRA
        MOVE.B  #2,DDRA     ;  Configure PORTA

        BSET.B  #2,CRA      ;   Address PORTA
WAIT    JMP     WAIT        ;   Wait for interrupt
```

Service Routine

```
        ORG     $000A00
        MOVE.B  PORTA, D1   ;   Input switch
        LSL.B   #1, D1      ;   Align LED data
        MOVE.B  D1, PORTA   ;   Output to LED
FINISH  JMP     FINISH      ;  Halt
```

Reset Vector

```
        ORG     0
        DC.L    $00001200
        DC.L    $00000300
```

Service Routine Vector

```
        ORG     $00007C
        DC.L    $00000A00
```

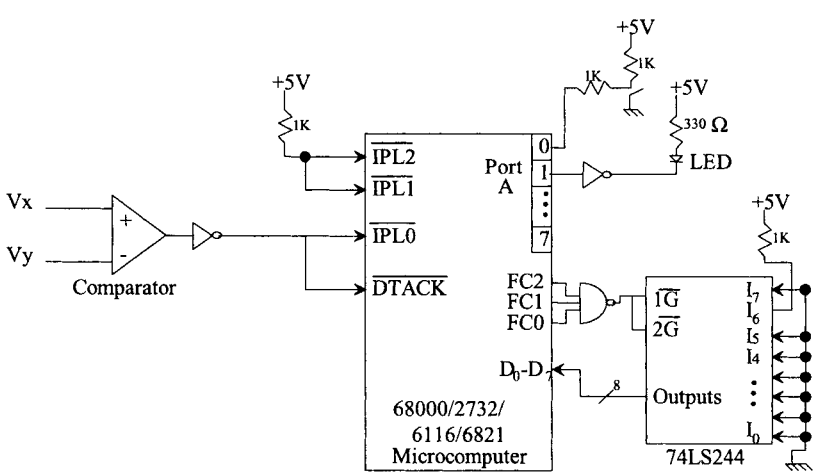


FIGURE 10.31 Example 10.18 using nonautovectors

Using Nonautovectoring (vector $40)

Figure 10.31 shows the pertinent connections for nonautovectoring interrupt.

Main Program

```
    CRA     EQU     $002002
    PORTA   EQU     $002000
    DDRA    EQU     PORTA
            ORG     $000300
            BCLR.B  #2,CRA      ;   Address DDRA
            MOVE.B  #2,DDRA     ;   Configure PORTA
            BSET.B  #2,CRA      ;   Address PORTA
            ANDI.W  #$0F8FF,SR  ;   Enable interrupts
    WAIT    JMP     WAIT        ;   Wait for interrupt
```

Service Routine

```
            ORG     $000A00
            MOVE.B  PORTA,D1    ;   Input switch
            LSL.B   #$01,D1     ;   Align LED data
            MOVE.B  D1,PORTA    ;   Output to LED
    FINISH  JMP     FINISH      ;   Halt
```

Reset Vector

```
            ORG     0
            DC.L    $00001200
            DC.L    $00000300
```

Service Routine Vector

```
            ORG     $000100
            DC.L    $00000A00
```

QUESTIONS AND PROBLEMS

10.1 What are the basic differences between the 68000, 68008, 68010, and 68012?

10.2 What does a HIGH on the 68000 FC2 pin indicate?

10.3 (a) If a 68000-based system operates in the user mode and an interrupt occurs, what will the 68000 mode be?
(b) If a 68000-based system operates in the supervisor mode, how can the mode be changed to user mode?

10.4 (a) What is the purpose of 68000 trace and X flags?
(b) How can you set or reset them?

10.5 Indicate whether the following 68000 instructions are valid or not valid. Justify your answers.
(a) `MOVE.B D0,(A1)`
(b) `MOVE.B D0,A1`

10.6 How many addressing modes and instructions does the 68000 have?

10.7 What happens after execution of the following 68000 instruction?

```
MOVE.L D0,$03000013
```

10.8 What is meant by 68000 privileged instructions?

10.9 Identify the following 68000 instructions as privileged or nonprivileged:

(a) `MOVE (A2),SR`
(b) `MOVE CCR,(A5)`
(c) `MOVE.L A7,A2`

10.10 (a) Find the contents of locations $305020 and $305021 after execution of the `MOVE D5,$305020`. Assume [D5] = $6A2FA150 prior to execution of this 68000 `MOVE` instruction.
(b) If [A0] = $203040FF, [D0] = $40F12560, and [$3040FF] = $2070, what happens after execution of the 68000 instruction: `MOVE (A0),D0`?

10.11 Identify the addressing modes for each of the following 68000 instructions:
(a) `CLR D0`
(b) `MOVE.L (A1)+,-(A5)`
(c) `MOVE $2000(A2),D1`

10.12 Determine the contents of registers / memory locations affected by each of the following 68000 instructions:
(a) `MOVE (A0)+,D1`
Assume the following data prior to execution of this `MOVE`:
[A0] = $50105020 [$105021] = $51
[D1] = $70801F25 [$105022] = $52
[$105020] = $50 [$105023] = $7F
(b) `MOVEA D5,A2`
Assume the following data prior to execution of this `MOVEA`:
[D5] = $A725B600
[A2] = $5030801F

10.13 Find the contents of register D0 after execution of the following 68000 instruction sequence:

```
EXT.W D0
EXT.L D0
```

Assume [D0] = $F215A700 prior to execution of the instruction sequence.

10.14 Find the contents of D1 after execution of `DIVS.W #6,D1`. Assume [D1] = $FFFFFFF7 prior to execution of the 68000 instruction. Identify the quotient and remainder. Comment on the sign of the remainder.

10.15 Write a 68000 assembly program to multiply a 16-bit signed number in the low word of D0 by an 8-bit signed number in the highest byte (bits 31–24) of D0.

10.16 Write a 68000 assembly program to divide a 16-bit signed number in the high word of D1 by an 8-bit signed number in the lowest byte of D1.

10.17 Write a 68000 assembly program to add the top two 16 bits of the stack. Store the 16-bit result onto the stack. Assume supervisor mode.

10.18 Write a 68000 assembly program to add a 16-bit number in the low word (bits

0–15) of D1 with another 16-bit number in the high word (bits 16–31) of D1. Store the result in the high word of D1.

10.19 Write a 68000 assembly program to add two 48-bit data items in memory as shown in Figure P10.19. Store the result pointed to by A1. The operation is given by

$$\begin{array}{r} \$00\ 02\ 03\ A1\ 07\ 20 \\ \underline{\$07\ 03\ 02\ 02\ 03\ 1A} \\ \$07\ 05\ 05\ A3\ 0A\ 3A \end{array}$$

Assume that the data pointers and the data are already initialized.

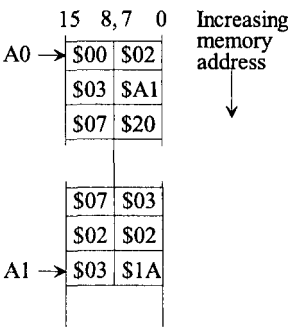


FIGURE P10.19

10.20 Write a 68000 assembly program to divide a 9-bit unsigned number in the high 9 bits (bits 31–23) of D0 by 8_{10}. Do not use any division instruction. Store the result in D0. Neglect the remainder.

10.21 Write a 68000 assembly program to compare two strings of 15 ASCII characters. The first string is stored starting at $502030. The second string is stored at location $302510. The ASCII character in location $502030 of string 1 will be compared with the ASCII character in location $302510 of string 2, [$502031] will be compared with [$302511], and so on. Each time there is a match, store $EEEE onto the stack; otherwise, store $0000 onto the stack. Assume user mode.

10.22 Write a subroutine in 68000 assembly language to subtract two 32-bit packed BCD numbers. BCD number 1 is stored at a location starting from $500000 through $500003, with the least significant digit at $500003 and the most significant digit at $500000. BCD number 2 is stored at a location starting from $700000 through $700003, with the least significant digit at $700003 and the most significant digit at $700000. BCD number 2 is to be subtracted from BCD number 1. Store the result as packed BCD digits in D5.

10.23 Write a subroutine in 68000 assembly language to compute

$$Z = \sum_{i=1}^{100} X_i$$

Assume the X_i's are signed 8-bit and stored in consecutive locations starting at $504020. Assume A0 points to the X_i's. Also, write the main program in 68000 assembly language to perform all initializations, call the subroutine, and then compute Z/100.

10.24 (a) Write a subroutine in 68000 assembly language to convert a 3-digit unpacked BCD number to binary using unsigned multiplications by 10, and additions. The most significant digit is stored in a memory location starting at $3000, the next digit is stored at $3001, and so on. Store the binary result (*N)* in D3. Note that arithmetic operations for obtaining *N* will provide binary result. Use the value of the 3-digit BCD number,

$$N = N2 \times 10^2 + N1 \times 10^1 + N0$$
$$= ((10 \times N2) + N1 \times 10 + N0$$

(b) Assume 10-MHz 68000. Write a 68000 assembly language program to obtain a delay routine for one millisecond. Using this one-millisecond routine, write a 68000 assembly language program to provide a delay for 10 seconds.

10.25 Write a 68000 assembly program to compute the following:

$$I = 6 \times J + K/M$$

where the locations $6000, $6002, & $6004 contain the 16-bit signed integers *J*, *K*, and *M*. Store the result into a long word starting at $6006. Discard the remainder of *K/M*.

10.26 Write a subroutine in 68000 assembly language program to compute the trace of a 4×4 matrix containing 8-bit unsigned integers. Assume that each element is stored in memory as a 16-bit number with upper byte as zero in the row-major order form; that is, elements are stored in memory as row by row and within a row, elements are stored as column by column. Note that the trace of a matrix is the sum of the elements of the leading diagonal.

10.27 A 68000/68230 microcomputer-based microcomputer is required to drive the LEDs connected to bit 0 of ports A and B based on the input conditions set by switches connected to bit 1 of ports A and B. The I/O conditions are as follows:

- If the input at bit 1 of port A is HIGH and the input at bit 1 of port B is low, then the LED at port A will be ON and the LED at port B will be OFF.
- If the input at bit 1 of port A is LOW and the input at bit 1 of port B is HIGH, then the LED at port A will be OFF and the LED at port B will be ON.
- If the inputs of both ports A and B are the same (either both HIGH or both LOW), then both LEDs of ports A and B will be ON.

Write a 68000 assembly language program to accomplish this.

10.28 A 68000/6821-based microcomputer is required to test a NAND gate. Figure P10.28 shows the I/O hardware needed to test the NAND gate. The microcomputer is to be programmed to generate the various logic conditions for the NAND inputs, input the NAND output, and turn the LED ON connected at bit 3 of port A if the NAND gate chip is found to be faulty. Otherwise, turn the LED ON connected at bit 4 of port A. Write 68000 assembly language program to accomplish this.

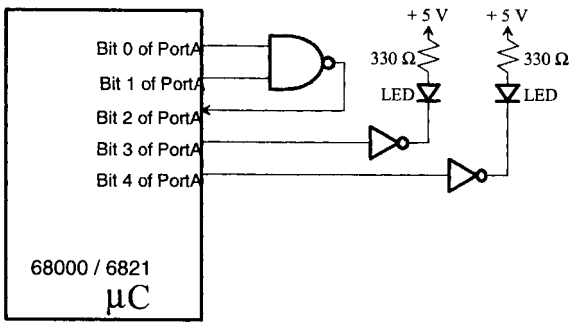


FIGURE P10.28 (Assume both LEDs are OFF initially).

10.29

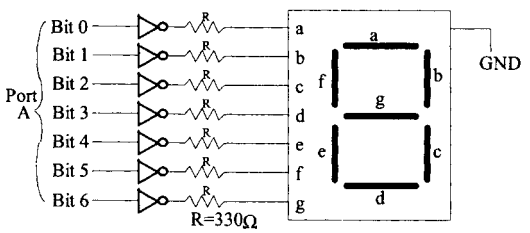


FIGURE P10.29

A 68000/68230-based microcomputer is required to add two 3-bit numbers stored in the lowest three bits of D0 and D1 and output the sum (not to exceed 9) to a common cathode seven-segment display connected at port A as shown in Figure P10.29.Write 68000 assembly language program to accomplish this by using a look-up table.

10.30 A 68000/68230-based microcomputer is required to input a number from 0 to 9 from an ASCII keyboard interfaced to it and output to an EBCDIC printer. Assume that the keyboard is connected to port A and the printer is connected to port B. Store the EBCDIC codes for 0 to 9 starting at an address $003030, and use this lookup table to write a 68000 assembly language program to accomplish the above.

10.31 Determine the status of $\overline{AS}$, FC2–FC0, $\overline{LDS}$, $\overline{UDS}$, and address lines immediately after execution of the following instruction sequence (before the 68000 tristates these lines to fetch the next instruction):

```
MOVE #$2050,SR
MOVE.B D0,$405060
```

Assume the 68000 is in the supervisor mode prior to execution of the instructions.

10.32 Suppose that three switches are connected to bits 0–2 of port A and an LED to bit 6 of port B. If the number of HIGH switches is even, turn the LED ON; otherwise, turn the LED OFF. Write a 68000 assembly language program to accomplish this.

(a) Assume a 68000/6821 system.

(b) Assume a 68000/68230 system.

10.33 Assume the pins and signal shown in Figure P10.33 for the 68000, 68230 (ODD), 2764 (ODD and EVEN). Connect the chips and draw a neat schematic. Determine the memory map and I/O map
(Addresses for PGCR, PADDR, PBDDR, PACR, PBCR, PADR, PBDR). Assume a 16.67-MHz internal clock on the 68000.

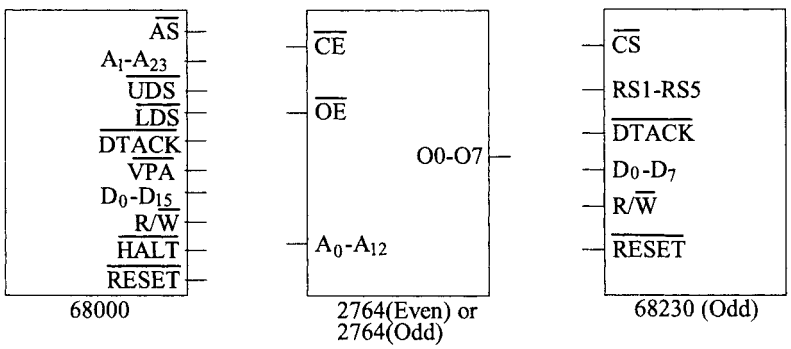


FIGURE P10.33

10.34 Find $\overline{LDS}$ and $\overline{UDS}$ after execution of the following 68000 instruction sequence:

```
MOVEA.L #$0005A123,A2
MOVE.B (A2),D0
```

10.35 (a) Write 68000 instruction sequence so that upon hardware reset, the 68000 will initialize the supervisor stack pointer to 1000_{10} and the program counter to 2000_{10}.

(b) Write a 68000 service routine at address $1000 for a hardware reset that will initialize all data registers to zero, address registers to $FFFFFFFF, supervisor SP to $502078, and user SP to $1F0524, and then jump to $7020F0.

10.36 Assume the 68000 stack and register values shown in Figure P10.36 before occurrence of an interrupt. If an external device requests an interrupt by asserting the $\overline{IPL2}$, $\overline{IPL1}$, and $\overline{IPL0}$ pins with the value 000_2, determine the contents of A7′ and SR during interrupt and after execution of RTE at the end of the service routine of the interrupt. Draw the memory layouts and show where A7′ points to and the stack contents during and after interrupt. Assume that the stack is not used by the service routine.

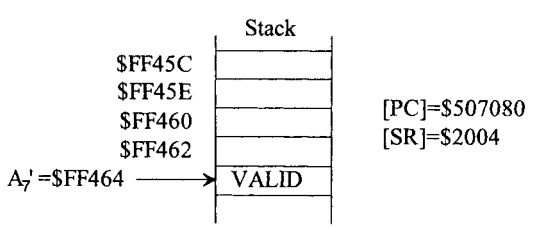


FIGURE P10.36

10.37 Consider the following data prior to a 68000 hardware reset:

[D0] = $7F2A1620
[A1] = $6AB11057
[SR] = $001F

What are the contents of D0, A1, and SR after hardware reset?

10.38 In Figure P.10.38, if $V_M > 12$ V, turn an LED ON connected at bit 3 of port A. If $V_M < 11$ V, turn the LED OFF. Using ports, registers, and memory locations as needed and level 1 autovectored interrupt:

(a) Draw a neat block diagram showing the 68000/6821 microcomputer and the connections to the diagram in Figure P10.38 to ports.

(b) Write the main program and the service routine in 68000 assembly language. The main program will initialize ports and wait for interrupt. The service routine will accomplish the above task and stop.

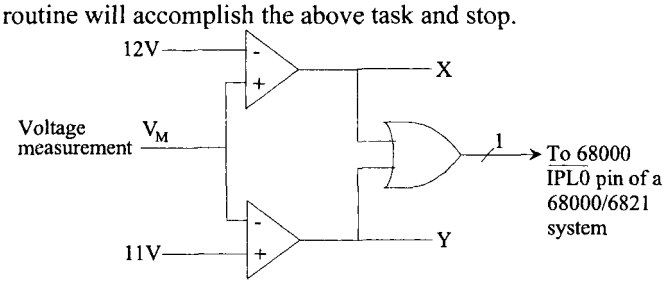


FIGURE P10.38

10.39 Write a subroutine in 68000 assembly language using the TAS instruction to find, reserve, and lock a memory segment for the main program. The memory is divided into three segments (0, 1, 2) of 16 bytes each. The first byte of each segment includes a flag byte to be used by the TAS instruction. In the subroutine, a maximum of three 16-byte memory segments must be checked for a free segment (flag byte = 0). The TAS instruction should be used to find a free segment. The starting address of the free segment (once found) must be stored in A0 and the low byte D0 must be cleared to zero to indicate a free segment and the program control should return to the main program. If no free block is found, $FF must be stored in the low byte of D0 and the control should return to the main program.

10.40 Will the circuit in Figure P10.40 work? If so, determine the I/O port addresses for PGCR, PADR, PADDR, PBDR, PBDDR, PCDR and PCDDR. If not, comment briefly, modify the circuit, and then determine the port addresses. Use only the pins and the signals shown. Assume all don't cares to be zeros.

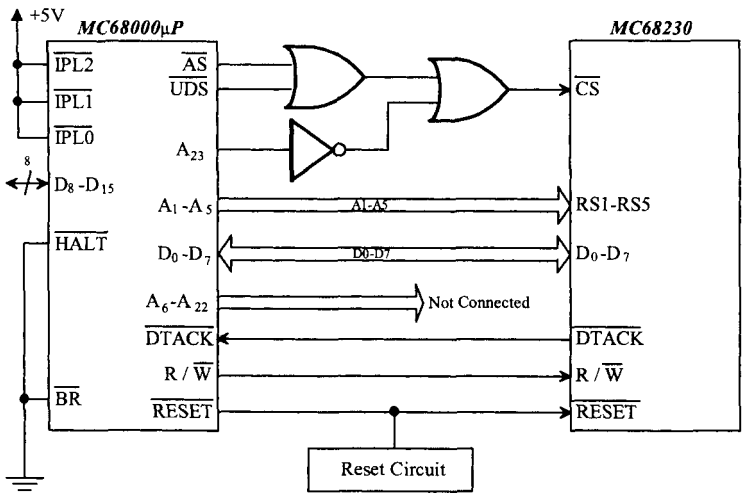


FIGURE P10.40