8

RISC MICROPROCESSORS: INTEL 80960, MOTOROLA MC88100 AND POWERPC

This chapter provides an overview of the hardware, software, and interfacing features associated with three popular RISC microprocessors namely, the Intel 80960 SA/SB, the Motorola MC88100, and the PowerPC. Finally, the basic features of typical 64-bit RISC mircoprocessors are discussed.

8.1 Basics of RISC

RISC is an acronym for Reduced Instruction Set Computer. This type of microprocessor emphasizes simplicity and efficiency. RISC designs start with a necessary and sufficient instruction set. The purpose of using RISC architecture is to maximize speed by reducing clock cycles per instruction. Almost all computations can be obtained from a few simple operations. The goal of RISC architecture is to maximize the effective speed of a design by performing infrequent operations in software and frequent functions in hardware, thus obtaining a net performance gain.

The following summarizes the typical features of a RISC microprocessor:

1. The microprocessor is designed using hardwired control with little or no microcode. Note that variable length instruction formats generally require microcode design. All RISC instructions have fixed formats, and therefore microcode design is not necessary.
2. A RISC microprocessor executes most instructions in a single cycle.
3. The instruction set of a RISC microprocessor typically includes only register-to-register, load, and store. All instructions involving arithmetic operations use registers, while load and store operations are utilized to access memory.
4. The instructions have simple fixed format with few addressing modes.
5. A RISC microprocessor has several general-purpose registers and large cache memories.
6. A RISC microprocessor processes several instructions simultaneously and thus includes pipelining.
7. Software can take advantage of more concurrency. For example, Jumps occur after execution of the instruction that follows. This allows fetching of the next instruction during execution of the current instruction.

RISC microprocessors are suitable for embedded applications. An embedded application is one in which the processor monitors and analyzes signals from one segment of the system and

produces output required by another segment of the system; thus, it behaves as a controller that bridges various parts of the entire system. It performs all its functions without any user input.

RISC microprocessors are well suited for applications such as image processing, robotics, graphics, and instrumentation. The key features of the RISC microprocessors that make them ideal for these applications are their relatively low level of integration in the chip and instruction pipeline architecture. These characteristics result in low power consumption, fast instruction execution, and fast recognition of interrupts.

The state-of-the-art 64-bit RISC microprocessors include Digital Equipment Corporation's Alpha 21164, Motorola/IBM/Apple PowerPC 620 and Sun Microsystems Ultrasparc. Among these processors, the Alpha 21164 is the fastest with a maximum clock frequency of 300 MHz, four-way superscalar design, and 128-bit data bus. These processors are compared later in this chapter.

8.2 Intel 80960

The Intel 80960 family includes two types of 16-bit RISC microprocessors. These are the 80960SA and 80960SB processors. The 80960SA is designed as an Intra-agent communication (IAC) microprocessor. IAC messages can be sent for execution into the bus interface of a 80960SA processor from software executing on another processor.

The 80960SB, on the other hand, is designed as a floating-point RISC microprocessor and includes on-chip floating-point hardware.

The 80960SA contains 32 32-bit registers while the 80960SB includes an additional four floating-point registers with a total of 36 32-bit registers.

The 80960SA/SB comes in two speeds: 10MHz and 16MHz. The clock input is divided by 2 internally to generate the internal processor clock.

8.2.1 Introduction

This section covers the basic architecture of the chip, its instruction set, typical 80960 based system design utilizing a burst controller with burst and non-burst memories.

8.2.2 Key Performance Features

The following summarizes the main features of the 80960SA/SB:

8.2.2.a Load and Store Model

Most operations are performed on operands in CPU registers rather than in memory. All of the arithmetic, comparison, branching, and bit operations are performed with registers and literals (5-bit and floating-point). Only LOAD & STORE are memory reference instructions.

8.2.2.b Large Internal Register Sets

Large internal register sets featuring 32 32-bit general purpose and specific function registers are divided into two types: global and local. Both of these types can be used for general storage of operands. The only difference between global and local registers is the global registers retain their contents across procedure boundaries, whereas the processor allocates a new set of local registers each time a procedure is called.

8.2.2.c On-Chip Code and Data Caching

To reduce memory accesses, two features are added: an instruction cache and multiple sets of local registers. The former allows pre-fetching of blocks of instructions from the main memory while the latter allows the processor to perform most procedure calls without having to write the local registers out to the stack in memory.

8.2.2.d Overlapped Instruction Execution

This is accomplished through a register scoreboarding scheme which enhances program execution speed. Register scoreboarding permits instruction execution to continue while data are being fetched from memory. When a load instruction is executed, the processor sets one or more scoreboard bits to indicate the target registers to be loaded. After the target registers are loaded, the scoreboard bits are cleared. While the target registers are being loaded, the processor is allowed to execute other instructions, called independent instructions, that do not use these registers. The net result of using this technique is that code can often be optimized in such a way as to allow some instructions to be executed in parallel.

8.2.2.e Single Clock Instructions

Most of the commonly used instructions are executed in a minimum number of clock cycles (usually one clock).

For example, instructions, either 32 or 64-bits long, are aligned on 32-bit boundaries allowing instructions to be decoded in one clock cycle. This eliminates the need for an instruction-alignment stage in the pipeline resulting in over 50 instructions that can be executed in a single clock cycle.

8.2.2.f Interrupt Model

To handle interrupts, the processor maintains an interrupt table of 248 interrupt vectors, of which 240 are available for general use. When an interrupt is generated, the processor uses a pointer from the interrupt table to perform an implicit call to an interrupt handler procedure. The processor automatically saves the state of the processor prior to receiving the interrupt, performs the interrupt routine, and then restores its previous state. A separate interrupt stack is also provided to segregate interrupt handling from application programs. Interrupt handling facilities feature prioritizing pending interrupts.

8.2.2.g Procedure Call Mechanism

Each time a call instruction is issued, the processor automatically saves the current set of local registers and allocates a new set of local registers for the called procedure. Likewise, on the return from a procedure, the current set of local registers is deallocated and the local registers for the procedure being returned to are restored. Thus, on a procedure call, the program never has to explicitly save and restore those local variables.

8.2.2.h Instruction Set and Addressing

The processor offers a full set of load, store, move, arithmetic, comparison, and branch instructions, with operations on both integer and ordinal data types. It also provides a complete set of Boolean and bit-field instructions, to simplify operations on bits and bit strings. The addressing modes are efficient and straightforward, while at the same time providing the necessary indexing and scaling modes required to address complex array and records.

The 32 address lines provide 4-gigabytes of address space for programs and data.

Table 8.1 lists the 80960SA/SB instruction set. The 80960SA does not include the floating-point instructions.

8.2.2.i Floating Point Unit (Available with 80960SB only)

The on-chip floating point unit includes a full set of floating point operations including add, subtract, multiply, divide, trigonometric functions, and logarithmic functions. These operations are performed on single precision (32-bit), double precision (64-bit), and extended precision (80-bit) data. Four 80-bit floating-point registers are provided to hold extended precision values.

8.2.3 80960 SA/SB Registers

Figure 8.1 shows the 80960SB registers. The processor provides three types of data registers: global, floating-point, and local. As their names imply, the global registers constitute a set of general-purpose registers whose contents are retained across procedure boundaries. The

TABLE 8.1 80960SA/SB Instruction Set

Data Movement	Arithmetic	Logical	Bit and Bit Field
Load, Store, Move, Load Address	Add, Subtract, Multiply, Divide, Shift, Remainder, Modulo, Extended Multiply, Extended Divide	And, Not And, And Not, Or, Xor, Not Or, Or Not, Nor, Exclusive Nor, Not, Nand, Rotate	Set Bit, Clear Bit, Not Bit, Check Bit, Alter Bit, Scan for Bit, Scan Over Bit, Extract, Modify
Comparison	**Branch**	**Call/Return**	**Fault**
Compare, Conditional Compare, Compare and Increment, Compare and Decrement	Unconditional Branch, Conditional Branch, Compare and Branch	Call, Call extended, Call system, Return, Branch and Link	Conditional Fault, Synchronous Fault
Debug	**Miscellaneous**	**Decimal**	**Conversion**
Modify Trace Controls, Mark, Force Mark	Atomic Add, Atomic Modify, Flush Local Register, Modify Arithmetic Controls, Modify Process Control, Scan Byte For Equal, Test Condition Code	Move, Add with Carry, Subtract with Carry	Convert Real to Integer, Convert Integer to Real
Floating-Point		**Synchronous**	
Move Real, Add, Subtract, Multiply, Divide, Remainder, Scale, Round, Square Root, Sine, Cosine, Tangent, Arctangent, Log, Log Binary, Log Natural, Exponent, Classify, Copy Real Extended, Compare		Synchronous Move, Synchronous Load	

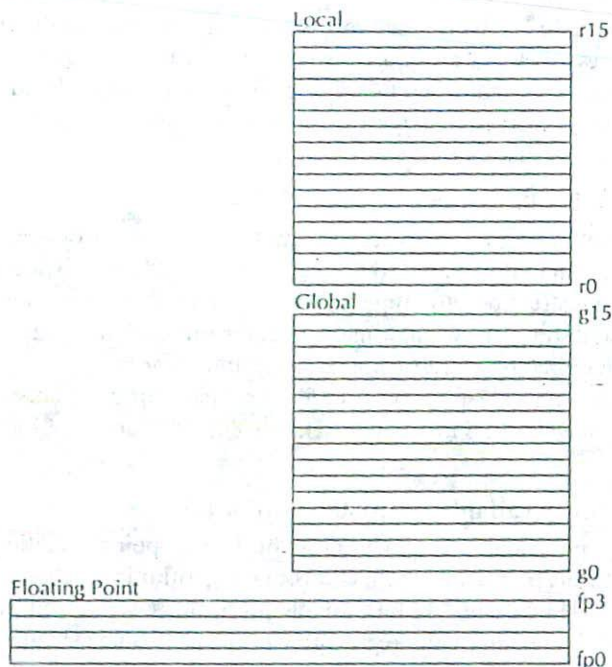


FIGURE 8.1 Local and global registers sets.

4 floating point registers are for extended precision floating point operations and are available only with the 80960SB. Their contents are also preserved across procedure calls. The 16 local registers are to hold local variables. For each procedure that is called, the processor allocates a separate set of local registers.

It should be noted that the global register g15 is reserved to hold the current frame pointer FP, while the others are available for general use. The local register r0 is used to hold the previous frame pointer (PFP), r1 is the stack pointer, r2 is used as a Return Instruction Pointer which is saved on the stack later and r3-r15 are available for general use.

Some special features of the 80960SB registers are provided in the following:

8.2.3.a Register Scoreboarding

The main purpose is to permit instructions to be executed concurrently provided that they are independent instructions.

8.2.3.b Instruction Pointer

The 32-bit r1 holds the address of the instruction currently being executed. Since the instructions are required to be aligned on a word boundary, the least significant 2 bits of IP are always 0. IP can not be read directly. However, IP can be used as an offset into address space. This addressing mode can be used with the load address (lda) instruction to read the current value of IP. When a break in instruction stream occurs due to an interrupt or procedure call, the IP contents will be stored in r2, and later saved on the stack.

8.2.3.c Process Control Register

The processor's process control register is made up of a set of 32 bits, as shown below:

```
Bit 0      :  Trace Enable
Bit 1      :  Execution Mode, 0 = User, 1 = Supervisor
Bit 10     :  Trace Fault Pending
Bit 13     :  State Flag, 0 = Execution Mode, 1 = Interrupted
               Mode
Bit 20-16  :  Priority
```

8.2.3.d Arithmetic Control

The arithmetic control bits include the condition code, arithmetic status, integer overflow flag and mask, floating point overflow, underflow, zero divide, invalid-op, inexact flags, masks, and faults. The processor sets or clears these bits to show the results of certain operations. For example, the processor modifies the condition code flags after each fault. These bits are set by the currently running program to tell the processor how to respond to certain fault conditions.

```
Bit 0-2    :  Condition Code
Bit 3-6    :  Arithmetic Status
```

```
Bit 8      :  Integer Overflow Flag
Bit 12     :  Integer Overflow Mask
Bit 15-20  :  Floating Point Condition Flags
Bit 24-29  :  Floating Point Condition Masks
Bit 30-31  :  Floating Point Normalizing and Rounding Mode
```

8.2.4 Data Types and Addresses

In order to be consistent with the data types included in the Intel 80960 manual, new terminologies such as ordinal and literals are introduced in this section.

8.2.4.a Data Types

The processor defines and operates on the following data types:

• Integer (8, 16, 32, and 64 bits)	signed whole numbers
• Ordinal (8, 16, 32, and 64 bits)	general-purpose, unsigned whole numbers
• Real (32, 64, and 80 bits) (floating-point)	conforms to IEEE single (32-bit), double (64-bits), and extended precision (80-bit) floating point representations
• Bit/Bit Field	span of 1 or more bits within register boundary
• Decimal (ASCII digits)	decimal values in ASCII format.
• Triple-Word (96 bits)	consecutive bytes
• Quad-Word (128 bits)	consecutive bytes

8.2.4.b Literals

The processor recognizes two types of literals: ordinal and floating-point, which can be used as operands in some instructions. An ordinal literal can range from 0 to 31 (5 bits). When an ordinal literal is used as an operand, the processor expands it to 32 bits by adding leading zeros. If an ordinal literal is used in an instruction that requires integer operands, the processor treats the literal as a positive integer value.

For floating-point, the processor recognizes two literals: +0.0 and +1.0. These floating point literals can only be used with floating point instructions. Ordinal literals can also be used in converting integer to real to get more values.

8.2.4.c Register Addressing

A register may be used as an operand in an instruction by giving the register number (e.g., g0, f5, fp3). Both floating-point and non-floating-point instructions can reference global and local registers in this way. However, floating-point registers can only be referenced in conjunction with a floating-point instruction.

If the instruction requires more than one word, the reference is to the lowest number, which must be even when 2 words are required, must be multiples of four when 3 or 4 words are required. This is called "Register Alignment."

8.2.4.d Memory Addressing Modes

Table 8.2 lists the 80960 memory addressing modes.

8.2.4.d.i Absolute Offset. Absolute offset is used to reference a memory location directly. An example is st g2, START which stores the word from register g2 into memory location START.

8.2.4.d.ii Register Indirect/Register Indirect with Offset. This mode permits an address to be specified with an ordinal value (32 bits) in a register or a displacement added to a value in a register. The register value is called the address base (abase). An example of register indirect is the ldob (r1), r2 which loads an ordinal byte from memory location addressed by r1 into r2. An example of

TABLE 8.2 80960 Memory Addressing Modes

Mode	Description	Assembler Syntax
Absolute offset	offset	exp
Register indirect	abase	(reg)
Register indirect with offset	abase + offset	exp(reg)
Register indirect with index	abase + (index * scale)	(reg)[reg*scale]
Register indirect with index and displacement	abase + (index*scale) + displacement	exp(reg)[reg*scale]
Index with displacement	(index*scale) + displacement	exp[reg*scale]
IP (instruction pointer) with displacement	IP + displacement + 8	exp(IP)

register indirect with offset is stl g4, BEGIN(g2) which stores double word from g4, g5 stored at memory location addressed by BEGIN+(g2).

8.2.4.d.iii *Register Indirect with Index/Register.* This mode allows a scaled index value in another register to be added to the value in a register. The scale factor can be 1, 2, 4, 8, or 16. A displacement may also be added to the abase value and scaled index. An example is ldq (r1) [r2*4], r4 which loads a quad-word starting at the memory location addressed by (r1)+(r2 scaled by 4) into register r4 through r7.

An example of register indirect with index and displacement is st g1, VALUE(g3)[g4*4] which loads word in g1 into memory location addressed by (g3) +VALUE+(g4*4).

8.2.4.d.iv *Index with Displacement.* This mode allows a scaled index to be used with a displacement. The index is contained in a register and is multiplied by a scaling constant before the displacement is added to it. An example is ldis VALUE [r8*2], r10 which loads short integer at memory location addressed by VALUE+(r8*2) into r10.

8.2.4.d.v *IP with Displacement.* This mode is often used with load and store instructions to make them IP relative. With this mode the displacement plus a constant of 8 is added to the IP of the instruction. An example is st r1, VALUE (IP) which stores words in r1 at memory location addressed by 8+IP+VALUE.

8.2.5 80960SA/SB Instruction Set

The 80960 includes 182 instructions. An assembly-language statement consists of an instruction mnemonic, followed by from 0 to 3 operands, separated by commas. The following example illustrates the assembly-language statement for the addo instruction:

```
addo g1, g3, g5
```

adds the ordinal operands in global register g1 and g3 and stores the result in g5.

The instructions can be classified into four categories:

1. Data Movement
2. Conversion
3. Arithmetic and Logic Operations
4. Comparison and Control

The following provides a list of operands used in the instructions:

reg — global (g0, g1, . . . , g15) or local (r0, r1, . . . , r15) registers
freg — global (g0, g1, . . . , g15) or local (r0, r1, . . . , r15) registers or floating-point (fp0 thru fp3) registers
lit — integer or ordinal literal of the range 0. . .31
flit — floating-point literal of value 1.0 or 0.0

disp — signed displacement of range -2^{22} to $+2^{22}-1$
mem — address defined with the full range of addressing modes
addr — address
efa — effective address

8.2.5.a Data Movement

The data movement instructions move data between the global and local registers, and between these registers and memory.

8.2.5.a.i Load Instructions.

Instruction	Operands
Load integer byte, ldib (8-bit)	mem, reg
Load integer short, ldis (16-bit)	
Load ordinal byte, ldob (8-bit)	
Load ordinal short, ldos (16-bit)	

The above instructions load a byte or half word (2 bytes) and convert it into a full 32-bit word. Integers are sign-extended, ordinals and zero-extended automatically. For example, ldib (r1), r0 loads the 8-bit integer in memory addressed by r1 into register r0.

```
Load - ld
Load long - ldl          mem, reg
Load triple - ldt
Load quad - ldq
```

The ld, ldl, ldt and ldq instructions copy 4, 8, 12, and 16 bytes, respectively, from memory into successive registers.

ldl mem, reg must specify an even numbered register (eg. g0, g2, . . ., g16).

ldt mem, reg and ldq mem, reg must specify a register that is a multiple of four (eg. g0, g4, g8, . . ., r0, r4, r8,. . .) For example, consider ldq 1254(r1), r4 loads the contents of memory location starting at address r1+1254 into register r4 thru r7.

8.2.5.a.ii Store Instructions. Each load instruction has a corresponding store instruction which stores bytes or words from registers to memory.

The store instructions are listed below:

```
st
stob
stos
stib
stis     reg, mem
stl
stt
stq
```

The stob and stib, and stos and stis instructions store a byte and half-word (16-bit), respectively, from the low order bytes of the specified source register. The st, stl, stt, and stq instructions store 4, 8, 12, and 16 bytes, respectively, from successive registers to memory.

For the stl instruction, the specified source register must specify an even numbered register (e.g. g0, g2, . . ., g16). For the stt and stq instructions, the specified source register number must be a multiple of four such as g0, g4, g8. . ., g16.

As an example, the instruction st g4, 2478(g8) stores the word in register g4 into memory location starting at offset 2478+(g8).

8.2.5.a.iii MOVE. The move instructions copy data from a register or group of registers to another register or group of registers. These are listed below:

```
move word - mov                 }
move longword - movl            }  reg/lit, reg
move triple word - movt         }
move quad word - movq           }
```

The movl, movt, and movq instructions specify the source and destination registers as the first (lowest numbered) register of several successive registers. For the movl, these registers must be even numbered such as g0, g2, . . , r0, r2, . . . while for the movt and movq instructions, these registers must be an integral multiple of four such as g0, g4. . . , r0, r4, ...

As an example, the instruction movt r4, g8 moves a triple word (three 32-bit) from registers r4, r5, r6 into g8, g9, g10.

8.2.5.a.iv Load Address. lda mem, reg computes an effective address specified with mem or efa and stores it in the destination, reg. Note that efa represents an effective address based on an addressing mode. This instruction loads a constant longer than 5 bits into a register. To load a register with a constant of 5 bits or less, the move instruction (mov) can be used with a literal as the source operand.

As an example, the instruction, lda 40(g7), g0 computes the effective address specified with 40+(g7) and stores it in g0.

lda 0x845, r4 loads the constant 845H into r4. Note that 0x indicates data in hexadecimal.

8.2.5.a.v Floating-Point Move (Available with 80960SB Only). The following move-real instructions (movr, movrl and movre) are provided for moving real number values between the global and local registers and the floating-point registers:

```
move real - movr                }
move longreal - movrl           }  freg/flit, freg
move extended real - movre      }
```

As an example, the following instruction sequence converts a real value in g1 to a long real value, which is stored in g8, g9.

```
movr    g1, fp0
movrl   fp0, g8
```

The two instructions cpysre and cpyrsre for real extended numbers are explained in the following:

```
cpysre   src1, src2, dst
 or
cpyrsre  freg/flit freg/flit freg
```

copies the absolute value of src1 into dst based on the sign of src2.

For cpysre: If src2 is positive then dst ← abs(src1); else dst ← abs(src1).
For cpyrsre: If src2 is negative then dst ← abs(src1); else dst ← abs(src1)

If the src1, src2, or dst operand specifies one of g0 thru g15 or r0 thru r15, this register (lowest) is the first of three successive registers. Also, this register number must be a multiple of 4 (e.g. g0, g4, g8, . .).

As an example, the instruction, cpysre g0, fp1, r4 means that the absolute value from g0g1g2 is copied to r4r5r6; the sign from fp1 is copied to r4r5r6.

8.2.5.b Conversion (Available With 80960SB Only)

As mentioned before, data can be converted from one length to another by means of the load and store instructions. For example, the ldis instruction loads a short integer from memory to a register and automatically converts the integer from a half word to a full word.

The 80960SB extended instruction set provides instructions to perform conversions between integer and real data types. These instructions are listed below:

```
Convert integer to real , cvtir          }
Convert long integer to real, cvtilr     }
Convert real to integer, cvtri           }  reg/lit, freg
Convert real to long integer, cvtril     }
Convert truncated real to integer, cvtzri          }
Convert truncated real to long integer, cvtzril    } freq/flit,reg
```

For the cvtilr instruction, the source operand specifies the first (lowest numbered) of two successive registers. This register must be even numbered (e.g. g0, g2, g4, . . .).

Converting an integer to long real format requires two instructions as follows:

1. cvtir or cvtilr can be used to convert the integer to extended real.
2. movrl can then be used to move the value from freg to two global or local registers.

For example the instruction sequence:

```
cvtir g2, fp0
movrl fp0, g4
```

converts an integer in g2g3 to real and stores it in fp0; movrl then converts the real value in fp0 to a long real value and stores the result in g4g5.

The cvtril and cvtzril instructions specify the destination operand as the first (lowest numbered) of two successive registers. This register must be even numbered. Also, the nontruncated version of cvtzri and cvtzril instructions round according to the current rounding mode in the Arithmetic Control register. The truncated version always rounds towards zero.

As an example, the following instruction sequence converts a long real value in g8g9 to a long integer in g2g3:

```
movrl g8, fp0    ; long real source in g8g9 is converted to
                 ; extended-real format in fp0
cvtril fp0, g2   ; extended real value in fp0 is converted
                   to long
                 ; integer in g2g3.
```

Synchronous Load and Move

Both the 80960SA and 80960SB include these instructions.

The 80960SA/SB executes the store instructions asynchronously with the memory controller. Once the processor outputs data for storing in main memory, it continues with execution of the next instruction in the program, and assumes that its bus control logic hardware will

complete the operation. The 80960SA/SB includes four special instructions for performing memory operations that perform store and move operations synchronously with memory.

The synchronous load instruction, synld reg/addr, reg copies a word from the source into a register. When this instruction is performed, the processor waits until a condition code bit is set in the arithmetic control register indicating that the operation has been completed, before it begins executing the next instruction. The synld instruction is primarily used to read the contents of the interrupt-control register.

The following instructions

Synmov
Synmovl } reg/addr, reg/addr
Synmovq

copies one (synmov), two (synmovl) or four (synmovq) words from memory location(s) specified by the source to the destination and waits for completion, including those operations initialed prior to this instruction. The primary function of these instructions is for sending IAC (Inter-agent communication) messages. The primary function of an IAC mechanism is to provide alternative to the external interrupt mechanism to communicate with the processor. Also, certain processor functions such as purging the instruction cache and setting breakpoint registers can only be done with the IAC mechanism. IAC messages are defined in such a way that processors can send them amongst themselves on the bus in a multiprocessor system. For example, a program on processor A can send a message to processor B telling it to flush its instruction cache. Without this facility, processor A would need to generate an interrupt to processor B to tell a program in processor B to flush the cache.

Since IAC messages carry out specific control functions that are not included in instructions, they are useful in single-processor systems. The 80960SA/SB can send an IAC message by writing the message to a special memory-mapped location. The memory mapping only occurs if one of the synchronous load/move instructions is used. A memory write to its specific memory-mapped location using one of these instructions does not cause a bus operation to occur; instead the data are interpreted by the processor as an IAC message and the message causes the same function to be performed by the processor. The function is performed synchronously (i.e. immediately after the synchronous load/move) instruction.

8.2.5.c Arithmetic and Logic Operations

8.2.5.c.i Table 8.3 lists 80960SA/SB add, subtract, multiply, divide, and shift instructions.

TABLE 8.3 80960SA/SB Add/Subtract/Multiply/Divide/Shift

Operations	Instructions and data types			
	Integer	Ordinal	Real	Long Real
add* src1, src2, dst reg/lit reg/lit reg dst ← src2 + src1 * = i or o or r or rl	addi	addo	addr	addrl
sub* src1, src2, dst reg/lit reg/lit reg dst ← src2 - src1 * = i or o or r or rl	subi	subo	subr	subrl
mul*src1, src2, dst freg/flit freg/flit freg dst ← src2 * src1 * = i or o or r or rl	muli	mulo	mulr	mulrl

TABLE 8.3 80960SA/SB Add/Subtract/Multiply/Divide/Shift (*continued*)

Operations	Instructions and data types			
	Integer	Ordinal	Real	Long Real
div* src1, src2, dst dst ← src2 / src1 reg/lit reg/lit reg * = i or o or r or rl No remainder is provided after div*, dst contain quotient.	divi	divo	divr	divrl
rem*src1, src2, dst freg/flit freg/flit freg Performs src2 / src1 and stores the remainder in dst. The sign of the result (if nonzero) is the same as the sign of src2. Calculation of the remainder is done by repeated subtraction. * = i or o or r or rl	remi	remo	remr	remrl
signed integer modulo: modi srcl, src2, dst reg/lit reg/lit reg dst = src2 – (src2 + src1)*src1	modi	—	—	—
Shift left shl* Len, src, dst reg/lit reg/lit reg Shifts src left by the number of bits specified in the Len operand and stores result in dst. For values greater than 32, the processor interprets the value as 32. * = i or o	shli	shlo	—	—
Shift right shr* len, src, dst reg/lit reg/lit reg shifts src right by the number of bits indicated with the len operand and stores the result in dst. For values of len greater than 32, the processor interprets the value as 32. * = i or di or o	shri shrdi	shro	—	—

Details of Table 8.3

Note the instructions addr/addrl, subr/subrl, roundr/roundrl and sqrtr/sqrtrl are only available with the 80960SB.

For addrl, subrl, mulrl, divrl and remrl instructions, if src1, src2, or dst operand specifies one of the registers from g0 thru g15 or r0 thru r15, the register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (eg. go, g2, g4, . . .).

The binary results from subi and subo are identical except that subi can signal an integer overflow.

For the divi instruction, an integer overflow can be signaled.

The shlo instruction shifts zeros into the least-significant bit and the shro instruction shifts zeros into the most-significant bit.

The shli instruction shifts zeros into the least-significant bit; if the bits shifted out are not the same as the sign bit, an overflow is generated. If overflow occurs, the sign of the result is the same as the sign of the src operand.

The shri instruction performs an arithmetic shift operation by shifting the sign bit in from the most-significant bit.

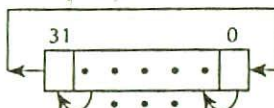
The shrdi instruction is provided for dividing an integer by a power of 2. With shrdi, one is added to the result if the bits shifted out are non-zero and the operand is negative, which produces the correct result for negative operands.

Remi and modi differ when there is a negative operand: the result of remi has the same sign as the dividend; that of modi has the same sign as the divisor. For example, if r3 = 3, r4 = (–7):

"remi r3, r4, r5" stores (–1) to r5, (–7) = –2 * 3 + (–1)
"modi r3, r4, r5" stores 1 to r5, (–7) = –2 * 3 – 1

shrdi adds 1 to the result if bits shifted out are non-zero and operand is negative, which produces the correct result for negative operands (if division is desired).

8.2.5.c.ii Rotate Instruction. The operation of the rotate instruction is provided below:

Instruction	Operation
rotate len, src, dst reg/lit reg/lit reg	dst ← rotate (len mod 32 (src)) copies src to dst and rotates the bits in the dst as follows:  The len operand specifies the number that the dst operand is rotated. The len operand can be from 0 to 31. The instruction can also be used to rotate bits to right.

8.2.5.c.iii Extended Arithmetic. There are four instructions for double precision integer arithmetic. These are described below:

1. Add ordinal with carry,
   addc src1, src2, dst
   reg/lit reg/lit reg
   Operation: dst ← src2 + src1 + carry
   Flags affected: carry (c) and overflow (v)
2. Subtract ordinal with carry,
   subc src1, src2, dst
   reg/lit reg/lit reg
   Operation: dst ← src2 – src1 – carry
   Flags affected: carry, overflow.
3. Extended multiply,
   emul src1, src2, dst
   reg/lit reg/lit reg
   Operation: dst + 1, dst ← src1 * src2
   The result is 64 bits and is stored in two adjacent registers. The dst operand specifies the lower numbered register, which receives the least significant bits of the result. The dst operand must be an even numbered register (ro, r2, r4, . . ., or g0, g2, . . .).
4. Extended Divide
   ediv src1, src2, dst
   reg/lit reg/lit reg
   Operation: dst ← Remainder of src2/src1
   dst + 1 ← Quotient of src2/src1
   Scr2 is a long ordinal (64 bits) which is contained in two adjacent registers. Src2 specifies the lower numbered register which contains the least significant bits of the operand. Src2 operand must be an even numbered register. Src1 value is a normal ordinal 32 bits. dst operand must be an even numbered register.

8.2.5.c.iv Floating-Point Arithmetic Instructions (Available with 80960SB Only). In addition to floating-point add (addr/addrl), subtract (subr/subrl), multiply (mulr/mulrl) and divide (divr/divrl) which were already explained, additional floating-point instructions are listed in Table 8.4.

Note that in Table 8.4,

For roundrl, sqrtrl, sinrl, cosrl, tanrl, logbnrl, exprl. → If the src or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. This register must be even numbered (g0, g2,. . ., r0, r2. . .).

For atanrl, logrl, logeprl, scalerl, → If the src1, src2, or dst references a global or local register, this register is the first (lowest numbered) of two successive registers. This register must be even numbered (g0, g2, . . ., r0, r2, . . .).

TABLE 8.4 80960SB Floating-Point Arithmetic Instructions Beyond Add/Subtract/Multiply/Divide

Operation		Instructions and Data Types	
		Real	Long Real
Basic:	round* src, dst freg/lit freg rounds src to the nearest integral value depending on the rounding mode and stores the result in dst. * = r or rl	roundr	roundrl
	sqrt* src, dst freg/flit freg calculates the square root of src and stores it in dst * = r or rl	sqrtr	sqrtrl
Trigonometric Operation			
Calculate the specified trigonometric function of src and stores the result in dst. The src value is in radians. The resulting dst value is in the range −1 to +1 inclusive for sine and cosine	Sin* src, dst freg/flit freg * = r or rl	sinr	sinrl
	cos* src, dst freg/flit freg * = r or rl	cosr	cosrl
For tangent, the source value is a finite real number between −∞ to +∞	tan* src, dst freg/lit freg * = r or rl	tanr	tanrl
atan calculates arctangent of src2/src1 and stores result in dst. The result is in radians and lies between −π to π inclusive. The sign of the result is same as the sign of src2.	atan* src1, src2, dst freg/lit freg/lit freg * = r or rl	atanr	atanrl
Operation (Logarithmic, Exponential, and Scale)			
Logbn calculates the $\log_2$(src) and stores the integral part of this value as real number in dst.	logbn* src, dst freg/flit freg * = r or rl	logbnr	logbnrl
Log* calculates src2*$\log_2$(src1) and stores result in dst. Compute y*$\log_2$(x).	log*src1, src2, dst freg/flit freg/flit freg * = r or rl	logr	logrl
logep* calculates rc2*$\log_2$(src1 + 1) and stores result in dst. Compute y*$\log_2$(x + 1).	logep*src1, src2, dst freg/flit freg/flit freg * = r or rl	logepr	logeprl
exp* performs dst ← (2^{src} − 1). The source value must be within −0.5 to +0.5 inclusive. Compute 2^x − 1.	exp* src, dst freg/flit freg	expr	exprl

TABLE 8.4 80960SB Floating-Point Arithmetic Instructions Beyond Add/Subtract/Multiply/Divide (*continued*)

Operation (Logarithmic, Exponential and Scale)		Instructions and Data Types Real	Long Real
scale* performs dst ← src2*(2^{src1}). src1 is integer, src2 and dst are reals. Multiply a floating-point value by a power of 2.	scale*src1, src2, dst reg/lit freg/flit freg	scaler	scalerl

8.2.5.c.v *Logical, Bit/Bit Field Operations.*

Table 8.5 lists these instructions:

TABLE 8.5 Logical Instructions

Instruction	Operation Performed
and src1, src2, dst reg/lit reg/lit reg	dst ← src2 ∧ src1
andnot src1, src2, dst reg/lit reg/lit reg	dst ← src2 ∧ (src)′
nand src1, src2, dst reg/lit reg/lit reg	dst ← (src2 ∧ src1)′
notand src1, src2, dst reg/lit reg/lit reg	dst ← (src2) ′ ∧ src1
or src1, src2, dst reg/lit reg/lit reg	dst ← src2 ∨ src1
ornot src1, src2, dst reg/lit reg/lit reg	dst ← src2 ∨ (src1)′
nor src1, src2, dst reg/lit reg/lit reg	dst ← (src2 ∨ src1)′
notor src1, src2, dst reg/lit reg/lit reg	dst ← (src2) ′ ∨ src1
xor src1, src2, dst reg/lit reg/lit reg	dst ← src2 ⊕ src1
xnor src1, src2, dst reg/lit reg/lit reg	dst ← (sr2 ⊕ src1)′

Note that in the above, ∧ = and, ∨ = or, ⊕ = exclusive or, ′ = NOT

Table 8.6 lists bit/bit field instructions.

TABLE 8.6 Bit/Bit Field Instructions

Instruction	Operation
alterbit bitpos, src, dst reg/lit reg/lit reg	copies the src to dst with one bit altered. The bitpos specifies the bit to be changed and the condition code determines the value the bit is to be changed to. If the condition code is 010_2, the selected bit is set to one; if the condition code is 000_2, the bit is cleared to zero.
chkbit bitpos, src reg/lit reg/lit	checks the bit in src specified by bitpos and sets the condition code according to the value found. If the bit is one, the condition code is set to 010_2; if the bit is zero, the condition code is cleared to 000_2.
clrbit bitpos, src, dst reg/lit reg/lit reg	copies src to dst with the bit specified by bitpos cleared to zero.
notbit bitpos, src, dst reg/lit reg/lit reg	copies src to dst with the bit specified by bitpos ones complemented.
scanbit src, dst reg/lit reg	searches src for most-significant set-bit. If the set-bit is found, its bit number is stored in dst and the condition code is set to 010_2. If src is zero, all ones are stored in dst and the condition code is cleared to 000_2.
setbit bitpos, src, dst reg/lit reg/lit reg	copies src to dst with the bit specified by bitpos set to one.

TABLE 8.6 Bit/Bit Field Instructions (*continued*)

Instruction	Operation
sparobit src, dst reg/lit reg	searches src for the most-significant clear-bit. If the clear-bit is found, its number is stored in dst and the condition code is set to 010_2. If the src value is all ones then all ones are stored in dst and the condition code is cleared to 000_2.
extract bitpos, len, src/dst reg/lit reg/lit reg	shifts a specified bit field in src/dst right and fills the bits to the left of the shifted bit field with zeros. The bitpos value specifies the least significant bit of the bit field to be shifted and the len value specifies the length of the bit field.
modify mask, src, src/dst reg/lit reg/lit reg	modifies selected bits in src/dst with bits from src. The mask operand selects the bits to be modified: Only the bits set in the mask operand are modified in src/dst. src/dst ← (src ∧ mask) ∨ (src/dst ∧ (mask)′)

8.2.5.c.vi Byte Operations. The scanbyte instruction performs a byte-by-byte comparison of two ordinals to determine if two corresponding bytes are equal.

The format of the scanbyte is as follows:

```
scanbyte src1, src2
reg/lit  reg/lit
```

The scanbyte performs a byte-by-byte comparison of src1 and src2 and sets the condition code to 010_2 if any two corresponding bytes are equal. If no corresponding bytes are equal, the condition code is cleared to 000_2.

The scanbyte operation is detailed below:

$$\text{If } (\text{src1} \wedge 000000\text{FF}_{16}) = (\text{src2} \wedge 000000\text{FF}_{16})$$
$$\text{or}$$
$$(\text{src1} \wedge 0000\text{FF}00_{16}) = (\text{src2} \wedge 0000\text{FF}00_{16})$$
$$\text{or}$$
$$(\text{src1} \wedge 00\text{FF}0000_{16}) = (\text{src2} \wedge 00\text{FF}0000_{16})$$
$$\text{or}$$
$$(\text{src1} \wedge \text{FF}000000_{16}) = (\text{src2} \wedge \text{FF}000000_{16})$$

then condition code = 010_2; else condition code = 000_2.

8.2.5.c.vii Decimal Arithmetic (Available with 80960SB Only). These instructions operate on 32-bit decimal operands that contain an 8-bit ASCII-coded decimal in the least-significant byte.

dmovt src, dst reg reg	copies src to dst. The least significant byte of src is tested to find whether or not it is a valid ASCII digit (30_{16} thru 39_{16}). If the value is a valid ASCII decimal, the condition code is cleared to 000_2; otherwise, it is set to 010_2. This instruction is normally used iteratively to validate decimal strings.
daddc src1, src2, dst reg reg reg	adds bits 0 thru 3 of src2 and src1 (with bit 1 of condition code used here as carry bit). The result is stored in bits 0 thru 3 of dst. If there is a carry after addition, bit 1 of condition code is set to one. Bits 4 thru 31 of scr2 are copied to dst unchanged. The instruction assumes that the least significant 4 bits of src1 and src2 are valid BCD digits.

The daddc is intended to be used iteratively to add BCD values in which the least significant four bits of the operands represent valid BCD numbers from 0 to 9.

dsubc src1, src2, dst reg reg reg	subtracts bits 0 thru 3 of src1 and src2 as follows: dst ← src2 − src1 − 1 + C.

Bit 1 of condition code is used as C (carry bit). The other characteristics of dsubc are same as the daddc instruction.

The dsubc is intended to be used iteratively to subtract BCD values in which the least significant four bits of the operands represent valid BCD numbers from 0 to 9.

8.2.5.c.viii Atomic Instructions. In multiprocessor systems, a mechanism is required to allow programs to manipulate shared data in an indivisible manner so that when such an operation is underway, another processor cannot perform the same operation. The 80960 includes two instructions called atomic instructions to implement higher-level synchronization mechanisms, such as locks and semaphores.

The atmod src, mask, src/dst
reg reg/lit reg
addr

instruction copies the src/dst value into the memory location specified in src. The src is a register containing the address and thus the name reg addr in the instruction. The bits set in the mask operand select the bits to be modified in memory. The initial value from memory is stored in src/dst.

For example, atmod g1, g3, g6 performs the following:

g1 ← g1 ANDed by g3 where g1 contains the address of a word in memory.
g6 ← Initial value stored at address g1 in memory.

The read and write of memory are done atomically (i.e. other processors are prevented from accessing the word of memory specified with the src/dst operand until the operation has been completed).

The memory location in src is the address of the first byte (least significant byte) of the word to be modified.

The atadd src/dst, src, dst
reg reg/lit reg
addr

adds the src value (full word) to the value in memory specified by src/dst. The initial value from memory is stored in dst.

The read and write of memory are done atomically. The memory location in src/dst is the address of the first byte (least significant byte) of the word.

The atadd instruction, therefore, adds a value of a word in memory and returns the original value of the word. For example, atadd r2, r4, r9 performs the following:

r2 ← r4 + (r2) where r2 specifies the address of a word in memory;
r9 ← initial value stored at address r2 in memory.

The atomic read operation waits until the LOCK line on the external bus is not asserted and then asserts the LOCK line and performs the read. The atomic write operation performs a write operation and deasserts the LOCK line. This ensures that another processor cannot perform an atomic read operation between read and write to the word in memory specified with the src/dst operand until the operation has been completed.

8.2.5.d Comparison and Control

Though this 80960SA/SB RISC processor has a condition code register, it is not affected by most arithmetic and movement instructions. An explicit comparison instruction is needed for

conditional branches. This feature has its advantage. Between the instruction that sets condition code and the instruction that performs conditional branching, many independent arithmetic operations can be inserted. That will increase the pipeline efficiency. Arithmetic instructions that change condition codes are: addc, subc, dmovt, daddc, dsubc.

8.2.5.d.i Comparison. These instructions compare integer (signed numbers) and ordinals (unsigned numbers):

Compare Integer (cmpi)/Ordinal (cmpo)
cmpi src1, src2
or reg/lit reg/lit
cmpo

compares src2 and src1 values and sets the condition code according to the following:

Condition Code	Comparison
100	src1 < src2
010	src1 = src2
001	src1 > src2

Compare and Increment Integer(cmpinci)/Ordinal (cmpinco)
cmpinci src1, src2, dst
or reg/lit reg/lit reg
cmpinco

compares src2 and src1 values and sets the condition code according to the results of the comparison. Src2 is then incremented by one and the result is stored in dst.

The condition codes are affected by the comparison result in exactly the same way as the cmpi/cmpo.

Conditional Compare Integer (concmpi)/Ordinal (concmpo)
concmpi src1, src2
or reg/lit reg/lit
concmpo

compares src2 and src1 value if bit 2 of the condition code is not set. If the comparison is performed, the condition code is set according to the comparison results in the same way as cmpi/cmpo.

Compare and Decrement Integer (cmpdeci)/Ordinal (cmpdeco)
cmpdeci rc1, src2, dst
or reg/lit reg/lit reg
cmpdeco

compares src2 and src1 values and sets the condition code according to the comparison results in the same way as cmpi/cmpo. The src2 is then decremented by one and the result is stored in dst.

The following instructions are for real and long real floating-point numbers (available with only 80960SB microprocessor):

Compare real (cmpr/longreal (cmprl))
compr src1, src
or freg/flit freg/flit
comprl

compares src2 with src1 and sets the condition code according to the result as follows:

Condition Code	Comparison
100	src1 < src2
010	src1 = src2
001	src1 > src2
000	if either src1 or scr2 is a NaN

cmpr/cmprl clears the condition code flags to 000_2 for the unordered condition. Note that the unordered relationship is true when at least one of the two values compared is a NaN.

Compare Ordered Real (cmpor)/Ordered Long Real (cmporl)

cmpor src1, src2
or freg/flit freg/flit
cmporl

compares src2 and src1 and sets the condition code in the same way as cmpr/cmprl.

Compor/comporl clears the condition code to 000_2 and an invalid-operation exception is signaled for the unordered condition. Note that the unordered condition is true when at least one of the two values being compared is a NaN.

Classify Real (cassr)/Long Real (classrl)

classr src
or freg/flit
classrl

checks classification of real number in src and stores the class in arithmetic-status bits (3 through 6) of the arithmetic controls as follows:

A Status	Classification
S000	Zero
S001	Denormalized number
S010	Normal finite number
S011	Infinity
S100	Quiet NaN
S101	Signaling NaN
S110	Reserved operand

The S bit is set to the sign of the src operand.

For cmprl and cmporl and classrl instructions, if src1 or src2 for cmprl/cmporl or src for calssrl specifies a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered.

8.2.5.d.ii Control Instructions. The 80960SA/SB include the following unconditional branch instructions:

Branch (b)/Branch Extended (bx)

b targ or bx targ
disp mem

branches to the instruction specified with the targ operand.

For the b instruction the range of targ operand is from -2^{23} to $(2^{23} - 4)$ bytes from the current IP. For bx, the targ can be farther than -2^{23} to $(2^{23} - 4)$ bytes for the current IP. Also, since the

targ operand for bx is a memory type, full range of addressing words including register indirect mode can be used.

Branch and Link (bal)/Link Extended (balx)
bal targ or balx targ, dst
disp mem reg

stores the address of the next instruction (next IP value) in a register and branches to the instruction specified with the targ operand. These instructions are intended for calling leaf procedures (procedures that do not call other procedures). Using the b or bx instruction, the leaf procedure can branch to the IP saved by bal or balx.

For bal, the address of the next instruction is saved in g14. The range of targ is from -2^{23} to $2^{23} - 4$.

The balx performs the same operation as the bal except that the address of the next instruction is stored in dst, allowing it to be stored in any available register. The range of targ can be farther -2^{23} to $(2^{23} - 4)$ bytes from the current IP.

Compare and Branch

These instructions compare two operands, then branch (or not) according to the result:

Compare Branch If	Integer (Signed)	Ordinal (Unsigned)
Equal	cmpibe src1, src2, targ	cmpobe src1, src2, targ
Not Equal	cmpibne src1, src2, targ	cmpobne src1, src2, targ
Less	cmpibl src1, src2, targ	cmpobl src1, src2, targ
Less or Equal	cmpible src1, src2, targ	cmpoble src1, src2, targ
Greater	cmpibg src1, src2, targ	cmpobg src1, src2, targ
Greater or Equal	cmpibge src1, src2, targ	cmpobge src1, src2, targ
Ordered	cmpibo src1, src2, targ	cmpobo src1, src2, targ
Unordered	cmpibno src1, src2, targ	cmpobno src1, src2, targ

In the above instructions, src1 = reg/lit, src2 = reg, and targ = disp. These instructions compare src1 and src2, and set the condition code based on the result. If the AND of the condition code and the mask part of the instruction is not zero, the processor branches to targ; otherwise, the processor goes to the next instruction. Note that the condition code 000_2 indicates no condition and is the unordered condition while condition code = 111_2 is the 'ordered' condition. The terms 'ordered' and 'unordered' are used when comparing two floating-point numbers. If, when comparing two floating-point values, one of the values is a NaN (Not a number), the relationship is said to be 'unordered'; otherwise, the releationship is 'ordered'.

Bit Instructions:
Check Bit and Branch if SET, bbs bitpos, src, targ
Check Bit and Branch if Clear, bbc reg/lit reg disp

bbs and bbc instructions check the bit in src-specified by bitpos and set the condition code according to the value. The processor then branches to targ according to the condition.

Test Condition Codes

These instructions cause a TRUE (1) to be stored in a destination register if the condition code matches. Otherwise, a FALSE (0) is stored.

teste dst Test if Equal testne dst Test if Not Equal
testl dst Test if Less testg dst Test if Greater

testle dst Test if Less or Equal	**testge** dst Test if Greater or Equal
testo dst Test Ordered	**testno** dst Test if Unordered

In the above, dst = reg.

Conditional Fault

These instructions permit a fault to be generated explicitly according to the state of the condition-code bits:

faulte	Fault if Equal	faultne	Fault if Not Equal
faultl	Fault if Less	faultg	Fault if Greater
faultle	Fault if Less or Equal	faultge	Fault if Greater or Equal
faulto	Fault Ordered	faultno	Fault if Unordered

Call and Return

The processor offers an on-chip call return mechanism for making procedure calls to local procedures and kernel procedures. These instructions support that mechanism:

call	targ	Calls where targ = disp
callx	targ	Calls Extended where targ = mem
calls	targ	Calls System where targ = reg/lit
ret		Return

The **call** and **callx** instructions call local procedures. They differ only in addressing mode. The processor will allocate a new set of local registers and a new stack frame for the called procedure. The **calls** instruction operates similarly, except that it gets its target procedure address from the system procedure table. Depending on the type of entry being pointed to in the procedure table, the **calls** instruction can cause a supervisor procedure call to be executed.

The **ret** instruction performs a return from a called procedure to a calling procedure. The same instruction is used to return from local and supervisor calls and from implicit calls to interrupt and fault handlers. The processor takes care of all the details.

Debug

The processor supports debugging and program tracing. These are the debugging tools:

modtc	Modify Trace Control
mark	Mark — generates a breakpoint trace event if breakpoint trace mode flag is enabled.
fmark	Force Mark — generates a breakpoint trace event regardless of the breakpoint trace mode flag.

Processor Management

The processor provides several instructions for use in controlling processor-related functions.

modpc src, mask, src/dst reg/lit reg/lit reg	stores the contents of src/dst in the process control register, with the bits set in the mask modified. The src/dst then contains the initial value of the process control register. The src/dst is a dummy operand and must be set equal to the mask operand. The processor must be in the supervisor mode for executing this instruction.
flushreg	copies each local register set except the current set to its associated stack-frame in memory and marks them as invalid, meaning that they will be reloaded from memory if and when they become the current local register set.

modac mask, src, dst reg/lit reg/lit reg	places the contents of src in the Arithmetic control register with the bits set in the mask modified register. The dst then contains the initial value of the Arithmetic controls.

Conditional Branches:

be	Branch if Equal	bne	Branch if Not Equal
bl	Branch if Less	bg	Branch if Greater
ble	Branch if Less or Equal	bge	Branch if Greater or Equal
bo	Branch if Ordered	bno	Branch if Unordered

These instructions are single-operand with the operand "targ" or "disp" defined in the same way as bal.

Value of π

The 80960SA/SB uses the value $41490FDA_{16}$ for π. The details of this computation are given in Intel i960SA/SB Microprocessor SA/SB reference manual. As an example, π can be located into a register such as r4 by using lda 0X41490fda, r4 where ox is used to represent hexadecimal number by the 80960SA/SB assembler.

80960 Assembler

The 80960 assembler uses the first operand of a two operand instruction as the source operand and the second operand as the destination. The assembler directive # is used before a comment. 0X before an immediate number is used to represent a hex number.

Example 8.1

Identify the addressing modes for the following 80960 instructions:

i) ldl 4816(r3),g4
ii) st r3,34(r8)[r4*4]

Solution

	source	destination
i)	register indirect	register
ii)	register	register indirect with scaled index and displacement.

Example 8.2

Determine whether the following 80960 instructions are valid or invalid. Comment.

i) ldq (g8) [g9], r2
ii) stl 46, 52(r5)

Solution

i) Not valid since for ldq instruction, the destination must specify a register number that is multiple of 4 such as r0, r4, r8, . . . , g0, g4, g8, ... Since register r2 is not a multiple of 4, the instruction is invalid.
ii) Valid since for stl, the source must be an even numbered register which is r6 in this case.

Example 8.3

Write an 80960 instruction sequence to read 32-bit elements 5, 6, and 7 from a table stored in memory into register r1, r2, and r3 respectively. Assume that register r5 points to the starting address containing element 0 (32-bit data) of the table.

Solution

```
ldt 5(r5), r8     #r8 ← ((r5+5))
                  #r9 ← ((r5+6))
                  #r10 ← ((r5+7))
mov r8, r1        #r1 ← r8
mov r9, r2        #r2 ← r9
mov r10, r3       #r3 ← r10.
```

Example 8.4

Write an assembly language program in 80960 assembly language to add two 64-bit numbers. Assume that the two 64-bit numbers are stored in r1, r0 and r3, r2 respectively. Store the 64-bit result in r0, r1.

Solution

```
       cmpo 1, 0        # clears bit 1(carry bit)
                        # of the condition code register
       addc r0,r2,r0    # r0 ← r2+r0+carry bit
       addc r1,r3,r1    # r1 ← r3+r1+carry bit
finish b finish         # halt
```

Example 8.5

Write an 80960 assembly language program to perform the following operation:

$$(A/B) + C * D$$

where A, B, C, D are stored in r0, r1, r2, r3 as 32-bit integers. Assume C*D generates 32-bit product. Discard remainder of A/B. Store the 32-bit result in r4.

Solution

```
       divi r1, r0, r4   # r4 ← r0/r1
       muli r2, r3, r5   # r5 ← r2*r3
       addi r5, r4       # r4 ← r4 + r5
finish b finish          # stop
```

Example 8.6

Write a program in 80960 assembly language that copies bits 3-6 of register r1 into bits 31-28 of register r2.

Solution

```
extract 3, 4, r1    # r1 = 000. . . 000aaaa
shlo 28, r1, r1     # r1 = aaaa000. . . 000
```

```
        shlo 28, 15, r3      # r3 = 1111000. . . 000
        modify r3, r1, r2    # r2 = aaaabbb. . . bbb
finish  b finish             # halt
```

Example 8.7

Write a program in 80960 assembly language to branch to a label 'start' if the 32-bit operand in register g2 is not a finite number.

Solution

```
        classr g2
        modac 0, 0, g1          # place condition code in g1
                                # arithmetic status in bits
                                #   3-6 of g1
        shro 3, g1, g1          # move arith status in bits
                                #   0-3 of g1
        and 7, g1, g1           # g1 = bits 0-2 of arith status
        cmpobge g1, 3, start    # branch if status not equal
                                # to s000, s001, or s010.
finish  b finish                # halt
```

Example 8.8

Write an 80960 instruction to add four 32-bit words of additional space to the stack.

Solution

```
        addo sp, 16, sp   # sp ← sp + 16
```

Note that the 80960 Intel assembler uses 'sp' to represent the stack pointer, r1. Also, sp in the above instruction is incremented in one-byte increments by the addo instruction so that sp must be incremented by 16.

Example 8.9

Write a program in 80960 assembly language to convert a long-real value in r0 to long-integer value in r8.

Solution

```
        movre ro, fp0      # long-real value in
                           # ro is converted to
                           # extended-real in fp0
        cvtril fp0, r8     # extended real-value
                           # in fpo is converted to
                           # long integer
finish  b finish           # halt
```

Example 8.10

Write a program in 80960 assembly language to compute the area of a circle by using $A = \pi r^2$ where 'A' is the area in 32-bit real to be stored in register r1 and 'r' is the radius of the circle stored in r0 as 32-bit real.

Solution

```
        mulr ro, ro, g10    #  calculate r²
        lda 0x41490fda,r1   #  Load π
        mulr g10, r1, r1    #  r1 contains the area
finish  b finish            #  halt
```

Example 8.11

Write an 80960 assembly language program to convert from polar coordinates to rectangular coordinates as follows:

$$x = r\cos\theta, \qquad y = r\sin\theta$$

where r0, r1 contain r and θ (in radians) of the polar coordinates and r2, r3 contain x, y of the rectangular coordinates respectively.

Solution

```
        cosr r1, g4        #  g4 = cos(r1)
        sinr r1, g5        #  g5 = sin(r1)
        mulr r0, g4, r2    #  r2 = r*cos(r1)
        mulr r0, g5, r3    #  r3 = r*sin(r1)
finish  b finish           #  halt
```

8.2.6 80960SA/SB Pins and Signals

Figure 8.2 shows the 80960SA/SB pins and signals.

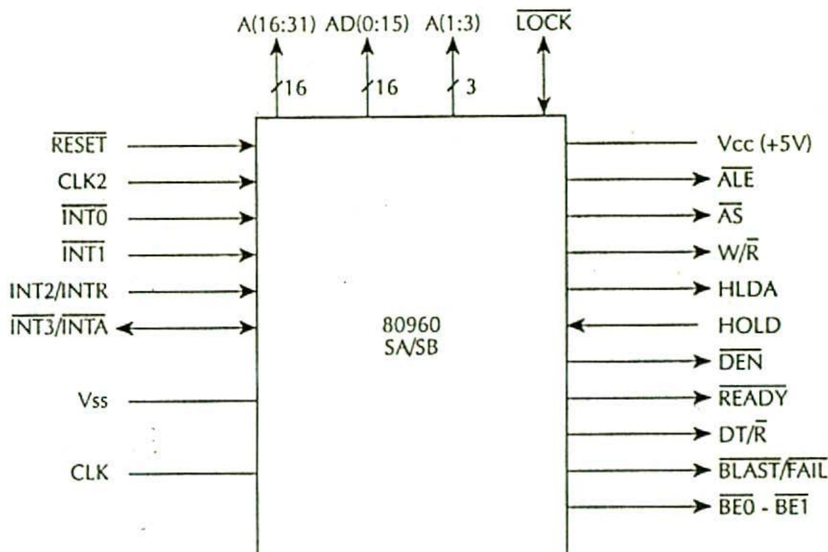


FIGURE 8.2 80960SA/SB pins and signals.

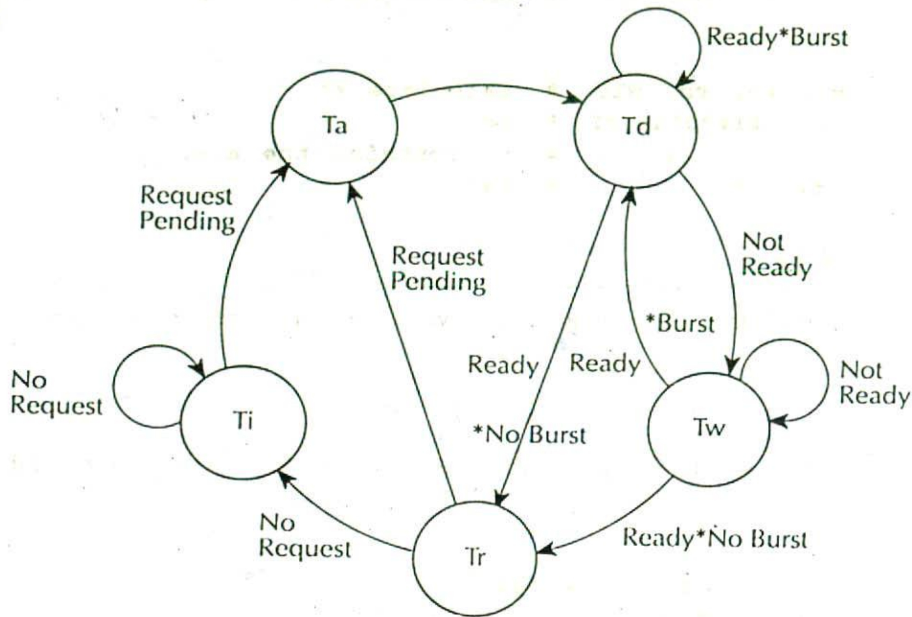


FIGURE 8.3 Basic L-bus states.

Some of the main features of the 80960SA/SB bus include the following:

- 32-bit addressing
- 16-bit multiplexed low 16-bit address/data bus.
- Two byte enables and an eight-word burst capability that allows transfers from 1 to 16 bytes in length.
- Basic bus states.

8.2.6.a Basic Bus States

There are five basic bus states: idle Ti, address Ta, data Td, recover Tr, and wait Tw as shown in Figure 8.3 assuming only one bus master resides on the bus.

Ti: the processor enters this state when no address or data transfer is in progress.
Ta: when processor receives a new request and starts transmitting address.
Td: following Ta, the processor transmits or receives data if $\overline{\text{READY}}$ input is asserted. If not the processor enters wait state Tw and remains there until data is ready. Tw may be repeated allowing sufficient time for I/O devices to respond.
Tr: following Td, the processor enters recovery state and comes back to Ti. In case of burst transactions, it exits Tw or Tr to transfer next data word. When done, it enters recovery state.

8.2.6.b Signals Groups

Address and Data Lines

The address/data signal consists of 35 lines.

A16-A31: ADDRESS BUS carries upper 16-bit of 32-bit addresses to memory. No latch is required.
AD(0-15): 16-bit LOW ADDRESS/DATA BUS represents addresses in Ta and data in Td.

A(1-3): ADDRESS BUS carries the burst addresses to memory. They are incremented during burst cycle indicating the next byte address of burst mode. They are duplicated with AD(1-3) during the address cycle.

Control Lines

Consists of 12 signals that permit the transfer of data.

$\overline{\text{ALE}}$: ADDRESS LATCH ENABLE, active high, is asserted during Ta and deasserted before the beginning of the Td state. It floats when processor is not-a bus master.

$\overline{\text{AS}}$: ADDRESS STATUS indicates an address state is asserted every address state and deasserted during the following Td and driven HIGH during RESET.

DT / $\overline{\text{R}}$: DATA TRANSMIT/RECEIVE indicates the flow of data. During READ operation at TA, Td, and Tr, it remains LOW. It is HIGH during Write operations.

$\overline{\text{DEN}}$: DATA ENABLE enables data transceivers and is asserted during Td and Tw.

$\overline{\text{READY}}$: INPUT from other devices indicates data on the bus ready to be read or written. If not asserted during a Td cycle, the Td cycle is extended for the next cycle by inserting a wait state, Tw.

$\overline{\text{LOCK}}$: BUS LOCK, prevents other devices from gaining the control of the bus. Asserted when processor performs READ MODIFIED WRITE or INTERRUPT ACKNOWLEDGE. WAITS if LOCK is asserted by other devices.

$\overline{\text{BE1}}$ - $\overline{\text{BE0}}$: BYE ENABLE indicates which data bytes (up to two) on the bus take part in the current bus cycle. BE1 corresponds to AD15-A8 and BE0 corresponds to AD7-AD0.

W / $\overline{\text{R}}$: Instructs memory or I/O devices to write or read data on the bus. It is asserted during Ta and remains valid during subsequent Td cycles.

HOLD, HLDA: Used for DMA.

$\overline{\text{BLAST}}$ / $\overline{\text{FAIL}}$: Indicates that an error occurred during the initialization. The failure state is indicated by a combination of $\overline{\text{BLAST}}$ asserted and both $\overline{\text{BE}}$ signals not asserted. $\overline{\text{FAIL}}$ is asserted while the processor performs a self-test. If the self test is successful, the $\overline{\text{FAIL}}$ is deasserted.

CLK2/CLK: The 80960SA/SB uses two clock signals (CLK2 and CLK). CLK2 provides the input clock to the 80960 and is double the specified processor frequency. CLK is the clock input signal for the peripheral devices and is the operating frequency of the processor.

The four interrupt pins of the 80960SA/SB are $\overline{\text{INT0}}$, INT1, INT2/INTR, and $\overline{\text{INT3}}$/ $\overline{\text{INTA}}$. The on-chip control register determines how these interrupts are used by the processor. The 80960SA/SB can be interrupted using any of the two methods as follows:

1. Receipt of a signal on any or all of the four direct interrupts ($\overline{\text{INT0}}$, INT1, INT2, and $\overline{\text{INT3}}$).
2. Receipt of a signal on the interrupt request (INTR) utilizes $\overline{\text{INTA}}$ to obtain an interrupt vector from an external device such as the 8259. The setting of the on-chip Interrupt Control Register selects one of the methods.

The $\overline{\text{RESET}}$ pin must be asserted for at least 41 CLK2 cycles. Upon hardware reset, the 80960 performs a self test if $\overline{\text{INT0}}$ INT1 $\overline{\text{INT3}}$ $\overline{\text{LOCK}}$ = 1x11. If the self test fails, the 80960SA/SB

enters the stopped state. Otherwise, the 80960 performs a checksum test of 16 words fetched from memory at physical address 00000000_{16}. After a successful checksum test, the 80960SA/SB uses some of the previously fetched words as addresses to initial data structures.

8.2.7 Basic READ and WRITE

READ:

1. During Ta state:
   - The processor places address on the address and address/data lines.
   - It asserts $\overline{ALE}$ used to latch address.
   - It asserts $\overline{AS}$.
   - W/$\overline{R}$ is low indicating read operation.
   - DT/$\overline{R}$ is low and used as direction input to data transceivers.
2. During Td state:
   - The processor reads data on the AD(0-15) pins.
   - It asserts $\overline{DEN}$ which is used to enable data transceivers.
   - The processor asserts $\overline{BE1}$ - $\overline{BE0}$ to specify which bytes the processor uses when reading the data word.
   - $\overline{READY}$ is asserted by external logic to indicate data is ready to be read. If not asserted, Tw is generated and repeated until $\overline{READY}$ is asserted.
3. The recovery states follow the data state allowing adequate time for external devices to remove their data from the bus before the 80960SA/SB generates the next address on the bus. W/$\overline{R}$, DT/$\overline{R}$, and $\overline{DEN}$ become inactive.

WRITE:

1. During Ta state:
   - The processor places address on address and address/data lines.
   - It asserts $\overline{ALE}$ used to latch address.
   - It then asserts $\overline{AS}$.
   - W/$\overline{R}$ is HIGH indicating WRITE operation.
   - DT/$\overline{R}$ is HIGH and used as direction input to data transceivers.
2. During Td state:
   - The processor places data on the AD(0-15) pins.
   - The processor asserts $\overline{BE1}$ - $\overline{BE0}$ to specify which bytes the processor is writing in the word.
   - It asserts $\overline{DEN}$ used to enable data transceivers.
   - $\overline{READY}$ is asserted by external logic to indicate data written. If not asserted, Tw is generated and repeated until $\overline{READY}$ is asserted. Data is held on the bus.
3. During Tw $\overline{READY}$ remains asserted and data is written into memory or storage device.
4. The recovery states follow the data state. W/$\overline{R}$, DT/$\overline{R}$, and $\overline{DEN}$ become inactive.

Burst READ and WRITE

This is an enhancement feature of the 80960SA/SB processor. It supports burst transactions that read or write up to eight 16-bit words at a maximum rate of one word per processor cycle.

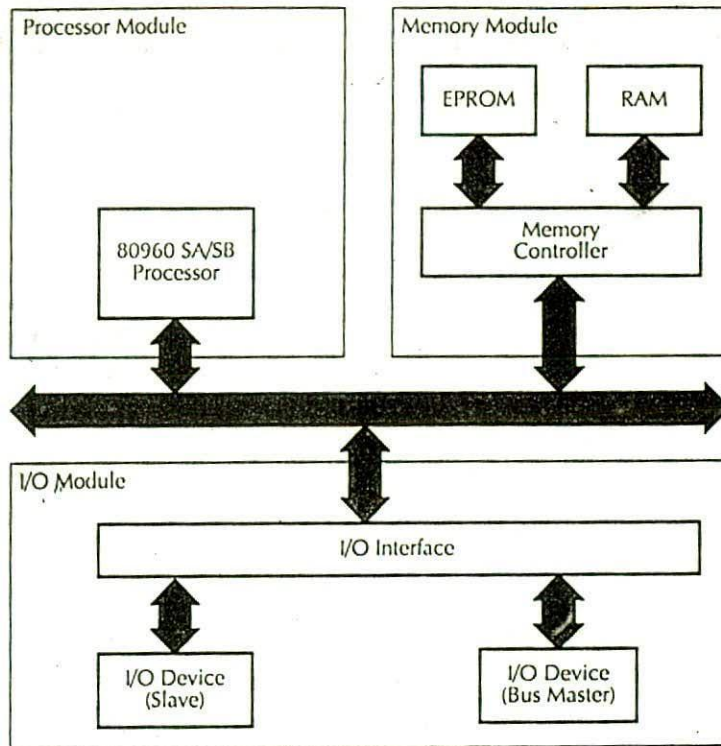


FIGURE 8.4 Basic80960SA/SB system configuration.

8.2.8 80960SA/SB-Based Microcomputer

Figure 8.4 depicts a typical 80960SA/SB system block diagram. The various components are described in the following.

The 80960SA/SB processor performs bus operations using multiplexed address and data signals and provides all the necessary control signals. For example standard control signals, such as Address Latch Enable ($\overline{ALE}$), Address Status ($\overline{AS}$), Write/Read command ($W/\overline{R}$), Data Transmit/Receive ($DT/\overline{R}$), and Data Enable ($\overline{DEN}$) are provided by the 80960SA/SB processor. The 80960SA/SB processor also generates byte enable signals that specify which bytes on the data lines are valid for the transfer.

To transfer control of the bus to an external bus master, the 80960SA/SB processor provides two arbitration signals: hold request (HOLD) and hold acknowledge (HLDA). After receiving HOLD, the processor grants control of the bus to an external bus master by asserting HLDA.

A memory module can consist of the memory controller, Erasable Programmable Read Only Memory (EPROM), and static or dynamic Random Access Memory (RAM). The memory controller first conditions the bus signals for memory operation. It demultiplexes the address and data lines, generates the chip select signals from the address and $\overline{BE}$ signals, detects the start of the cycle for burst mode operation, and latches the byte enable signals.

The memory controller generates the control signals for EPROM, SRAM, and DRAM. In particular, it provides the control signals, multiplexed row/column address, and refresh control for dynamic RAMs. The controller can be designed to accommodate the burst transaction

of the 80960SA/SB processor. In addition to supplying the operation signals, the controller generates the $\overline{READY}$ signal to indicate that data has been transferred to or from the 80960SA/SB processor.

The 80960SA/SB processor directly addresses up to 4G bytes of physical memory.

The I/O module consists of the I/O components and the interface circuit. I/O components can be used to allow the 80960SA/SB processor to use most of its clock cycles for computational and system management activities.

The interface circuit performs several functions. It demultiplexes the address and data lines, generates the chip select signals from the address, produces the I/O read or I/O write command from the processor's $W/\overline{R}$ signal, latches the byte enable signals, and generates the $\overline{READY}$ signal. Because these functions are the same as some of the functions of the memory controller, the same logic can be used for both interfaces.

The 80960SA/SB processor uses memory-mapped addresses to access I/O devices. This allows the CPU to use many of the same instructions to exchange information for both memory and peripheral devices.

Typical I/O chips such as Intel 82C64 timer and Z8536 parallel port/timer can be used with the 80960SA/SB

8.3 Motorola MC88100 RISC Microprocessor

MC88100 is a 32-bit RISC microprocessor designed using HCMOS technology. The 88100 includes the following:

- Hardwired control design with no microcodes
- 20- or 25-Mhz internal clock frequency
- Packaged in 17×17 (180 pins used) PGA (Pin Grid Array) with a maximum size of 1.78″ × 1.78″
- Includes 51 instructions
- Contains four fully parallel on-chip execution units (pipelined)
- Unlike the 80960SA/SB, the 88100 does not support 80-bit extended floating-point format
- Unlike the 80960SA/SB, the 88100 does not include instructions for computing trigonometric and logarithmic functions
- User and supervisor modes
- 32-bit on-chip combinational multiplier
- Separate data and instruction buses that include 32-bit data bus, 32-bit instruction address bus, 32-bit data address bus, and 32-bit instruction bus (fixed instruction length of 32 bits)
- Directly interfaces to memory or to 88200 cache/memory management unit
- 4 gigabytes of directly addressable memory

The 88100 performs register-to-register operations for all data manipulation instructions. Source operands are contained in source registers or are included as an immediate value inherent in the instruction. A separate destination register stores the results of an instruction. This means that source operand registers can be reused in the subsequent instructions. Register contents can be read from or written to memory only with ld (load) and st (store) instructions. A xmem (memory exchange) instruction is included for semaphore testing and multiprocessor application.

The 88100 contains 51 instructions. All instructions are executed in one cycle. The instructions requiring more than one cycle are executed in effectively one cycle via pipelining. All instructions are decoded by hardware and no microcode is used.

The 88100 includes all data manipulation instructions as register-to-register or register plus immediate value instructions. This eliminates memory access delays in data manipulation. Only 10 memory addressing modes are provided: three modes for data memory, four modes for instruction memory, and three modes for registers.

All 88100 instructions are 32 bits wide. This fixed instruction format minimizes instruction decode time and eliminates the need for alignment. All instructions are fetched in a single memory access. The 88100 implements delayed branching to minimize pipeline delay. For pipelined architecture, branching instructions can slow down execution speed due to the time required to flush and refill the pipeline. The 88100 delayed branching feature allows fetching of the next instruction before the branch instruction is executed.

The 88100 provides two modes: supervisor and user. The supervisor mode is used by the operating system, while the application programs are executed in user mode.

The 88100 includes four execution units which operate independently and concurrently. The 88100 can perform up to five operations in parallel.

Scoreboard bits are associated with each of the general-purpose registers. When an instruction is executed or dispatched, the scoreboard bit of the destination register is set, reserving that register for that instruction. Other instructions are executed or dispatched as long as their source and destination operands have clear scoreboard bits. When an instruction completes execution, the scoreboard bit of the destination is cleared, thus freeing that register to be used by other instructions.

The 88100 memory devices can interface directly to memory. Most 88100 designs implement at least two 88200 CMMUs (one for data memory and one for instruction memory). The P-bus provides the interface to the 88200/memory system. The 88200 is an optional external chip that provides paged virtual memory support and data/instruction cache memory.

Conditional test results are provided to any specified, general-purpose register instead of a dedicated condition code register. Conditions are computed at the explicit request of the programmer using compare instructions. This eliminates contention between concurrent execution units accessing a dedicated condition code register.

8.3.1 88100/88200 Interface

Figure 8.5 shows typical 88100 interfaces to several 88200s. The PBUS (processor bus) contains logical addresses, while MBUS (memory bus) contains all physical addresses. Up to 4 88200s can reside on each PBUS. Note that in the figure, the MC88000 includes the entire RISC microprocessor family, with 88100 being the first microprocessor.

Figure 8.6 shows the 88100/88200 block diagram. Each unit in the 88100 can operate independently and simultaneously. Each unit may be pipelined.

The integer unit performs 32-bit arithmetic, logic, bit field, and address operations. All operations are performed in one clock cycle. The integer unit includes 21 control registers. The floating-point unit supports IEEE 754-1985 floating-point arithmetic, integer multiply, and divide. This unit contains 11 control registers with a five-stage add pipeline and a six-stage multiply pipeline. Six optional SFUs (special function units) are reserved in the architecture. The SFUs can be added to or removed from a given system with no impact on the architecture.

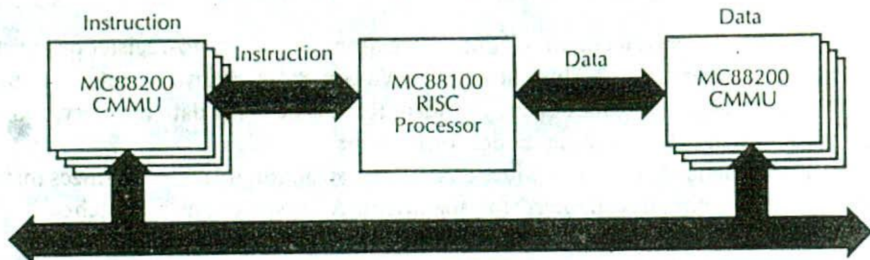
The data unit performs address calculation and data access and includes a three-stage pipeline.

The instruction unit fetches instruction codes and contains a two-stage pipeline.

The register file includes 32 32-bit general-purpose registers.

The sequencer uses a scoreboard to control register reads/writes. It dispatches instructions and recognizes exceptions.

Figure 8.7 shows the 88200 internal block diagram.



MC88100 32-Bit RISC Microprocessor
- 1.5 Micron HCMOS, 180 pins
- Highly pipelined
- Separate instruction and data buses (Harvard architecture)

MC88200 32-Bit Cache/Memory Management Unit
- 1.5 Micron HCMOS, 180 pins
- 56 entry Page Address Traslation Cache (PATC)
- 10 entry Block Address Translation Cache (BATC)
- 16 Kbyte code/data cache

M88000 Processor BUS
- Synchronous, non-multiplexed, pipelined
- 33-bit logical addresses, 32-bit data path
- 1 word each clock cycle, maximum transfer rate

M88000 Memory BUS
- Synchronous, multiplexed
- 32-bit physical addresses, 32-bit data path
- N words each N + 1 clock cycle maximum transfer rate

FIGURE 8.5 Typical 88100 interface to 88200s. Note that MC88000 represents the 88000 family which includes 88100, 88200, and all future products.

The block address translation cache (BATC) contains 10 entries and is fully associative with software replacement.

The page address translation cache (PATC) includes 56 entries and is fully associative with hardware replacement. The SRAM (static RAM) array contains 16K bytes of static RAM and is set associative.

8.3.2 88100 Registers

Figure 8.8 shows the 88100 registers. All registers are 32 bits wide.

Three types of registers are included:

- 32 32-bit registers, r0-r31, containing program data (source operand and instruction results). All of these registers except r0 (constant 0) have read/write access.
- Internal registers control instruction execution and data transfer
- Control registers in the various execution units containing status, execution control, and exception processing information

The internal registers cannot be directly accessible in software, while most control registers can be accessed in supervisor mode. The internal registers can only be modified and used indirectly.

The control registers include shadow registers and exception time registers, integer-unit control registers, and floating-point unit control registers.

The shadow registers are associated with several internal registers. Shadowing is utilized by the 88100 to keep track of the internal pipeline registers at each stage of the instruction execution.

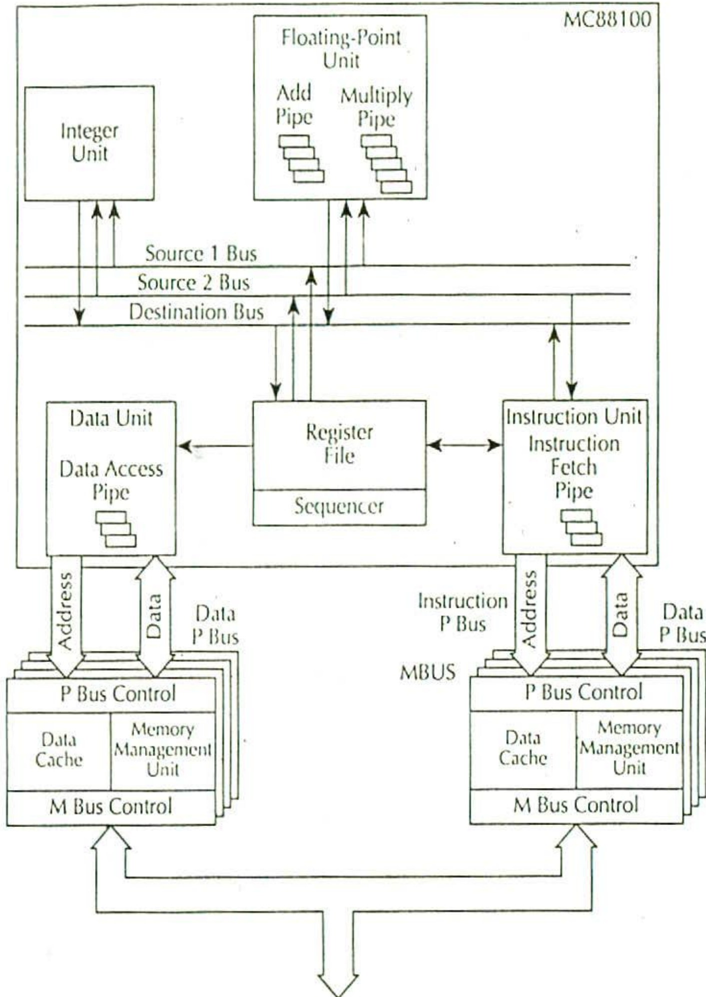


FIGURE 8.6 MC88100/MC88200 block diagram.

There are 21 32-bit control registers (cr0 through cr20) in the integer unit. Fourteen of these registers provide exception information for integer unit or data unit exceptions. The other seven registers include status information, the base address of the exception vector table, and general-purpose storage.

The floating point includes 11 control registers (fcr0-fcr8, fcr62, and fcr63). fcr0 through fcr8 contain exception information such as the exception type, source operands and results, and the instruction in progress. These registers can only be accessed in supervisor mode. Registers fcr62 and fcr63 are not privileged. These two registers can be used to enable user-supplied exception handler software and to report exception causes in user mode.

The supervisor programmer's model contains all general-purpose and control registers. The general-purpose registers provide data and address information, while the control registers provide exception recovery and status information for the integer unit.

In user mode, all general-purpose registers can be accessed. Two control registers (floating-point control and status) can be accessed in the user mode.

Among the 32 general-purpose registers, r0 always contains the value 0, r1 is loaded with the subroutine return address, and r2 through r31 are general-purpose.

Figure 8.9 shows the 88000 register data formats. The 88100 supports two types of data formats, namely, integer (signed or unsigned) and floating-point real numbers. The integers

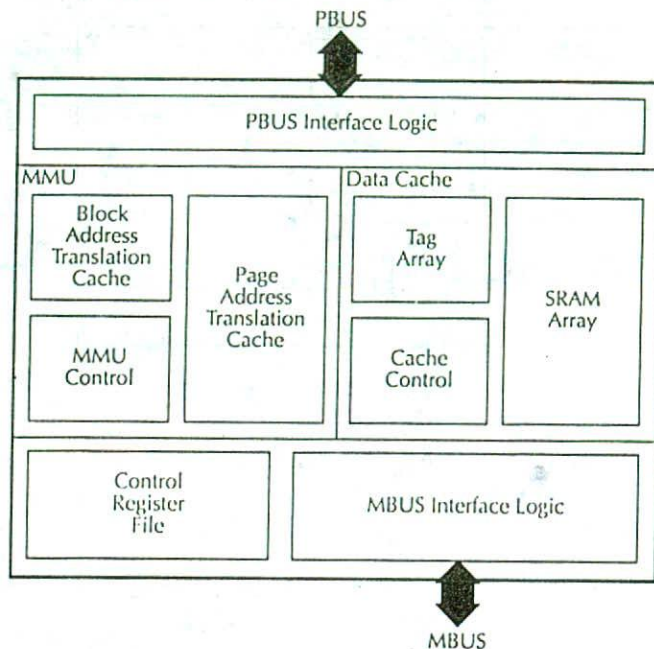


FIGURE 8.7 88200 internal block diagram.

can be byte, half-word (16-bit), and word (32-bit). All operations affect all 32 bits of a general-purpose register. The half-word or byte pads the sign bit.

The floating-point data can be single precision (32-bit) and double precision (64-bit). Figure 8.10 shows formats for Fcr62 and Fcr63.

The reserved bits in fcr62 and fcr03 are always read as zero. The FPcr defines the desired rounding mode and which exceptions are handled by user exception handles. The FPSR indicates which floating-point exceptions have occurred but were not processed by a user exception handler.

The 88100 general-purpose register convention is shown in Figure 8.11. r31 addresses the top of the stack. r30 contains the address of the current data frame in the stack.

Figure 8.12 shows the 88100 stack operation. The SP must always be 32-bit aligned. The stack grows from high memory to low memory addresses.

Next, consider the supervisor programmer model:

- The VBR contains the base address of the exception vector table.
- The 88100 does not automatically use SR0-SR3. They are reserved for operating system use.

Figure 8.13 shows the 88100 processor status register format. In Figure 8.13, Big Endian means the most significant byte at the highest byte address. Serial instruction is to complete before the next one begins. Note that not all adds/subtracts affect C. The SFDI bit enables or disables the floating-point unit. When SFDI = 1, attempted execution of any floating-point or integer multiply/divide instructions causes a floating-point precise exception.

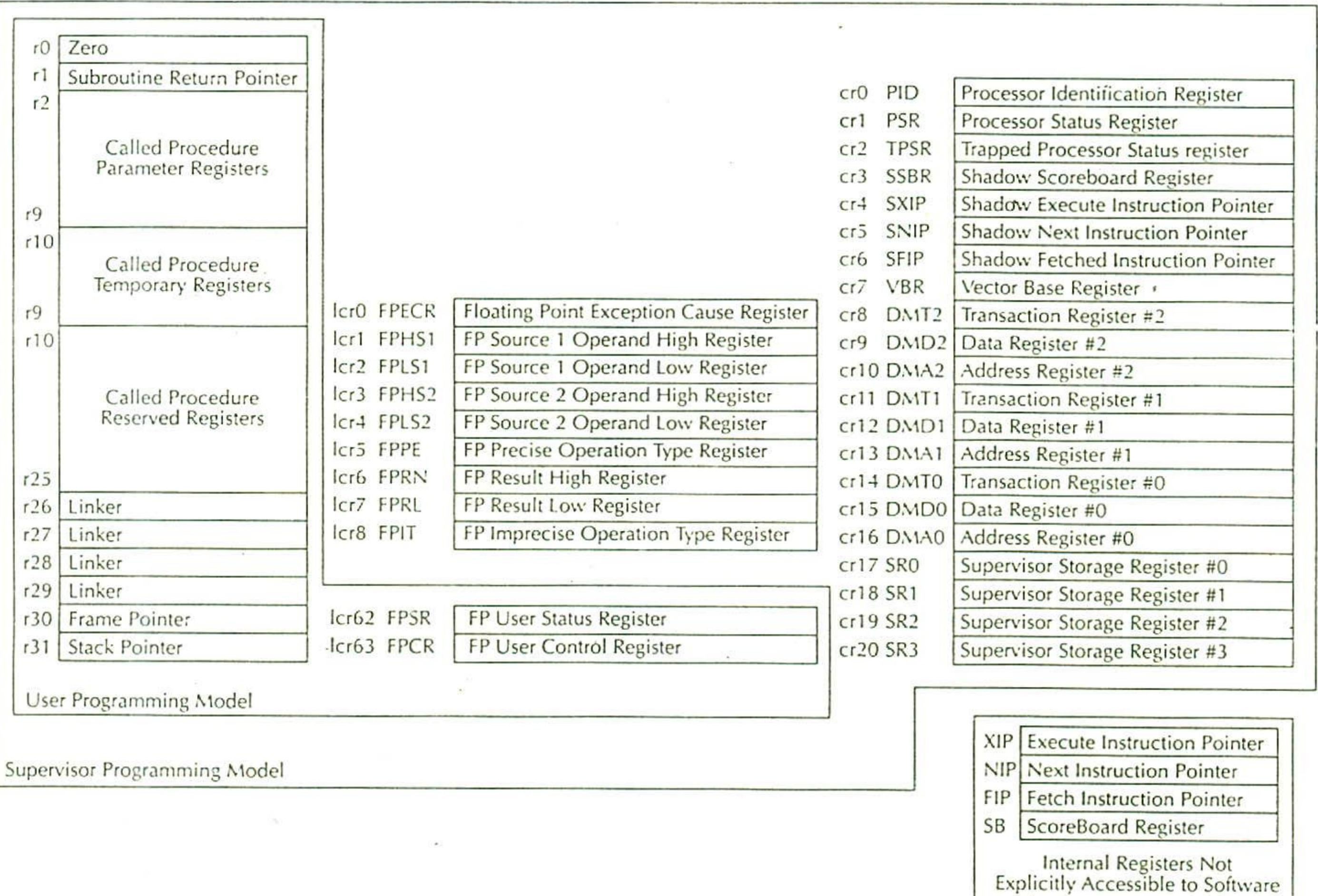


FIGURE 8.8 MC88100 register organization.

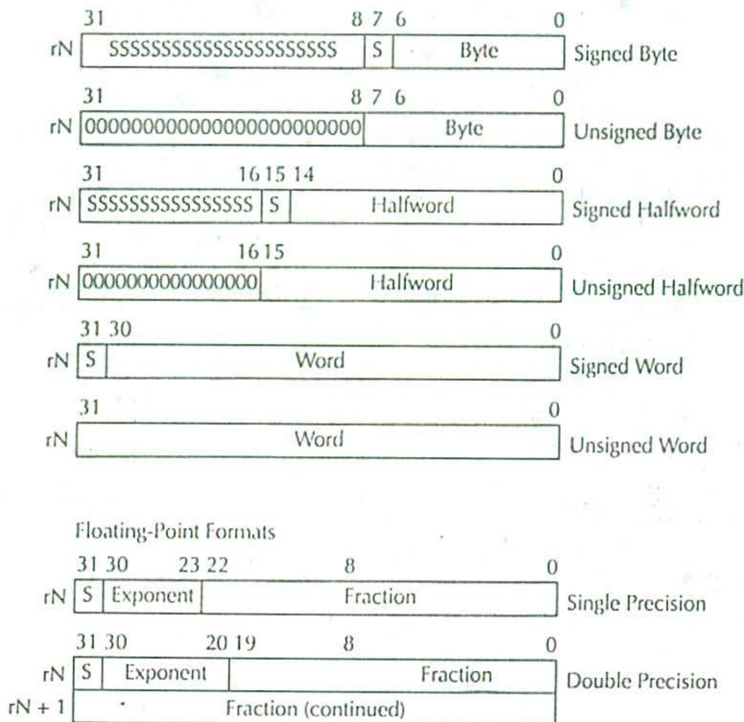
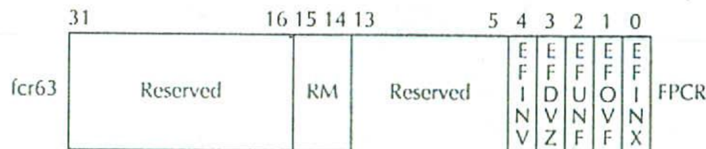


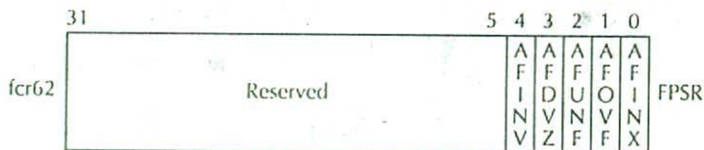
FIGURE 8.9 88100 register data formats. An "S" in the diagram above indicates a sign bit.



RM: Floating Point Rounding Mode

= 00 Round to rearest
= 01 Round to zero
= 10 Round to negative infinity
= 11 Round to positive infinity

EFINV: Enable Invalid Operation User Exception Handler
EFDVZ: Enable Divide By Zero User Exception Handler
EFUNF: Enable Underflow User Exception Handler
EFOVF: Enable Overflow User Exception Handler
EFINX: Enable Inexact User Exception Handler



AFINV: Accumulated Invalid Operation Flag
AFDVF: Accumulated Divide By Zero Flag
AFUNF: Accumulated Underflow Flag
AFOVF: Accumulated Overflow Flag
AFINX: Accumulated Inexact Flag

FIGURE 8.10 88100 fcr 62 and fcr 63 formats.

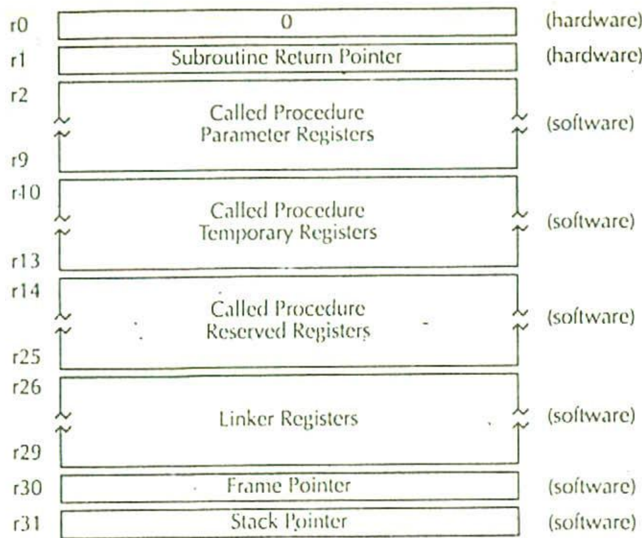


FIGURE 8.11 88100 register convention.

8.3.3 88100 Data Types, Addressing Modes, and Instructions

Tables 8.7a and 8.7b list the data types and addressing modes supported by the 88100. Table 8.8 summarizes the 88100 instructions.

The 51 instructions listed in Table 8.8 of the MC88100 can be divided into 6 classes: integer arithmetic, floating-point arithmetic, logical, bit field, load/store/exchange, and flow control.

These simple instructions must be used to obtain complex operations. Shift and rotate operations are special cases of bit field instructions. Only ld, st, and xmem can access memory.

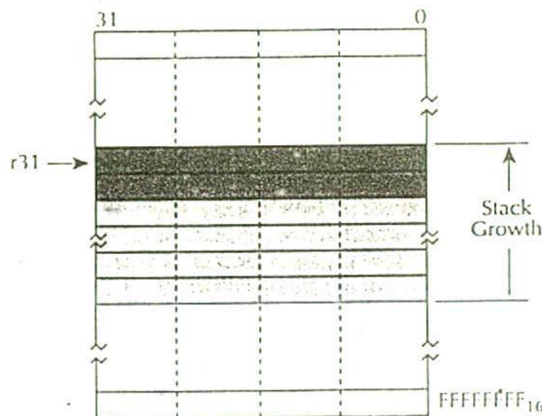


FIGURE 8.12 88100 stack operation.

MODE = 0 Processor is in user mode.
= 1 Processor is in supervisor mode.

BO = 0 Big Endian byte order in memory.
= 1 Little Endian byte order in memory.

SER = 0 Concurrent instruction execution.
= 1 Serial instruction execution.

C = 0 No carry / borrow generated.
= 1 Carry / borrow generated.

DEXC = 0 Data memory exception not pending.
= 1 Data memory exception pending.

SFD1 = 0 SFU1 enabled.
= 1 SFU1 disabled.

* MXM = 0 Misaligned memory accesses generate exceptions.
= 1 Misaligned memory access truncate.
Misaligned address to next lower aligned address.

IND = 0 Interrupt enabled.
= 1 Interrupt disabled.

SFRZ = 0 Shadow registers enabled.
= 1 Shadow registers frozen.

* 88100 can address each byte. For halfword, aligned address include all addresses in multiples of 2, for word in multiples of 4 and for doubleword in multiple of 8.

FIGURE 8.13 88100 processor status register format.

TABLE 8.7a Data Types

Data type	Represented as
Bit fields	Signed and unsigned bit fields from 1 to 32 bits
Integer	Signed and unsigned byte (8 bits)
	Signed and unsigned half-word (16 bits)
	Signed and unsigned word (32 bits)
Floating-point	IEEE P754 single precision (32 bits)
	IEEE P754 double precision (64 bits)

TABLE 8.7b Addressing Modes

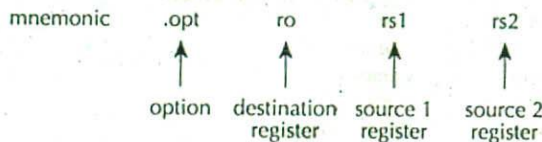
Data addressing mode	Syntax
Register indirect with unsigned immediate	rD,rS1,imm16
Register indirect with index	rD,rS1,rS2
Register indirect with scaled index	rD,rS1[rS2]
Instruction addressing mode	**Syntax**
Register with 9-bit vector number	m5,rS1,vec9
Register with 16-bit signed displacement	m5,rS1,d16
Instruction pointer relative (26-bit signed displacement)	d26
Register direct	rS2

TABLE 8.8 Instruction Set Summary

Mnemonic	Description
	Integer Arithmetic Instructions
add	Add
addu	Add unsigned
cmp	Compare
div	Divide
divu	Divide unsigned
mul	Multiply
sub	Subtract
subu	Subtract unsigned
	Floating-Point Arithmetic Instructions
fadd	Floating-point add
fcmp	Floating-point compare
fdiv	Floating-point divide
fldcr	Load from floating-point control register
flt	Convert integer to floating point
fmul	Floating-point multiply
fstcr	Store to floating-point control register
fsub	Floating-point subtract
fxcr	Exchange floating-point control register
int	Round floating point to integer
nint	Floating-point round to nearest integer
trnc	Truncate floating point to integer
	Logical Instructions
and	AND
mask	Logical mask immediate
or	OR
xor	Exclusive OR
	Bit-Field Instructions
cir	Clear bit field
ext	Extract signed bit field
extu	Extract unsigned bit field
ff0	Find first bit clear
ff1	Find first bit set
mak	Make bit field
rot	Rotate register
set	Set bit field
	Load/Store/Exchange Instructions
ld	Load register from memory
lda	Load address
ldcr	Load from control register
st	Store register to memory
stcr	Store to control register
xcr	Exchange control register
xmem	Exchange register with memory
	Flow Control Instructions
bb0	Branch on bit clear
bb1	Branch on bit set
bcnd	Conditional branch
br	Unconditional branch
bsr	Branch to subroutine
jmp	Unconditional jump
jsr	Jump to subroutine
rte	Return from exception
tb0	Trap on bit clear
tb1	Trap on bit set
tbnd	Trap on bounds check
tcnd	Conditional trap

Also, only compare instructions affect condition codes. Most MC88100 instructions can have one of the three formats:

1. Triadic register instructions (three operands). The general format is



   An example is add.ci r2, r7, r4. Note that Motorola's assembler expects the 88100 instructions in lower case. In the example, the mnemonic is add and the option is ci, meaning use 'carry in' in the operation. The source registers r7 and r4 remain unchanged unless one of them is used as destination. This add instruction adds [r7] and [r4] with carry and stores the result in r2.
2. Triadic register instructions with 16-bit field instruction. The general format is mnemonic.opt rD, rs1, imm16. Consider

```
add.Co r2, r4, 0XA125
```

   0X before data A125 means that A125 is in hex. This notation is used by the Motorola assembler.
3. Dyadic register instructions (two operands). The general format is mnemonic . opt rd, rs2. An example is flt.sw r5, r6 — which converts the integer source word in r6 into floating-point in r5.

Table 8.9 lists the 88100 load, store, and exchange instructions.

TABLE 8.9 MC88100 LOAD, STORE, Exchange Instructions

Instructions		Exceptions
ld[.<opts>]	rD,rS1,<imm16>	
ld[.<opts>][.<space>]	rD,rS1,rS2	
ld[.<opts>][.<space>]	rD,RS1 [rS2]	
st[.<size>]	rS,rS1,<imm16>	Data access
st[.<size>][.<space>]	rS,rS1,rS2	Misaligned access
st[.<size>][.<space>]	rS,rS1[rS2]	Privilege violation
xmem[.bu]	rS,rS1,<imm16>	
xmem[.bu][.<space>]	rS,rS1,rS2	
xmem[.bu][.<space>]	rS,rS1[rS2]	
ldcr	rD,crCRS	
stcr	rS,crCRD	
xcr	rD,rS1,crCRS/D	
		Privilege violation
fldcr	rD,crFCRS	
fstcr	rS,crFCRD	
fxcr	rD,rS1,crFCRS/D	

<opts>	For	ld	<size>	For	st
.b	—	Signed byte	.b	—	Byte
.bu	—	Unsigned byte	.h	—	Halfword
.h	—	Signed halfword	none	—	Word
.hu	—	Unsigned halfword	.d	—	Doubleword
none	—	Word			
.d	—	Doubleword			

TABLE 8.9 MC88100 LOAD, STORE, Exchange Instructions (*continued*)

<space>	For	ld,st,xmem
.usr	—	Access user space regardless of mode bit in PSR
none	—	Access space indicated by PSR MODE bit

- rs = Source register.
- crCRS = Source control register.
- crCRD = Destination control register.
- crCRS/D = Source/destination control register.
- crFCRS = Source floating-point control register.
- crFCRD = Destination floating-point control register.
- crFCRS/D = Source/destination floating-point control register.
- Memory accesses for xmem are indivisible.

- ld loads a general-purpose register from data memory. There are three operands with this instruction. Two source operands are used to calculate the address. Three forms of ld use the three addressing modes available. The (.opt) for ld specifies the size of data read from memory.
- The st instructions are similar to ld instructions, except they are used to store source data.
- The exchange instructions (xmem, xcr, fxcr) swap the contents of a general-purpose register with data memory or with a control register.
- Consider ld r47, r31, 0X4. The mode used here is register indirect with unsigned immediate. If [r31] = 00005000_{16}, the effective address is computed by adding rsl (r31) with unsigned 16-bit immediate data. Therefore, the 88100 loads the register r47 with 32-bit data from a memory location addressed by 00005004_{16}; r31 is the SP. Therefore, the access occurs within the stack. Since the immediate data are unsigned, the accessed address cannot be less than r31. This means that the stack grows toward the lower address.
- Consider st.b rs, rs1, rs2. This instruction has register indirect with index mode.

The access address is (rs1) + (rs2) where rsl is the base register and rs2 is the index register. For example, consider st.b r1, r0, r5. If [r5] = 00010200_{16}; then since r0 is always 0, the low 8-bit content of r1 is stored at address 00010200_{16}. The 88100 ignores any carry generated during address calculation. Note that in the above r0 is the base address and r5 is the index register.

Finally, consider st.hu rs, rs1 [rs2]. The mode is register indirect with index. The access address is (rs1) + (rs2)* (operand size). The scaling is specified by surrounding the index register rs2 by square brackets. Operand size is 1 for byte, 2 for halfword, 4 for word, and 8 for double word.

As an example, consider st. r5, r31 [r1]. If [r1] = 00000003_{16}, [r31] = 00005000_{16}, then the effective address is

$$00005000 + 4 * 00000003 = 0000500C_{16}$$

scaled by 4 for word since the instruction without any option specified means 32-bit word.

Therefore, the above store instruction stores the 32-bit contents of r5 into a memory location addressed by $0000500C_{16}$.

Table 8.10 shows the integer arithmetic instructions.

TABLE 8.10 MC88100 Integer Arithmetic Instructions

Instructions		Exceptions
add[.<opt>]	rD,rS1,rS2	Integer overflow
add	rD,rS1,<imm16>	
addu[.<opt>]	rD,rS1,rS2	None
addu	rD,rS1,<imm16>	
sub[.<opt>]	rD,rS1,rS2	Integer overflow
sub	rD,rS1,<imm16>	
subu[.<opt>]	rD,rS1,rS2	None
subu	rD,rS1,<imm16>	
mul	rD,rS1,rS2	None
mul	rD,rS1,<imm16>	
div	rD,rS1,rS2	
div	rD,rS1,<imm16>	Integer divide
divu	rD,rS1,rS2	
divu	rD,rS1,<imm16>	
lda [.<size>]	rD,rS1,<imm16>	
lda [.<size>]	rD,rS1,rS2	None
lda [.<size>]	rD,rS1,[rS2]	

<opt> FOR add/addu/sub/subu	
none	No carry
.ci	Use carry in
.co	Propagate carry out
.cio	Use carry in and propagate carry out

<size> FOR lda	
.b	Scale rS2 by 1
.h	Scale rS2 by 2
none	Scale rS2 by 4
.d	Scale rS2 by 8

- mul yields correct signed and unsigned results.
- Division by zero signals the integer divide exception.
- An integer divide exception occurs when either source operand is negative for div.
- Unscaled lda is functionally equivalent to addu.

- Consider add [.<opt>] rd, rs1, rs2. Three options can be used with this instruction as follows:
  - add.Ci rD, rs1, rs2 adds the 32-bit contents of rs1 with rs2 and the C-bit in processor status register, and stores the 32-bit result in rD without providing any carry-out, and thus the C-bit in PSR is unchanged.
  - add.Co rD, rs1, rs2 adds the 32-bit contents of rs1 with rs2 without any carry-in and stores the result in rD and reflects the carry-out in the C-bit of PSR.
  - add.Cio rD, rs1, rs2 adds the 32-bit contents of rs1 with rs2 and the C-bit from the PSR and stores the result in rD and reflects any carry-out in the C-bit in the PSR.
- Consider add.Cio r2, r1, r5. If the C-bit in the PSR is 0, [r1] = 8000 F102$_{16}$, [r5] = F1101100$_{16}$, then after this add [r2] = 71110202$_{16}$ and the C-bit in the PSR is set to one.
- If no option is specified in an instruction such as add or addu, the carry-in is not included in addition and also no carry-out from the addition is provided. For example, consider addu r1, r5, 0XF112. The 16-bit immediate data F112$_{16}$ is converted to an unsigned 32-bit number 0000-F112$_{16}$ and is added with the 32-bit contents of r5. The

32-bit result is stored in r1. The carry-out is not provided. The add instruction performs signed arithmetic. If the result cannot be accommodated in a 32-bit integer, an integer overflow exception occurs. The immediate 16-bit data in an add instruction are sign-extended to 32 bits before addition.

- The sub instructions are similar to the add instructions. The content of rs2 is subtracted from the content of rs1 with the C-bit in the PSR as borrow if .Ci is used for <opt>.
- mul instructions multiply a 32-bit number (signed or unsigned) in rs2 by a 32-bit number (signed or unsigned) in rs1 and store the low 32-bit result in rD and discard the upper 32 bits of the result.
- div instructions perform signed division while divu carries out unsigned division. divu (unsigned division) instructions divide the 32-bit content of rs1 by the 32-bit content of rs2 or a 16-bit immediate value. The 32-bit quotient is stored in rD and the remainder is discarded. If the divisor is zero, an integer divide exception is taken and rD is unaffected. div (signed division) operates similarly except that the integer divide exception is taken if either the dividend (rs1) or the divisor (rs2) has a negative value. The exception handler must convert the negative value to positive, perform the signed integer divide, and convert the sign of the result.
- lda is the load address instruction. lda calculates the access address using one of three indirect addressing modes. lda loads rD with the access address. Unscaled lda is functionally equivalent to addu with the same operands.

Table 8.11 lists the 88100 floating-point arithmetic instructions.

- The MC88100 permits a mixture of single and double precision source and destination operands.
- trnc performs "round to zero" rounding.
- nint performs "round to nearest" rounding.
- int performs rounding specified by the RM field of the FPCR.
- fadd adds the contents of rs1 with rs2 and stores the result in rD.
- fsub subtracts the contents of rs2 from rs1 and stores the result in rD.
- fmul multiplies the contents of rs1 by rs2 and stores the result in rD.
- fdiv divides the contents of rs1 by rs2 and stores the quotient in rD.

TABLE 8.11 MC88100 Floating-Point Arithmetic Instructions

Instructions		Exceptions
fadd.<sizes>	rD,rS1,rS2	Floating point reserved operand
fsub.<sizes>	rD,rS1,rS2	Floating point overflow
		Floating point underflow
fmul.<sizes>	rD,rS1,rS2	Floating point inexact
fdiv.<sizes>	rD,rS1,rS2	Floating point divide by zero
trnc.<sizes>	rD,rS2	
nint.<sizes>	rD,rS2	Floating point integer conversion overflow
int.<sizes>	rD,rS2	Floating point reserved operand
flt.<sizes>	rD,rS2	Floating point inexact

<sizes> FOR fadd/fsub/fmul/fdiv
sss, ssd, sds, sdd, dss, dsd, dds, ddd
<sizes> FOR trnc/nint/int
ss, sd
<sizes> FOR flt
ss, ds

TABLE 8.12 MC88100 Logical Instructions

Instructions		Exceptions
and [.c]	rD,rS1,rS2	
and [.u]	rD,rS1,<imm16>	
mask [.u]	rD,rS1,<imm16>	
or [.c]	rD,rS1,rS2	None
or [.u]	rD,rS1,<imm16>	
xor [.c]	rD,rS1,rS2	
xor [.u]	rD,rS1,<imm16>	

• Option .c ones-complements the contents of rs2 before performing the operation.
• Option .u performs the specified logical operation between 16-bit immediate data <imm16> and the high 16 bits of rs1.

- The 88100 utilizes hardware to perform these IEEE floating-point computations.
- The 88100 allows single and double precision operands. <sizes> specify the operand sizes of rD, rs1, and rs2 as single or double precision. For example, .ssd means that rD and rs1 are single precision, while rs2 is double precision.
- trnc, nint, int, and flt provide conversions between integer and floating-point values. These instructions have two operands with one operand having a floating point value and the other having an integer value. trnc, nint, and int convert a floating-point format to equivalent format. The difference between them is the type of rounding performed. trnc rounds toward zero, and nint rounds to the nearest value, int rounds as specified by the RM field in FPCR.
- Two exceptions are provided for floating-point instructions. An integer conversion overflow exception occurs when the operand value cannot be expressed as a high word. A reserved operand exception occurs with certain floating-point values.
- flt converts a signed 32-bit number into a floating-point format. The integer operand size is always specified by <sizes> indicating signed word size. A 'd' for double or 's' for single defines the floating-point operand's precision.

Table 8.12 lists the MC88100 logical instructions.

If the option .u is omitted, the specified logical operation is performed between the 16-bit immediate data <imm16> and the low-order 16 bits of rs1.

- The mask always affects all 32 bits of rD. The mask logically ANDS the 16-bit immediate value with the low 16 bits or highest 16 bits of rs1 and clears the other 16 bits to zero. If the .u option is omitted, the AND is performed with the low-order 16 bits of rs1 and if .u is included, the AND is performed with the high-order 16 bits of rs1.
- When both operands are registers (rs1 and rs2) for AND, OR, and XOR, the 32-bit logical operation is performed. When the .c option is used, the MC88100 ones-complements the contents of rs2 before performing the operation. The MC88100 only performs a 16-bit operation when the second source operand is a 16-bit immediate. The .u option, when present in AND, OR, and XOR, performs the logical operation between the high 16-bit value of rs1 and the 16-bit immediate data and then stores the result in the high 16 bits of rD. The low 16 bits of rs1 are copied into the low 16 bits of rD. If the .u option is not present, only the low 16 bits of rs1 are used in the operation.

Table 8.13 lists the 88100 bit field instructions.

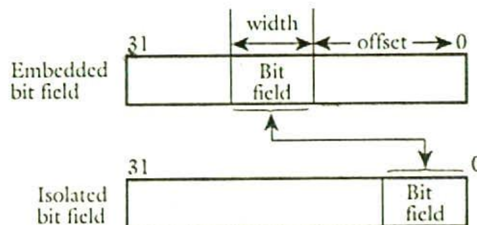
- W5 is five bit width and <05> is a five bit offset.
- The number of bits in a bit field is called width. Width can be from 1 to 32. The least significant bit in bit field is called the offset. When the sum of width and offset is greater than 32, the bit field may be imagined to extend beyond the most significant bit of the register.

TABLE 8.13 MC88100 Bit Field Instructions

Instructions		Exceptions
clr	rD,rS1,w5 <05>	None
clr	rD,rS1,rS2	
set	rD,rS1,w5 <05>	
set	rD,rS1,rS2	
ext	rD,rS1,w5 <05>	
ext	rD,rS1,rS2	
extu	rD,rS1,w5 <05>	
extu	rD,rS1,rS2	
mak	rD,rS1w5 <05>	
mak	rD,rS1,rS2	
rot	rD,rS1, <05>	
rot	rD,rS1,rS2	
ff1	rD,rS2	
ff0	rD,rS2	

- ext and extu affect all 32 bits of rD.
- ext and extu perform shift right operations when the width equals 32.
- mak clears rD before inserting the bit field.
- mak performs shift left operations when the width equals 32.
- rot rotates the bit field to the right.

An isolated bit field ends in the least significant bit of a register with an implicit offset of zero. Bit fields may contain signed and unsigned values. For unsigned isolated bit fields, the high-order bits in a register are all zero and for signed isolated bit field, they are the 2's complement sign bit. In either case, the entire register contains the word value equivalent to the isolated bit field value. The 88100 includes instructions to isolate embedded bit field for arithmetic manipulation. The 88100 instruction moves an isolated bit field back into an embedded bit field. This is illustrated in the following:



Two bit field formats are used. These are literal width with offset and register width with offset. The literal width with offset uses immediate width and offset values. An example is set r5, r1, 3 <7>. The destination register is r5; the source bit field is 3 bits wide with an offset of 7 in r1. With the Motorola assembler, the offset must be included in angle brackets < >.

The register width with offset uses three operands. An example is clr r1, r3, r4. The source bit field is in r3 with offset and width determined from r4. The MC88100 obtains the offset from bits 0-4 of r4 and the width from bits 5-9 of r4. The upper 22 bits of r4 are don't cares. Both formats use an offset of 0 to 31 and width of 1 to 32 with 32 encoded as 0. The destination register (r1 in this case) stores the final result. The content of the source register (r3) does not change after the operation.

ext (signed) and extu (unsigned) instructions extract the register value from rs1 and convert it to an isolated bit field in rD. For a bit field width of 32 (encoded as 0), ext and extu perform a shift right operation. The content of rs1 is shifted to the right by the

TABLE 8.14 MC88100 Integer Compare

Instructions	Exceptions
cmp	rD,rS1,<imm16>
cmp	rD,rS1,rS2
cmp	

cmp Predicate Bit String

rD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

11	10	9	8	7	6	5	4	3	2	1	0
hs	lo	ls	hi	ge	lt	le	gt	ne	eq	0	0

Predicate Bits	Bit Set If And Only If:	
eq	rS1 == rS2	
ne	rS1 ≠ rS2	
gt	rS1 > rS2	Signed Evaluation
le	rS1 <= rS2	Signed Evaluation
lt	rS1 < rS2	Signed Evaluation
ge	rS1 >= rS2	Signed Evaluation
hi	rS1 > rS2	Unsigned Evaluation
ls	rS1 <= rS2	Unsigned Evaluation
lo	rS1 < rS2	Unsigned Evaluation
hs	rS1 >= rS2	Unsigned Evaluation

number of bits specified in the offset and the result is stored in rD. extu performs a logical shift, while ext performs an arithmetic shift.

- mak creates an imbedded bit field in rD with an offset specified by an immediate value or by the content of rs2. The MC88100 stores the least significant bits of rs1 in the imbedded bit field. The bits outside the imbedded bit field in rD are cleared to zero. The mak is the inverse operation of ext and extu.
- The shift left operation may be performed by a bit field width of 32. The offset specifies the number of positions to be shifted.
- rot reads rs1 and rotates it to the right by the number of bits specified in <05> or in bits 0-4 of rs2. The result is stored in rD.
- ff1 finds the most significant set bit in rs2 and stores the bit number in rD.
- If all bits are cleared, the 88100 loads 32 into rD. ff0 operates similarly but finds the most significant clear bit.

Table 8.14 summarizes the 88100 Integer Compare instructions.

- cmp provides integer data comparison. The 88100 compares the rs1 contents with either an unsigned 16-bit immediate number or the content of rs2. The 16-bit immediate data are converted to a 32-bit value with zeros in the high 16 bits before comparison. The result of the comparison is stored in rD.

Table 8.15 shows the 88100 floating-point compare instructions. Table 8.16 lists the 88100 conditional branch instructions.

An instruction using the .n option must not be followed by another flow control instruction. (Error undetected by the MC88100.)

<cond> for bcnd/tcnd

eq0
ne0
gt0
lt0
ge0
le0

TABLE 8.15 MC88100 Floating-Point Compare

Instructions	Exceptions
fcmp.sss rD,rS1,rS2	Floating point reserved operand
fcmp.ssd rD,rS1,rS2	
fcmp.sds rD,rS1,rS2	
fcmp.sdd rD,rS1,rS2	

fcmp Predicate Bit String

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rD	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	ob	in	ib	ou	ge	lt	le	gt	ne	eq	cp	nc

Predicate Bits	Bit Set If And Only If:
nc	operands are not comparable
cp	operands are comparable
eq	rS1 == rS2
ne	rS1 ≠ rS2
gt	rS1 > rS2
le	rS1 <= rS2
lt	rS1 < rS2
ge	rS1 >= rS2
ou	(rS1 > rS2 OR rS1 < 0) AND rS2 > 0
ib	rS1 <= rS2 AND rS1 >= 0 and rS2 > 0
in	rS1 < rS2 AND rS1 > 0 AND rS2 > 0
ob	(rS1 >= rS2 OR rS1 <= 0) AND rS2 > 0

- <d16> ‖ signed 16-bit displacement.
- <vec9> ‖ 9-bit vector number (0-511).
- <b5> ‖ 5-bit bit number (0-31).
- The .n option indicates "execute next". The next instruction executes whether or not the branch takes effect.
- tbnd traps if the value in rS1 is greater than the value in rS2 or <imm16>, or if rS1 is negative.
- bcnd tests the content of rs1 for = 0, ≠ 0, >0, <0, ≤0, and ≥0 and branches with 16-bit signed displacement if the condition is true. The 16-bit signed displacement is sign-extended to 32 bits, shifted twice to the left, and adds to address of bcnd to branch with a displacement of (2^{-18}) to ($2^{18} - 4$) bytes. .n indicates 'execute next'. If .n is present and the condition is true, the bcnd executes the next instruction before taking the branch. The 'execute next' allows the 88100 to branch without flushing the execution pipeline and thus provides faster execution.

TABLE 8.16 MC88100 Conditional Flow Control Instructions

Instructions		Exceptions
bcnd [.n]	<cond>,rS1,<d16>	None Trap vec9 Privilege violation
bb1 [.n]	<b5>,rS1,<d16>	None
bb0 [.n]	<b5>,rS1,<d16>	
tb1	<b5>,rS1,<vec9>	Trap vec9
tb0	<b5>,rS1,<vec9>	Privilege violation
tbnd	rS1,rS2	
tbnd	rS1,<imm16>	

TABLE 8.17 MC88100 Unconditional Flow Control Instructions

Instructions		Exceptions
br [.n]	<d26>	
bsr [.n]	<d26>	None
jmp [.n]	rS2	
jsr [.n]	rS2	
rte		Privilege violation

- <d26> || signed 26-bit displacement.
- .n || "execute next".
- bsr and jsr save the return address in r1.
- "jmp rl" performs return from subroutine.
- The last instruction of typical exception handlers is rte.

- tcnd also tests the content of rs1 for the condition but traps if the condition is true. The 88100 includes 512 vectors in the vector table. <vec9> specifies a 9-bit vector number from 0 to 511. .bb1 (branch on bit set) tests the content of rs1 for a set bit. If it is set, the 88100 takes a branch. <b5> specifies a bit number from 0 to 31. bb1 usually follows cmp and fcmp instructions. tbb1 is similar to bb1 except a trap is taken if the bit is set. bb0 (branch on bit clear) is similar to bb1 except a branch is taken if the specified bit is 0. tb0 is similar to tb1 except a trap is taken if the specified bit is 0.
- tbnd (trap on bound check) generates bound check violation if rs1 contents are out of bounds; 0 is the implicit lower bound. The upper bound is either unsigned 16 bits or is contained in rs2. The value of the upper bound is treated as an unsigned number.

Table 8.17 lists the unconditional jump instructions.

- br always unconditionally branches with signed 26-bit displacement with a range of (2^{-26}) to ($2^{+26} - 4$) bytes.
- bsr is an unconditional subroutine call and saves the return address in r1. When [.n] is specified, the return address is the address of bsr plus 8.
- jmp branches to the address specified by the contents of rs2. The 88100 rounds the least 2 bits of rs2 to 00 before branching for alignment. However, the contents of rs2 are unchanged by the instruction. .jsr is similar to jmp except it is a subroutine jump to an address specified by the contents of rs2 and also saves the return address in r1.
- The 88100 does not provide any return from subroutine instruction. jump r1 fetches the next instruction from the return address saved by bsr or jsr.
- rte provides an orderly termination of an exception handler. It uses the shadow registers to restore the state that existed before the exception. rte can only clear the mode bit in the PSR and ensures that the instruction is executed in user mode.

Example 8.12

Show the contents of registers and memory after the 88100 executes the following instructions:

i) st.h r2, r3, r5
ii) xmem r2, r2[r3]
Assume the following data:

Memory

$[r2] = 00000004_{16}$

$[r3] = 00002000_{16}$

$[r5] = 00000002_{16}$

Address				
2000	FF	25	71	02
2004	00	01	02	03
2008	A2	71	36	25
8000	01	02	A2	71
8004	B1	11	26	05

Solution

i) $[r3] + [r5] = 2002_{16}$. Consider st.h r2, r3, r5 where .h stands for half-word (16 bits). The low 16 bits of r2 is stored in 2002_{16} and 2003_{16}. Therefore

2000	FF	25	00	04

ii) xmem r2, r2[r3]. This is a 32-bit word operation. Hence, the scale factor is 4. The effective address

```
= [r2] + 4 * [r3]
= 0000 0004₁₆ + 4 * 00002000₁₆
= 0000 8004₁₆
```

Therefore, after the xmem, $[r2] = B111\ 2605_{16}$; and $[00008004] = 0000\ 0004_{16}$.

Example 8.13

Write an instruction to logically shift right by 3 bits the value of r2 into r5.

Solution

Since r2 is 32 bits wide, extu performs logical right shift and the width 32 is encoded as 0. extu r5, r2, 0<3>.

Example 8.14

Write an MC88100 instruction sequence to logically AND the 32-bit contents of r7 with $F2710562_{16}$ and store the result in r5.

Solution

The instruction "and" logically ANDs 16-bit operands.

```
and r5, r7, 0x0562     ;  and logically ANDs 0562₁₆
                       ;  with low 16 bits of
                       ;  r7 and stores result in
                       ;  low 16 bits of r5
and.u r5, r7, 0xF271   ;  and.u logically ANDs
                       ;  F271₁₆ with high 16 bits
                       ;  of r7 and stores result
                       ;  in high 16 bits of r5
```

Example 8.15

Find an MC88100 instruction to load register r5 with the constant value 9.

Solution

add r5, r0, 9.

Example 8.16

Find an MC88100 instruction to branch to the instruction with label START if the value in r5 is equal to zero.

Solution

bcnd eq0, r5, START.

Example 8.17

Write an 88100 assembly language program to add two 64-bit numbers in registers r2 r3 and r4 r5. Store result in r4 r5.

Solution

```
        add r5, r3, r5      ; add low 32-bits
        add.cio r4, r2, r4  ; add next 32 bits with carry
finish  br finish
```

Example 8.18

Write an 88100 assembly language program to perform the following operation:

$$(A/B) + C^* D$$

where A, B, C, D are stored in r2, r3, r4, r5 as signed 32-bit integers. Assume C*D generates a 32-bit product. Discard the remainder of A/B. Store the 32-bit result in r6.

Solution

```
        div r8, r2, r3      ; r8 ← r2/r3
        mul r7, r4, r5      ; r7 ← r4*r5
        add.co r6, r7, r8   ; store result in r6
finish  br finish           ; halt
```

Example 8.19

Write a program in 88100 assembly language to compute the area of a circle by using $A = \pi r^2$ where A is the area in 32-bit single-precision to be stored in register r4 and r is the radius of the circle stored in r2 as a 32-bit floating-point number.

Solution

```
        fmul.sss r4, r2, r2 ; calculate r²
        add r5, r0, 7       ; move 7 into r5
```

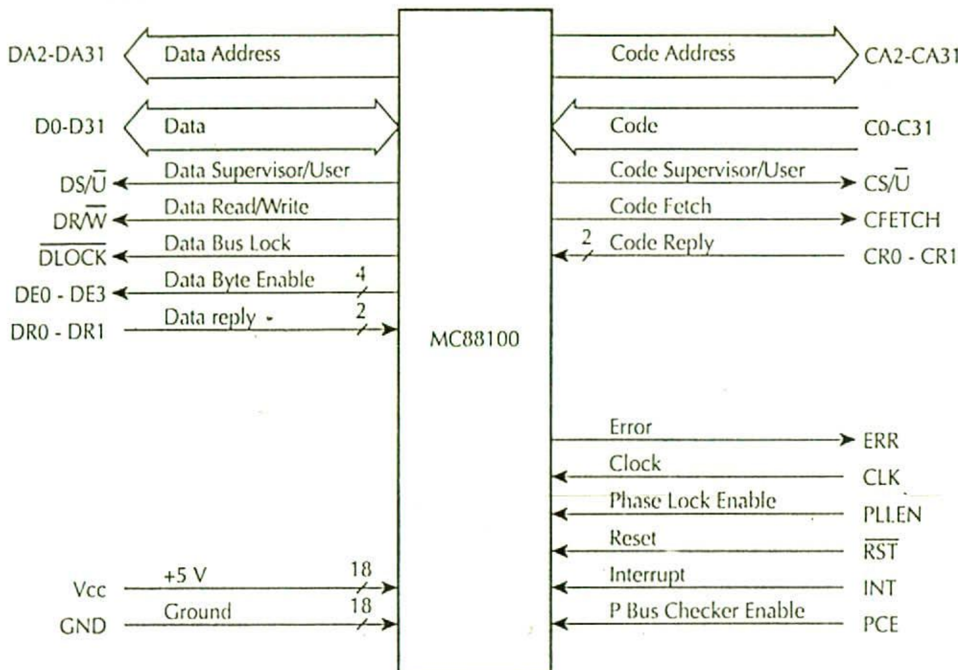


FIGURE 8.14 MC88100 signal functional diagram.

```
         flt.ss r6, r5          ; convert 22 to floating-point
         fdiv.sss r8, r4, r5    ; compute r²/7
         add r7, r0, 22         ; move 22 into r7
         flt.ss r9, r7          ; convert 22 into floating-point
         fmul.sss r4, r8, r9    ; compute πr²
finish   br finish              ; halt
```

8.3.4 88100 Pins and Signals

Figure 8.14 shows the 88100 pins by functional group. Table 8.18 provides a brief description of the functions of these pins.

TABLE 8.18 Signal Index

Signal name	Mnemonic	Function
Data address bus	DA2-DA31	Provides the 30-bit word address to the data memory space; an entire data word (32 bits) is always addressed; individual bytes or half words are selected using the data byte strobe signals
Data bus	D0-D31	32-bit bidirectional data bus interfacing the MC88100 to the data memory space
Data supervisor/user	DS/$\overline{U}$	This signal selects between the supervisor select data address space and the user data address space; DS/$\overline{U}$ is determined by the value of the MODE bit in the processor status register, or by the .usr option of the ld and st instructions
Data read/write	DR/$\overline{W}$	Indicates whether the memory transaction is a read (DT0 = 1) or a write (DT0 = 0)
Data bus lock	$\overline{DLOCK}$	The memory lock pin is used by the xmem instruction in conjunction with the CMMU; when asserted, the CMMU maintains control of the memory bus during the two xmem accesses; data are guaranteed to be unaccessed between the read and write accesses of the xmem instruction

TABLE 8.18 Signal Index (*continued*)

Signal name	Mnemonic	Function
Data byte enable	DBE0-DBE3	Used during memory accesses, these signals indicate which bytes are accessed at the addressed location; DEB0-DEB3 are always valid during memory write cycles; a memory read is always 4 bytes wide, and the processor uses the enables to extract the valid data; that is, during an ld instruction, the memory system should drive all 32 data signals, regardless of whether 1, 2, or 4 bytes enables are asserted; when DEB0-DEB3 are negated, the transaction is a null; otherwise, the transaction is a valid load or store operation
Data reply	DR0-DR1	Indicates the status of the data memory transaction
Code address bus	CA2-CA31	Provides the 30-bit word address to the instruction memory space; all instructions are 32 bits wide and are aligned on 4-byte boundaries; therefore, the lower two bits of the address space are not required and are implied to be zero
Code bus	C0-C31	This read-only, 32-bit data bus interfaces the MC88100 to the instruction memory space; instructions are always 32 bits wide
Code supervisor/user select	$CS/\overline{U}$	Selects between the user and supervisor instruction memory spaces; when asserted, selects supervisor memory and when negated, user memory; this signal is determined by the value of the MODE bit in the processor status register
Code fetch	CFETCH	When asserted, signals that an instruction fetch is in progress; when negated, the transaction is a null transaction (code P bus idle)
Code reply	CR0-CR1	Signals the status of the instruction memory transaction
Error	ERR	Asserted when a bus comparator error occurs, ERR indicates that the desired signal level was not driven on the output pin; ERR is used in systems implementing a master/checker configuration of MC88100s
Clock	CLK	Internal clock normally phase locked to minimize skew between the external and internal signals; since CLK is applied to peripherals (such as CMMU devices), exact timing of internal signals is required to properly synchronize the device to the P bus
Phase lock enable	PLLEN	Asserted during reset to select phase locking, PLLEN controls the internal phase lock circuit that synchronizes the internal clocks to CLK
Reset	$\overline{RST}$	Used to perform an orderly restart of the processor; when asserted, the instruction pipeline is cleared and certain internal registers are cleared or initialized; when negated, the reset vector is fetched from memory, with execution beginning in supervisor mode
Interrupt	INT	Indicates that an interrupt request is in progress; when asserted, the processor saves the execution context and begins execution at the interrupt exception vector; software is responsible for handling all recognized interrupts (those between instructions when no higher priority exception occurs)
P bus checker enable	PCE	Used in systems incorporating two or more MC88100s redundantly; when negated, the processor operates normally and when asserted, the processor monitors (but does not drive) all of its outputs except ERR as inputs
Power supply	Vcc	+ 5-volt power supply
Ground	GND	Ground connections

The 88100 uses memory-mapped I/O. The 88100 can readily be interfaced to memory and I/O chips using its bus control signals.

8.3.5 88100 Exception Processing

The 88100 includes the following exceptions:

- Reset the hardware interrupts which are activated externally via the respective input pins
- Externally activated errors such as a memory access fault
- Internally generated errors such as divide by zero
- Trap instructions

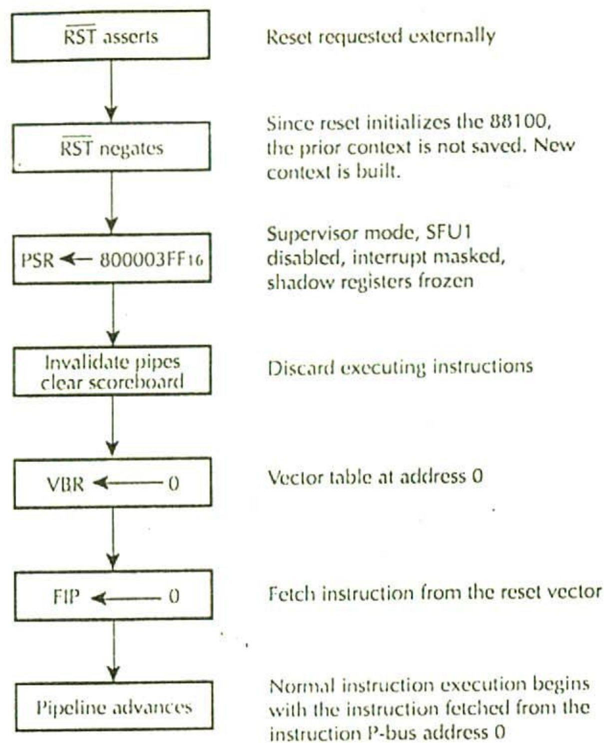


FIGURE 8.15 88100 reset exception flowchart.

Exceptions are processed by the 88100 after completion of the current instructions. When an exception is acknowledged, the 88100 freezes the contents of shadow and exception time control registers, disables interrupts and all SFUs, and enters the supervisor mode.

Figure 8.15 shows the 88100 reset flowchart. When the 88100 $\overline{RST}$ pin is asserted, all outputs go into high impedance state except ERR which indicates no error. Upon hardware reset, the 88100 initializes the PSR with appropriate data ($800003FF_{16}$ for supervisor mode operation, SFU1 disabled, interrupt masked, and shadow registers frozen), VBR with zero value, and fetches two 32-bit instructions from the reset vector 0.

VBR contains the address of the 88100 vector table. The vector table contains 512 vectors. Each vector corresponds to an exception. Each vector address contains the first two instructions of its exception routine. The instruction stored in the first vector is usually a branch instruction such as ' br.n START' (delayed branch) where START is the starting address of the exception routine. The second vector is normally used to save the current SP, and therefore the instruction such as stcr r31, cr17 is stored at the second vector. The first instruction of the exception routine should load the new SP to be used in the exception routine by using an instruction such as ldcr r31, cr18. The last instruction of the exception handling routine should be rte which returns control to the interrupting routine.

8.4 IBM/Motorola/Apple PowerPC 601

This section provides an overview of the basic features of PowerPC microprocessor. The PowerPC 601 is jointly developed by Apple, IBM, and Motorola. It is available from IBM as PP 601 and from Motorola as MPC 601.

The PowerPC 601 is the first implementation of the PowerPC family of Reduced Instruction Set Computer (RISC) microprocessors. There are two types of PowerPC implementations:

32-bit and 64-bit. The PowerPC 601 implements the 32-bit portion of the IBM PowerPC architectures and Motorola 88100 bus control logic. It includes 32-bit effective (logical) addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits. For 64-bit PowerPC implementations, the PowerPC architecture provides 64-bit integer data types, 64-bit addressing, and other features necessary to complete the 64-bit architecture.

The 601 is a pipelined superscalar processor and is capable of executing three instructions per clock cycle. A pipelined processor is one in which the processing of an instruction is broken down into discrete stages, such as decode, execute, and writeback (result of the operation is written back in the register file).

Because the tasks required to process an instruction are broken into a series of tasks, an instruction does not require the entire resources of an execution unit. For example, after an instruction completes the decode stage, it can pass on to the next stage, while the subsequent instruction can advance into the decode stage. This improves the throughput of the instruction flow. For example, it may take three cycles for an integer instruction to complete, but it there are no stalls in the integer pipeline, a series of integer instructions can have a throughput of one instruction per cycle. Each unit is kept busy in each cycle.

A superscalar processor is one in which multiple pipelines are provided to allow instruction to execute in parallel. The PowerPC 601 includes three execution units. These are a 32-bit integer unit (IU), a branch processing unit (BPU), and a pipelined floating-point unit (FPU).

The PowerPC 601 contains an on-chip, 32-Kbyte unified cache (combined instruction and data cache) and an on-chip memory management unit (MMU). It has a 64-bit data bus and 32-bit address bus. The 601 supports single-beat and four-beat burst data transfer for memory accesses. Note that a single-beat transaction indicates data transfer of up to 64 bits. The PowerPC 601 uses memory-mapped I/O. Input/Output devices can also be interfaced to the PowerPC 601 by using I/O controller. The 601 is designed by using an advanced, CMOS process technology and maintains full compatibility with TTL devices.

The main features of the PowerPC 601 are compared with a similar pipelined superscaler RISC microprocessor manufactured by Digital Equipment Corporation, the Alpha 21064. Finally, typical 64-bit RISC microprocessors are discussed.

8.4.1 PowerPC 601 Block Diagram

Figure 8.16 shows the functional block diagram of the PowerPC 601.

The 601 contains the following on-chip hardware:

a. RTC (Real Time Clock)
b. Instruction Unit
c. Execution Unit
d. Memory Management Unit (MMU)
e. Cache Unit
f. Memory Unit
g. System Interface

8.4.1.a RTC (Real Time Clock)

The RTC has normally been an I/O device completely outside the CPU in the most earlier microcomputers. While the RTC appearing inside the microcomputer chip is common in single chip microcomputers, this is the first time the RTC is implemented inside a top-of-the line microprocessor such as the PowerPC. The implication is that modern multi-tasking operating systems require time keeping for task switching as well as keeping the calendar date.

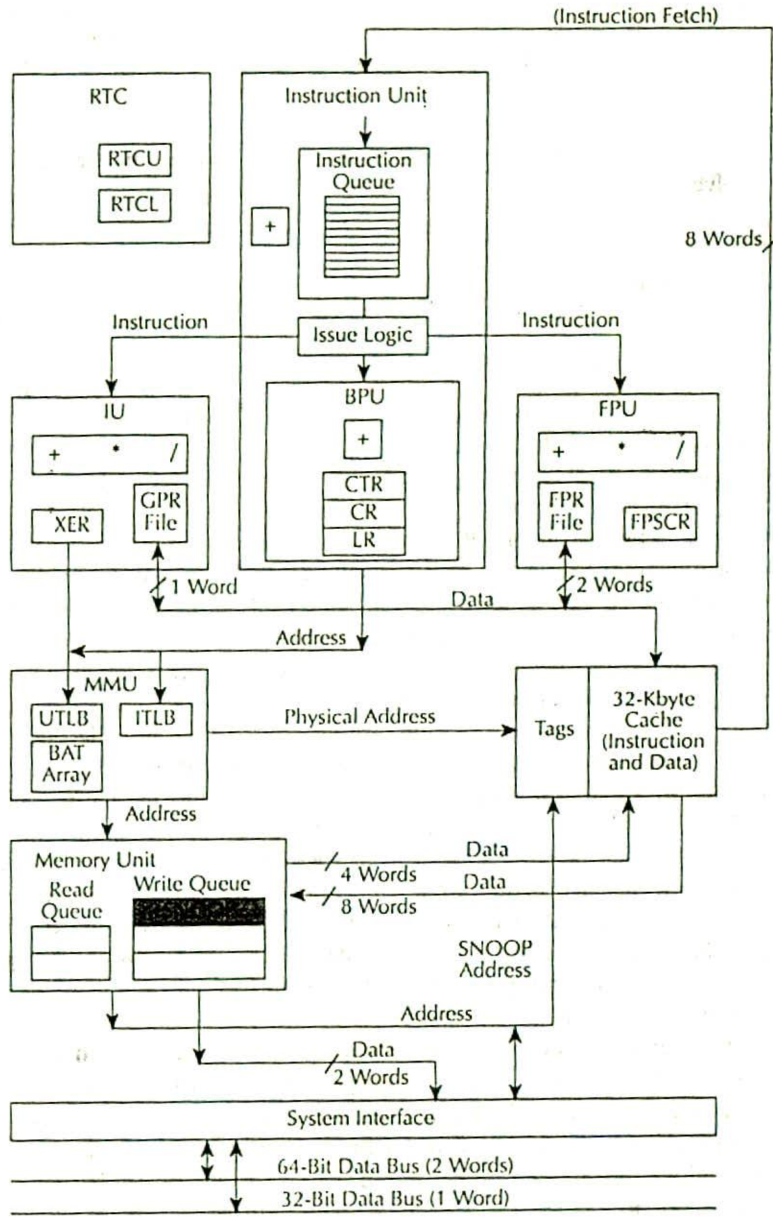


FIGURE 8.16 PowerPC 601 microprocessor block diagram.

The 601 real-time clock (RTC) on-chip hardware provides a measure of real-time in terms of time of day and data with a calendar range of 136.19 years. The RTC contains two registers. These are the RTC upper (RTCU) reigster and the RTC lower (RTCL) register. The RTCU register maintains the number of seconds from a point in time specified by software. The RTCL register counts nanoseconds. The contents of these registers can be copied to any 601 general purpose reigster.

8.4.1.b Instruction Unit

The 601 instruction unit computes the address of the next instruction to be fetched. The instruction unit includes an instruction queue and a Branch Processing Unit (BPU).

The instruction queue holds up to eight instructions and can be filled from the cache during a single cycle.

The BPU searches through the instruction queue for a conditional branch instruction and tries to resolve it early in order to achieve zero-cycle branch in many instances.

8.4.1.c Execution Unit

The 601 execution unit includes three on-chip hardware components. These are floating-point unit (FPU), integer unit (IU), and branch processing unit (BPU). These units operate independently and in parallel.

The FPU includes a single-precision multiply-add (combinatorial) array, the floating-point status and control register (FPSCR), and thirty-two 64-bit floating-point registers. The PowerPC FPU is a cominatorial unit that provides a complete product followed by a sum (with previously accumulated products) in a single clock cycle. Note that this is the heart of a DSP (Digital Signal Processor) chip. The implication is that the power PC chip can replace a DSP. IBM has intentions along these lines for multimedia work in PowerPC based PC's. It is expected that while the machine 'may' be fast enough, the software and timing complexity will be so great that distributing the tasks to other processors (like DSPs) can be more likely to result in an error free product. The PowerPC is pipelined so that most floating-point instructions can be issued back to back. The FPU has no feed-forwarding capabilities. In other words, as a floating point operation completes, another floating-point instruction that may be waiting for those results must wait for the data to be written into the register file before decode can begin. The multiply-add array allows the 601 to implement floating-point operations such as multiply, add, divide, and multiply-add. The 601 FPU supports all IEEE754 floating-point data types (normalized, denormalized, NaN, Zero, and infinity) in hardware.

The Integer Unit (IU) executes all integer instructions (computational and logical instructions). Most integer instructions are single-cycle instructions. The IU interfaces with the cache and MMU for all instructions that access memory. The IU executes one integer instruction at a time by utilizing its arithmetic logic unit (ALU), multiplier, divider, integer exception register (XER), and the general purpose register (GPR) file.

The branch processing unit (BPU) is used for prediction of 601 conditional march instructions early in order to achieve zero-cycle branch. The BPU contains an adder to compute branch target addresses and three special-purpose, user-control registers namely, the link register (LR), the count register (CTR), and the condition register (CR). The LR is used to save the return pointer (computed by the BPU) for subroutine calls. The LR also contains the branch target address for certain types of branch instructions such as branch conditional to link register instruction (bclrx). The CTR, on the other hand, contains the branch target address for some other instructions such as branch conditional to count register (bcctrx) instruction. The CR reflects the result of certain operations and provides a mechanism for testing and branching.

8.4.1.d Memory Management Unit (MMU)

The memory management unit (MMU) of the PowerPC 601 supports up to 4 peta bytes (2**52) of virtual memory and 4 Gigabytes of physical memory.

The main functions of the 601 on-chip MMU hardware are to:

- Translate Logical (effective) addresses to physical addresses for memory accesses.
- Translate I/O accesses (most I/O accesses are assumed to be memory mapped).
- Translate I/O controller interface accesses.

- Provides access protection on blocks and pages of memory by the operating system in relation to the supervisor/user privilege level of the access and in relation to whether the access is load or store.

The 601 generates three types of accesses that require address translation. These are instuction accesses, data accesses to memory generated by load and store instructions, and I/O controller interface accesses generated by load and store instructions.

The 601 MMU supports demand-paged virtual memory. Virtual memory management allows execution of programs larger than the size of the physical memory. Demand-paged means that individual pages are loaded into physical memory from system memory only when they are first accessed by an executing program.

To accomplish the above functions, the 601 MMU hardware contains three components. These are UTLB (Unified Translation Lookaside Buffer), ITLB (Instruction Translation Lookaside Buffer) and BAT (Block Address Translation) array.

For instruction accesses, the 601 MMU first performs a lookup in the four entries of the ITLB for both block- and page-based physical address translations. Instruction accesses that miss in the ITLB and all data accesses cause a lookup in the UTLB and BAT array for the physical address translation. In most cases, the physical address translation resides in one of the TLBs and the physical address bits are readily available to the on-chip cache. However, if the physical address translation misses in the TLBs, the 601 automatically performs a search of the translation tables in memory using the information in the table search descriptor register (SDR1) and the corresponding segment register.

8.4.1.e Cache Unit

The PowerPC 601 includes a 32-Kbyte, eight-way set associative, unified (instruction and data) cache. The cache line size is 64 bytes, divided into two eight-word sectors, each of which can be snooped, loaded, cached out, or invalidated independently. Note that snooping means monitoring addresses driven by a bus master to detect the need for coherency actions. The 601 controls cacheability, write policy, and memory coherency. The cache uses a least recently used (LRU) replacement policy.

The instruction unit provides the cache with the address of the next instruction to be fetched. In the case of a cache hit, the cache returns the instruction and as many of the instructions following it as can be placed in the eight-word instruction queue up to the cache sector boundary.

The cache tag directory has one address port dedicated to the instruction fetch and load/store accesses and one port dedicated to snooping transactions on the system interface.

8.4.1.f Memory Unit

The 601's on-chip memory unit consists of read and write queues that buffer operations between the external interface and the cache. These operations are comprised of operations resulting from load and store instructions that are cache misses, read and write operations required to maintain cache coherency, and table search operations. The read queue contains two elements and write queue contains three elements. The read queue receives requests from the cache unit for arbitration onto the 601 bus interface. Each element of the write queue can contain up to eight words (one sector) of data. One element of the write queue marked snoop is dedicated to writing cache sectors to system memory after a modified sector is hit by a snoop from another processor or snooping device on the system bus. The other two elements in the write queue are used for storing operations and writing back modified sectors that have been deallocated by updating the queue. That is, when a cache location is full, the least-recently used cache sector is deallocated by first being copied into the write queue and from there to system memory.

8.4.1.g System Interface

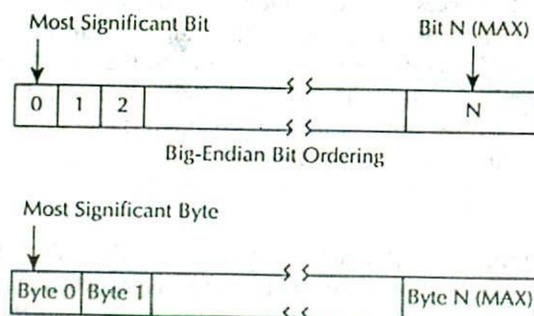
The 601 system interface includes a 32-bit address bus, a 64-bit data bus, and 52 control and information signals.

The 601 control and information signals allow for functions such as address arbitration, address start, address termination, address transfer, data arbitration, data start, and data termination.

In a multiprocessor system, the system interface supports bus pipelining. The 601 supports split bus transactions for systems with potential multiple bus masters. Allowing multiple bus transactions to occur simultaneously increases the available bus bandwidth for other activity and as a result, improves performance.

8.4.2. Byte and Bit Ordering

The PowerPC 601 supports both big- and little-endian byte ordering. The default byte and bit ordering is big-endian is shown below:



For example, to specify the ordering of four bytes (ABCD) within 32 bits, the 601 can use either the ABCD (big-endian) or DCBA (little-endian) ordering. The 601 big- or little-endian modes can be selected by setting the LM bit (bit 28) in the HID0 register.

Note that big-endian ordering (ABCD) assigns the lowest address to the highest-order eight bits of the multibyte data. On the other hand, little-endian byte ordering (DCBA) assigns the lowest address to the lowest order (rightmost) 8 bits of the multibyte data.

Note that Motorola 68XXX supports big-endian byte ordering while Intel 80XXX supports little-endian byte ordering.

8.4.3 PowerPC 601 Registers and Programming Model

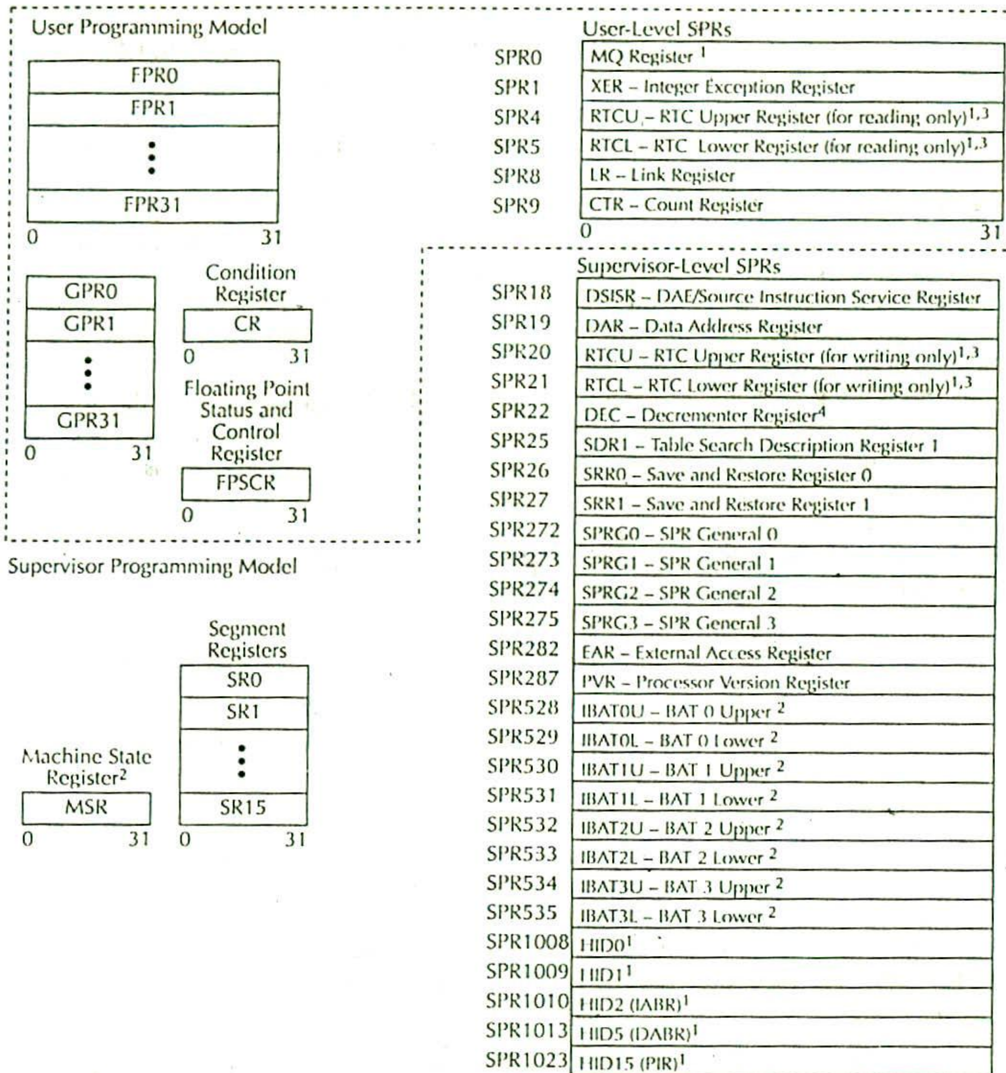
Figure 8.17 shows the register of the PowerPC 601 32-bit implementation. For 64-bit implementation, most registers are 64-bit wide.

These registers can be accessed depending on the program's access privilege level (supervisor or user mode). The privilege level is determined by the privilege level (PR) bit in the machine status register (MSR). The supervisor mode of operation is typically used by the operating system while the user mode is used by the application software.

The PowerPC 601 programming model contains user- and supervisor-level registers as follows:

8.4.3.a User-Level Registers

The user-level register can be accessed by all software with either user or supervisor privileges.



[1] 601-only registers. These registers are not necessarily supported by other PowerPC processors.
[2] These registers may be implemented differently on other PowerPC processors. The PowerPC architecture defines two sets of BAT registers–eight IBATs and eight DBATs. The 601 implements the IBATs and treats them as unified BATs.
[3] RTCU and RTCL registers can be written only in supervisor mode, in which case different SPR numbers are used.
[4] DEC register can be read by user programs by specifying SPR6 in the mfspr instruction (for POWER compatibility).

FIGURE 8.17 PowerPC 601 microprocessor programming model-registers.

The 32 32-bit GPRs (General Purpose registers, GPR0-GPR31) can be used as the data source or destination for all integer instructions. They can also provide data for generating addresses.

The 32 32-bit FPRs (Floating-Point registers, FPR0 through FPR31) can be used as data source and destination for all floating point instructions.

The Floating-point status and control register (FPCSR) is a user-control register in the Floating-Point Unit (FPU). It contains floating-point status and control bits such as floating-point exception signal bits, exception summary bits, and exception enable bits.

The condition register (CR) is a 32-bit register, divided into eight 4-bit fields, CR0-CR7. These fields reflect the results of certain arithmetic operations and provide mechanisms for testing and branching.

The remaining user-level registers are 32-bit special purpose registers, SPR0, SPR1, SPR4, SPR5, SPR8, and SPR9.

SPR0 is known as the MQ register and is used as a register extension to hold the product for the multiplication instructions and the dividend for the divide instructions. The MQ register is also used as an operand of long shift and rotate instructions.

SPR1 is called the Integer exception register (XER). The XER is a 32-bit register that indicates carries and overflow bits for integer operations. It also contains two fields for load string and compare byte indexed instruction.

SPR4 and SPR5 respectively represent two 32-bit read-only register and hold the upper (RTCU) and lower (RTCL) portions of the Real Time Clock (RTC). The RTCU register maintain the number of second from a time specified by software. The RTCL register maintains the fraction of the current second in nanoseconds.

SPR8 is the 32-bit link register (LR). The link register can be used to provide the branch target address and to hold the return address after branch and link instructions.

SPR9 represents the 32-bit count register (CTR). The CTR can be used to hold a loop count that can be decremented during execution of certain branch instructions. The CTR can also be used to hold the target address for the branch conditional to count register instruction.

8.4.3.b Supervisor-Level Registers

The supervision-level registers can only be accessed by the programs executed with supervisor privileges. These include the following:

8.4.3.b.i Machine State Register (MSR). The MSR is a 32-bit register that defines the state of the processor. When an exception occurs, the bits in MSR are changed according to the exception. The bits in the MSR indicate processor information such as privilege level and single step trace enable.

The privilege level (PR bit, bit 17 of MSR) indicates whether the 601 can execute both user and supervisor level instructions (PR = 0) or can only execute user level instructions (PR = 1).

The single-step trace enable (SE bit, bit 21 of MSR) indicates whether the processor executes instructions normally (SE = 0) or executes instructions in single step mode (SE = 1).

8.4.3.b.ii Segment Registers. The 601 includes sixteen 32-bit registers (SR0-SR15). The bits in the segment register are interpreted differently depending on the value of the T-bit (bit 0). For example, if T= 0 in the selected segment register, the effective address is a reference to an ordinary memory segment. On the other hand, if T=1, in the selected segment register, the effective address is a reference to an I/O controller interface segment.

8.4.3.b.iii Supervisor-Lever SPRs. Many of the SPRs can be accessed only by supervisor-level instructions. Any attempt to access these SPRs with user-level instructions will result in a "privilege exception". These registers consist of the following:

- The 32-bit data access exception (DAE)/source instruction service register (DSISR) defines the cause of data access and alignment exceptions.
- Real-Time clock (RTC) register includes two 32-bit registers namely, RTC upper (RTCU) and RTC lower (RTCL). The registers can be read from by user-lever software, but can be written to only by supervisor-level software.
- Decrement register (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay. The 601 implements a separate clock input rather than the processor clock that serves both the DEC and the RTC facilities.
- The 32-bit Table search description register 1 (SDR1) specifies the page table format used in logical-to-physical address translation for pages.

- The 32-bit machine status save/restore register 0 (SPR0) is used by the 601 for saving the address of the instruction that caused the exception, and the address to return to when a return from Interrupt (rfi) instruction is executed.
- General SPRs, SPRG0-SPRG3, are 32-bit registers provided for operating system use.
- The 32-bit external access register (EAR) control facility through the External Control Input Word Indexed (eciwx) and External Control Output Word Indexed (ecowx) instructions.
- The 32-bit processor version register (PVR) is a read-only register that identifies the version (model) and revision level of the PowerPC processor.
- The eight 32-bit block address translation (BAT) register are grouped into four pairs of BATs (BAT0U-BAT3U and BAT0L-BAT3L).

  The block address translation mechanism in the 601 is implemented as a software controlled BAT array. The BAT array maintains the address translation information for four blocks of memory. The BAT array in the 601 is maintained by the system software and is implemented as a set of eight special-purpose registers (SPRs). Each block is defined by a pair of SPRs called upper and lower BAT registers.
- Five 32-bit hardware implementation registers (HID0-HID2, HID5, and HID15) are provided primarily for implementing debugging features such as break point and single stepping. HID15 holds the four-bit processor identification tag (PID) that is useful for differentiating processor in multiprocessor system designs.

8.4.4 PowerPC 601 Memory Addressing: Effective Address (EA) Calculation

The effective address (EA) is the 32-bit address computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction.

Effective address computations for both data and instruction accesses use 32-bit unsigned binary arithmetic. If the sum of the effective address and the operand length exceeds the maximum effective address, a carry from bit is ignored. Arithmetic and logic instructions do not read or modify memory. To use the contents of a memory location in a computation, and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location using load or store instructions. This is a consequence of a RISC architecture. Note that the RISC ideology caused the lack of capability for instructions such as ADD instruction to directly modify memory. ADDs are register/register operations in a RISC while ADDs are memory operations in a CISC. The concept of alignment is also applied more generally to data in memory. For example, 12 bytes of data are said to be word-aligned if its address is a multiple of 4. The operand of a single-register memory access instruction has a natural boundary equal to the operand length. That is, the "natural" address of an operand is an integral multiple of the operand length.

A memory operand is said to be aligned if it is aligned at its natural boundary, otherwise it is misaligned. The PowerPC can transfer both aligned and misaligned data between the processor and memory. However, the placement (location and alignment) of operands in memory affects the relative performance of memory accesses. Best performance is guaranteed if memory operands are aligned on natural boundaries

Operands for single-register memory access instructions have the characteristics shown below:

Memory Operand (if aligned)	Length	Address (28–31)
Byte	8 bits	XXXX
Half word	2 bytes	XXX0
Word	4 bytes	XX00
Double word	8 bytes	X000

In the above, an "X" indicates that the bit can be 0 or 1 independence of the state of the bits in the address.

Load and store operations have two types of effective address generation :

8.4.4.a Register Indirect with Immediate Index Mode

Instructions using this mode contain a signed 16-bit index (d operand in the 32-bit instruction) which is sign extended to 32 bits, and added to the contents of a general purpose register specified by five bits in the 32-bit instruction (rA operand) to generate the effective address. A zero in the rA operand causes a zero to be added to the immediate index (d operand). The option to specify rA or 0 is shown in the instruction descriptions of the 601 user's manual as the notation (rA|0).

An example is lbz rD, d (rA) where rA specifies a general purpose register (GPR) containing an address, d is the 16-bit immediate index and rD specifies a general purpose register as destination. Consider lbz r1, 20 (r3). The effective address (EA) is the sum r(3|0) + 20. The byte in memory addressed by the EA is loaded into bits 31 through 24 of register r1. The remaining bits in r1 are cleared to zero. Note that registers r1 and r3 represent GPR1 and GPR3 respectively.

8.4.4.b Register Indirect with Index Mode

Instructions using this addressing mode add the contents of two general purpose registers (one GPR holds an address and the other GPR holds the index). An example is lbzx rD, rA, rB where rD specifies a GPR as destination, rA specifies a GPR as the index, and rB specifies a GPR holding an address. Consider lbzx r1, r4, r6. The effective address (EA) is the sum (r4|0) + (r6). The byte in memory addressed by the EA is loaded into register r1 (24-31). The remaining bits in register rD are cleared to 0.

PowerPC 601 conditional and unconditional Branch instructions compute the effective address (EA) or the next instruction address using various addressing modes. A few of them are described below:

8.4.4.b.i Branch Relative. Branch instructions (32-bit wide) using the relative mode generate the address of the next instruction by adding an offset and the current program counter contents. An example of this mode is an instruction "bc start" unconditionally jumps to the address PC + start.

8.4.4.b.ii Branch Absolute. Branch instructions using this mode include the address of the next instruction to be exectuted. For example, the instruction ba begin unconditionally branches to the absolute address "begin" specified in the instruction.

8.4.4.b.iii Branch to Link Register. Branch instructions using this mode branch to the address computed as the sum of the immediate offset and the address of the current instruction. The instruction address following the instruction is placed into the link register. For example, the instruction bl start unconditionally jumps to the address computed from current PC contents plus start. This return address is also placed in the link register.

8.4.4.b.iv Branch to Count Register. Instructions using this mode branch to the address contained in the current register. Consider bcctr BO,BI means branch conditional to count register. This instruction branches conditionally to the address specified in the count register.

The BI operand specifies the bit in the condition register to be used as the condition of the branch. The BO operand specifies how the branch is affected by or affects condition or count registers. Numerical values specifying BI and BO can be obtained from the 601 manual.

Note that some instructions combine the link register and count register modes. An example is bcctrl BO, BI. This instruction first performs the same operation as the bcctr and then places the instruction address following the instruction into the link register. This instruction is a form of "conditional call" as the return address is saved in the link register.

8.4.5 PowerPC 601 Typical Instructions

The 601 instructions are divided into the following categories:

a. Integer Instructions
b. Floating-point Instructions
c. Load/store Instructions
d. Flow control Instructions
e. Processor control Instructions

Integer instructions operate on byte (8-bit), half-word (16-bit), and word (32-bit) operands. Floating-point instructions operate on single-precision and double precision floating-point operands.

Since the PowerPC is based on the RISC as opposed to the CISC architecture, arithmetic and Logical instructions do not read or modify memory.

8.4.5.a Integer Instructions

The integer instructions include integer arithmetic, integer compare, integer rotate and shift, and integer logical instructions.

The integer arithmetic instructions always set integer exception register bit, CA, to reflect the carry out of bit 7. Integer instructions with the overflow enable (OE) bit set will cause the XER bits SO (summary overflow — overflow bit set due to exception) and OV (overflow bit set due to instruction execution) to be set to reflect overflow of the 32-bit result.

Some examples of integer instructions are provided in the following.

Note that rS, rD, rA, and rB in the following examples are 32-bit general purpose registers (GPR) of the 601 and SIMM is 16-bit signed immediate define:

- add rD, rA, SIMM
  Performs the following immediate operation: rD ← (rA|0) + SIMM; (rA|0) can be either (rA) or 0.
- add rD, rA, rB performs rD ← rA + rB
- add.rD, rA, rB adds with CR update as follows: rD ← rA + rB
  The dot suffix enables the update of the condition register.
- Subf rD, rA, rB performs rD ← rB – rA
- Subf.rD, rA, rB performs the same operation as Subf but updates the condition code register.
- addme rD, rA performs the (add to minus one extended)
  Operation: rD ← (rA) + FFFF FFFFH + CA bit in XER
- Subfme rD, rA performs the (subtract from minus one extended)
  Operation: rD ← (rA)′ + FFFF FFFFH + CA bit in XER where (rA)′ represents the ones complement of the contents of rA.
- mulhwu rD, rA, rB performs an unsigned multiplication of two 32-bit numbers in rA and rB. The high-order 32 bits of the 64-bit product are placed in rD.
- mulhw rD, rA, rB performs the same operation as the mulhwu except that the multiplication is for signed numbers.

- mullw rD, rA, rB places the low-order 32 bits of the 64-bit product (rA) * (rB) into rD. The low-order 32-bit products are independent whether the operands are treated as signed or unsigned integers.
- mulli rD, rA, SIMM places the low-order 32 bits of the 48-bit product (rA) * SIMM16 into rD. The low-order bits of the 32-bit product are independent of whether the operands are treated as signed or unsigned integers.
- divw rD, rA, rB divides the 32-bit signed dividend in rA by the 32-bit signed divisor in rB. The 32-bit quotient is placed in rD and the remainder is discarded.
- divwu rD, rA, rB is same as the divw instruction except that the division is for unsigned numbers.
- cmpi crfD, L, rA, SIMM compares 32 bits in rA with immediate value SIMM treating operands as signed integer. The result of comparison is placed in crfd field (0 for CR0, 1 for CR1 and so on) of the condition register. L = 0 indicates 32-bit operands while L=1 represents 64-bit operands. For example, cmpi 0, 0, rA, 200 compares 32-bits in register rA with immediate value 200 and CR0 is affected according to the comparison.
- xor, rA, rS, rB performs exclusive-or operation between the contents of rS and rB. The result is placed into register rA.
- extsb rA, rS places bits 21–31 of rS into bits 21–31 of rA. Bit 24 of rS is the sign extended through bits 0–23 of rA.
- slw rA, rS, rB shifts the contents or rS left the shift count specified by rB [27–31]. Bits shifted out of position 0 are lost. Zeros are placed in the vacated positions on the right. The 32-bit result is placed into rA.
- Srw rA, rS, rB is similar to slw rA, rS, rB except that the operation is for right shift.

8.4.5.b Floating-Point Instructions

Some of the 601 floating-point instructions are provided below:

- fadd frD, frA, frB adds the contents of the floating-point register, frA to the contents of the floating-point register frB. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the specified precision under control of the FPSCR register. The result is then placed in frD.

Note that this 'fadd' instruction requires one cycle in execute stage, assuming normal operations; however, there is an execute stage delay of three cycles if next instruction is dependent.

The 601 floating-point addition is based on "exponent comparison and add by one" for each bit shifted, until the two exponents are equal. The two significants are then added algebraically to form an intermediate sum. If a carry occur, the sum's significand is shifted right on bit position and the exponent is increased by one.

- fsub frD, frA, frB performs frA – frB, normalization and rounding of the result are performed in the same way as the fadd.
- fmul frD, frA, frC performs frD ← frA * frC.

Normalization and rounding of the result are performed in the same way as the fadd. Floating-point multiplication is based on exponent addition and multiplication of the significands.

- fdiv frD, frA, frB performs the floating-point division frD ← frA/frB. No remainder is provided. Normalization and rounding of the result are performed in the same way as the fadd instruction.
- fmsub frD, frA, frC, frB performs frD ← frA * frC – frB. Normalization and rounding of the result are performed in the same way as the fadd instruction.

8.4.5.c Load/Store Instructions

Some examples of the 601 load and store instructions are listed below:

- lhzx rD, rA, rB loads the half word (16- bit) in memory addressed by the sum (rA|0) + (rB0) into bits 16 through 31 of rD. The remaining bits of rD are cleared to zero.
- sthux rS, rA, rB stores the 16-bit half word from bits 16-31 of register rS in memory addressed by the sum (rA|0) + (rB). The value (rA|0) + rB is placed into register rA.
- lmw rD, d(rA) loads n (where n = 32-D and D = 0 through 31) consecutive words starting at memory location addressed by the sum (rA|0) + d into the general purpose register specified by rD through r31.
- stmu rS, d(rA) is similar to lmw except that the stmw stores n consecutive words.

8.4.5.d Flow Control Instructions

Flow control instructions include conditional and unconditional branch instructions. An example of one of these instructions is provided below:

- bc (branch conditional) BO, BI, target branch with offset target if the condition bit in CR specified by bit number BI is true (The condition 'true' is specified by a value in BO).

For example, bc 12, 0, target means that branch with offset target if the condition specified by bit 0 in CR (BI = 0 indicated result is negative) is true (specified by the value BO = 12 according to Motorola PowerPC 601 manual).

8.4.5.e. Processor Control Instructions

Processor control instructions are used to read from and write to the machine state register (MSR), condition register (CR), and special status register (SPRs). Some examples of these instructions are provided below:

- mfcr rD places the contents of the condition register into rD.
- mtmsr rS places the contents of rS into the MSR. This is a supervisor-level instruction.
- mfmsr rD places the contents of MSR into rD. This is a supervisor-level instruction.

8.4.6 PowerPC 601 Exception Model

All 601 exceptions can be described as either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions are caused by events external to the processor's execution. Synchronous exceptions, on the other hand, are handled precisely by the 601 and are caused by instructions; precise exception means that the machine state at the time the exception occurs is known and can be completely restored. That is, the instructions that invoke trap and system call exceptions complete execution before the exception is taken. When exception processing completes, execution resumes at the address of the next instruction.

An example of a maskable asynchronous, precise exception is the external interrupt. When an asynchronous, precise exception such as the external interrupt occurs, the 601 postpones its handling until all instructions and any exceptions associated those instructions complete execution.

System reset and machine check exceptions are two nonmaskable exceptions that are asynchronous and imprecise. These exceptions may not be recoverable or may provide a limited degree of recoverability for diagnostic purpose.

Asynchronous, imprecise exceptions have the highest priority with the synchronous, precise exceptions the next priority and the asynchronous, precise exceptions have the lowest priority.

The 601 exception mechanism allows the processor to change automatically to supervisor state as a result of exceptions. When exceptions occur, information about the state of the

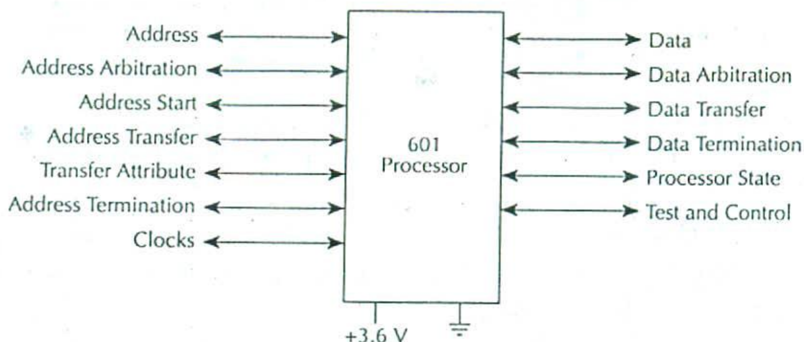


FIGURE 8.18 System interface.

processor is saved to certain registers rather than in memory as is usually done with other processors in order to achieve high speed. The processor then begins execution at an address (exception vector) predetermined for each exception. The exception handler at the specified vector is then processed with processor in supervisor mode.

8.4.7 601 System Interface

Figure 8.18 shows the system interface signals of the PowerPC 601.

The pins and signals of the PowerPC 601 include a 32-bit address bus, a 64-bit data bus, and 52 control and information signals.

The 601 control and information signals include the address arbitration, address start, address transfer, transfer attribute, address termination, data arbitration, data transfer, and data signals. Test and control signals provide diagnostics for selected internal circuitry.

The processor supports multiple master through a bus arbitration scheme that allows various devices to compete for the shared resource. The arbitration logic can implement priority protocols and can park masters to avoid arbitration overhead.

The following sections describe the 601 bus support for memory and I/O controller interface operations.

8.4.7.a Memory Accesses

Memory accesses allow transfer sizes of 8, 16, 24, 32, 40, 48, 56, or 64 bits in one bus clock cycle. Data transfers occur in either single-beat transactions or four-beat burst transactions. A single-beat transaction transfers as much as 64 bits. Single-beat transactions are caused by non cached accesses that access memory directly. An example is reading and writing when the cache is disabled. Burst transactions, which always transfer an entire cache sector (32 bytes), are initiated when a sector in the cache is read from or written to memory.

8.4.7.b I/O Controller Interface Operations

Both memory and I/O accesses can use the same bus transfer protocols. The 601 also has the ability to define memory areas as I/O controller interface areas.

The 601 uses $\overline{TS}$ pin for memory-mapped accesses and XATS pin for I/O controller interface accesses.

8.4.7.c 601 Signals

The 601 signals are grouped as follows:

1. Address arbitration signals — these signals provide arbitration for address bus mastership.
2. Address transfer start signals — these signals indicate that a bus master has begun a transaction on the address bus.
3. Address transfer signals — these signals consisting of the address bus, address parity, and address parity signals, are used to transfer the integrity of the transfer.
4. Transfer attribute signals — these signals provide information about the type of transfer, such as the transfer size, and whether the transaction is bursted or cache-inhibited.
5. Address transfer termination signals — these signals are used to acknowledge the end of the address phase of transaction. They also indicate whether a condition exists that requires the address phase to be replaced.
6. Data arbitration signals — these signals are used to arbitrate for data bus mastership.
7. Data transfer signals — these signals consisting of the data bus, data parity, and data parity error signals, are used to transfer the data and to ensure the integrity of the transfer.
8. Data transfer termination signals — In a single-beat transaction, these signals indicate the end of the tenure, while in burst access, the data termination signals apply to individual beats and indicate the end of the tenure only after the final data beat.
9. System status signals — these signals include the interrupt signal and reset signals.
10. Clock signals — these signals determine the system clock frequency. These signals can also be used to synchronize multiprocessor systems.
11. Processor state signals — these signals are used to set the reservation coherency bit. The 601 uses this bit for atomic memory updating by using its atomic instructions. Note that in multiprocessor systems, a mechanism is required to allow programs to manipulate shared data in an indivisible manner so that when such an operation is underway, another processor cannot perform the same operation. In order to implement higher-level synchronization mechanisms, such as locks and semaphores, atomic instructions are included in the 601. Memory can be updated atomically by setting a reservation on the load instructions (lwarx) and checking that the reservation is still valid (stwcx) is executed.

The reservation (RSRV) output signal is always driven by bus clock cycle and reflects the status of the reservation coherency bit in the reservation address register.

8.4.8 PowerPC 601 Vs. Alpha 21064

Both Motorola/IBM/Apple PowerPC 601 and Digital Equipment Corporation's Alpha 21064 are RISC-based superscalar microprocessors. That is, they can execute two or more instructions per cycle.

The PowerPC 601 contains powerful instructions while the Alpha 21064 includes a simplified instruction set with a very fast clock.

Both the PowerPC 601 and the Alpha are based on load/store architectures. This means that all instructions that access memory are either loads or stores, and all operate instructions are from register to register. They both have 32-bit fixed-length instructions along with 32 integer and 32 floating-point registers.

The PowerPC 601 includes two primary addressing modes. These are register plus displacement and register plus register. In addition, the 601 load and store instructions perform the load or store operation and also modify the index register by placing the just-computed effective address in it. The Alpha 21064, on the other hand, has only one primary addressing mode called register plus displacement. In Alpha, load and store instructions do not update the index register.

There are significant differences in the way the two microprocessors handle branching. In both architectures, branch target addresses are normally determined by using program counter

TABLE 8.19 Summary of Implementation Characteristics

Characteristic	PowerPC 601	Alpha 21064
Technology	0.6-micron CMOS	0.75-micron CMOS
Die size	1.09 cm square	2.33 cm square
Transistor count	2.8 million	1.68 million
Total cache (instruction + data)	32 Kbyte	16 Kbyte
Package	304-in QFP	431-pin PGA
Clock frequency	50 and 66 Mhz	150 to 200 Mhz

relative mode. That is, the branch target address is determined by adding a displacement to the program counter. However, as mentioned before, conditional branches in the 601 may test fields in the condition code register and the contents of a special register called the count register (CTR). A single 601 branch instruction can implement a loop-closing branch by decrementing the CTR, testing its value, and branching if it is nonzero.

In the Alpha 21064, on the other hand, conditional branches test a general-purpose register relative to zero or by low-order bit (a 1 or 0 in low order bit respectively mean odd or even) odd/even register contents. Thus, results of most instructions can be used directly by conditional branch instructions, as long as they are tested against zero or odd/even.

There are also differences in the way the return address is saved by certain control transfer instructions such as the subroutine call. For example, special jump instructions are used to save the return address in a general-purpose register. The 601, on the other hand, does this in any branch by setting the link (LK) bit to one. The return address is saved in the link register.

Next, the implantation characteristics of the 601 and 21064 are considered. Table 8.19 summarizes these differences.

Both the PowerPC 601 and Alpha 21064 utilize sophisticated pipelines. The 601 uses relatively short independent pipelines with more buffering while the 21064 includes longer pipelines with less internal buffering. The 601 does a lot of computation in each pipe stage. Furthermore, the clock of the Alpha is approximately three times faster than the 601.

The two microprocessors utilize different cache memory designs. For example, the 601 has a unified (combined) 32-Kbyte cache. That is, instructions and data reside in the same cache in the 601. The 21064, on the other hand, has separate data and instruction caches of 8 Kbytes each. Therefore, the 601 is expected to have a higher hit rate than the 21064.

Finally, the 601 offers high performance by utilizing sophisticated design tricks. The 21064 gains performance by design simplicity. For example, the 601 includes powerful instructions such as floating-point multiply-add and update load/store that perform more tasks with fewer instructions. The 21064 architecture's simplicity, on the other hand, lends itself better to very high clock rate implementations.

8.5 64-Bit RISC Microprocessors

Typical 64-bit RISC microprocessors include the Alpha 21164 (Digital Equipment Corporation), the PowerPC 620 (Motorola/IBM/Apple) and the Ultrasparc (Sun Microsystems). These 64-bit processors are ideal candidates for data crunching machines and high-performance desktop systems/workstations.

The number of instructions issued per cycle has been increasing steadily. For example, the PowerPC 601 can issue instructions to the integer, floating-point, and branch-processing units in one and the same cycle. The 64-bit RISC microprocessors can typical issue four instructions to six independent units or more.

These 64-bit processors include multiple integer units which allow multiple integer operations in each cycle. For example, the PowerPC 620 contains three integer units — two for

single cycle and one for multiple cycle operations. Some 64-bit RISC processors such as the Ultrasparc include multiple floating-point units.

The clock frequencies of the 64-bit RISC microprocessors vary from 133 MHz to 300 MHz. These processors can issue a maximum of four instructions per cycle. In order to keep the data and instructions flowing, many 64-bit RISC processors such as the Alpha 21164 are provided with 128-bit data bus. The Alpha 21164 is the fastest microprocessor available today with a maximum of 300 MHz clock. The Alpha 21164 is a four-way superscalar processor.

Table 8.20 compares the various features of typical 64-bit RISC microprocessors.

TABLE 8.20 Comparison of Various Features of Typical 64-bit RISC Microprocessors

Features	Digital Equipment Corp. Alpha 21164	Motorola/IBM/Apple PowerPC 620	Sun Microsystems Ultrasparc
Clock speed	300 MHz	133 MHz	167 MHz
Millions of transistors	9.3	7	3.8
On-chip data/instruction cache, K Byte	8/8 Primary 96 Unified secondary	32/32	16/16
Power, W	50	30	30
Data bus size	128-bit	128-bit	128-bit
Address bus size	40-bit	40-bit	41-bit
Maximum number of instructions per cycle	4	4	4
Number of independent units (integer, floating-point, etc.)	4	6	9

Questions and Problems

8.1 Summarize the basic features of RISC microprocessors. Identify how some of these features are implemented in the 80960SA/SB and 88100.

8.2 What operations are controlled by the 80960SA/SB and 88100 register scoreboard?

8.3 Compare the main on-chip features of the 80960SA/SB with those of the 88100. Comment on the floating-point and real data types.

8.4 Identify the 80960SA/SB and 88100 stack pointers.

8.5 Compare 80960SA/SB cache with the 88200 cache.

8.6 Assume a 80960SA/SB with the condition code 010. Write an instruction to set bit 20 to one in register g8 and store the result in g8.

8.7 What happens after execution of the following instructions:

i) cmpo ox10, r7
ii) cmpinco r12, g4, g7

8.8 Find the contents of r8 with (r2) > (r4) after the following 80960SA/SB instruction sequence is executed:

```
compo r2, r4
testg r8
```

8.9 Find 80960 single instructions which are equivalent to the following instruction sequences:

i) cmpi 0,g0
   ble begin
ii) chkbit 1, g8
   be start

8.10 For the following 80960 instruction, what will be the size of the result: addr ro, r1, fp1.

8.11 Assume 80960. Find the operation performed along with the register in which the result is stored after execution of each of the following instructions:

i) logbnrl r8, fp0
ii) logepr r0, r4, fp0
iii) modi r1, r2, r3
iv) notand g3, g4, g6
v) sqrtrl r2, fp1

8.12 What functions are performed by the following 80960SA/SB pins: BE_0-BE_1, BLAST/FAIL, A(1:3).

8.13 Discuss the 80960SA/SB interrupts.

8.14 Describe briefly the functional blocks included in the 88200. What does the 88200 provide to an 88100 system?

8.15 What is the maximum number of 88200s that can be present in one 88000 processing mode?

8.16 How many pipelines are in the 88100?

8.17 Since there is no return from subroutine instruction, how does the 88100 return from subroutine?

8.18 What 88100 floating-point control registers can be accessed by user mode programs?

8.19 Show the contents of registers and memory locations after the 88100 executes the following instructions:

i) st.h r1, r2, r0
ii) ld.h r1, r2, 0X0A

Assume [r1] = $0000\ 0020_{16}$, [030A] = 2018_{16}
[r2] = $0000\ 0300_{16}$

All numbers are in hexadecimal.

8.20 Find the contents of r5 after execution of the following 88100 instructions:

i) mask.u r5, r2, 0XFFFF
ii) mask r5, r2, r6

Assume [r5] = $\text{AAAA}\ 0100_{16}$
[r2] = $0020\ \text{05FF}_{16}$
[r6] = $7777\ 7777_{16}$

prior to execution of the above instructions.

All numbers are in hexadecimal.

8.21 What is the effect of the 88100 tb1 r0, r1, 200 instruction?

8.22 What are the functions of the 88100 $CS/\overline{U}$, BE0-BE3, C0, and C2 pins?

8.23 What 88100 registers are affected by hardware reset?

8.24 Discuss briefly the 88100 exceptions.

8.25 Write an 80960 or 88100 assembly language instruction sequence to logically shift the content of r2 into r1 to the right by 8 bits.

8.26 Write an assembly language program in 80960 or 88100 assembly language to subtract a 64-bit number in r4 r5 from another 64-bit number in r0 r1. Store result in r0 r1.

8.27 Write a program in 80960 or 88100 assembly language to compute the volume of a sphere $V = 4/3\ \pi r^3$ where r is the 32-bit radius stored in register r2.

8.28 Write a program in 80960 assembly language to perform the following: A + (B/C) where A is an 80-bit floating-point number contained in a floating-point register. B and C are respectively 64-bit floating-point numbers stored in r2 r3 and r4 r5 respectively. Store 80-bit result in a floating-point register. Discard the remainder of B/C.

8.29 Repeat 8.28 for the 88100 except that A, B, and C are 64-bit floating-point numbers. Assume that the number A is in fp1. Store the result in fp2.

8.30 Write program in 80960 or 88100 assembly language to compute the roots of the quadratic equation $ax^2 + bx + c = 0$ by using

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Use registers of your choice.

8.31 Discuss the types of PowerPC architectures.

8.32 How many execution units are included in the PowerPC 601? Comment.

8.33 How does the PowerPC 601 achieve zero-cycle branch?

8.34 What do you mean by the unified cache of the 601? What is its size?

8.35 What is meant by the snoop controller of the 601? What is its purpose?

8.36 List the user-level and supervisor-level registers of the 601?

8.37 How does the 601 MSR indicate the following:
- i) The 601 executes both the user- and supervisor-level instructions.
- ii) The 601 executes only the user-level instructions.

8.38 Explain the operation performed by each of the following 601 instruction:

i) lbz r2,30(r4)
ii) lbzx r1,r2
iii) add.r1,r2,r3
iv) divwu r2,r3,r4
v) extsb r1,r2
vi) fsub fr2,fr3,fr4

8.39 Repeat Examples 8.13 through 8.19 using the 601 assembly language instructions of the PowerPC.

8.40 Discuss briefly the exceptions included in the PowerPC 601.

8.41 What is the purpose of the reservation coherency bit of the 601?

8.42 Compare the basic features of the 601 with the Alpha 21064.

8.43 Compare the basic features of the Alpha 21164, the PowerPC620, and the Ultrasparc.

9

PERIPHERAL INTERFACING

This chapter describes interfacing characteristics of a microcomputer with typical peripheral devices such as a hexadecimal keyboard, display, DMA controller, printer, CRT (Cathode Ray tube) terminal, and coprocessor.

9.1 Keyboard Interface

9.1.1 Basics of Keyboard and Display Interface to a Microprocessor

A common method of entering programs into a microcomputer is via a keyboard. A popular way of displaying results by the microcomputer is by using seven segment displays. The main functions to be performed for interfacing a keyboard are

1. Sense a key actuation.
2. Debounce the key.
3. Decode the key.

Let us now elaborate on the keyboard interfacing concepts. A keyboard is arranged in rows and columns. Figure 9.1 shows a 2 × 2 keyboard interfaced to a typical microcomputer. In Figure 9.1, the columns are normally at a HIGH level. A key actuation is sensed by sending a LOW to each row one at a time via PA0 and PA1 of port A. The two columns can then be input via PB2 and PB3 of port B to see whether any of the normally HIGH columns are pulled LOW by a key actuation. If so, the rows can be checked individually to determine the row in which the key is down. The row and column code in which the key is pressed can thus be found.

The next step is to debounce the key. Key bounce occurs when a key is pressed or released — it bounces for a short time before making the contact. When this bounce occurs, it may appear to the microcomputer that the same key has been actuated several times instead of just once. This problem can be eliminated by reading the keyboard after 20 ms and then verifying to see if it is still down. If it is, then the key actuation is valid.

The next step is to translate the row and column code into a more popular code such as hexadecimal or ASCII. This can easily be accomplished by a program.

There are certain characteristics associated with keyboard actuations which must be considered while interfacing to a microcomputer. Typically, these are two-key lockout and N-key rollover. The two-key lockout takes into account only one key pressed. An additional key

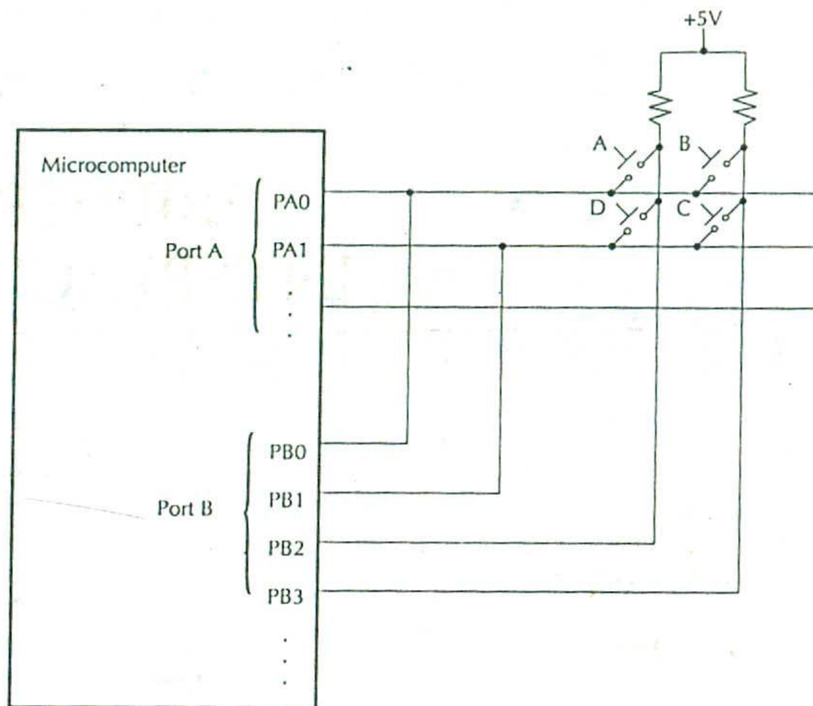


FIGURE 9.1 A 2 × 2 keyboard interfaced to a microcomputer.

pressed and released does not generate any codes. The system is simple to implement and most often used. However, it might slow down the typing since each key must be fully released before the next one is pressed down. On the other hand, the N-key rollover will ignore all keys pressed until only one remains down.

Now let us elaborate on the interfacing characteristics of typical displays. The following functions are to be typically performed for displays:

1. Output the appropriate display code.
2. Output the code via right entry or left entry into the displays if there is more than one display.

The above functions can easily be realized by a microcomputer program. If there is more than one display, they are typically arranged in rows. A row of four displays is shown in Figure 9.2. In the figure, one has the option of outputting the display code via right entry or left entry. If it is entered via right entry, then the code for the most significant digit of the four-digit display should be output first, then the next digit code, and so on. Note that the first digit will be shifted three times, the next digit twice, the next digit once, and the last digit (least significant digit in this case) does not need to be shifted. The shifting operations are so fast that visually all four digits will appear on the display simultaneously. If the displays are entered via left entry, then the least significant digit must be output first and the rest of the sequence is similar to the right entry.

FIGURE 9.2 A row of four displays.

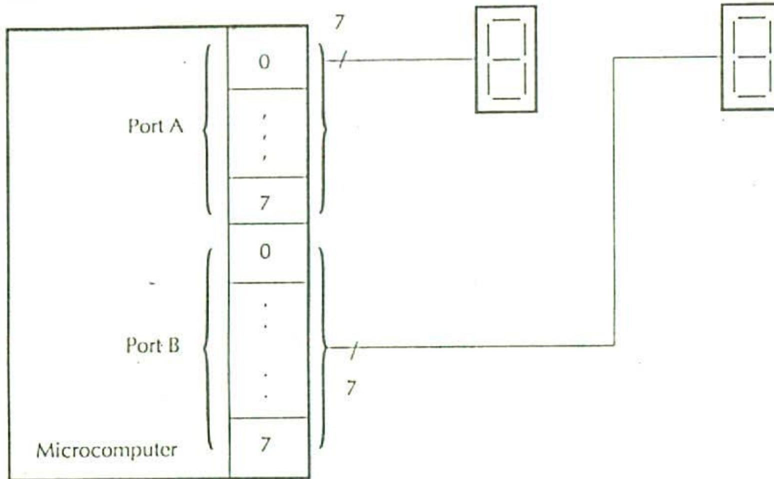


FIGURE 9.3 Nonmultiplexed hexadecimal displays.

Two techniques are typically used to interface a hexadecimal display to the microcomputer. These are nonmultiplexed and multiplexed. In nonmultiplexed methods, each hexadecimal display digit is interfaced to the microcomputer via an I/O port. Figure 9.3 illustrates this method.

BCD to seven-segment conversion is done in software. The microcomputer can be programmed to output to the two display digits in sequence. However, the microcomputer executes the display instruction sequence so fast that the displays appear to the human eye at the same time.

Figure 9.4 illustrates the multiplexing method of interfacing the two hexadecimal displays to the microcomputer.

In the multiplexing scheme, seven-segment code is sent to all displays simultaneously. However, the segment to be illuminated is grounded.

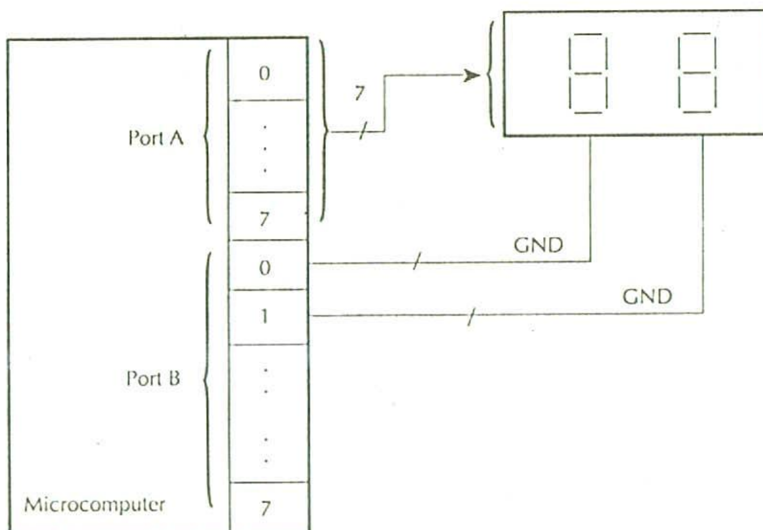


FIGURE 9.4 Multiplexed displays.

The keyboard and display interfacing concepts described here can be realized by either software or hardware. In order to relieve the microprocessor of these functions, microprocessor manufacturers have developed a number of keyboard/display controller chips such as the Intel 8279. These chips are typically initialized by the microprocessor. The keyboard/display functions are then performed by the chip independent of the microprocessor.

The amount of keyboard/display functions performed by the controller chip varies from one manufacturer to another. However, these functions are usually shared between the controller chip and the microprocessor.

9.1.2 8086 Keyboard Interface

In this section, an 8086-based microprocomputer is designed to display 4 hexidecimal digits entered via a keypad (16 keys). Figure 9.5 shows the hardware schematic.

9.1.2.a Hardware

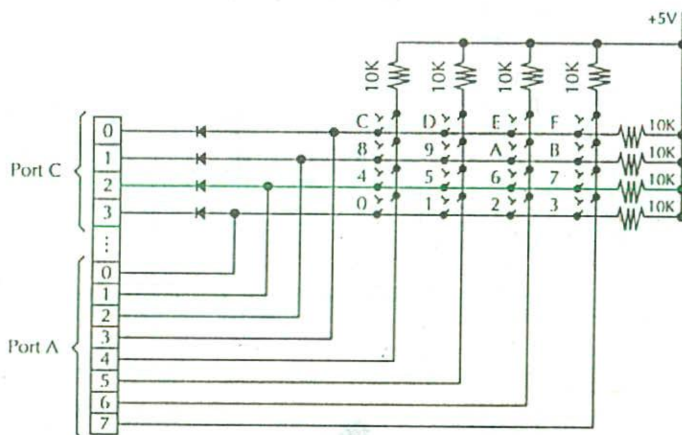
The 8086/8255 microcomputer is designed using standard I/O. For simplicity, only seven address lines are used to directly access the system memory. Therefore, only 128 bytes of system memory can be accessed by the microcomputer. Furthermore, RAM is not available in the system, although RAM should have been used in the design for interrupts and subroutines. However, a small system like this will work without any RAM. Finally, only 8-bit even addressed I/O ports are available in the system. The ports are configured as follows:

1. Port A is configured as an input port to receive the row-column code
2. Port B is configured as an output port to display the key(s) pressed
3. Port C is configured as an output port to control the row-column code

Table 9.1 shows memory and I/O maps.

The system is designed to run at 2 Mhz. Debouncing is provided to avoid unwanted oscillation caused by the opening and closing of the key contacts. In order to ensure stability of the input signal, a delay of 20 msec is used for debouncing the input.

The following diagram shows the internal layout of the keypad used:



9.1.2.b Software

The program begins by performing all necessary initializations. Next, it makes sure that all the keys are opened (not pressed). A delay loop of 20 msec is included for debouncing. The initial loop counter values is calculated as follows:

```
mov    reg/imm      (4 cycles)
loop   label        (19/5 cycles)
```

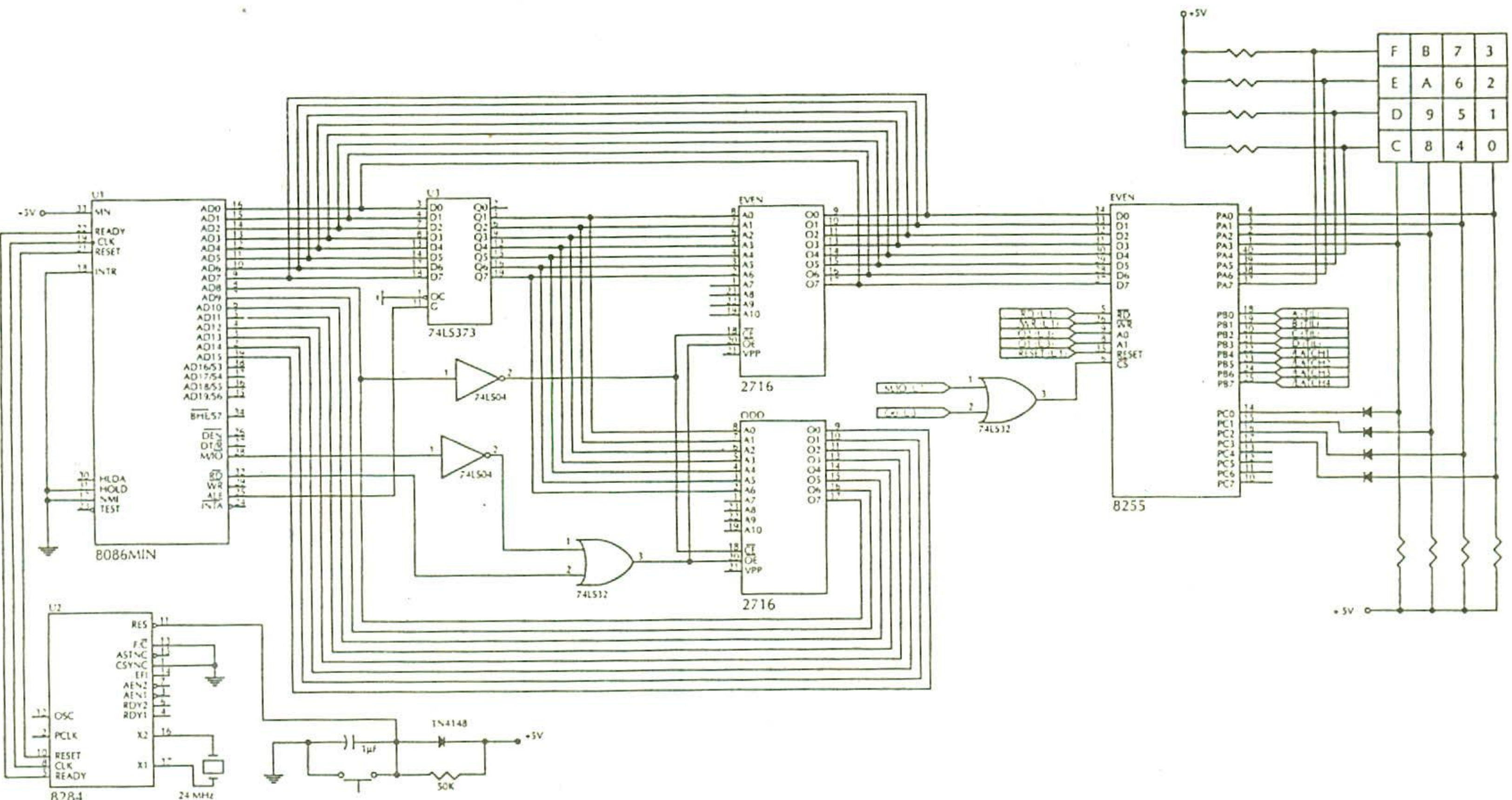


FIGURE 9.5 8086 Keyboard interface.

TABLE 9.1 Memory and I/O Maps

Memory Map:

A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	0
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

I/O Map:

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Port Selected	Hex
1	1	1	1	1	0	0	0	Port A	F8
1	1	1	1	1	0	1	0	Port B	FA
1	1	1	1	1	1	0	0	Port C	FC
1	1	1	1	1	1	1	0	CSR	FE

$$20 \text{ msec} * (2{,}000{,}000 \text{ cycle/msec}) = 40{,}000 \text{ cycles}$$
$$4 + 19 * (\text{count} - 1) + 5 = 40{,}000$$
$$\rightarrow \text{count} = 38D2_{16}$$

The next three lines detect for a key closure. If a key closure is detected, it is first debounced. It is necessary to determine exactly which key is pressed. To do this, a sequence of row-control codes (0FH, 0EH, 0DH, 07H) are output via Port C. The row-column code is input via Port A to determe if the column code changes corresponding to each different row code. If the column code is not 0FH (changed), the input key is identified. The program then indexes through a look-up table to determine the row-column code. If the code is found, the corresponding index value which equals to the input key's face value (a single hexidecimal digit) is buffered. Since the microcomputer does not have access to any RAM, the input key's face value along with three previously entered values are saved into register bx. The upper 4 bits of bh correspond to the most significant digit (least current input), while the lower 4 bits correspond to the next significant digit. The upper 4 bits of bl correspond to the second to the least significant digit. Finally, the lower 4 bits of bl correspond to the least significant digit (most current input). All four digits are output via Port B (one digit at a time). In other words, the displays (four TIL311's) are refreshed for every key input. The program is written such that it will continuously scan for input key and update the display for each new input. Note that lower-case letters are used to represent the 8086 registers in the program. For example, al, ah, and ax in the program represent the 8086 AL, AH, and AX registers, respectively.

A listing of the 8086 assembly language program is given in the following:

```
                          .MODEL SMALL
                          .8086            ;  restrict to 8086
                                               instructions only!!!
0000                      .CODE            ;  ASSUME CS:CODE, DS:DATA
0000          main        PROC
= 00F8                    PORTA EQU 0F8h   ;  hex keyboard input (row/
                                               coln)
= 00FA                    PORTB EQU 0FAh   ;  LED displays/controls
                                               (c3-c0)
= 00FC                    PORTC EQU 0FCh   ;  hex keyboard row controls
= 00FE                    CSR EQU 0FEh     ;  control status register
= 00F0                    OPEN EQU 0F0h    ;  row/coln codes if all
                                               keys are opened
```

```
0000                 start:                      ;
0000 B0 90                      mov al, 90h      ;  config PortA,B,C as i/o/o
0002 E6 FE                      out CSR, al      ;
0004 2A C0                      sub al, al       ;  clear al
0006 E6 FA                      out PORTB, al    ;  enable/initialize displays
0008 2B DB                      sub bx, bx       ;  clear bx (content for
                                                 ;   displays)
000A                 scan-key:
000A 2A C0                      sub al, al       ;  clear al
000C E6 FC                      out PORTC, al    ;  set row controls to zero
000E                 key-close:
000E E4 F8                      in al, PORTA     ;  read PORTA
0010 3C F0                      cmp al, OPEN     ;  Are all keys opened?
0012 75 FA                      jnz key-close    ;  repeat if closed
0014 B9 38D2                    mov cx, 38d2h    ;  delay of 20 msec
0017 E2 FE           delay1:    loop delay 1     ;  debounce key opened
0019                 key-open:
0019 E4 F8                      in al, PORTA     ;  read PORTA
001B 3C F0                      cmp al, OPEN     ;  Are all keys closed?
001D 74 FA                      jz key-open      ;  repeat if opened
001F B9 38D2                    mov cx, 38d2h    ;  delay of 20 msec
0022 E2 FE           delay2:    loop delay2      ;  debounce key closed
0024 B0 FF                      mov al, 0FFh     ;  set al to all one's
0026 F8                         clc              ;  clear carry
0027                 next-row:
0027 D0 D0                      rcl al, 1        ;  set up row mask
0029 8A c8                      mov cl, al       ;  save row mask in al
002B E6 FC                      out PORTC, al    ;  set a row to zero
002D E4 F8                      in al, PORTA     ;  read PORTA
002F 8A D0                      mov dl, al       ;  save row/coln codes in
cl
0031 24 0F                      and al, 0Fh      ;  mask row code
0033 3C 0F                      cmp al, 0Fh      ;  Is coln code affected?
0035 75 05                      jnz decode       ;  if yes, decode coln code
0037 8A C1                      mov al, cl       ;  restore row mask to al
0039 F9                         stc              ;  if no, set carry
003A EB EB                      jmp next-row     ;  check next row
003C                 decode:
003C BE FFFF                    mov si, -1       ;  initialize index register
003F B1 0F                      mov cl, 0Fh      ;  set up counter
0041                 search:
0041 46                         inc si           ;  increment index
0042 2E: 3A 94                  cmp dl,          ;  index thru table of
 0075 R                           [TABLE+si]     ;   codes
0047 E0 F8                      loopne search    ;  loop if not found
0049 B1 04                      mov cl, 04h      ;  amount to be shifted (1
                                                 ;   digit)
004B D3 E3                      shl bx, cl       ;  advanced [bx] by 1 digit
004D 03 DE                      add bx, si       ;  append current digit
004F 8A c7                      mov al, bh       ;  extract 1st/2nd digits
0051 24 F0                      and al, 0F0h     ;  mask 2nd digit
0053 D2 E8                      shr al, cl       ;  move digit to a3-a0
```

```
0055 0C 70              or al, 70h       ; enable /L3 (set low)
0057 E6 FA              out PORTB, al    ; display 1st digit (MSD)
0059 8A C7              mov al, bh       ; extract 1st/2nd digits
005B 24 0F              and al, 0Fh      ; mask 1st digit
005D 0C B0              or al, 0B0h      ; enable /L2 (set low)
005F E6 FA              out PORTB, al    ; display 2nd digit
0061 8A C3              mov al, bl       ; extract 3rd/4th digits
0063 24 F0              and al, 0F0h     ; mask 4th digit
0065 D2 E8              shr al, cl       ; move digit to a3-a0
0067 0C D0              or al, 0D0h      ; enable /L1 (set low)
0069 E6 FA              out PORTB, al    ; display 3rd digit
006B 8A C3              mov al, bl       ; extract 3rd/4th digits
006D 24 0F              and al, 0Fh      ; mask 3rd digit
006F 0C E0              or al, 0E0h      ; enable /L0 (set low)
0071 E6 FA              out PORTB, al    ; display 4th digit (LSD)
0073 EB 95              jmp scan-key     ; return to scan another
                                         ;  key input
0075 E7        TABLE    DB 0E7h          ; code for 0
0076 EB                 DB 0EBh          ; code for 1
0077 ED                 DB 0EDh          ; code for 2
0078 EE                 DB 0EEh          ; code for 3
0079 D7                 DB 0D7h          ; code for 4
007A DB                 DB 0DBh          ; code for 5
007B DD                 DB 0DDh          ; code for 6
007C DE                 DB 0DEh          ; code for 7
007D B7                 DB 0B7h          ; code for 8
007E BB                 DB 0BBh          ; code for 9
007F BD                 DB 0BDh          ; code for A
0080 BE                 DB 0BEh          ; code for B
0081 77                 DB 77h           ; code for C
0082 7B                 DB 7Bh           ; code for D
0083 7D                 DB 7Dh           ; code for E
0084 7E                 DB 7Eh           ; code for F
0085 B8 4C00            mov ax, 4C00h    ; these two lines are
                                         ;  required
0088 CD 15              int 21           ; to exit DOS
008A           main     ENDP             ; end of procedure
                        END main         ; end of program
```

9.2 DMA Controllers

As mentioned before, direct memory access (DMA) is a type of data transfer between the microcomputer's main memory and an external device such as disk without involving the microprocessor. The DMA controller is an LSI (Large-Scale Integration) chip in a microcomputer system which supports DMA-type data transfers. The DMA controller can control the memory in the same way as the microprocessor, and, therefore, the DMA controller can be considered as a second microprocessor in the system, except that its function is to perform I/O transfers. DMA controllers perform data transfers at a very high rate. This is because several functions for accomplishing the transfer are implemented in hardware. The DMA controller is provided with a number of I/O ports. A typical microcomputer system with a DMA controller is shown in Figure 9.6.

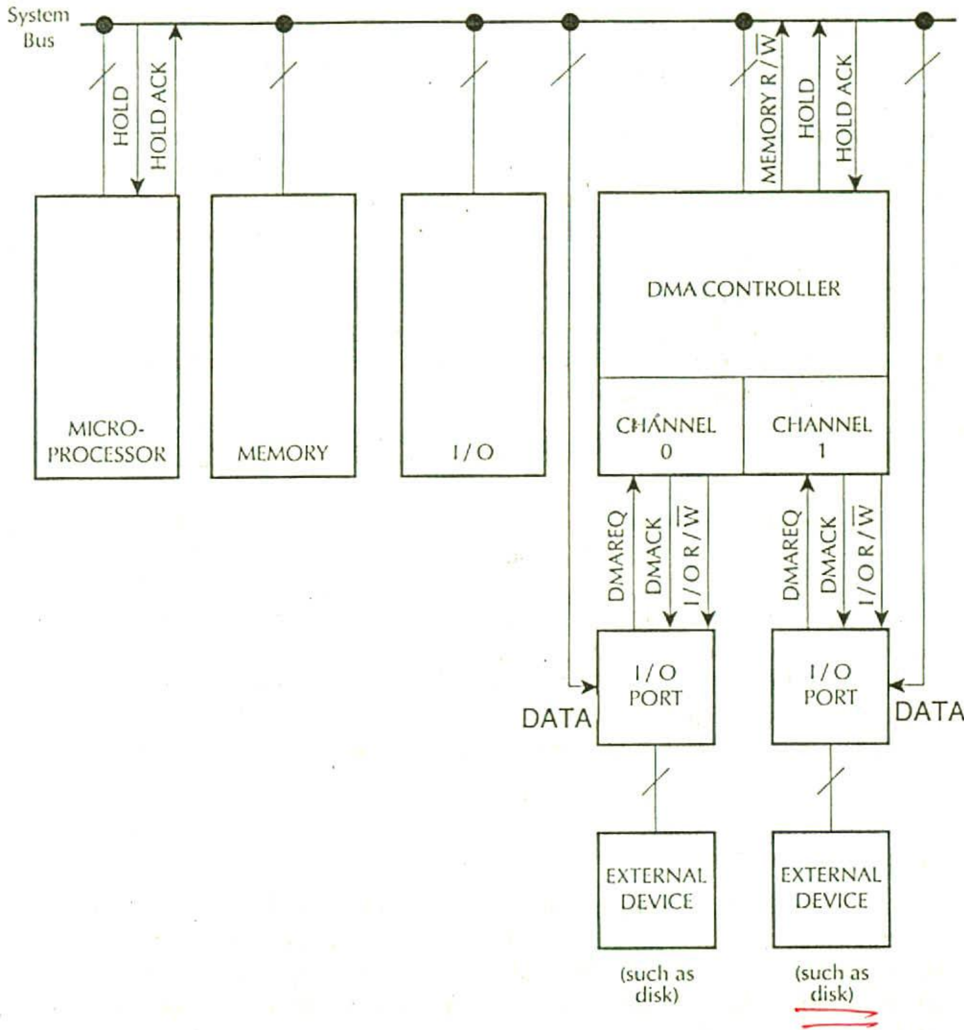


FIGURE 9.6 A microcomputer system with a DMA controller.

The DMA controller in the figure connects one or more ports directly to memory so that data can be transferred between these ports and memory without going through the microprocessor. Therefore, the microprocessor is not involved in the data transfer.

The DMA controller in the figure has two channels (Channel 0 and 1). Each channel contains an address register, a control register, and a counter for block length. The purpose of the DMA controller is to move a string of data between the memory and an external device. In order to accomplish this, the microprocessor writes the starting address of memory where transfer is to take place in the address register, and controls information such as the direction of transfer in the control register and the length of data to be transferred in the counter.

The DMA controller then completes the transfer independent of the microprocessor. However, in order to carry out the transfer, the DMA controller must not start the transfer until the microprocessor relinquishes the system bus and the external device is ready.

The interface between an I/O port and each channel has typically a number of control signals which include DMAREQ, DMACK, and I/O read/write signals. When the I/O port is ready with an available buffer to receive data or has data ready to write into memory location, it activates the DMAREQ line of the DMA controller. In order to accomplish the transfer, the

DMA controller sends the DMACK to the port, telling the port that it can receive data from memory or send data to memory.

DMACK is similar to a chip select. This is because when the DMACK signal on the port is activated by the DMA controller, the port is selected to transfer data between the I/O device and memory. The main difference between a normal and DMA transfer is that the read or write operations have opposite meanings — that is, if the DMA controller activates the read line of the port, then data are read from a memory location to the port. However, this is a write operation as far as the port is concerned. This means that a read from a memory location is a write to the port. Similarly, a write to a memory location is equivalent to a read from the port. The figure shows two types of $R/\overline{W}$ signals. These are the usual memory $R/\overline{W}$ signal and the I/O $R/\overline{W}$ for external devices. The DMA controller activates both of these lines at the same time in opposite directions. That is, for reading data from memory and writing into a port, the DMA controller activates the memory $R/\overline{W}$ HIGH and I/O $R/\overline{W}$ LOW. The I/O ports are available with two modes of operation: non-DMA and DMA.

For non-DMA (microprocessor-controlled transfers), the ports operate in a normal mode. For DMA mode, the microprocessor first configures the port in the DMA mode and then signals the DMA controller to perform the transfer. The $R/\overline{W}$ line is complemented for providing proper direction of the data transfer during DMA transfer.

The DMA controller has a HOLD output signal and a HOLD ACK input signal. The port, when ready, generates the DMAREQ's signal for the DMA controller. The DMA controller then activates the HOLD input signal of the microprocessor, requesting the microprocessor to relinquish the bus, and waits for a HOLD ACK back from the microprocessor.

After a few cycles, the microprocessor activates the HOLD acknowledge and tristates the output drivers to the system bus. The DMA controller then takes over the bus. The DMA controller:

1. Outputs the starting address in the system bus
2. Sends DMACK to the I/O port requesting DMA
3. Outputs normal $R/\overline{W}$ to memory and complemented $R/\overline{W}$ to the I/O port

The I/O port and memory then complete the transfer. After the transfer, the DMA controller disables all the signals including the HOLD on the system bus and tristates all its bus drivers. The microprocessor then takes over the bus and continues with its normal operation.

For efficient operation, the DMA controller is usually provided with a burst mode in which it has control over the bus until the entire block of data is transferred.

In addition to the usual address, control, and counter registers, some DMA controllers are also provided with data-chain registers which contain an address register, a control register, a counter, and a channel identification. These data-chain registers store the information for a specific channel for the next transfer. When the specified channel completes a DMA transfer, its registers are reloaded from the data-chain registers and the next transfer continues without any interruption from the microprocessor. In order to reload the data-chain registers for another transfer, the microprocessor can check the status register of the DMA controller to determine whether the DMA controller has already used the contents of the data-chain registers. In case it has, the microprocessor reinitializes the data-chain registers with appropriate information for the next block transfer and the process continues.

In order to illustrate the functions of a typical DMA controller just described, Motorola's MC68440 dual channel DMA controller is described.

The MC68440 is designed for the MC68000 family microprocessors to move blocks of data between memory and peripherals using DMA.

The MC68440 includes two independent DMA channels with built-in priorities that are programmable. The MC68440 can perform two types of DMA: cycle stealing and burst. In addition, it can provide noncontinuous block transfer (continue mode) and block transfer restart operation (reload mode).

Figure 9.7 shows a typical block diagram of the MC68000/68440/68230 interface to a disk.

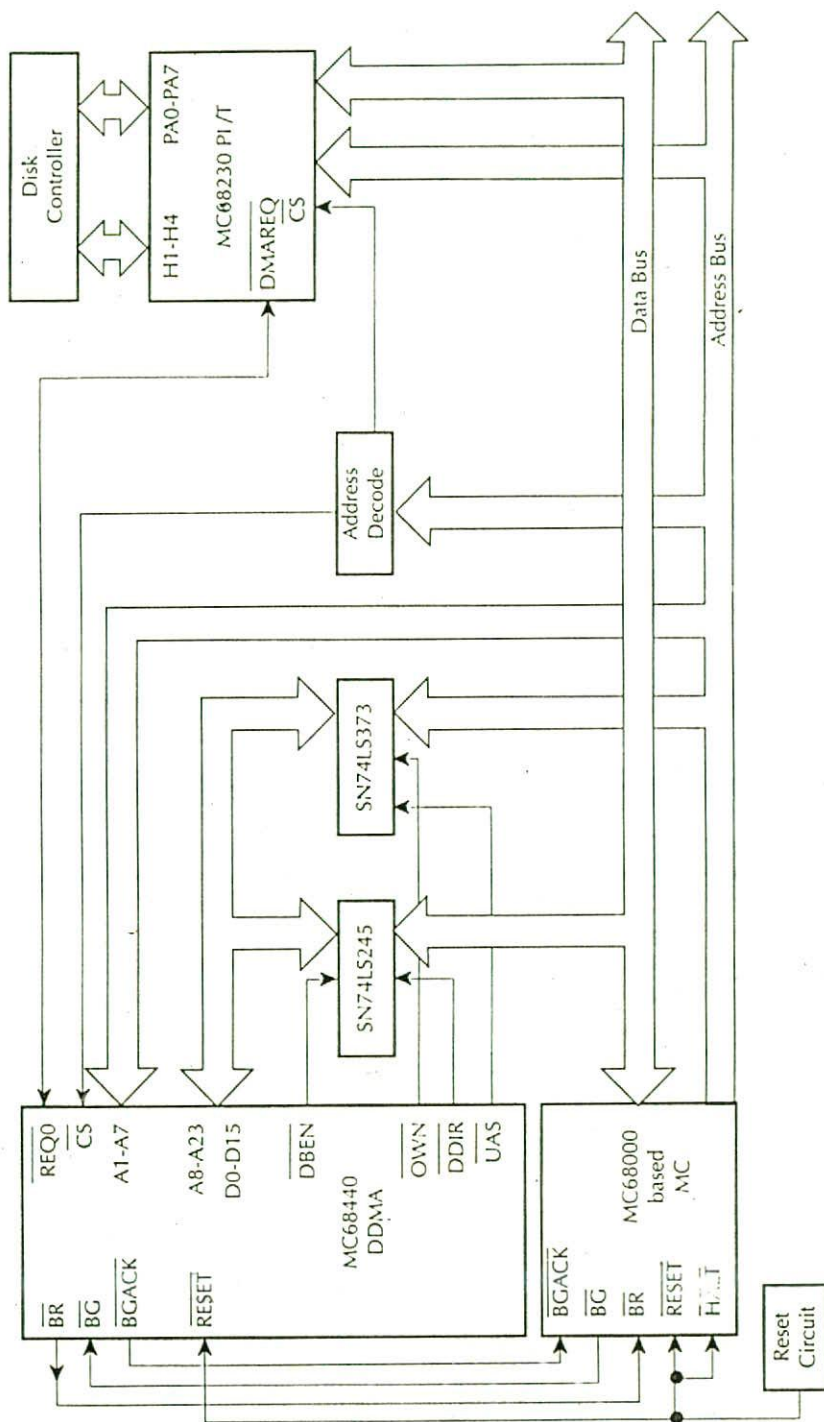


FIGURE 9.7 Typical system configuration.

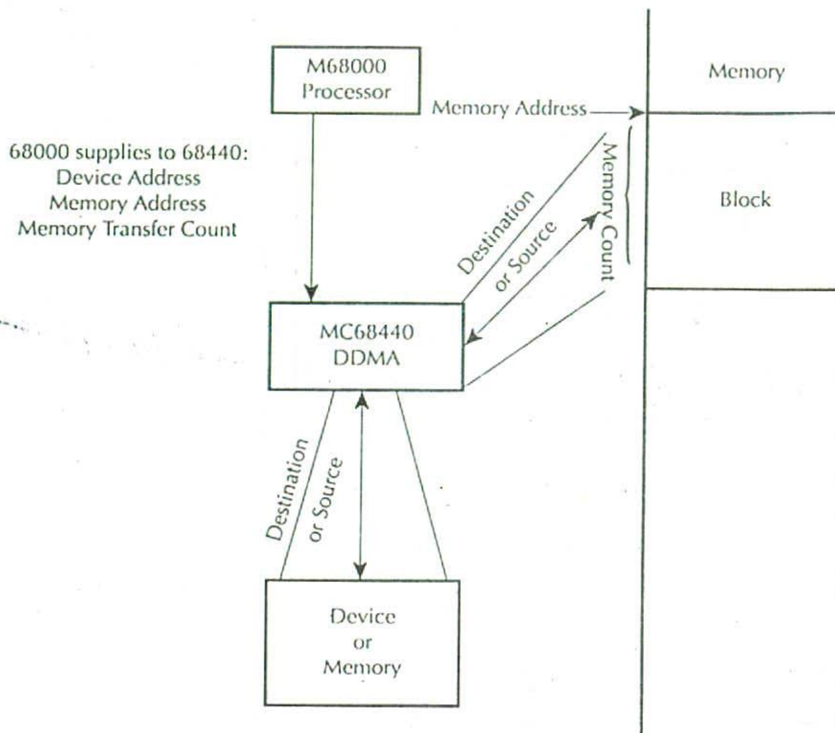


FIGURE 9.8 Data block format.

Data transfer between the disk and the memory takes place via port A of the MC68230, using handshaking signals H1-H4.

The A8/D0 through A23/D15 lines are multiplexed. The MC68440 multiplex control signals $\overline{\text{OWN}}$, $\overline{\text{UAS}}$ (upper address strobe), $\overline{\text{DBEN}}$ (data buffer enable), and $\overline{\text{DDIR}}$ (data direction) are used to control external demultiplexing devices such as 74LS245 bi-directional buffer and 74LS373 latch to separate address and data information on the A8/D0-A23/D15 lines. The MC68440 has 17 registers plus a general control register for each of the two channels and is selected by the lower address lines (A1-A7) in the MPU mode. A1-A7 also provides the lower 7 address outputs in the DMA mode.

A1-A7 lines can select 128 (2^7) registers; however, with A1-A7 lines, only seventeen registers with addresses are defined in the range from 00_{16} through FF_{16} and some addresses are not used. As an example, the addresses of the channel status register and the channel priority register are, respectively, 00_{16} and $2D_{16}$.

The MC68440 registers contain information about the data transfer such as:

1. Source and destination addresses along with function codes
2. Transfer count
3. Operand size and device port size
4. Channel priority
5. Status and error information on channel activity

The processor service request register (PSRR) of the MC68230 defines how the $\overline{\text{DMAREQ}}$ pin should be used and how the DMA transfer should take place, whether via handshaking or ports.

A data block contains a sequence of bytes or words starting at a particular address with the block length defined by the transfer count register. Figure 9.8 shows the data block format.

There are three phases of a DMA transfer. These are channel initialization, data transfer, and block termination. During channel initialization, the MC68000 loads the MC68440 registers with control information, address pointers, and transfer counts, and then starts the channel.

During the transfer phase, the MC68440 acknowledges data transfer requests and performs addressing and bus controls for the transfer. Finally, the block termination phase takes place when the transfer is complete. During this phase, the 68440 informs the 68000 of the completion of data transfer via a status register. During the three phases of a data transfer operation, the MC68440 will be in one of the three modes of operation. These are idle, MPU, and DMA. The MC68440 goes into the idle mode when it is reset by an external device and waits for initialization by the MC68000 or an operand transfer request from a peripheral.

The MPU mode is assumed by the MC68440 when its $\overline{CS}$ (chip select) is enabled by the MC68000. In this mode, the MC68440 internal registers can be read or written for controlling channel operation and for checking the status of a block transfer.

The MC68440 assumes the DMA mode when it takes over the bus to perform an operand transfer.

In Figure 9.7, upon reset, the MC68440 goes into idle mode. In order to initialize the MC68440 registers, the MC68000 outputs appropriate register addresses on the bus. This will enable the MC68440 $\overline{CS}$ line and places the MC68440 in the MPU mode. The MC68000 initializes the MC68440 registers in this mode. The MC68000 then executes the RESET instruction to place the MC68440 back to the idle mode.

The MC68000 now waits for a transfer request from the 68230. When the 68000 desires a DMA transfer between the disk and memory, it enables the $\overline{CS}$ line of the 68230. The 68230, when ready, activates the $\overline{DMAREQ}$ line low, which in turn drives the $\overline{REQ0}$ line of the MC68440 to low. The MC68440 then outputs low on its $\overline{BR}$ line requesting the MC68000 to relinquish the bus. The MC68000, when ready, sends a low on its $\overline{BG}$ pin. This tells the MC68440 to take over the bus. The MC68440 then enters the DMA mode and sends a low on its $\overline{BGACK}$ pin to inform the MC68000 of its taking over the bus. The MC68440 transfers data between the disk and memory via the MC68230. Each time a byte is transferred, the MC68440 decrements the transfer counter register and increments the address register. When the transfer is completed, the MC68440 updates a bit in the status register to indicate this. It also asserts the $\overline{DTC}$ (data transfer complete) to indicate completion of the transfer.

The MC68440 $\overline{DTC}$ pin can be connected to the MC68230 PIRQ pin. The MC68230 then outputs high to the MC68440 $\overline{REQ0}$ pin, which in turn places a HIGH on the MC68000 $\overline{BR}$, and the MC68000 takes over the bus and goes back to normal operation.

9.3 Printer Interface

Microprocessors are typically interfaced to two types of printers: serial and parallel.

Serial printers print one character at a time, while parallel printers print a number of characters on a single line so fast that they appear to be printed simultaneously. Depending on the character generation technique used, printers can be classified as impact or nonimpact. In impact printers, the print head strikes the printing medium, such as paper, directly, in order to print a character. In nonimpact printers, thermal or electrostatic methods are used to print a character.

Printers can also be classified based on the character formation technique used. For example, character printers use completely formed characters for character generation, while matrix printers use dots or lines to create characters.

The inexpensive serial dot matrix impact printer is very popular with microcomputers. An example of such a printer is the LRC7040 manufactured by LRC, Inc. of Riverton, Wyoming. The LRC7040 can print up to 40 columns of alphanumeric characters. The printer includes four major parts. These are the frame, the printhead, the main drive, and the paper handling

	T0	T1	T2	T3	T4
S0					
S1					
S2					
S3					
S4					
S5					
S6					

FIGURE 9.9 5 × 7 Dot matrix pattern for generating the character 'c'.

components. The LRC7040 provides 8 inputs in the basic configuration. One input turns the main drive motor ON or OFF, while the other seven inputs control the print solenoids for the printhead, using TTL drivers.

The LRC7040 utilizes a 5 × 7 matrix of dots to generate characters. The columns are labeled T0 through T4 and rows are labeled S0 through S6. Each row corresponds to one of the solenoids. The entire printhead assembly is moved from left to right across the paper so that at some time the printhead is over the column T0, then it's over column T1, and so on.

A character is generated by energizing the proper solenoids at each one of the columns T0 through T4. Figure 9.9 shows how the character C is formed.

At T0, solenoids S0 through S6 are ON and at T1 through T4 solenoids S0 and S6 are active to form the character C. A number of characters can be formed by the microcomputer by sending appropriate data to the printhead to generate the correct pattern of active solenoids for each of the five instants of time. The code for the character C consists of 5 bytes of data in the sequence $7F_{16}$, 41_{16}, 41_{16}, 41_{16}, 41_{16} as follows:

	S6	S5	S4	S3	S2	S1	S0	
Column T0	1	1	1	1	1	1	1	$= 7F_{16}$
Column T1	1	0	0	0	0	0	1	$= 41_{16}$
Column T2	1	0	0	0	0	0	1	$= 41_{16}$
Column T3	1	0	0	0	0	0	1	$= 41_{16}$
Column T4	1	0	0	0	0	0	1	$= 41_{16}$

Note that in the above, it is assumed that a 1 will turn a solenoid ON and a 0 will turn it OFF. Also it is assumed that S7 is zero.

The interface signals to the printer include a pair of wires for each solenoid, a pair of wires for each motor (main drive motor and line feed motor), a pair of wires indicating the state of the HOME microswitch, and a pair of wires indicating the state of the LINEFEED microswitch.

Paper feed is accomplished by activating the line feed motor. The LINEFEED microswitch is activated by the print logic when the actual paper feed takes place. The control logic can use the trailing edge of the signal generated by the LINEFEED microswitch to turn the line feed motor OFF. The LRC7040 also has an automatic line feed version.

The HOME microswitch is activated HIGH when the printhead is at the left-hand edge of the paper. When the printhead is over the print area and moves from left to right, the HOME microswitch is deactivated to zero.

The solenoids must be driven from 40 ± 4 volts with a peak current of 3.6 A. An interface circuit is required at the microcomputer's output to provide this drive capability.

There are two ways of interfacing the printer to a microcomputer. These are

1. Direct microcomputer control
2. Indirect microcomputer control using a special chip called the Printer Controller

The direct microcomputer control interfaces the printer via its I/O ports and utilizes mostly software. The microcomputer performs all the functions required for printing the alphanumeric characters.

Indirect microcomputer control, on the other hand, utilizes a printer control chip such as the Intel 8295 Dot Matrix Printer Controller. The benefits of each technique depend on the specific application.

The direct microcomputer approach provides an inexpensive interface and can be appropriate when the microcomputer has a light load. The indirect microcomputer approach, on the other hand, may be useful when the microcomputer has a heavy load and cost is not a major concern.

9.3.1 LRC7040 Printer Interface Using Direct Microcomputer Control

The steps involved in starting a printing sequence by the microcomputer are provided below:

1. The microcomputer must turn the Main Drive motor (MDM) ON by sending a HIGH output to the MDM.
2. The microcomputer is required to detect a HIGH at the HOME microswitch. This will ensure that the printhead is at the left-hand margin of the print area.
3. The microcomputer is then required to send five bytes of data for an alphanumeric character in sequence to energize the solenoids. Each solenoid requires a pulse of about 400 ms to generate a dot on the paper. A pause of about 900 ms is required between these pulses to provide a space between dots.

Figure 9.10 shows a block diagram interfacing the LRC7040 printer to an MC68000/6821/6116/2716-based microcomputer.

Using the hardware of Figure 9.10, an MC68000 assembly program can be written to send the start pulse for the main drive motor, detect the HOME microswitch, and then, by utilizing the timing requirements of 400 μs and 900 μs of the printer, a hexadecimal digit (0 to F) stored in a RAM location can be printed. An MC68000 assembly language program for printing the character C is shown in Figure 9.11 assuming the 68000 user mode so that USP can be initialized.

The program assumes a look-up table which stores the 5-byte code for the character C starting at $003000. Furthermore, the program assumes that the delay routines DELAY400 for 400 μs and DELAY900 for 900 μs are available. The program prints only one character C and then stops. The program is provided for illustrating the direct microcomputer control technique for printing.

9.3.2 LRC7040 Printer Interface to a Microcomputer Using the 8295 Printer Controller Chip

With direct microcomputer control, the microcomputer spends time in a "wait" loop for polling the status of the HOME signal from the LRC7040 printer. In order to unload the microcomputer of polling the printer status and other functions, typical LSI printer controller chips such as the Intel 8295 can be used.

The 8295 is a dot matrix printer controller. It provides an interface for microprocessors such as the 8085 and 8086 to dot matrix impact printers such as the LRC7040. The 8295 is packaged in a 40-pin DIP and can operate in a serial or parallel communication mode with the 8085 or 8086. In parallel mode, command and data transfers to the printer by the processor occur via

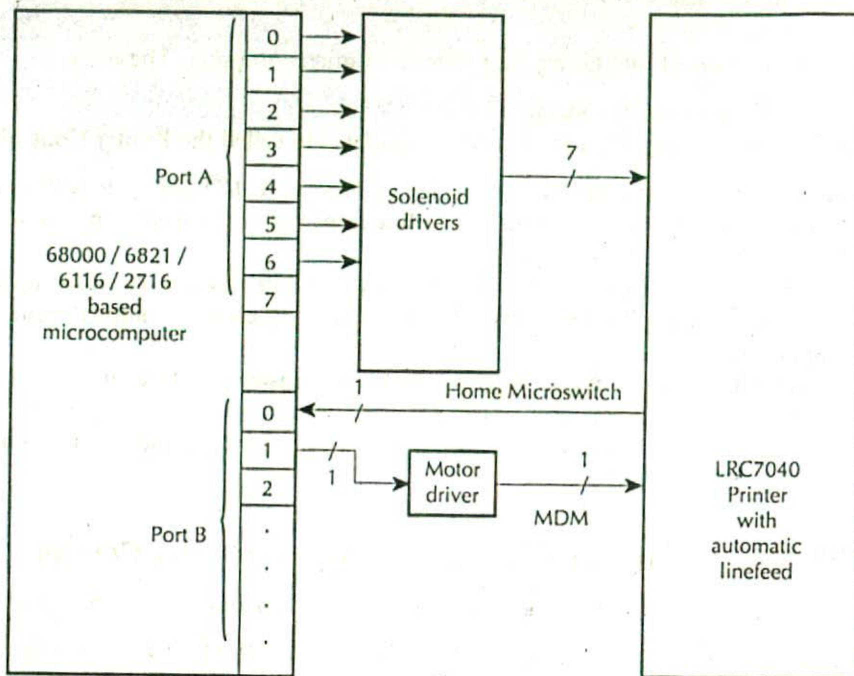


FIGURE 9.10 MC68000-based microcomputer interface to the printer.

polling, interrupts, or DMA using commands. The processor specifies the format of the printed character and controls all printer functions such as linefeed and carriage return.

The 8295 includes a 40-character buffer. When the buffer is full or a carriage return is received, a line is printed automatically. The 8295 has the buffering capability of up to 40 characters and contains a 7 × 7 matrix character generator, which includes 64 ASCII characters. The mode selection (serial or parallel) is not software programmable and is inherent in system hardware. For example, by connecting the 8295 $IRQ/\overline{SER}$ pin to ground, the serial mode is enabled; otherwise the parallel mode is enabled. The two modes cannot be mixed in a single application. Note that the $IRQ/\overline{SER}$ pin is also the 8295 interrupt request to the processor in the parallel mode.

9.3.2.a 8295 Parallel Interface

Two 8295 registers (one for input and the other for output) can be accessed by the processor in the parallel mode. The registers are selected as follows:

$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	Register Selected
1	0	0	Input data register
0	1	0	Output status register

Two types of data can be written in the input data register by the processor:

1. A command to be executed. The command can be 0XH or 1XH. For example, the command 08H will enable DMA mode. On the other hand, the command 11_{16} will enable normal left-to-right printing for printers whose printhead home is on the right.
2. A character data (defined in the 8295 data sheet) such as 37H for '7' or 41H for 'A' to be stored in the character buffer for printing.

The 8295 status is available in the output status register at all times. Typical status bits indicate whether the input buffer is full or DMA is enabled. For example, the IBF (Input buffer

```
            ORG $500000                  ;   Address DDRA
            BCLR.B#$2, CRA               ;   Configure Bits 0-6
            MOVE.B # $7F, DDRA           ;   As outputs and access Port A
            BSET.B # $2, CRA             ;   Address DDRB
            BCLR.B # $2, CRB             ;   Configure bit 0 and
            MOVE.B # $02, DDRB           ;   Bit 1 of Port B as input
            BSET.B # $2, CRB             ;   and output respectively
                                         ;   and access Port B
            MOVEA.L # $040000, A1        ;   Initialize
            MOVE.L A1, USP               ;   USP
            MOVE.B # $02, PORT B         ;   Turn MDM on
HOME        MOVE.B PORT B, D1            ;   Input HOME switch
            ANDI.B # $01, D1             ;   Wait for HOME
            BEQ HOME                     ;   Switch to be on
            MOVEA.L # $3000, A2          ;   Move starting address
                                         ;     of character
            MOVE.B $05, D3               ;   Initialize
                                         ;   Character
                                         ;   Counter
CHARACTER   MOVE.B # $00, PORT A         ;   Generage
            MOVE. B (A2)+, PORT A        ;   Solenoid pulses
            CALL DELAY 400               ;   Generate 400µs
            MOVE.B # $00, PORT A         ;   Pulses
            CALL DELAY900                ;   Delay 900µs
            SUBQ.B # $01, D3             ;   Subtract character counter
            BNE CHARACTER                ;   Loop to output all five bytes
            MOVE.B # $00, PORT B         ;   Turn MDM OFF
STOP        JMP STOP                     ;   Halt
            ORG $003000                  ;
            DB $7F, $41, $41, $41, $41   ;   5-byte
            END                          ;   Code for C
```

FIGURE 9.11 Assembly language program for printing the character C.

full; bit 1 of the status register) is set to one whenever data are written to the input data register. When IBF = 1, no data should be written to the 8295. The DE bit (DMA Enabled; bit 4 of the status register) is set to one whenever the 8295 is in DMA mode. Upon completion of the DMA transfer, the DE is cleared to zero.

The 8295 IRQ/$\overline{\text{SER}}$ pin is used for interrupt driven systems. This output is activated HIGH when the 8295 is ready to receive data. Using polling in parallel mode, the 8295 IRQ/$\overline{\text{SER}}$ pin can be input via the processor I/O port and data can be sent to the 8295 input data register.

Using interrupt in parallel mode, the 8295 IRQ/$\overline{\text{SER}}$ pin can be connected to a processor's interrupt pin to provide an interrupt-driven system.

Using polled or interrupt techniques in parallel mode, the processor typically communicates with the 8295 by performing the following sequence of operations in the main program (polled) or service routine (interrupt):

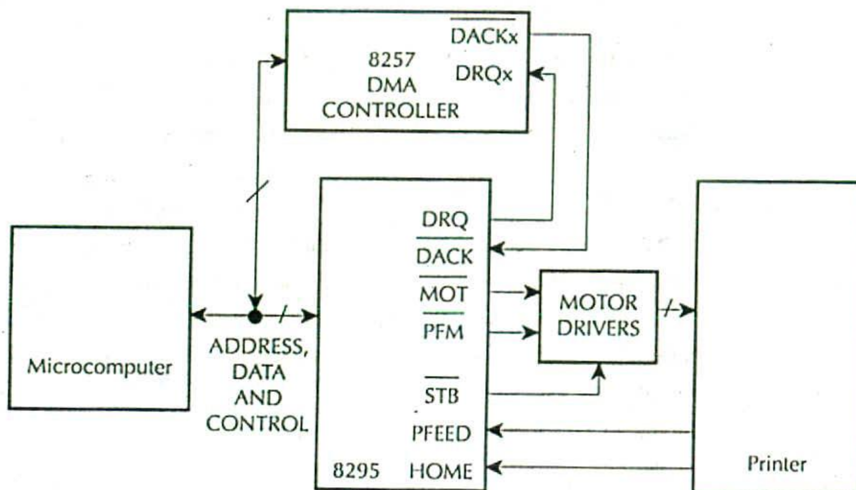


FIGURE 9.12 8295 DMA transfer.

- The processor reads the 8295 status register and checks the IBF flag for HIGH.
- If IBF = 1, the processor waits in a wait loop until IBF = 0. The processor writes data to be printed to the 8295 input data register. The IBF flag is then set to one indicating no data should be written.

Data can also be transferred from the main memory to the 8295 via the DMA method using a DMA controller such as 8257.

The processor initializes the 8257 by sending a starting address and a block length. The processor also enables the 8295 DMA channel by sending it the "ENABLE DMA" command (08H) followed by two bytes specifying the block length to be transferred (low byte first). The 8295 will then activate the DMA request line of the 8257 without any further involvement by the main processor. The DMA enable (DE) flag in the status register will be HIGH until the data transfer is completed. As soon as the data transfer is completed, the DE flag is cleared to zero and IRQ/$\overline{\text{SER}}$ is set to HIGH. The 8295 then goes back to the non-DMA mode of operation.

Figure 9.12 shows a block diagram of the 8295 DMA transfer.

Typical control signals between the 8257 and the processor include HOLD, HLDA, $\overline{\text{RD}}$, and $\overline{\text{WR}}$. The 8295 control signals for the processor include $\overline{\text{CS}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, RESET, and IRQ/SER. $\overline{\text{CS}}$, $\overline{\text{RD}}$, and $\overline{\text{WR}}$ pins are used to select either 8295 input or output registers. The 8295 control signals for the printer include $\overline{\text{MOT}}$, $\overline{\text{PFM}}$, $\overline{\text{STB}}$, PFEED, and HOME.

The 8295 $\overline{\text{MOT}}$ output pin, when LOW, drives the motor. The MOT output is automatically in LOW on power-up. This will make the 8295 HOME input pin HIGH, indicating that the printhead of the printer is in HOME position.

The $\overline{\text{PFM}}$ signal, when LOW, drives the paper feed motor, and this is LOW on power-up. The PFEED is an 8295 input and indicates status of paper feed. A LOW on the $\overline{\text{PFM}}$ indicates that the paper feed mechanism is 'disabled' and a one indicates that the S1 through S7 signals, when LOW, drive the seven solenoids of the printer. Each character datum, when written into the 8295 input data register, is automatically converted to the five-byte code by the 8295 and provides the proper ON/OFF sequence for the solenoids. The $\overline{\text{STB}}$ output is used to determine the duration of solenoid activation and is automatically provided by the 8295.

9.3.2.b 8295 Serial Mode

The 8295 serial mode is enabled by connecting the IRQ/$\overline{\text{SER}}$ pin to LOW. The serial mode is enabled immediately upon power-up. The serial baud rate is programmed by the D2, D1,

D0 data lines. For example, D2 D1 D0 = 001 means 150 baud (bits/sec) and is used to set the serial transfer data rate. In this mode, $\overline{RD}$ must be tied to high and $\overline{CS}$ and $\overline{WR}$ must be tied to ground. The processor needs a UART (Universal asynchronous receiver transmitter) such as the 8251. The 8295 $\overline{DACK}$/SIN signal (data input for serial mode) must be connected to the 8251 TXD output (8251 transmit data output bit). Also, the 8295 DRQ/$\overline{CTS}$ (clear to send in serial mode) must be connected to the 8251 $\overline{CTS}$ output. Note that a UART chip converts parallel to serial data and vice versa. The processor must wait for 8295 $\overline{CTS}$ to go LOW before sending data via TXD.

9.4 CRT (Cathode Ray Tube) Controller and Graphics Controller Chips

The CRT terminal is extensively used in microcomputer systems as an efficient man-machine interface. The users communicate with the microcomputer system via the CRT terminal. It basically consists of a typewriter keyboard and a CRT display. In order to relieve the microprocessor from performing the tedious tasks of CRT control, manufacturers have designed an LSI chip called the CRT Controller. This chip simplifies and minimizes the cost of interfacing the CRT terminal to a microcomputer.

The CRT controller supports all the functions required for interfacing a CRT terminal to a microprocessor. The microprocessor and the CRT controller usually communicate via a shared RAM. The microprocessor writes the characters to be displayed in this RAM; the CRT controller reads this memory using DMA and then generates the characters on the video display. The CRT controller provides functions such as clocking and timing, cursor placement, and scrolling. The CRT controller chip includes several registers that can be programmed to generate timing signals and video interface signals required by a terminal. The display functions are driven by clock pulses generated from a master clock. The CRT controller chip normally produces a special symbol such as a blinking signal or an underline on the CRT. This signal is commonly called the 'cursor'. It can be moved on the screen to a specific location where data need to be modified. The scrolling function implemented in the CRT controller moves currently displayed data to the top of the screen as new data are entered at the bottom.

In this section, fundamentals of CRT, character generation techniques, and graphics controllers are discussed.

A typical CRT controller such as the Intel 8275 is then considered to illustrate its basic functions. Finall,, the graphics functions provided by Intel 82786 are covered.

9.4.1 CRT Fundamentals

A CRT consists of an evacuated glass tube, a screen with an inner fluorescent coating, and an electron gun for producing electron beams. When the electrons generated by the gun are focused on the fluorescent inner coating of the screen, an illuminated phosphor dot is produced.

The position of the dot can be controlled by deflecting the electron beam by using an electromagnetic deflection technique. A complete display is produced by moving the beam horizontally and vertically across the entire surface of the screen and at the same time by changing its intensity.

Most modern CRT terminals generate the display by using horizontal and vertical scans. In the horizontal scan, the beam moves from the upper left-hand corner to the extreme right-hand of the line and thus travels across the screen. The beam then goes off and starts at the left of the next lower line for another scan. After several horizontal scans, the beam reaches the bottom of the screen to complete one vertical scan. The beam then disappears from the screen and begins another vertical scan from the top. This type of scan is also called 'raster' scan. This

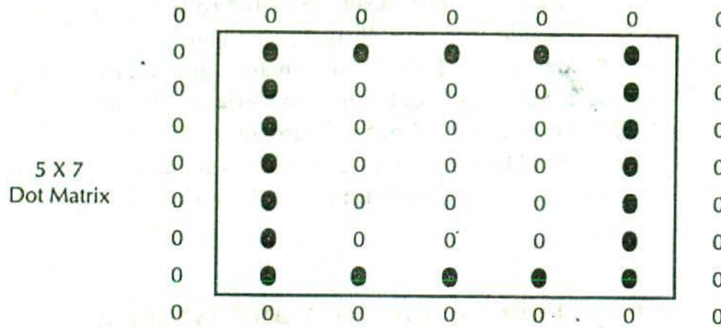


FIGURE 9.13 Generation of '0' using 5 × 7 dot matrix.

is because the display is produced on the screen by continuously scanning the beam across the screen for obtaining a regular pattern of closely spaced horizontal lines, or raster covering the entire screen. One of the most common examples of a raster display is the home TV set. The typical bandwidth used in these TV sets is 4.5 MHz. The raster displays used with microcomputers include a wider bandwidth from 10 MHz to 20 MHz for displaying detailed information. In most modern CRT terminals, each sweep field contains the entire picture or text to be displayed.

In order to display characters, the screen is divided by horizontal and vertical lines into a dot matrix. A matrix of 5 × 7 or 7 × 9 dots is popular for representing a character. For example, a 5 × 7 dot matrix can be used to represent the number '0' as shown in Figure 9.13.

To provide space around the character, one top, one left, one right, and one bottom line are left blank. Each character is generated using 5 × 7 dot matrix. Therefore, each character requires 35 dots, which can be turned ON or OFF depending on the dot pattern required by the character. The pattern of dots is usually stored in ROM. A ROM pattern for '0' is shown in Figure 9.14.

One character requires a 35-bit word. Each row is addressed by three bits. After reading each row data, it is transferred to a parallel to serial shift register. These data are then shifted serially by a clock to the CRT. For a standard 64-character set with each character represented by a 5 × 7 dot matrix, a total of 2240-bit (64 × 7 × 5) ROM is required. Each character in the 64 (2^6)-

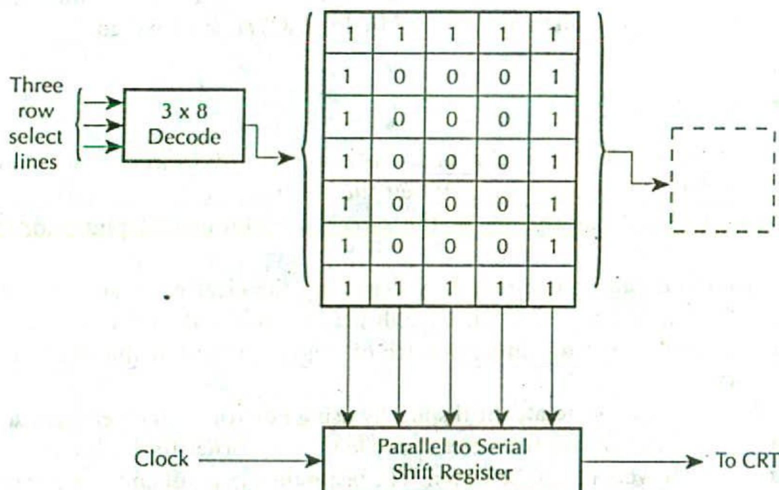


FIGURE 9.14 ROM pattern for '0'.

character set can be addressed using 6 address lines and three row select or scan lines (row select counter typically used) are required to identify the dot row of the character. The ROM, addressing logic, and parallel to serial shift register are referred to as a character generator. Also, a memory known as display memory is required in the CRT to store the character data to be displayed. When a character is entered via the CRT keyboard, it is stored in the display memory.

Graphics can display any figure on the CRT screen. An example of such a VLSI chip is the Intel 82786. In this chip, linked lists are used to update the display and can thus generate displays at a high speed.

Most modern graphics use the bit mapping technique rather than character generation. In order to understand bit mapping, consider a CRT screen as divided into 512 by 128 dots. Each dot is called a Pixel, or picture element, which can be illuminated by an electron beam. Each dot is a single bit in a 64K ($512 \times 128 = 65{,}536$) by 1 RAM and is called a bit plane. If a '1' is stored in a specific bit location, the associated Pixel is turned ON (white). On the other hand, if a '0' is stored, the corresponding Pixel is turned OFF (black). The video refresh circuitry implemented in the VLSI chip converts the ones and zeros in the bit plane to whites and blacks on the CRT screen.

Resolution is an important factor to be considered in graphics. In order to provide various colors and intensity, more than one bit is utilized in representing a Pixel. For example, Apple's 68000-based LISA microcomputer uses four bits per Pixel on a 364×720 Pixel screen. Therefore, a high speed RAM of over 1 megabit ($364 \times 720 \times 4 = 1{,}048{,}320$) is required to support such a resolution.

Therefore, graphics generation requires a bit-mapped RAM array and the LSI video interface chip. The software involves determining the information written to the bit plane array to generate the desired graphic display. Most graphics systems generate figures by combinations of straight-line segments. The software is required to generate a straight line by identifying each Pixel and writing information to its corresponding bit-map positions.

The concepts associated with CRT controllers and graphics described above are illustrated by using the Intel 8275 and Intel 82786 in the following.

9.4.2 Intel 8275 CRT Controller

The INTEL 8275 is a single chip (40-pin) CRT controller. It provides the functions required to interface CRT raster scan displays with Intel microcomputer systems using the 8051, 8085, 8086, and 8088. It refreshes the display by storing (buffering) the information to be displayed from memory and controls the display position on the screen. The 8275 provides raster timing, display row buffering, visual attribute decoding, cursor timing, and light pen detection. The 8275 can be interfaced with the Intel 8257 DMA controller and character generator ROM for dot matrix decoding.

Figure 9.15 shows the 8275's interface to a microcomputer system and the 8257.

The 8275 obtains display characters from memory and displays them on a row-by-row basis. There are two row buffers in the 8275. It uses one row for display, and at the same time fills the other row with the next row of characters to be displayed. The number of display characters per row and the number of character rows are software programmable.

The 8275 utilizes the 8257 DMA controller to fill the row buffer that is not being used for display. It displays character rows one line at a time.

The 8275 controller provides visual attribute codes such as graphics symbols, without the use of the character generator, and blinking, highlighting, and underlining of characters. The raster timing is controlled by the 8275. This is done by generating the horizontal retrace and vertical retrace signals on the HRTC and the VRTC pins.

The 8275 provides the light pen input and associated registers. The light pen input is used to read the registers. A command can be used to read the light pen registers. The light pen consists

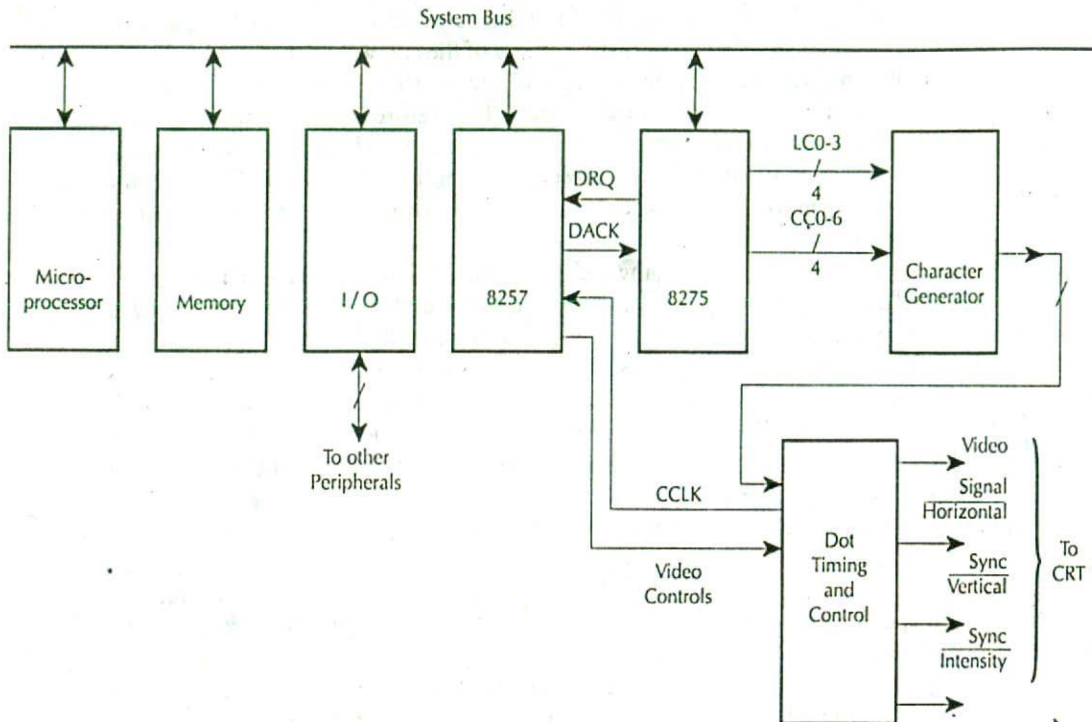


FIGURE 9.15 Microcomputer/8275/8257 interface.

of a microswitch and a tiny light sensor. When the light pen is pressed against the CRT screen, the microswitch enables the light sensor. When the raster sweep reaches the light sensor, it enables the light pen output. If the output of the light pen is presented to the 8275 LPEN pin, the row and character position coordinates are stored in a pair of registers. These registers can be read by a command. A bit in the status register in the 8275 is set, indicating detection of the light pen signal. The 8275 can generate a cursor. The cursor location is determined by a cursor row register and a character position register which are added by a command to the controller.

The cursor can be programmed to appear on the display in many forms such as a blinking underline and a nonblinking underline. The 8275 does not provide a scrolling function.

The 8275 outputs the line count (LC0-LC3) and character code (CC0-CC6) signals for the character generation. The LC0-LC3 signals are contents of the 8275 line counter which are used to address the character generator for the line positions on the screen. The CC0-CC6 outputs of the 8275 are the contents of the row buffers used for character selection in the character generator.

The 8275 video control signals typically include line attribute codes, highlight, and video suppression. The two line attribute codes (LA0 and LA1 pin outputs) must be decoded by the dot timing logic to produce the horizontal and vertical line combinations for the graphic displays defined by the character attribute codes. The video suppress (VSP pins) output signal is used to blank the video signal to the CRT. The highlighted (HLGT) output signal is used to intensify the display at a specific position on the screen, as defined by the attribute codes.

The dot timing and interface logic must provide the character clock (CCLK pin) input of the 8275 for proper timing.

9.4.3 Intel 82786 Graphics Controller

The Intel 82786 is a single VLSI chip providing bitmapped graphics. It is designed for microcomputer graphics applications, including personal computers, engineering worksta-

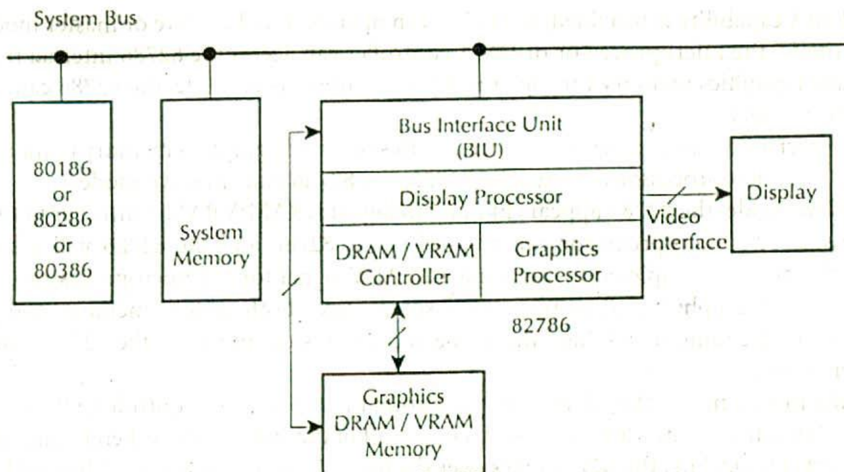


FIGURE 9.16 80186/80286/80386 interface to 80786.

tions, terminals, and laser printers. The 82786 is designed using Intel's CHMOS III process. It is capable of both drawing and refreshing raster displays. It supports high resolution displays with a 25-MHz Pixel clock and can display up to 256 colors simultaneously. It can be interfaced to all Intel microcomputers such as 80186, 80286, and 80386. Figure 9.16 shows a block diagram of the 80186/80286/80386 interfaces to the 82786.

The 82786 includes three basic components. These are a display processor (DP), a graphics processor (GP), and a bus interface unit (BIU), with a DRAM/VRAM controller.

The display processor controls the CRT timing and provides the serial video data stream for the display. It can assemble several windows (portions of bitmaps) on the screen from different bitmaps scattered across the memory accessible to it.

The graphics processor executes commands from a graphics command block placed in memory by the 80186/80286/80386 and updates the bitmap memory for the display processor. The graphics processor has high level video display interface-like commands and can draw graphical objects and text at high speeds.

The BIU controls all communication between the 82786, 80186/80286/80386, and memory. The BIU contains a DRAM/VRAM controller that can perform block transfers. The display processor and graphics processor use the BIU to access the bitmaps in memory.

The system bus connects the 80186/80286/80386 and system memory to the 82786. The video interface connects the 82786 to the CRT or other display. The video interface is controlled directly by the display processor. The 82786 can be programmed to generate all the CRT signals for up to 8 bits/Pixel (256 colors) displays. The other interfaces are controlled by the BIU. The BIU interfaces the graphics and display processors to the 80186/80286/80386 and system memory as well as the graphics memory via the internal DRAM/VRAM controller.

The dedicated graphics DRAM/VRAM memory provides the 82786 with fast access to memory without contention with the microprocessor and system memory.

Usually, the bitmaps to be drawn and displayed, the characters, and commands for the 82786 are all stored in this memory. The 82786 DRAM/VRAM controller interfaces directly with a number of dynamic RAMS without external logic.

Figure 9.15 shows the most common configuration. The microprocessor can access the system memory, while the 82786 accesses its dedicated graphics memory simultaneously. However, when the microprocessor accesses the graphics memory, the 82786 cannot access the system memory. Also, when the 82786 accesses the system memory, the microprocessor cannot access the graphics memory.

If DMA capability is provided, the 82786 can operate in either slave or master mode. In the slave mode, the microprocessor or DMA controller can access the 82786 internal registers or dedicated graphics memory through the 82786. In the master mode, the 82786 can access the system memory.

The microprocessor software can access both system and graphics memory in the same way. When the microprocessor accesses the 82786, the 82786 runs in slave mode.

In slave mode, the 82786 appears like an intelligent DRAM/VRAM controller to the microprocessor. The microprocessor can chip-select the 82786 and the 82786 will acknowledge when the cycle is completed by asserting a READY signal for the microprocessor.

The 82786 graphics and display processor accesses both system memory and graphics memory in the same way. When the 82786 accesses system memory, the 82786 must run in master mode.

In the master mode, the 82786 acts as a second microprocessor controlling the system bus. The 82786 activates the HOLD line to take control of the system bus. When the microprocessor asserts HLDA line, the 82786 takes over the bus. When the 82786 is finished with the bus, it will disable the HOLD line and the microprocessor can remove HLDA and take over the bus.

The 82786 provides two different video interfaces when using standard DRAMS. The 82786 reads the video data from memory and internally serializes the video data to generate the serial video data stream. When using VRAMS, the 82786 loads the VRAM shift register. Periodically, the shift register and external logic then generate the serial video data stream.

9.5 Coprocessors

In Chapters 1 and 7, the basics of coprocessors, along with the functions provided by Motorola coprocessors such as the MC68851 and MC68881, are covered. In this section, a brief overview of the Intel coprocessors will be provided.

Intel offers a number of coprocessors which include numeric coprocessors such as 8087/80287/80387, DMA coprocessors such as the 82258, and graphic coprocessors such as the 82786. In the following, a brief overview of Intel's numeric coprocessors, which include the 8087, 80287, and 80387, is given.

9.5.1 Intel 8087

Intel 8087 numeric data coprocessor is designed using HMOS III technology and is packaged in a 40-pin DIP. When an 8087 is present in a microcomputer system, it adds 68 numeric processing instructions and eight 80-bit registers to the microprocessor's register set. The 8087 can be interfaced to Intel microprocessors such as the 8086/8088 and 80186/80188.

The 8087 supports seven data types which include 16-, 32-, and 64-bit integers, 32-, 64-, and 80-bit floating point, and 18-digit BCD operands. The 8087 is compatible with the IEEE floating-point format. It includes several arithmetic, trigonometric, exponential, and logarithmic instructions.

The 8087 is treated as an extension to the microprocessor, providing additional register data types, instructions, and control at the hardware level. At the programmer's level, the microprocessor and the 8087 are viewed as a single processor. For the 8086/8088, the microprocessor's status (S0-S2) and queue status lines (QS0-QS1) enable the 8087 to monitor and decode instructions in synchronization. For 80186/80188 systems, the queue status signals of the 80186/80188 are synchronized to the 8087 by the 8288 bus controller. The 8087 can operate in parallel with and independent of the microprocessor. For resynchronization, the 8087's BUSY pin tells the microprocessor that the 8087 is executing an instruction and the microprocessor's WAIT instruction tests this signal to ensure that the 8087 is ready to execute subsequent instructions. The 8087 can interrupt the microprocessor when it detects an error

or exception. The 8087's interrupt register line is typically connected to the microprocessor through an 8259 programmable interrupt controller for 8086/8088 systems and INT for 80186/80188 systems.

The 8087 uses the request/grant lines of the microprocessor to gain control of the system bus for data transfer.

The microprocessor controls overall program execution while the 8087 utilizes the coprocessor interface to recognize and perform numeric operations.

9.5.2 Intel 80287

Intel 80287 is an enhanced 8087 that extends the 80286 microprocessor. The 80287 adds over 50 instructions to the 80286 instruction set. The 80287 is designed using HMOS technology and is housed in a 40-pin special package called 'CERDIP'.

The 80287 supports the IEEE floating-point format. The 80287 expands the 80286 data types to include 32-bit, 64-bit, and 80-bit floating point, 32-bit and 64-bit integers, and 18-digit BCD operands. It extends the 80286 instruction set to trigonometric, logarithmic, exponential, and arithmetic instructions for all data types.

The 80287 executes instructions in parallel with an 80286. The 80287 has two operating modes like the 80286. Upon reset, the 80287 operates in real address mode. It can be placed in the protected virtual address mode by executing an instruction on the 80286. The 80287 cannot be placed back to the real address mode unless reset. Once in protected mode, all memory references for numeric data or status information follow the 80286 memory management and protection rules and thus the 80287 extends the 80286 protected mode.

The 80287 receives instructions and data via the data channel control signals ($\overline{PEREQ}$ — Processor Extension Data Channel Operand Transfer Request); $\overline{PEACK}$ — Processor Extension Data Channel Operand Transfer Acknowledge; $\overline{BUSY}$; $\overline{NPRD}$ — Numeric Processor $\overline{RD}$; $\overline{NPWR}$ — Numeric Processor $\overline{WR}$). When in protected mode, all information received by the 80287 is validated by the 80286 memory management and protection unit. When the 80287 detects an exception, it will indicate this to the 80286 by asserting the ERROR signal.

The 80286/80287 is programmed as a single processor. All memory addressing modes, physical memory, and virtual memory of the 80286 are available in the 80287.

9.5.3 Intel 80387

Intel 80387 is a numeric coprocessor that extends the 80386 processor with floating-point, extended integer, and BCD data types. It is compatible with IEEE floating point. The 80387 includes 32-, 64-, and 80-bit floating point, 32- and 64-bit integers, and 18-digit BCD operands. It extends the 80386 instruction set to include trigonometric, logarithmic, exponential, and arithmetic instructions of all data types. The 80387 can operate in the real, protected, or virtual 8086 modes of the 80386. It is designed using CHMOS III technology and is packaged in a 68-pin PGA (Pin Grid Array).

The 80387 operates in the same manner whether the 80386 is executing in real address mode, protected mode, or virtual 86 mode. All memory access is handled by the 80386; the 80387 operates on instructions and values passed to it by the 80386. Therefore, the 80387 is independent of the 80386 mode.

The 80387 includes three functional units that can operate in parallel. The 80386 can be transferring commands and data to the 80387 bus control logic for the next instruction while the 80387 floating-point unit is performing the current numeric instruction. This parallelism improves system performance.

The 80387 adds to an 80386 system additional data types, registers, instructions, and interrupts. All communication between the 80386 and 80387 is transparent to application software. Thus, the 80387 greatly enhances the 80386 capabilities.

Questions and Problems

9.1 Interface a hexadecimal matrix keyboard and four LED displays to an 8086/8255-based microcomputer.

i) Draw a hardware schematic of the design. Show only the pertinent signals.

ii) Write an 8086 assembly language program to display the hex digit on the display from 0-F each time a digit is pressed on the keyboard.

9.2 Describe the basic functions of a DMA Controller. How does it control the I/O $R/\overline{W}$ and memory $R/\overline{W}$ signals? Why is the DMA Controller faster than the microprocessor for data transfer?

9.3 Describe briefly the main features of Motorola's MC68440 DMA controller.

9.4 Draw a functional block diagram showing the pertinent signals of the MC68020/68230/68440 interface.

9.5 Define the MC68440 modes of operation.

9.6 Which mode and which address lines are required by the MC68440 to decode the register addresses? Why does the MC68440 require more address lines than it requires for register address decoding?

9.7 Draw a functional block diagram of the MC68440/68008 interface.

9.8 What is the difference between the following?

i) Serial and Parallel printers

ii) Impact and Nonimpact printers

iii) Character and Matrix printers

9.9 Assume an LRC7040 printer. Draw a functional block diagram of the LRC7040 printer to an 8086-based microcomputer. Write an 8086 assembly language program to print the hexadecimal digit '0' on the printer.

9.10 Draw a functional block diagram of the 8295 printer controller interface to an 8085-based microcomputer.

9.11 How are the 8295 input data register and output status registers accessed? What are the functions of these registers?

9.12 How are the 8295 serial and parallel modes of operation selected?

9.13 In the 8295 parallel mode, describe briefly how printers are interfaced for polled, interrupt, or DMA operation.

9.14 Summarize the basics of a CRT. What is the main difference between character generation displays and graphics displays?

9.15 What are the typical functions of a CRT controller? Relate these typical functions to the Intel 8275.

9.16 Draw a functional block diagram showing an 8086-based microcomputer interface to an 8275. Show only pertinent signals.

9.17 What is meant by bitmapping? How does it apply to graphics?

9.18 Describe briefly the functions provided by the Intel 82786 graphics controller.

9.19 Draw a functional block diagram showing an 80386/82786 interface. Show only pertinent signals.

9.20 Summarize the basic differences between the Intel 8087, the 80287, and the 80387 numeric coprocessors. Why are these three separate chips for the same coprocessor family provided by Intel?

10

DESIGN PROBLEMS

This chapter includes a number of design problems that utilize external hardware. The systems are based on typical microprocessors such as the 8085, 8086, and 68000.

The concepts presented can be extended to other microprocessors.

10.1 Design Problem No. 1

10.1.1 Problem Statement

An 8085-based digital voltmeter is designed which will measure a maximum of 5 V DC via an A/D converter and then display the voltage on two BCD displays. The upper display is the integer part (0 to 5 V DC) and the lower display is the fractional part (0.0 to 0.9 V DC).

10.1.2 Objective

A digital voltmeter capable of measuring DC voltage up to and including 5 V will be built and tested. The voltmeter is to be implemented using the Intel 8085 microprocessor and an analog-to-digital converter of the designer's choice. The measured voltage is to be displayed on two seven-segment LEDs.

10.1.3 Operation

Figure 10.1 shows a block diagram of the digital voltmeter. It is composed of the microprocessor, 2K bytes of EPROM, 256 bytes of RAM with I/O, the A/D converter, and the display section.

The Intel 8085 microprocessor provides control over all address, data, and control information involved in program execution. It also provides for manipulation of data as taken from the A/D and sent to the display section.

The EPROM is a memory unit which stores the instructions necessary for system operation. The RAM and I/O section is a memory unit which provides for data storage as well as data transfers to and from the A/D and display. The A/D converter takes the voltage measured across Vin(+) and Vin(–) pins and converts it to an 8-bit binary value. The binary information is taken into the microprocessor via the I/O and converted to its decimal equivalent. The display section takes the converted binary information from the processor so that it may be read on two seven-segment LED displays. The leftmost display provides the integer portion of the measured voltage, while the rightmost provides the decimal portion.

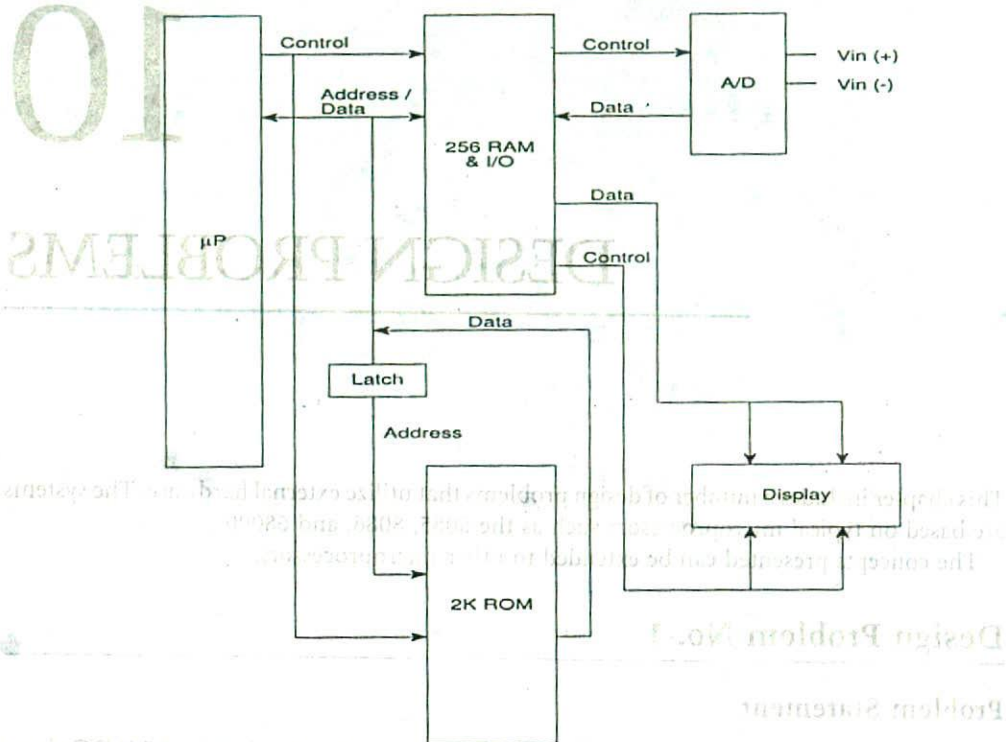


FIGURE 10.1 Voltmeter block diagram.

10.1.4 Hardware

Figure 10.2 shows the detailed hardware schematic. The system uses standard I/O with RAM memory mapped at 0800H-08FFH. The I/O ports of the 8155 are all used and are I/O mapped starting at 08H. Configured as an input port, port B is connected to the output of the 0804 A/D chip. Configured as an output port, port C is connected to the TIL311 displays. Bits 0–3 are the data outputs while bits 4 and 5 are connected to the latches of the TIL311. Only three bits of port A are used, configured as an output port, to control the select, read, and write lines of the A/D chip.

Using the fully decoded memory addressing, the 74LS318 decoder is used to select either the RAM or the EPROM. Also, a 74LS373 is used to latch the address lines for the EPROM. The RAM does not require such a chip because the 8155 RAM has its own internal latches. The ALE line of the 8085 microprocessor controls the latches as seen in the schematic.

The EPROM contains the instructions for converting the binary representation of the analog voltage (applied to the A/D converter) back to decimal representation. The instructions are used to control the system operation. The algorithm uses repeated subtraction to obtain the correct voltage in decimal form. The left display is the integer part and the right display is the decimal part.

The displays, as stated before, are TIL311 hexadecimal displays. In addition, the displays have their own latches which are active low. In the 8085 microprocessor, the interrupt RST 6.5 is used to jump to the address with the algorithm to convert and display the voltage. The $\overline{\text{INTR}}$

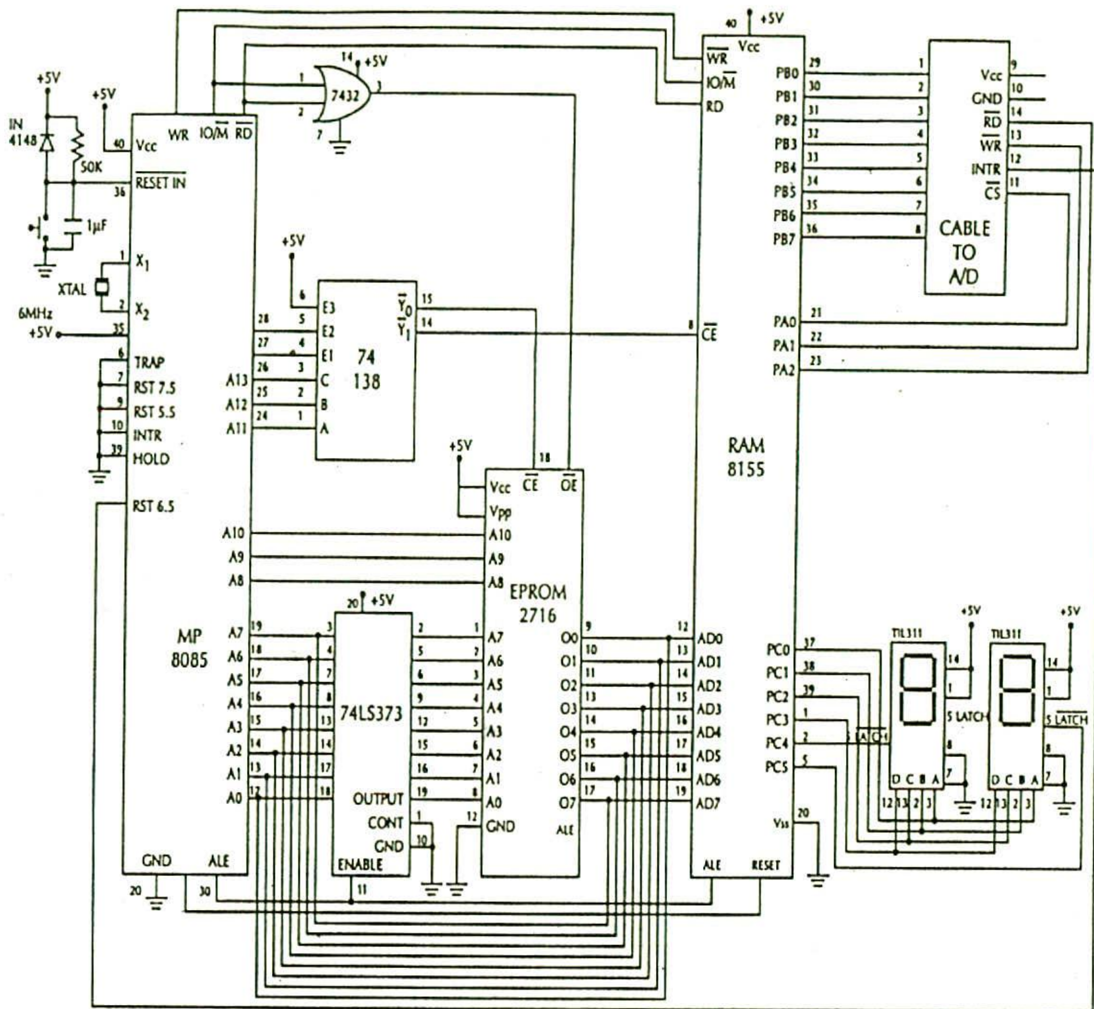


FIGURE 10.2 Digital voltmeter detailed schematic.

pin of the A/D is connected directly to the RST 6.5 pin of the microprocessor. First, an active low is sent to the chip select pin, and then the write pin of the A/D converter is toggled.

Upon completion of the A/D conversion, the 8085 is interrupted. The service routine outputs an active low onto the read pin of the A/D, which latches the data. After inputting the data via the port, the read pin is toggled which then tristates the A/D output.

10.1.5 Software

An important part of the software is to convert the A/D's 8-bit binary data into its decimal equivalent for the display. The decimal data will have two digits: one integer part and one fractional part. Two approaches can be used to accomplish this as follows.

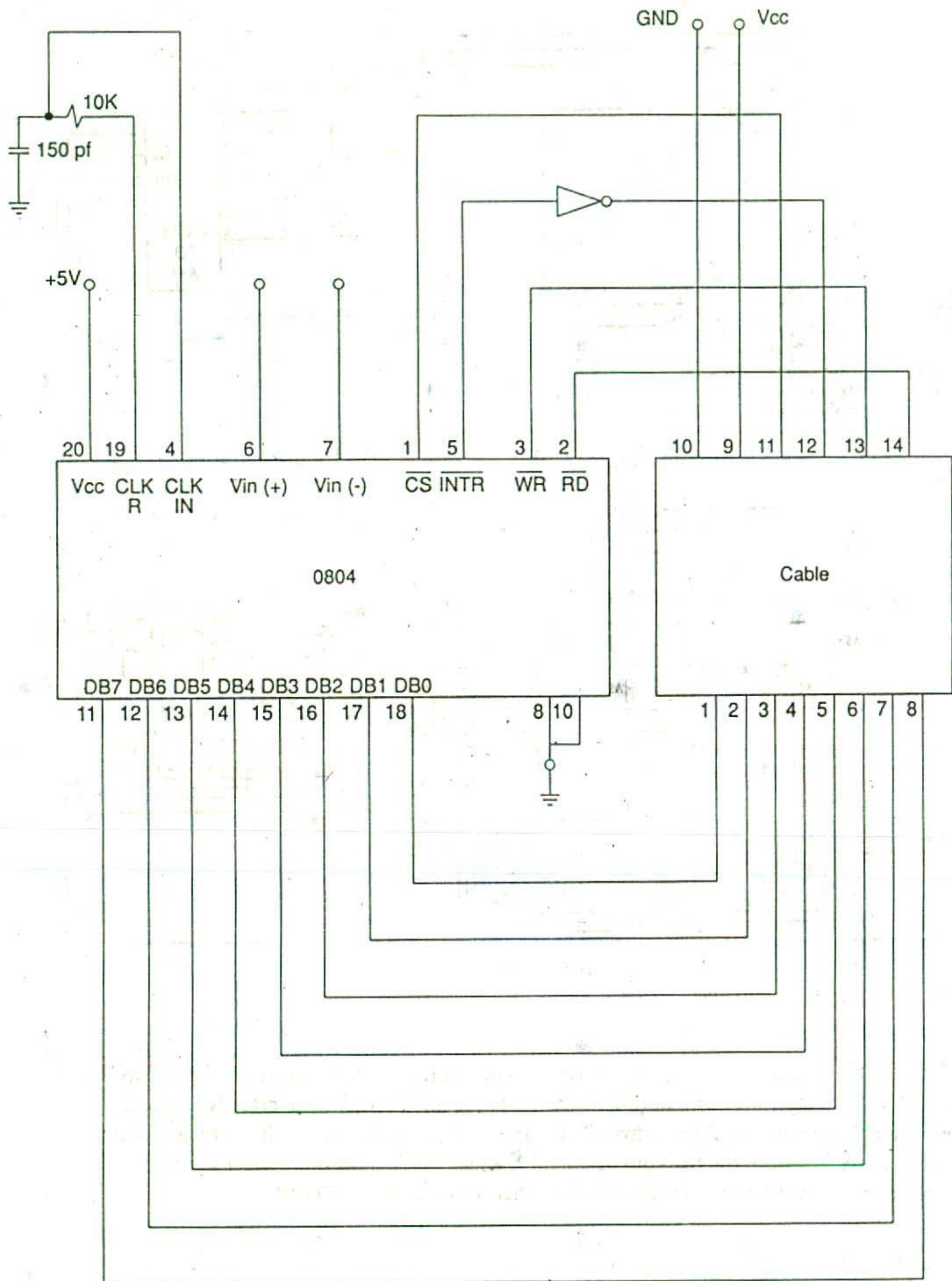


FIGURE 10.2 *Continued.*

Approach 1

Since the maximum decimal value that can be accommodated in 8 bits is 255_{10} (FF_{16}), the maximum voltage of 5 V will be equivalent to 255_{10}. This means that the display in decimal is given by:

```
D = 5*(Input/255)
  = Input/51
  = Quotient + (Remainder/51)
        ↑
  Integer part
```

The fractional part in decimal is

```
F = (Remainder/51)*10
  ≅ Remainder/5
```

Approach 2

In the second approach, the equivalent of 1 V (255/5 V = $51_{10} = 33_{16}$) is subtracted from the input data. If the input data are greater than 1 V, a counter initially cleared to zero is incremented by one. This process continues until the data are less than 1 V. The register keeps count of how many subtractions take place with a remainder greater than 1 V and thus contains the integer portion of the measured voltage in decimal.

The decimal portion of the fractional part is obtained in the same way except that if the input data are less than 1 V, then they are compared with the decimal equivalent (51/10 ≅ 5) of 0.1 V. If the measured data are greater than 0.1 V, a counter initially cleared to zero is incremented by one and the process continues until the input data are less than 0.1 V. The counter contains the fractional part of the display.

Approach 2 is used as a solution to this problem. A complete listing of the 8085 assembly language program to control the digital voltmeter is given below. The program is used to begin and end the A/D conversion process as well as to manipulate the binary data into their decimal form so that they can be displayed in an easily readable format.

```
FILE: LIST1;RAT001  HEWLETT-PACKARD: 8085 Assembler

LOCATION        OBJECT
CODE            LINE            SOURCE LINE

                1  "8085"
<09A0>          2  STACKP   EQU   09A0H
<0009>          3  PORTA    EQU   0009H
<000A>          4  PORTB    EQU   000AH
<000B>          5  PORTC    EQU   000BH
<0008>          6  CSR      EQU   0008H
<0034>          7  INTR     EQU   0034H
<0000>          8  PROG     EQU   0000H
                9
                10          ORG   PROG
0000 3109A0     11          LXI   SP,STACKP   ;  INIT, STACK
0003 3E0D       12 START    MVI   A,0DH       ;  SET INTERRUPT MASK
0005 30         13          SIM               ;  SET INTERRUPT MASK 6.5
0006 FB         14          EI                ;  ENABLE INTERRUPT
0007 D308       15          OUT   CSR         ;  DEFINE PORTS A,B,C
0009 3E30       16          MVI   A,30H       ;  SET DISPLAY ENABLES
000B D30B       17          OUT   PORTC       ;
000D 3EFF       18          MVI   A,0FFH      ;
000F D309       19          OUT   PORTA       ;  SET /CS,/WR/RD HIGH
0011 3EFE       20          MVI   A,0FEH      ;
0013 D309       21          OUT   PORTA       ;  SEND /CS LOW
0015 3EFC       22          MVI   A,0FCH      ;
0017 D309       23          OUT   PORTA       ;
```

```
FILE: LIST1;RAT001 HEWLETT-PACKARD: 8085 Assembler (continued)
LOCATION        OBJECT
CODE            LINE              SOURCE LINE
0019 3EFE       24          MVI    A,0FEH        ;
001B D309       25          OUT    PORTA         ;  TOGGLE /WR
001D 9B         26  WAIT    JMP    WAIT          ;  WAIT FOR INTERRUPT
                27                               ;
                28                               ;
                29          ORG    INTR          ;  INTERRUPT VECTOR
                                                    ADDRESS
0034 210900     30          LXI    H,0900H       ;  INIT, MEM. POINTER
0037 1600       31          MVI    D,00H         ;  INIT, INTEGER COUNTER
0039 0E33       32          MVI    C,33H         ;
003B 3EFA       33          MVI    A,0FAH        ;  SEND /RD LOW
003D D309       34          OUT    PORTA         ;
003F 00         35          NOP
0040 DB0A       36          IN     PORTB         ;  INPUT DATA
0042 47         37          MOV    B,A           ;  MOVE DATA TEMP. TO B
0043 3EFE       38          MVI    A,0FEH        ;
0045 D309       39          OUT    PORTA         ;  TOGGLE /RD
0047 70         40          MOV    M,B
0048 78         41          MOV    A,B           ;  MOVE DATA TO 0900H
0049 91         42  SUB1:   SUB    C             ;
004A DA0052     43          JC     CONT1         ;
004D 14         44          INR    D             ;
004E 77         45          MOV    M,A           ;
004F C30049     46          JMP    SUB1          ;
0052 7A         47  CONT1:  MOV    A,D           ;
0053 D30B       48          OUT    PORTC         ;
0055 E63F       49          ANI    3FH           ;
0057 D30B       50          OUT    PORTC         ;
0059 1600       51          MVI    D,00H         ;
005B 0E05       52          MVI    C,05H         ;
005D 7E         53          MOV    A,M           ;
005E 91         54  SUB2:   SUB    C             ;
005F DA0066     55          JC     CONT2         ;
0062 14         56          INR    D             ;
0063 C3005E     57          JMP    SUB2          ;
0066 JA         58  CONT2:  MOV    A,D           ;
0067 F610       59          ORI    10H           ;
0069 D30B       60          OUT    PORTC         ;
006B E63F       61          ANI    3FH           ;
006D D30B       62          OUT    PORTC         ;
006F 3EFC       63          MVI    A,0FCH        ;
0071 D309       64          OUT    PORTA         ;
0073 3EFE       65          MVI    A,0FEH        ;
0075 D309       66          OUT    PORTA         ;
0077 CB         67          RET                  ;
Errors = 0
```

Lines 2–8 are assembler directives which equate a recognizable label with a hex value. This is useful for values which are to be used throughout the program.

Line 10 is another assembler directive which sets the beginning of the program at address 0000H.

Line 11 initializes the stack pointer at address 09A0H. This is necessary if we are to return to a current program after an interrupt has been serviced.

Lines 12–14 set the mask bits and enable interrupt RST6.5.

Line 15 defines port A as output, port B as input, and port C as output. Note that the data to configure the ports were already in the accumulator as per line 12.

Lines 16–17 send an active high to each display's data latch enable pin. This insures that the displays will output the correct data on the next high-to-low transition at the latch enable pins.

Lines 18–19 send a high to the chip's select ($\overline{CS}$), write ($\overline{WR}$), and read ($\overline{RD}$) pins of the A/D converter. This insures proper start-up of the converter.

Lines 20–21 first send an active low to the converter's $\overline{CS}$ pin. Next, lines 22–25 toggle the $\overline{WR}$ pin so that conversion starts. The combination of $\overline{CS}$ and $\overline{WR}$ active low resets the A/D internally and sets it up for the start of the conversion. By sending $\overline{WR}$ high, the conversion starts. Figure 10.3 shows the timing diagram for the A/D.

Line 26 is a "WAIT LOOP" which is provided as a delay to wait for the interrupt request. This is necessary since it may take as long as 127 μs for the interrupt to be asserted. This is equivalent to approximately 380 clock cycles for the 8085 operating at 3 MHz.

Line 29 continues the program at the interrupt vector for interrupt RST6.5.

Line 30 loads the HL register pair with a memory address to be used later in the program.

Lines 31–32 initialize the D and C registers. D register is to hold the integer portion of the measured voltage, while C register holds a hex value equivalent to 1 V for this system.

Lines 33–34 send an active low to the $\overline{RD}$ pin of the A/D converter so that the binary information corresponding to the measured voltage may be read by the microprocessor.

Lines 36–37 take the data from the A/D converter and store it into register B.

Lines 38–39 toggle the $\overline{RD}$ pin back to active high.

Lines 40–41 move the 8-bit data into memory location 0900H and then into the accumulator.

Lines 42–46 convert the binary data into their decimal equivalent so that the integer portion may be displayed. First the equivalent of 1 V is subtracted from the input data. If the measured voltage is less than 1 V, the program jumps to line 47. If the voltage is greater than one, the program continues at line 44 where register D is incremented by one. The remainder from the subtraction is temporarily stored in memory. The program then unconditionally jumps back to line 42 so that another subtraction takes place. This loop occurs until the remainder from the subtraction is less than 1 V. Register D keeps count of how many subtractions took place with a remainder greater than 1 V and thus counts the integer number of volts measured.

Lines 47–50 send the contents of register D to the leftmost display. The AND operation unlatches the data at the display.

Lines 51–52 again initialize registers D and C, but this time register D will be counting the fractional portion of the measured voltage and register C will hold the hex equivalent of 0.1 V.

Line 53 moves the last positive remainder from memory into the accumulator.

Lines 54–57 perform the same function as lines 42–46 but with the fractional portion of the measured data.

The remaining lines output the contents of register D into the rightmost display, toggle $\overline{WR}$, and returns to the main program.

10.2 Design Problem No. 2

10.2.1 Display Scroller Using the Intel 8086

10.2.1.a Introduction and Problem Statement

The objective of this project is to design and build an 8086-based system as shown in the block diagram of Figure 10.4. The system scans a 16-key keyboard and drives three seven-segment displays. The keyboard is scanned in a 4 × 4 X-Y matrix. The system will take each key pressed

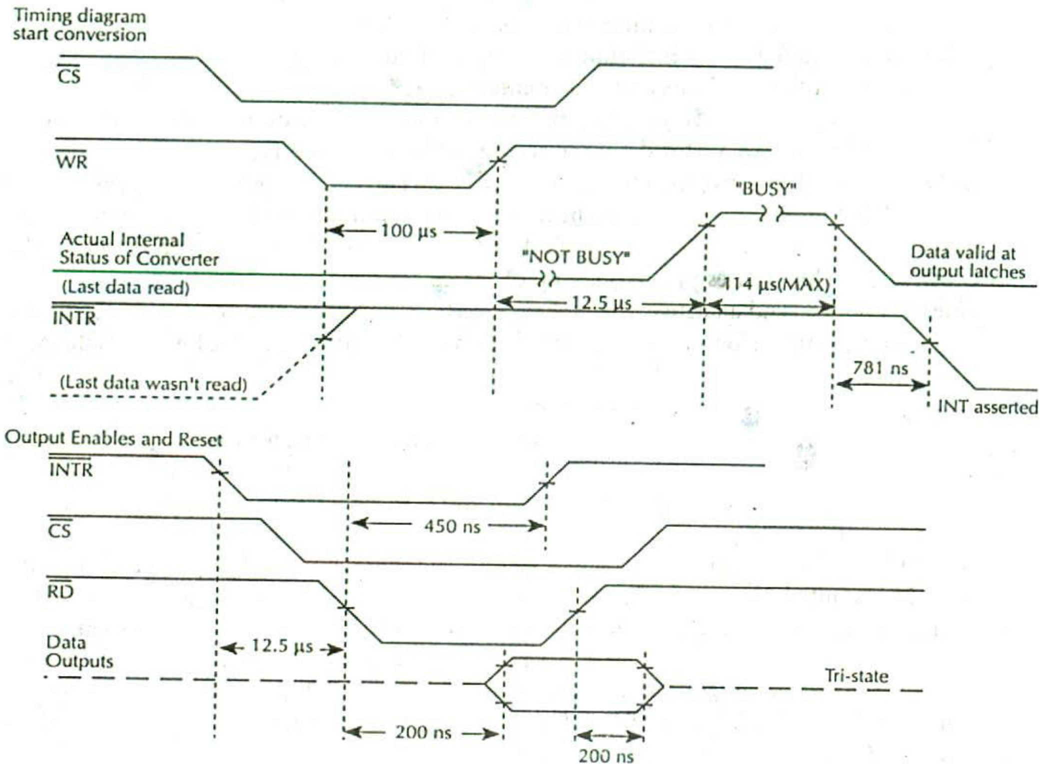


FIGURE 10.3 A/D timing diagram.

and scroll them in from the right side of the displays and keep scrolling as each key is pressed. The leftmost digit is discarded. The system continues indefinitely.

10.2.1.b Hardware Description

Figure 10.5 shows the hardware schematic. The microcomputer is designed using the 8086, 8255 I/O port chip and two 2716 EPROMs. The system does not contain any RAM since no stack is required.

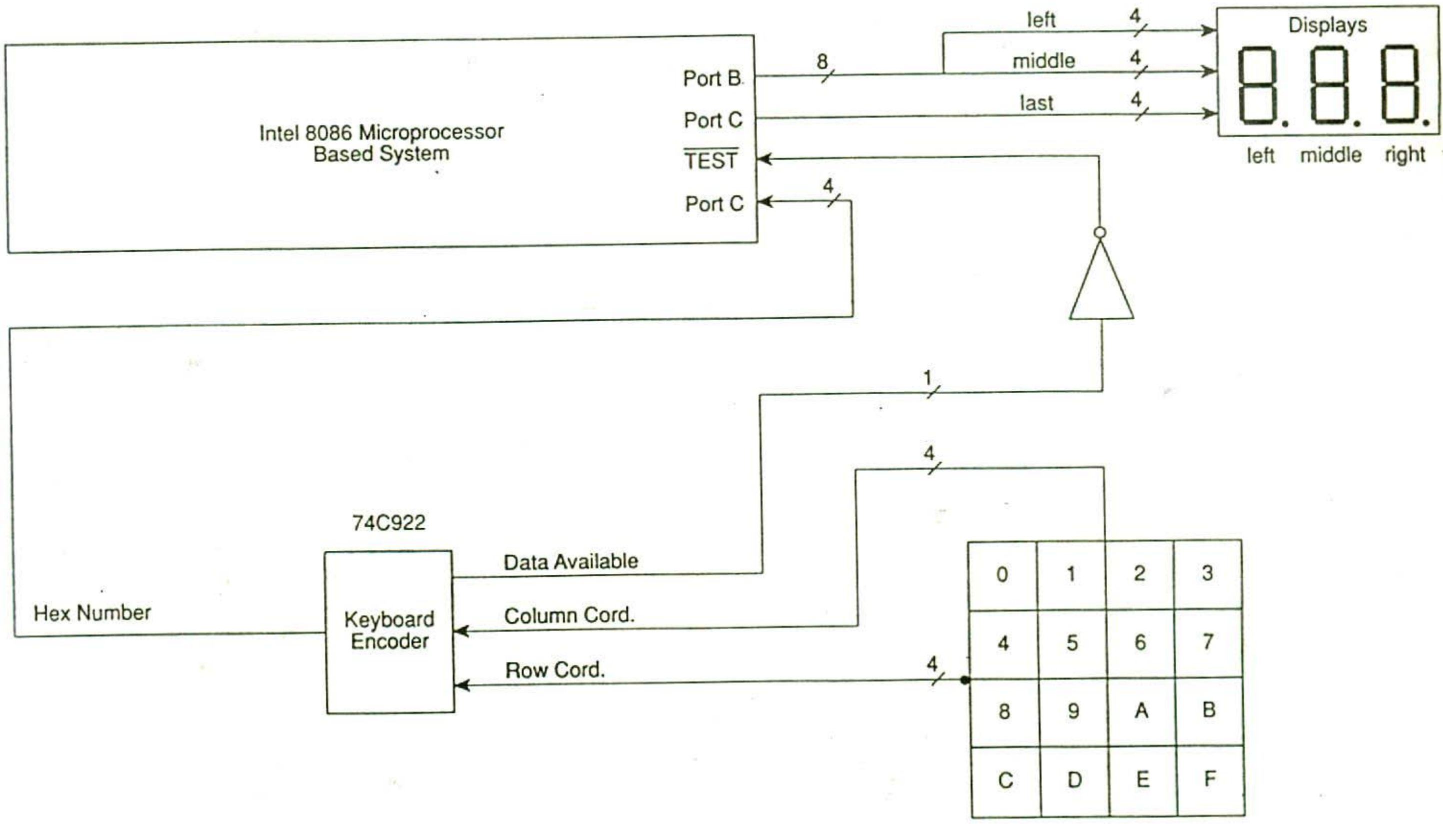


FIGURE 10.4 Keyboard scroller block diagram.

FIGURE 10.5 Keyboard scroller schematic diagram.

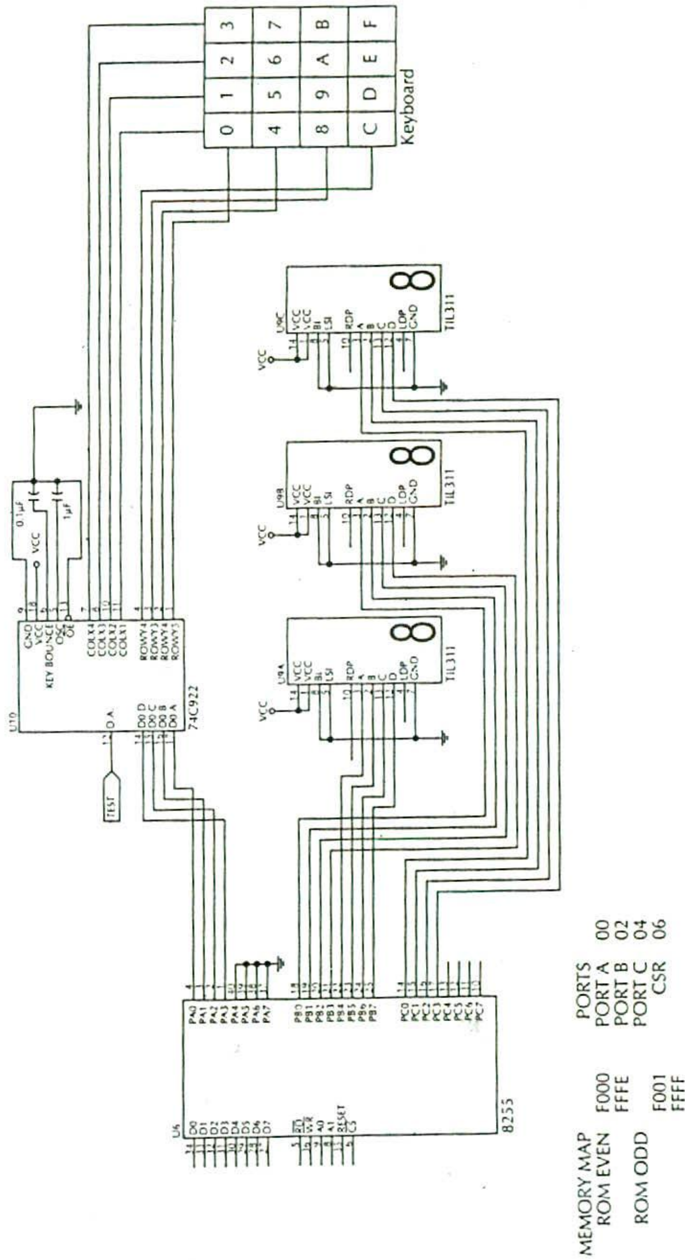


FIGURE 10.5 *Continued.*

```
PROG    SEGMENT
        ASSUME CS:PROG, DS:PROG
        .8086
PORTA   equ  00h
PORTB   equ  02h
PORTC   equ  04h
CSR     equ  06h
start:  mov CX, 0FFFh
LP1:    loop LP1          ;  Delay test.
        mov AL, 90h       ;  set ports A input, B & C output
        out CSR, AL
        xor BL, BL        ;  clear BL
        xor AX, AX        ;  clear AX
        out PORTB, AL     ;  clear Displays
        out PORTC, AL
        mov CL, 04h       ;  set CL to 04
over:   wait              ;  wait for TEST pin to go low.
        in AL, PORTA      ;  Get data
        out PORTC, AL     ;  Out first number and
        shl AL, CL        ;  rotate into place for
        shl AX, CL        ;  2nd and 3rd position.
        mov AL, BL        ;  Move old inputed nibbles
        out PORTB, AL     ;  to be outputed.
        mov BL, AH        ;  Copy AH new to BL.
        mov CX, 0FFFh     ;  Delay so D.A. signal
LP2:    loop LP2          ;  can return to low.
        mov CL, 04h       ;  Reset CL to 04h
        jmp over
PROG    ENDS
        END start
```

FIGURE 10.6a 8086 Assembly language program for the keyboard scroller.

Keyboard encoding is accomplished via hardware. The 74C922 chip is used for this purpose. This chip inputs a key pressed from a hexadecimal keypad and outputs the corresponding binary equivalent on its data lines. In order to indicate that a key has been pressed, the 74C922 sends a HIGH on its Data Available (DA) output pin. This signal is inverted and then connected to the TEST pin of the 8086. This means that the key actuation is indicated by a LOW on the 8086 TEST pin.

The displays are three TIL311s. The rightmost TIL311 is connected to bits 0–3 of port C. This display outputs the most recent key pressed. The middle and the leftmost displays are connected to port B. These two displays show the previous two keys pressed.

10.2.1.c Software Development

Figure 10.6a shows the 8086 assembly language program.

The program first initializes the ports and then waits in a loop for a key to be pressed. In this loop, the 8086 WAIT instruction checks the TEST pin for a LOW. As soon as a key is pressed, the DA pin of the 74C922 goes to a HIGH. This, in turn, drives the TEST pin of the 8086 to a LOW indicating that the data is available.

The 4-bit equivalent of the hex key pressed is input into the 8086 AL register and output to port C. The last two keys pressed are saved in BL. This data is moved to AL and then output to port B. The program loops back to the WAIT instruction and waits for the next key.

Figure 10.6b shows how the contents of the 8086 registers change as the keys are pressed on the keyboard.

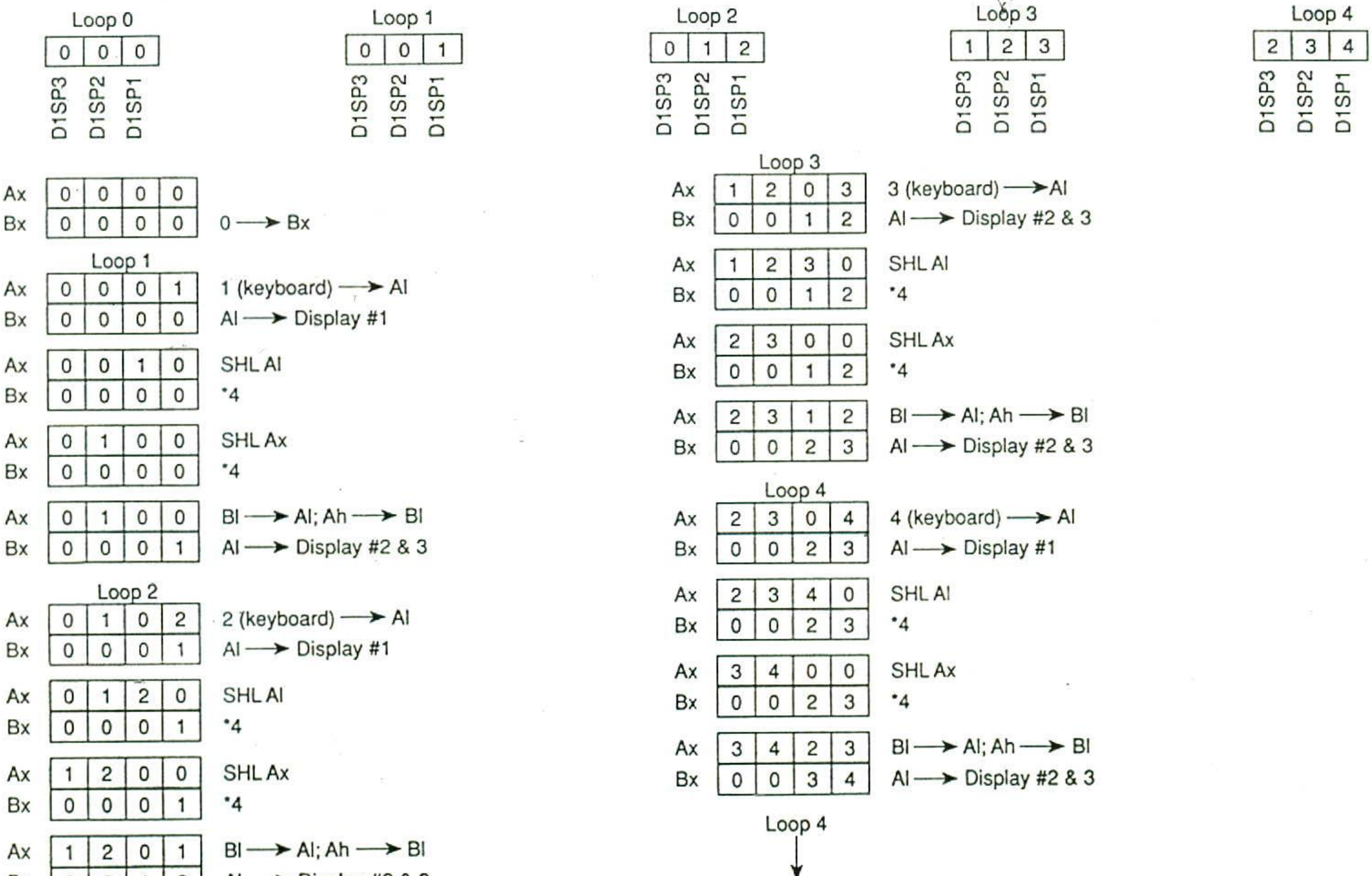


FIGURE 10.6b Contents of program registers.

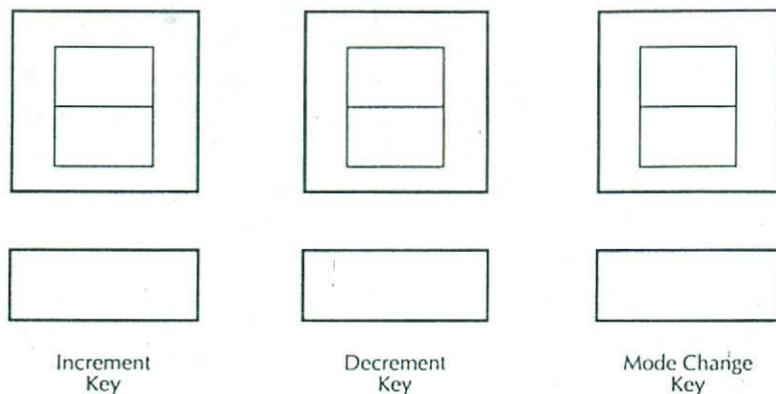


FIGURE 10.7 Block diagram for design problem no. 3.

10.3 Design Problem No. 3

10.3.1 Problem Statement

A 68000-based system is designed to drive three seven-segment displays and monitor three key switches. The system starts by displaying 000. If the increment key is pressed, it will increment the display by one. Similarly, if the decrement key is pressed, it will decrement the display by one. The display goes from 00-FF in the hex mode and from 000-255 in the BCD mode. The system will count correctly in either mode. The change mode key will cause the display to change from hex to decimal or vice versa, depending on its present mode. Figure 10.7 depicts the block diagram.

Two solutions are provided for this problem. Solution one uses programmed I/O with no interrupts, while solution two utilizes interrupt I/O but no programmed I/O.

10.3.2 Solution No. 1

The simplest and the most straightforward system possible is built to obtain the required results. This means that there will be no RAM in the system; therefore, no subroutine will be used in the software and only programmed I/O (no interrupt) is used.

10.3.2.a Hardware

Figure 10.8 shows the detailed hardware schematic. The circuit is divided into the following sections.

10.3.2.a.i Reset Circuit. The reset circuit for the system is basically the same as the one used for the 8085. The circuit has a 0.1-μF capacitor and 1K resistor to provide an RC time constant of 10^{-4} s for power on reset. The $\overline{RESET}$ and $\overline{HALT}$ pins of the 68000 and the $\overline{RESET}$ pin of the 6821 are tied together for complete and total reset of the system.

10.3.2.a.ii Clock Signal. An external pulse-generator is used to generate the clock signal for the system. The system is driven up to 3 MHz, the limit of the generator, without any problems.

10.3.2.a.iii Address Mapping. The system has two 2K EPROM (2716s) and one 6800's peripheral I/O chip (6821). The 68000 address lines A1 through A11 are needed to address the EPROMs. So

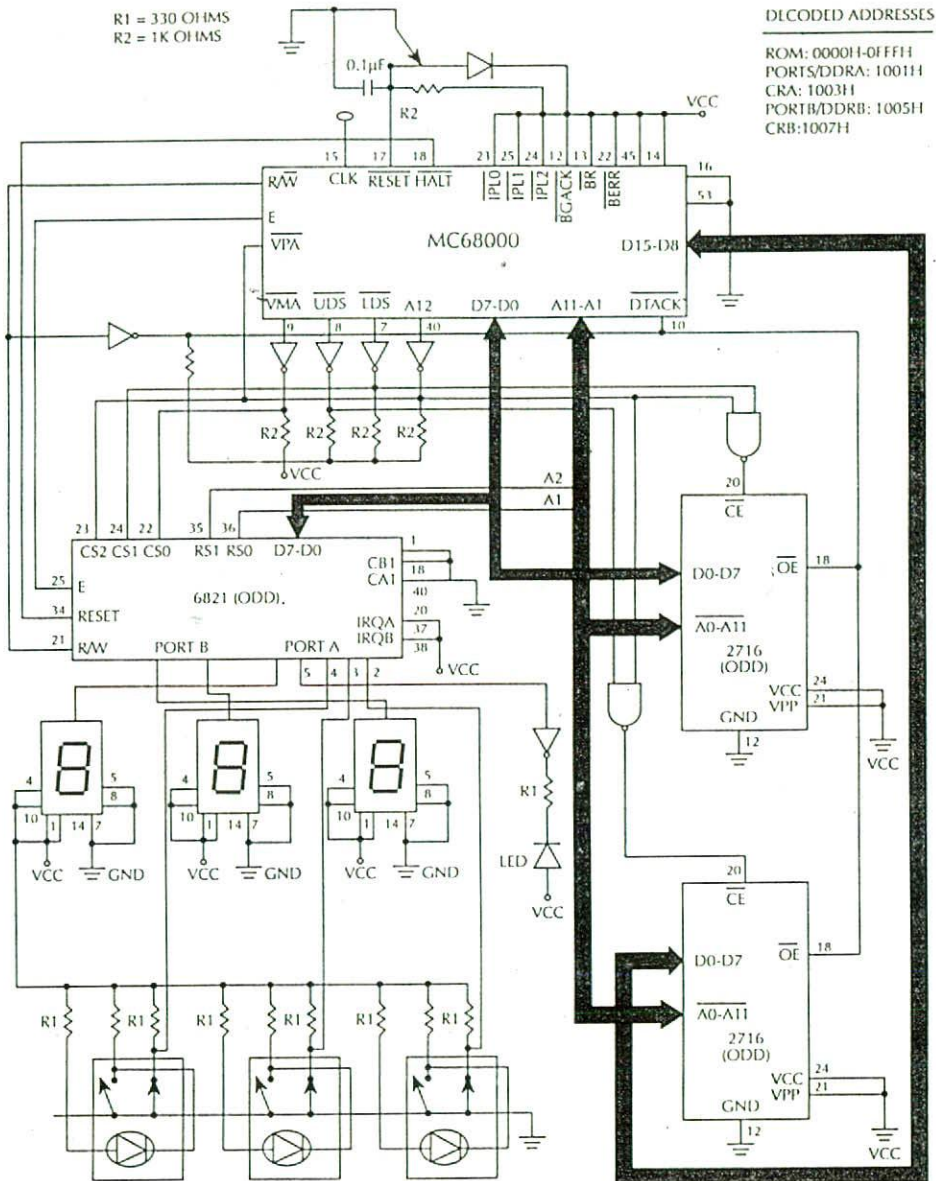


FIGURE 10.8 Detailed hardware schematic.

A12 is used to select between the 2716s and the 6821 (0 for 2716s and 1 for 6821). Memory access for the EPROMs is asynchronous, while the 6821 is synchronized with the E-clock. A12 is inverted, through the buffer, so the output of the inverter goes to CS2 of the 6821 and also to VPA of the 68000 for synchronization. The 68000 VMA pin is also buffered and inverted and it goes to CS0 of the 6821. The 6821 is chosen to be odd, so CS1 is activated by the inverted LDS line. Finally, address lines A1 and A2 are connected to RS0 and RS1, respectively.

The CE for the two 2716s comes from two NAND gates. They are the results of the inverted A12 NANDed with the inverted LDS or the inverted UDS, depending on whether the

EPROM is odd or even. The $\overline{DTACK}$ pin of the 68000 and the $\overline{OE}$ pins of the 2716s are activated by the signal of $R/\overline{W}$ inverted. When the 68000 wants to read the EPROMs this signal will be high, so its inverted signal will provide a low to $\overline{DTACK}$. This does not cause any problem because when the 68000 accesses the 6821, $\overline{VPA}$ is activated and so the 68000 will not look for $\overline{DTACK}$.

The configuration above causes the memory map to be as follows:

	A23...A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	HEX
4K of EPROM Memory	0 0	0	0	0	0	0	0	0	0	0	0	0	0	000000_{16}
	THROUGH													to
	0 0	0	1	1	1	1	1	1	1	1	1	1	1	$000FFF_{16}$
PA/DDRA	0 0	1	0	0	0	0	0	0	0	0	0	0	1	001001_{16}
CRA	0 0	1	0	0	0	0	0	0	0	0	0	1	1	001003_{16}
PB/DDRB	0 0	1	0	0	0	0	0	0	0	0	1	0	1	001005_{16}
CRB	0 0	1	0	0	0	0	0	0	0	0	1	1	1	001007_{16}

10.3.2.a.iv I/O. There are 3 seven-segments displays in the system (TI311), an LED, and 3 switches. The 3 displays have internal latches and hex decoders. So the two least significant displays are connected directly to port B of the 6821 chip, and the most significant display is connected to the upper 4 bits of port A. The latches are tied to ground so as to enable the displays at all times. The LED, when ON, will indicate that the display is in the BCD mode. Each of the three switches, double-pole single throw type, with LED indicator are connected to the lowest 3 bits of port A.

10.3.2.a.v Unused-Input Pins Connection. For the 68000 there are 6 unused, active-low, input pins which must be disabled by connecting them to 5 V. These are $\overline{IPL0}$, $\overline{IPL1}$, $\overline{IPL2}$, $\overline{BERR}$, $\overline{BR}$, and $\overline{BGACK}$. Two of the 6821 unused pins ($\overline{IRQA}$ and $\overline{IRQB}$) are also disabled this way, while CA1 and CB1 are disabled by connecting them to ground.

10.3.2.b Microcomputer Development System

Hewlett-Packard (HP) 64000 is used to design, develop, debug, and emulate the 68000-based system. Some details are given in this section.

The emulator is a very important part of the development of the software and hardware of the system. The 68000 emulator has most of the functions for emulation such as display memory or registers, modify memory or registers. But there is no single-step function. The HP 64000 emulator is divided into three modes of operations: initialization, emulation, and EPROM programming.

10.3.2.b.i Initialization

Edit. The edit function is used to create the application program in mnemonic form. The first line of the program must be "68000" to indicate that the program is to be assembled by the 68000 assembler, and that a 68000 microprocessor is to be used for emulation.

Assemble. When the application program has been completed and properly edited, the file is then assembled into a relocatable object code file. All errors indicated by the assembler should be corrected at this point.

In order to use the 68000 emulator special functions, a special monitor program is required. This can be copied as follows:

```
COPY Mon_68K:HP:source to MON_68K
```

Note upper and lower case. Once this monitor is copied, it must be assembled for no errors. Assume MON_68K as user file name.

Linking. The two relocatable files must be linked together to create an absolute file for the emulation process. The files can be linked as follows:

```
Link <cr>
Object files? Mon_68K
Library file? <cr>
Prog, Data, Comm = 000100H, 000000H, 000000H
More files? yes
Object file? (name of application program file)
Library file? <cr>
Prog, Data, Comm = 001100H, 000000H, 000000H
More files? no
Absolute file? (name of absolute file)
```

There are reasons why the two files were linked this way. The monitor program must be stored at 0100H through 0FFFH, and since this I/O port is mapped starting at 1001H, the application program must be stored starting at a different address, so 1100H was used. Also, address 000H through 0FFF is used for exception vectors by the 68000 microprocessor.

10.3.2.b.ii Emulation. The emulator was used as a replacement for the actual 68000 chip to test the software logic and hardware before it was actually installed into the circuit.

To start the emulation process the following soft-key parameters were entered:

```
EMULATELOAD                  (absolute file name)
Processor clock?             external
Restrict to real time?       no
Memory block size?           256
Significant bits?            20
Break on write to ROM?       yes
Memory map:
0000H thru 0FFFH emulation RAM (monitor & exception vector)
1000H thru 10FFH user RAM (I/O PORT addresses)
1100H thru 1FFFH emulation ROM (application program)
Modify simulated I/O?        no
Reconfigure pod?             no
Command file name?           (name of emulation command file)
```

Note: Usually external clock requires external $\overline{\text{DTACK}}$; however, since the system only has EPROMs and for the purpose of emulation these EPROMs are not used, the external $\overline{\text{DTACK}}$ is not required.

Once the required files and memory maps are loaded, the system is ready for emulation. The monitor program must be running before the application program is executed. To run the monitor program, the following is used:

Then the application program is run using:

```
run from 1100H <cr>
```

Another important part that the user should keep in mind is the processor status. There are three messages for the processor status which indicate that the emulator is not generating any bus cycles. They are

1. Reset — Indicates that the user's hardware is asserting the Reset input. The condition can only be terminated by releasing the user's hardware.
2. Wait — Indicates that the 68000 is waiting for a $\overline{DTACK}$ or other memory response. The condition can be terminated by asserting $\overline{DTACK}$, $\overline{BERR}$, $\overline{VPA}$, or entering "reset" from the keyboard.
3. No memory cycle — Indicates that the 68000 has executed a STOP instruction. The condition can be terminated by asserting "break" or "reset" from the keyboard.

10.3.2.b.iii EPROM Programming. After the software and hardware have been emulated and they work properly, the final step is to program the EPROM and put the final circuit together. But before this, the program must be changed to include the addresses for the stack pointer and the initial PC. This is done by using the "ORG" and "DC" assembler directives. Then this new program is assembled and linked again. The EPROM is then programmed with the contents of this final "absolute file".

Programming EPROMs with the HP 64000 for 68000 is done by odd and even EPROMs. To program the lower 8 bits of data (odd ROM), the option bit 0 is selected, and bit 8 for the upper 8 bits (even ROM) is chosen as follows:

```
Prom_Prog        <cr>
2716             <cr>
Program from     (filename: absolute: bit 0 or 8)
```

Also, to check if the EPROM is clear, the command "check_sum" is used and if the result is F800H then the EPROM is clear.

10.3.2.c Software

The program consists of three major functions: initialize I/O ports and data registers, monitor and debounce key switches, and increment, decrement, or change mode. The program configures port B of the 6821 as an output port which will be used to display the two lower significant nibbles of data. The higher 4 bits of port A are configured as output to display the most significant nibble of the data. Bit 3 is also an output bit which turns ON and OFF the LED. The lowest 3 bits of port A are configured as inputs to detect the positions of three key switches. Register D3 is used to store the data in hex. Registers D4 and D7 are used to store the data in BCD mode with the low order byte in D4 and the high order byte in D7. Bit 3 of D0 contains a logic 1 representing BCD mode and logic 0 representing hex mode. Register D5 contains a 1 which will be used for incrementing BCD data, since ABCD doesn't have immediate mode. Register D6 contains 999 which is used for decrementing BCD.

The program monitors the three switches and stores the three input bits into register D0 if any of the keys is pressed. The processor then waits until the depressed key is released and then checks the input data one by one. The processor then branches to the increment, decrement, or mode change routine according to the depressed key. After execution, the processor will display the result on the three seven-segment displays.

Figure 10.9 shows the software flowchart. Note that the flowchart and the corresponding software does not include 'Mode LED' on/off feature.

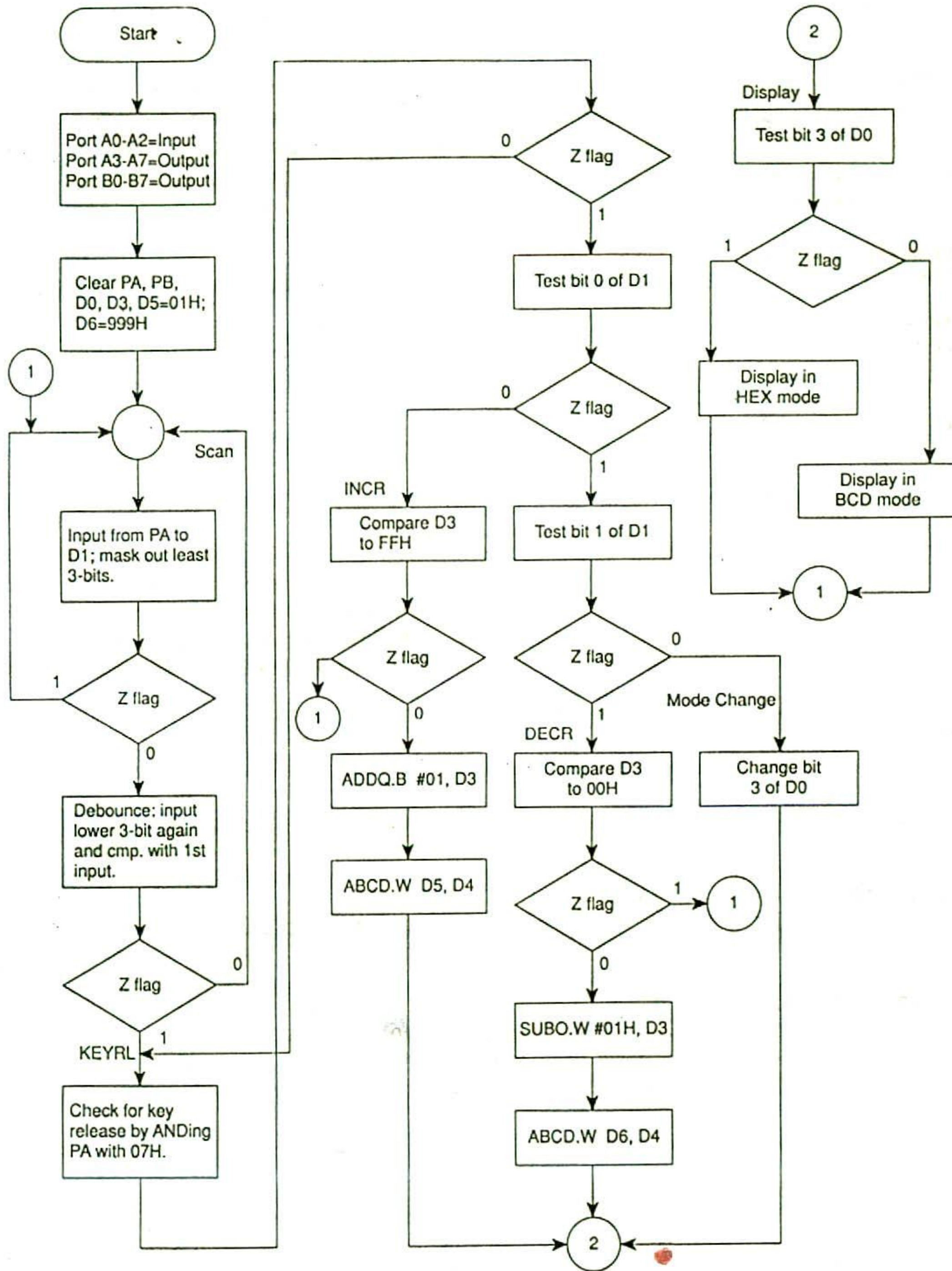


FIGURE 10.9 Software flowchart.

The assembly language program is listed below:

```
FILE: LAB2:KH0A22 HEWLETT-PACKARD: 68000 Assembler

1 "68000"
2 ***********************************************************
3 *THIS PROGRAM STARTS DISPLAYING 000 AND MONITORS THREE KEY*
4 *SWITCHES THEN INCREMENT, DECREMENT, OR CONVERT HEX TO BCD*
5 *OR VICE VERSA, DEPENDING ON WHICH KEY IS DEPRESSED, THE  *
6 *DISPLAY GOES FROM 00-FF IN HEX MODE OR 00-255 IN BCD MODE*
7 ***********************************************************

LOCATION OBJECT
CODE LINE                                 SOURCE LINE
<1001>                 8 PA        EQU      001001H
<1001>                 9 DDRA      EQU      001001H
<1003>                10 CRA       EQU      001003H
<1005>                11 PB        EQU      001005H
<1005>                12 DDRB      EQU      001005H
<1007>                13 CRB       EQU      001007H
                      14           ORG      00000000H
000000 FFFF FFFF      15           DC.L     0FFFFFFFFH
000004 0000 0008      16           DC.L     START
                      17 *
                      18 * CONFIGURE THE INPUT AND OUTPUT PORTS,
                      19 * DISPLAY 000 ON THE 7-SEGMENT DISPLAYS,
                      20 * AND INITIALIZE ALL THE DATA REGISTERS.
                      21 * THE HEX MODE IS STORED IN D3 AND THE
                      22 * BCD MODE IS STORED IN D7 AND D4.
000008 4238 1003      23 START     CLR.B    CRA
00000C 11FC 00FB      24           MOVE.B   #0F8H,DDRA   ;BIT 0-2 OF
                                                         ;PORT A AS
                                                         ;INPUT
000012 0BF8 0002      25           BSET.B   #02H,CRA     ;BIT 3-7 OF
                                                         ;PORT AS OUT-
                                                         ;PUT
000018 4238 1007      26           CLR.B    CRB
00001C 11FC 00FF      27           MOVE.B   #0FFH,DDRB   ;ALL 8 BITS OF
                                                         ;PORT B AS
                                                         ;OUTPUT
000022 08F8 0002      28           BSET.B   #02H,CRB
000028 4200           29           CLR.B    D0
00002A 11C0 1001      30           MOVE.B   D0,PA        ;DISPLAY 000
00002E 11C0 1005      31           MOVE.B   D0,PB
000032 4203           32           CLR.B    D3
000034 4244           33           CLR.W    D4
000036 7A01           34           MOVEQ.W  #01H,D5
000038 3C3C 0999      35           MOVE.W   #999H,D6
00003C 4207           36           CLR.B    D7
                      37 *
                      38 * DEBOUNCE THE KEY SWITCHES
                      39 *
00003E 1238 1001      40 SCAN      MOVE.B   PA,D1        ;MONITOR THE
                                                         ;KEYS
000042 0201 0007      41           ANDI.B   #07H,D1      ;MASK OUT THE
                                                         ;OUTPUT PINS
```

FILE: LAB2:KH0A22 HEWLETT-PACKARD: 68000 Assembler (*continued*)

```
LOCATION OBJECT
CODE LINE                                          SOURCE LINE
000046 67F6          42          BEQ      SCAN     ;IF NO KEY IS
                                                   ;DEPRESSED GO TO
                                                   ;SCAN
000048 1438 1001     43          MOVE.B   PA,D2    ;READ THE DATA
                                                    AGAIN
00004C 0202 0007     44          ANDI.B   #07H,D2
000050 B401          45          CMP.B    D1,D2    ;CHECK TO SEE IF
                                                   ;THE DATA REMAIN
                                                   ;UNCHANGED
000052 66EA          46          BNE      SCAN     ;IF IT CHANGES GO
                                                   ;TO SCAN
                     47 *
                     48 * CHECK TO MAKE SURE THAT THE KEY IS
                     49 * RELEASED BEFORE THE NEXT KEY CAN BE
                     50 * ENTERED.
000054 1438 1001     51 KEYRL    MOVE.B   PA,D2
000058 0202 0007     52          ANDI.B   #07H,D2
00005C 66F6          53          BNE      KEYRL
                     54 * CHECK TO SEE WHICH KEY HAS BEEN EN-
                     55 * TERED. BITS 0,1, AND 2 OF D1 REPRESENT
                     56 * INCREMENT, DECREMENT, AND MODE EX-
                     57 * CHANGE RESPECTIVELY.
00005E 0801 0000     58          BTST.B   #0H,D1
000062 6600 003C     59          BNE      INCR     ;IF BIT-0 OF D1
                                                   ;IS 1 GOTO INCR
000066 0801 0001     60          BTST.B   #1H,D1
00006A 6700 0044     61          BEQ      MODE     ;IF BIT-1 IS 1
                                                   ;DECREMENT, OTHER-
                                                   ;WISE GOTO MODE
                     62 *
                     63 * DECREMENT BOTH HEX AND BCD AT THE SAME
                     64 * TIME.
00006E 0C03 0000     65          CMPI.B   #00H,D3
000072 67CA          66          BEQ      SCAN     ;IF THE NUMBER IS
                                                   ;0 NO DECREMENT,
                                                   ;GOTO SCAN
000074 5303          67          SUBQ.B   #1H,D3   ;DECREMENT HEX BY 1
000076 C603          68          AND.B    D3,D3    ;CLEAR THE CARRY
000078 C906          69          ABCD.B   D6,D4    ;DECREMENT BCD BY
                                                   ;1 BY ADDING IT
                                                   ;WITH 999
00007A CF06          70          ABCD.B   D6,D7
                     71 *
                     72 * DISPLAY THE NUMBER IN HEX IF BIT-3 OF
                     73 * D0 IS 0, OTHERWISE DISPLAY IN BCD.
                     74 *
00007C 0800 0003     75 DISPLAY  BTST.B   #3H,D0
000080 6700 0014     76          BEQ      HEX      ;IF BIT-3 OF D0
                                                   ;IS 0, GOTO HEX
000084 11C4 1005     77          MOVE.B   D4,PB    ;OUTPUT THE LSB
                                                   ;TO PORT B
000088 E94F          78          LSL.W    #4H,D7   ;SHIFT LEFT 4
                                                   ;TIMES
```

```
FILE: LAB2:KH0A22   HEWLETT-PACKARD: 68000 Assembler (continued)

LOCATION OBJECT
CODE LINE                                          SOURCE LINE

00008A 08C7 0003   79          BSET.B   #3H,D7    ;TURN OFF THE LED
00008E 11C7 1001   80          MOVE.B   D7,PA     ;OUTPUT THE MSB
                                                  ;TO UPPER 4 BITS
                                                  ;OF PORT A
000092 E84F        81          LSR.W    #4,D7
000094 60A8        82          BRA      SCAN
000096 11C0 1001   83 HEX      MOVE.B   D0,PA     ;OUTPUT 0 TO PORT A
00009A 11C3 1005   84          MOVE.B   D3,PB     ;OUTPUT THE HEX
                                                  ;NUMBER TO PORT B
00009E 609E        85          BRA      SCAN
                   86          *
                   87          * INCREMENT BOTH HEX AND BCD.
                   88          *
0000A0 0C03 00FF   89 INCR     CMPI.B   #0FFH,D3
0000A4 6798        90          BEQ      SCAN      ;IF THE NUMBER
                                                  ;IS FF NO
                                                  ;INCREMENT
0000A6 5203        91          ADDQ.B   #1H,D3    ;INCREMENT HEX
                                                  ;NUMBER BY 1
0000A8 4202        92          CLR.B    D2
0000AA C905        93          ABCD.B   D5,D4     ;INCREMENT LSB OF
                                                  ;BCD BY 1
0000AC CF02        94          ABCD     D2,D7     ;INCREMENT MSB OF
                                                  ;BCD BY 1 IF CARRY
                                                  ;IS 1
0000AE 60CC        95          BRA      DISPLAY
                   96 *
                   97 * EXCHANGE THE MODE THEN DISPLAY THE
                   98 * NUMBER.
0000B0 0840 0003   99 MODE     BCHG     #3H,D0    ;EXCHANGE MODE BY
                                                  ;CHANGING BIT-3
                                                  ;OF D0
0000B4 60C6        100         BRA      DISPLAY
Errors = 0

LINE#   SYMBOL    TYPE   REFERENCES

***     B         U      23,24,25,26,27,28,29,30,31,32,36,40,41,43,44,
                          45,51,52,58,60,65,67,68,69,70,75,77,79,80,
                          83,84,89,91,92
10      CRA       A      23,25
13      CRB       A      26,28
9       DDRA      A      24
12      DDRB      A      27
75      DISPLAY   A      95,100
83      HEX       A      76
89      INCR      A      59
51      KEYRL     A      53
***     L         U      15,16
99      MODE      A      61
8       PA        A      30,40,43,51,80,83
11      PB        A      31,77,84
40      SCAN      A      42,46,66,82,85,90
23      START     A      16
***     W         U      33,34,35,78,81
```

TABLE 10.1 Memory Map

\$000000-\$000FFF	EPROM
\$003000-\$003FFF	RAM
\$005000-\$005FFF	DISPLAYS
\$009000-\$009FFF	SWITCHES

10.3.3 Solution No. 2

The second solution approach uses interrupt I/O but no I/O ports.

10.3.3.a Hardware

The system includes a 3-digit display and three momentary function switches (increment, decrement, and mode select). In order to minimize the complexity of the project, no I/O chips are used. Instead, a buffer and some latches are used as the I/O ports. The buffer is used to input the status of the momentary switches and the latches are used to input the information coming from the data bus. To further the design, three TIL311 displays are used because they contain internal data latches. Because the 68000 has 23 address lines (not including A0), the memory is linearly decoded. The even and odd memory chips are enabled by decoding pins $\overline{UDS}$, $\overline{LDS}$, and $\overline{AS}$.

To display the three-digit number, the data lines are connected to the inputs of the three TIL311 displays (D0–D3 = LSD, D4–D7 = middle digit, D8–D11 = MSD). The address strobe ($\overline{AS}$) is NANDed with the address line A14 to latch the data onto the three displays. The memory map for the displays is given in Table 10.1. Because of linear decoding, the problem of foldback exists.

Two 2716s are used for the EPROM and two 6116s are used for the RAM. Both the RAM and the EPROM chips are divided into even and odd memory. The configuration enables the 68000 to access an even or an odd data bytes or a complete word in one bus cycle. The even and odd select lines are generated by ANDing the $\overline{UDS}$ and $\overline{AS}$ pins and the $\overline{LDS}$ and $\overline{AS}$ pins, respectively. To access a word, both the even and the odd enable signals are asserted. These signals are then NANDed with address lines A12 and A13 to select the EPROM and the RAM, respectively (see Figure 10.10). The odd memory chip data lines are connected to D0–D7 of the 68000. The even memory chip data lines are connected to D8–D15. Table 10.1 shows the memory map.

In the system, the interrupt pins are implemented by ANDing the status of the momentary switches and connecting the output of the gate to $\overline{IPL2}$. To achieve a level 5 interrupt, $\overline{IPL1}$ and $\overline{IPL0}$ pins are connected to Vcc and ground, respectively (see Figure 10.10). To reduce the number of components, the 68000 is instructed to generate an internal autovector to service the interrupt. This is accomplished by asserting $\overline{VPA}$ and $\overline{IPL2}$ at the same time. If an interrupt occurs (switch pressed), the 68000 will compute the autovector number \$1D and the vector address \$74. The processor will then go to a service routine that will find the switch that was pressed.

A 4-MHz crystal oscillator is used to clock the processor. Since the 68000 is operating at 4 MHz, $\overline{AS}$ is directly connected to $\overline{DTACK}$. This gave the EPROMs (450 ns access time) about 500 ns to provide valid data. A reset circuit similar to the one used in the 8085 system is used for the 68000-based microcomputer. However, on the 68000, both the $\overline{RESET}$ and $\overline{HALT}$ pins are tied together (see Figure 10.10). Figure 10.11 shows the board layout of all the chips.

10.3.3.b Software

The first major feature of the software is the inclusion of a start-up routine. The advantage of the start-up routine is to visibly verify the system performance. For example, if one of the displays malfunctions, the fault will not be known unless the user is able to see the display patterns. This requirement leads to the development of a start-up routine in which all three

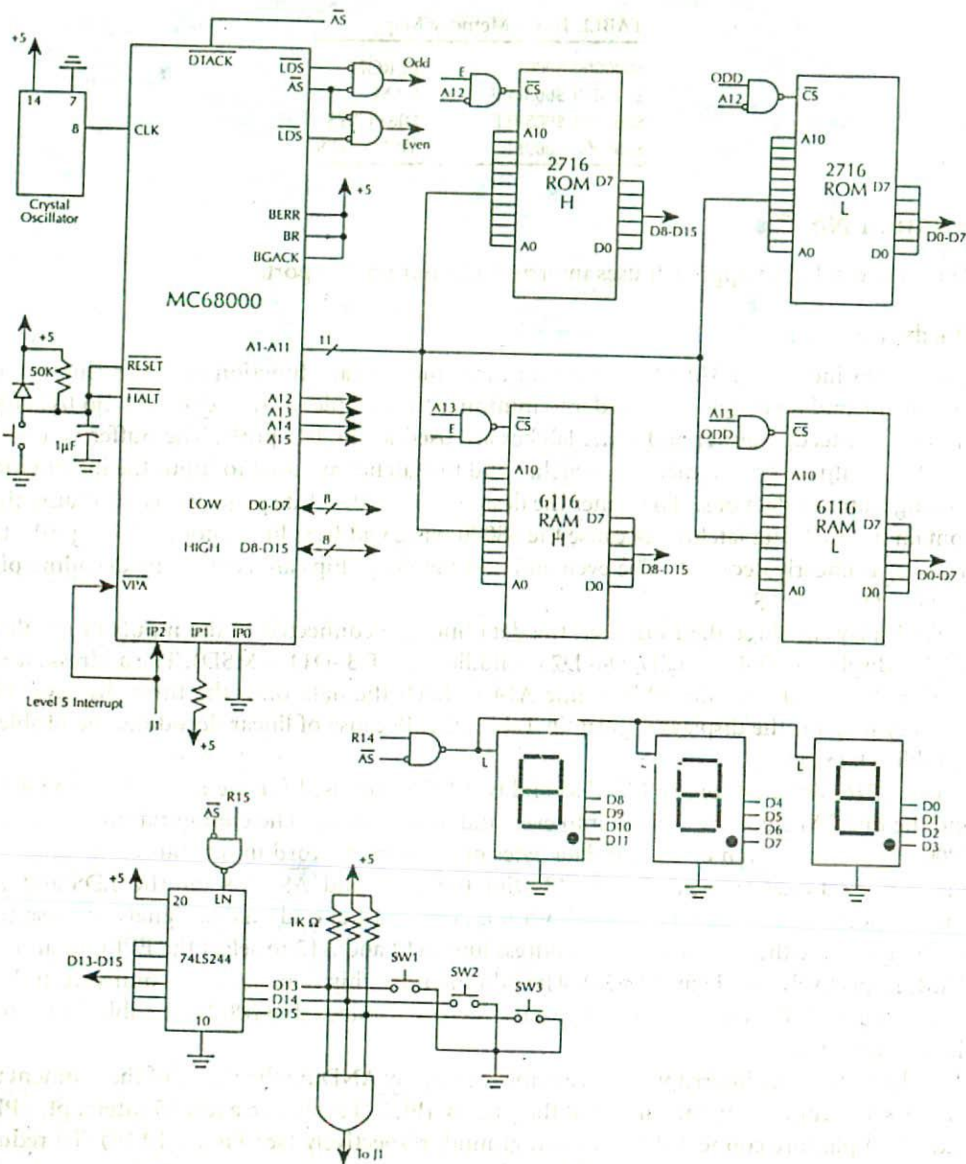


FIGURE 10.10 68000-based system for design problem no. 2.

displays count F down to 0 (in parallel). This routine uses a DBF loop in which the counter's value is duplicated to the two higher hex digits. The following is the actual start-up routine implemented in the program.

```
        MOVEQ    #0FH,D0      ; INITIALIZE LOOP COUNTER
                              ; TO $0000000F
LOOP    MOVE.W   D0,D1        ; COPY D0 TO D1
        ASL.W    #4,D1        ; SHIFT D1 LEFT 4 TIMES
        ADD      D0,D1        ; ADD D0 TO D1
        ASL.W    #4,D1        ; SHIFT D1 LEFT 4 TIMES
```

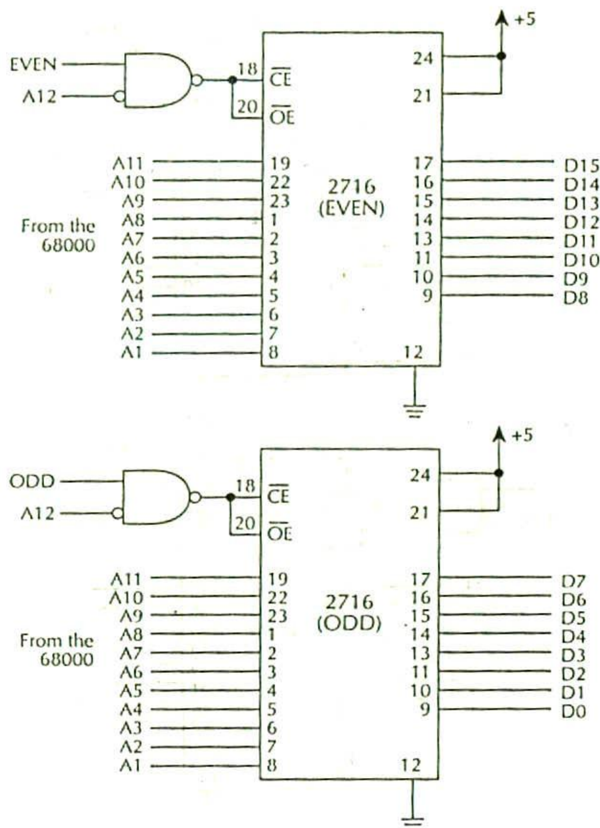


FIGURE 10.10 *continued.*

```
        ADD.W     D0,D1           ; ADD D0 TO D1
        MOVE.W    D1,DISPADDR     ; SEND RESULT TO DISPLAY
        MOVE.L    #VISIBLE,D6     ; LOAD DELAY TIME
        JSR       DELAY           ; CALL DELAY SUBROUTINE
        DBF.W     D0,LOOP         ; DEC BRANCH IF D0 ≠ -1, NOT
                                  ; TO THE LOOP
        CLR.L     D0              ; INITIALIZE COUNTER TO ZERO
        CLR.L     D7              ; INITIALIZE MODE TO DECIMAL
        MOVE.W    D0,DISPADDR     ; INITIALIZE DISPLAYS TO ZERO
        MOVE.W    #INTRMASK,SR    ; SET INTR AT LEVEL 5 AND
                                  ; SUPERVISOR MODE
WAIT    BRA.B     WAIT            ; WAIT FOR INTERRUPT
```

Upon successful completion of the start-up routine, the software directs the 68000 to enter an infinite wait loop. The wait loop serves to occupy the 68000 until a level 5 interrupt signals the processor. Upon interrupt, SR is pushed and the PC is also stacked. The 68000 accesses the long word located at address $74 and jumps to that service routine. The service routine exists at location $500. In response to the interrupt, the software directs the 68000 to move in the status switches to the low word of D0. A "C"-type priority case statement executes.

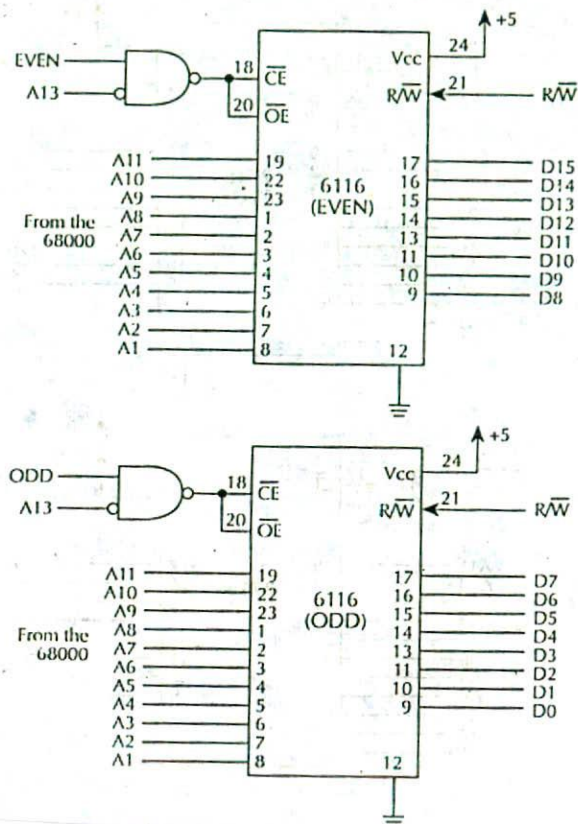


FIGURE 10.10 *continued.*

The case statement has the priority of up, down, then mode. Implementation of the case statement eliminates uncertainties when multiple keys are depressed. In the following, the case statement is shown.

```
RESPONSE  NOP                     ; ENTRY NO OPERATION
          MOVE.W  STATUS,D1       ; MOVE IN BUTTON STATUS WORD
          BTST.W  #UPBIT,D1       ; TEST INCREMENT BIT
          BEQ.W   INCREASE        ; IF UPBIT=0 BRANCH TO INCREASE
          BTST.W  #DOWNBIT,D1     ; TEST DECREMENT BIT
          BEQ.W   DECREASE        ; IF DOWNBIT=0 BRANCH TO
                                    DECREASE
          BTST.W  #MODEBIT,D1     ; TEST MODEBIT
          BEQ.B   CHMODE,D1       ; IF MODEBIT=0 BRANCH TO CHANGE
                                    MODE
          BRA.B   RESPONSE        ; NO RESPONSE, THEN SEARCH AGAIN
```

This segment utilizes the test bit facilities of the 68000. The algorithm first loads the switches. The switch word is then tested by the BTST instruction. The first bit test is the upbit. If the bit is found to be 0, the program branches to an increase-update routine. If the downbit

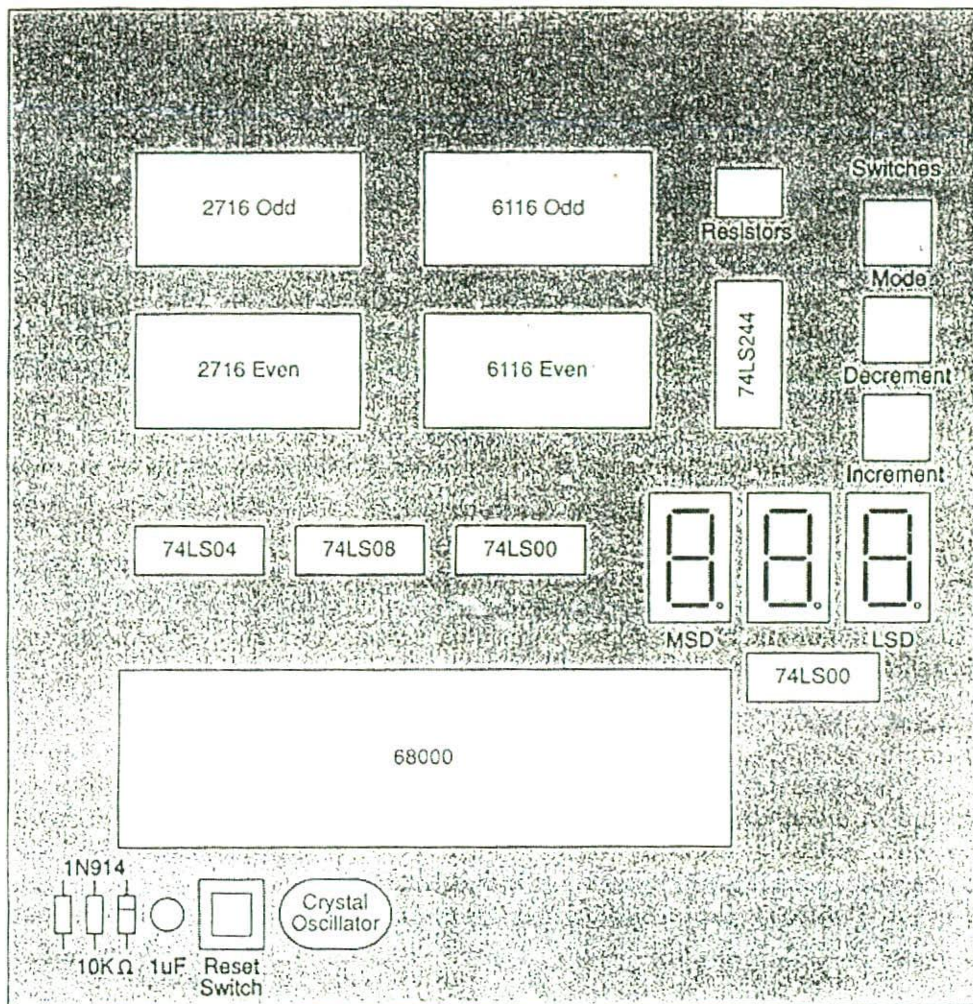


FIGURE 10.11 Board layout.

is found to be low, then the program reacts to decrement the displays. If the mode bit is found low, then the response is the base conversion of the displayed output.

The user may be tempted to indefinitely press a button or press multiple buttons. The habit is permissible. The program implements a 0.4-second wait loop at the end of any press of a key. This is a post-debounce. Without this feature, the 68000 will either count or change modes at speeds beyond recognition. The debounce routine also contains a priority. If the user constantly depresses multiple keys, the 68000 will service the input with the highest priority.

At this point, a deviation of the problem was made. The deviation was for ease of checking out the project. During checkout, when one wants to see a rollover, the increment or decrement key must be pressed 255 times. This is futile. At the end of the service routine, the software will not lock out a key entry, but rather the 68000 will immediately go to the wait state where the next interrupt may take place. To the user, it will appear that the 68000 is either autoincrementing, autodecrementing, or automatically changing modes. The post-debounce segment is displayed below.

```
VIEWER  NOP                      ; ENTRY NO OPERATION
        MOVE.1   #VISIBLE,D6     ; PLACE DELAY INTO D6
```

```
         JSR      DELAY          ; JUMP TO DELAY SUBROUTINE
         RTE                     ; RETURN FROM EXCEPTION
DELAY    NOP                     ; ENTRY NO OPERATION
         DBF.W    D6,DELAY       ; DECREMENT FOR WAIT
         RTS                     ; RETURN FROM SUBROUTINE
```

The debounce routine implements a dummy loop that utilizes a large loop count. The routine is initialized by an immediate move long to D6. The debounce routine is called via the jump subroutine command. The delay loop contains a no-operation to increase loop time. After the NOP, the DBF.W will decrement D6 and branch if D6 is not equal to negative 1.

The software uses a hex base for counting; that is, all numbers whether decimal or hex will originate from a hex byte in data register 0 (D0). The display status exists in data register 7 (D7). If the contents of D7 are zero, this informs the program to display a decimal number on the next update. Otherwise, the program will send a hex value to the display. A typical decision-making segment (below) uses the 68000's ability to update flags on a move operation.

```
MOVE.   D7,D7      ; MOVE TO UPDATE FLAGS
BNE.B   HEX        ; IF Z=0 THEN SEND HEX TO DISPLAYS
BRA.B   DECIMAL    ; OTHERWISE, DECIMAL TO DISPLAYS
```

To convert a hex number to decimal format, the program uses the division/modulo algorithm shown in the following.

```
DECIMAL  NOP                     ; ENTRY NO OPERATION
         CLR.L    D2             ; INITIALIZE D2 TO ZERO
         CLR.L    D1             ; INITIALIZE D1 TO ZERO
         MOVE.B   D0,D1          ; COPY COUNT
         DIVU     #10,D1         ; DIVIDE D1 BY 10 MSD HEX →
                                 ; DECIMAL
         SWAP     D1             ; PLACE REMAINDER IN LOW WORD
                                 ; D1
         MOVE.W   D1,D2          ; MOVE REMAINDER TO D2
         CLR.W    D1             ; CLEAR REMAINDER
         SWAP     D1             ; REPLACE REMAINDER
         DIVU     #10,D1         ; DIVIDE D1 BY 10
         SWAP     D1             ; REMAINDER TO LOW WORD D1
         ASL.W    #4,D1          ; SHIFT REMAINDER UP ONE DIGIT
         ADD.W    D1,D2          ; ADD IN SECOND SIG FIG
         SWAP     D1             ; REPLACE QUOTIENT
         ASL.W    #4,D1          ; SHIFT QUOTIENT UP ONE DIGIT
         ASL.W    #4,D1          ; SHIFT QUOTIENT ANOTHER DIGIT
         ADD      D1,D2          ; ADD IN LSD HEX & DECIMAL
         MOVE.W   D2,DISPADDR    ; SEND DECIMAL RESULTS TO
                                 ; DISPLAY
         BRA.W    VIEWER         ; GO TO DISPLAY BRANCH
```

The algorithm exploits the DIVU (unsigned division) facilities of the 68000. The hex byte is moved to a long word register with zero-extend (assumed by CLR.L followed by a MOVE.B operation). The number is then divided by 10. The quotient remains in the low word of the destination register (D1); the remainder lies in the high word. With the use of SWAP, the remainder and quotient words are swapped. The remainder is moved (MOVE.W D1,D2) to another register D2 (initialized to zero). At this point, the remainder is cleared in D1, and swap is used to replace the quotient in the low word of D1. The next lower significant digit is

extracted. Again, DIVU uses an immediate source of 10. The remainder in D1 is swapped into the low word, shifted up four times, then added to D2. The quotient is swapped back to the low word, shifted left eight times, then added to D2. The result of this routine is (at most) a three-digit BCD number which is suitable to send to the displays.

After the update of the displays, a time delay subroutine allows execution delays from the order of microseconds to the order of seconds. The time delay subroutine is shown below.

```
DELAY  NOP                    ; ENTRY NO OPERATION
       DBF.W   D6,DELAY       ; DECREMENT FOR WAIT
       RTS                    ; RETURN FROM SUBROUTINE
```

The NOP serves to increase the delay time of the loop. The NOP takes 4 clock cycles. DBF.W (decrement and branch on false) takes 10 clocks on a branch and 14 on a skip. JSR (jump subroutine) to the delay takes 23 cycles. The time analysis is simplified when consideration is taken only of the duration of the delay loop. A suitable delay for this project is about 0.4 s. This equates to (4-MHz clock) 1.6 million clock cycles! Because of the high number of cycles required, the "calls" and "returns" can be avoided because of their insignificance when compared to the massive number of clock delays required. A delay of about 0.4 s is used, which requires about 100,000 loops in the delay routine.

Some mention should be made of the mode features of the software. The change mode allows the user to liberally change the viewing format from hex to BCD or vice versa. The activation of this software feature simply complements D7 and then updates the displays via the previously mentioned methods. The increment facility increments D0 and updates the displays. Similarly, the decrement facility decrements D0 and updates the displays.

Expansion of the system is possible. Maybe for user entertainment, an uptone or downtone can be implemented. The tone can be generated through a variable delay routine. One of the address lines may be tied in series to a small speaker and every time the address is accessed, the speaker will "tick". Otherwise, the software, as it is, is suitable for the project. A listing of the assembly language program is provided below:

```
"68000"
;
; Microcomputer Applications
; M. Rafiquzzaman
; November 9, 1987
;
;
; This is the software routine for Design Problem
; No. 2
; THE BCD<>HEX COUNTER
; Language is 68000 MACRO
;
;
              NAME     BCD<>HEX_COUNTER
AUTO6         EQU      00000074H          ;SERVICE ROUTINE
                                          ;ADDRESS LOCATION
TSTACK        EQU      000037FCH          ;STACK INITIALIZE
RESPONSE      EQU      00000500H          ;INTERRUPT VECTOR
                                          ;ADDRESS
DISPADDR      EQU      00005000H          ;ADDRESS OF DISPLAY
PCINIT        EQU      00000400H          ;PC STARTUP ADDRESS
```

```
VISIBLE         EQU     00100000H       ;DELAY TIME APPROX. 0.4
                                        ;SECONDS
UPBIT           EQU     0FH             ;INCREMENT BIT LOCATION
DOWNBIT         EQU     0EH             ;DECREMENT BIT LOCATION
MODEBIT         EQU     0DH             ;MODE BIT LOCATION
STATUS          EQU     00009000H       ;STATUS WORD LOCATION
INSTRMSK        EQU     2500H           ;INTERRUPT MASK, LEVEL 7
;
; Top of the stack, program origin, and interrupt service
; location
;
                ORG     00000000H       ;STARTUP
                DC.L    TSTACK          ;INITIAL SUPERVISOR
                DC.L    PCINIT          ;FIRST PROGRAM INSTR
                                        ;LOC
                ORG     AUTO6           ;LOCATION OF AUTOVECTOR
                                        ;RESPONSE
                DC.L    RESPONSE        ;ADDRESS OF SERVICE
                                        ;ROUTINE 5
                DC.L    RESPONSE        ;ADDRESS OF SERVICE
                                        ;ROUTINE 6
                DC.L    RESPONSE        ;ADDRESS OF SERVICE
                                        ;ROUTINE NMI
;
; Startup routine
;
                ORG     PCINIT          ;BOOTUP AND TEST
                                        ;ROUTINE
                NOP                     ;ENTRY NO OPERATION
                MOVEQ   #0FH,D0         ;INITIALIZE LOOP
                                        ;COUNTER TO $0000000F
LOOP            MOVE.W  D0,D1           ;COPY D0 TO D1
                ASL.W   #4,D1           ;SHIFT D1 LEFT FOUR
                                        ;TIMES
                ADD.W   D0,D1           ;ADD D0 TO D1
                ASL.W   #4,D1           ;SHIFT D1 LEFT FOUR
                                        ;TIMES
                ADD.W   D0,D1           ;ADD D0 TO D1
                MOVE.W  D1,DISPADDR     ;SEND RESULT TO DISPLAY
                MOVE.L  #VISIBLE,D6     ;LOAD DELAY TIME
                JSR     DELAY           ;CALL DELAY SUBROUTINE
                DBF     D0,LOOP         ;DEC BRANCH IF D0 !=
                                        ;-1, NOT TO LOOP
                CLR.L   D0              ;INITIALIZE COUNTER TO
                                        ;ZERO
                CLR.L   D7              ;INITIALIZE MODE TO
                                        ;ZERO
                MOVE.W  D0,DISPADDR     ;INITIALIZE DISPLAYS TO
                                        ;ZERO
                MOVE.W  #INTRMASK, SR   ;SET INTR AT 5 AND
                                        ;SUPER MODE
WAIT            BRA.B   WAIT            ;WAIT FOR INTERRUPT
;
; INTERRUPT ROUTINE
;
                ORG     RESPONSE
RESPONSE        NOP                     ;ENTRY NO OPERATION
                MOVE.W  STATUS,D1       ;MOVE IN BUTTON STATUS
                                        ;WORD
```

```
                BTST.W   #UPBIT,D1          ;TEST INCREMENT BIT
                BEQ.W    INCREASE           ;IF UPBIT=0 THEN BRANCH
                                            ;TO INCREASE
                BTST.W   #DOWNBIT,D1        ;TEST DECREMENT BIT
                BEQ.W    DECREASE           ;IF DOWNBIT=0 BRANCH TO
                                            ;DECREASE
                BTST.W   #MODEBIT,D1        ;TEST MODEBIT
                BEQ.B    CHMODE,D1          ;IF MODEBIT=0 THEN
                                            ;BRANCH TO CHANGE MODE
                BRA.B    RESPONSE           ;NO RESPONSE, THEN
                                             SEARCH AGAIN
;
; Change mode and update displays
;
CHMODE          NOP                         ;ENTRY NO OPERATION
                EORI.B   #0FFH,D7           ;COMPLEMENT MODE MASK
                BNE.B    HEX                ;IF EORI IS NOT ZERO
                                            ;THEN HEX OUT
                BRA.B    DECIMAL            ;EORI IS ZERO, THEN
                                            ;DECIMAL OUT
;
; Increment display count
;
INCREASE        NOP                         ;ENTRY NO OPERATION
                ADDQ.B   #1,D0              ;INCREMENT THE COUNT D0
                MOVE.B   D7,D7              ;MOVE TO UPDATE FLAGS
                BNE.B    HEX                ;IF Z=0 THEN SEND HEX
                                            ;TO DISPLAYS
                BRA.B    DECIMAL            ;OTHERWISE, DECIMAL TO
                                            ;DISPLAYS
;
; Decrement display count
;
DECREASE        NOP                         ;ENTRY NO OPERATION
                SUBQ.B   #1,D0              ;DECREMENT COUNT
                MOVE.B   D7,D7              ;MOVE TO UPDATE FLAGS
                BNE.B    HEX                ;IF Z=0 THEN SEND HEX
                                            ;TO DISPLAYS
                BRA.B    DECIMAL            ;OTHERWISE, DECIMAL
                                            ;DISPLAYS
; This routine sends hex contents of D0 to the displays
;
HEX             NOP                         ;ENTRY NO OPERATION
                MOVE.W   D0,DISPADDR        ;HEX DATA IS SENT TO
                                            ;DISPLAYS
                BRA.W    VIEWER             ;GO TO DELAY BRANCH
;
; HEX → Decimal converter
;
DECIMAL         NOP                         ;ENTRY NO OPERATION
                CLR.L    D2                 ;INITIALIZE D2 TO ZERO
                CLR.L    D1                 ;INITIALIZE D1 TO ZERO
                MOVE.B   D0,D1              ;COPY COUNT
                DIVU     #10,D1             ;DIVIDE D1 BY 10 MSD
                                            ;HEX → DECIMAL
                SWAP     D1                 ;PLACE REMAINDER IN LOW
                                            ;WORD D1
                MOVE.W   D1,D2              ;MOVE REMAINDER TO D2
                CLR.W    D1                 ;CLEAR REMAINDER
```

```
                SWAP    D1              ;REPLACE REMAINDER
                DIVU    #10,D1          ;DIVIDE D1 BY 10
                SWAP    D1              ;REMAINDER TO LOW WORD
                                        ;D1
                ASL.W   #4,D1           ;SHIFT REMAINDER UP ONE
                                        ;DIGIT
                ADD.W   D1,D2           ;ADD IN SECOND SIG FIG
                SWAP    D1              ;REPLACE QUOTIENT
                ASL.W   #4,D1           ;SHIFT QUOTIENT UP ONE
                                        ;DIGIT
                ASL.W   #4,D1           ;SHIFT QUOTIENT ANOTHER
                                        ;DIGIT
                ADD     D1,D2           ;ADD IN LSD HEX →
                                        ;DECIMAL
                MOVE.W  D2,DISPADDR     ;SEND DECIMAL RESULTS
                                        ;TO DISPLAY
                BRA.W   VIEWER          ;GO TO DISPLAY BRANCH
;
; This sends output to displays and implements delay of 0.7
; seconds
;
VIEWER          NOP                     ;ENTRY NO OPERATION
                MOVE.L  #VISIBLE,D6     ;PLACE ENTRY INTO D6
                JSR     DELAY           ;JUMP TO DELAY
                                        ;SUBROUTINE
                RTE                     ;RETURN FROM EXCEPTION
;
; Delay subroutine
;
DELAY           NOP                     ;ENTRY NO OPERATION
                DBF.B   D6,DELAY        ;DECREMENT FOR WAIT
                RTS                     ;RETURN FROM
                                        ;SUBROUTINE
```

The following shows 68000 Delay Analysis:

```
                MOVE.L  #VISIBLE,D6     ;MOVE IN DELAY LOOP
                                        ;COUNT
                JSR     DELAY           ;CALL SUBROUTINE TO
                                        ;DELAY LOOP
DELAY           NOP                     ;TIME DELAY INCREASE
                                        ;LOOP
                DBF     D6,DELAY        ;DECREMENT BRANCH FALSE
                RTS                     ;RETURN SUBROUTINE
```

Execution Time:

```
MOVE.L   #REG        12      CLOCKS
JSR      ADDR        10      CLOCKS
NOP                  4       CLOCKS
DBF      REG,ADDR    14/10   CLOCKS
RTS                  16      CLOCKS
```

First Pass: t1 = 12 + 20 + 4 + 14 = 50 CLOCKS

Middle Pass: t(n–2) = (n – 2) (4 + 14) CLOCKS

Last Pass: tn = 4 + 10 + 16 = 30 CLOCKS

General Pass: (for n > 3)

Typical Pass: (for large n)

For 0.45-s delay with a 4-MHz clock, 1.8 million clock cycles are required (0.45 s was chosen for ease of calculations).

Therefore,

$$18n = 1,800,000$$
$$n = 100,000$$

Figure 10.12 shows the flowchart for the start-up routine.

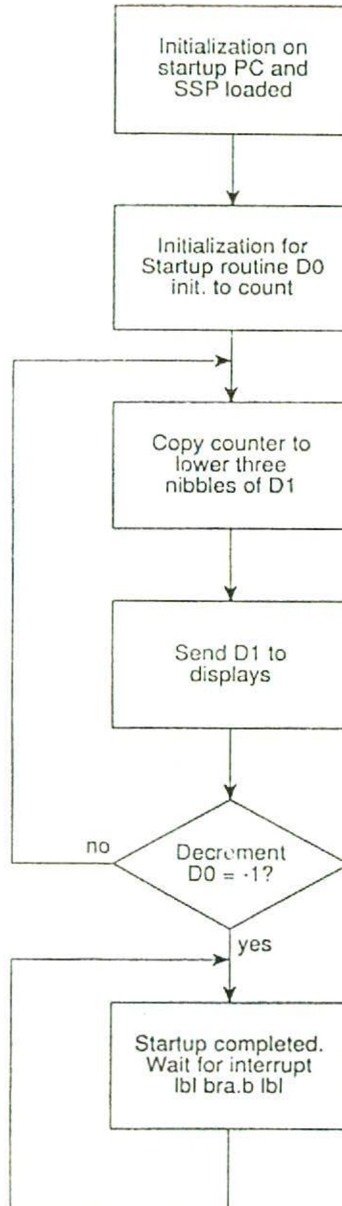


FIGURE 10.12 Start-up routine flowchart.

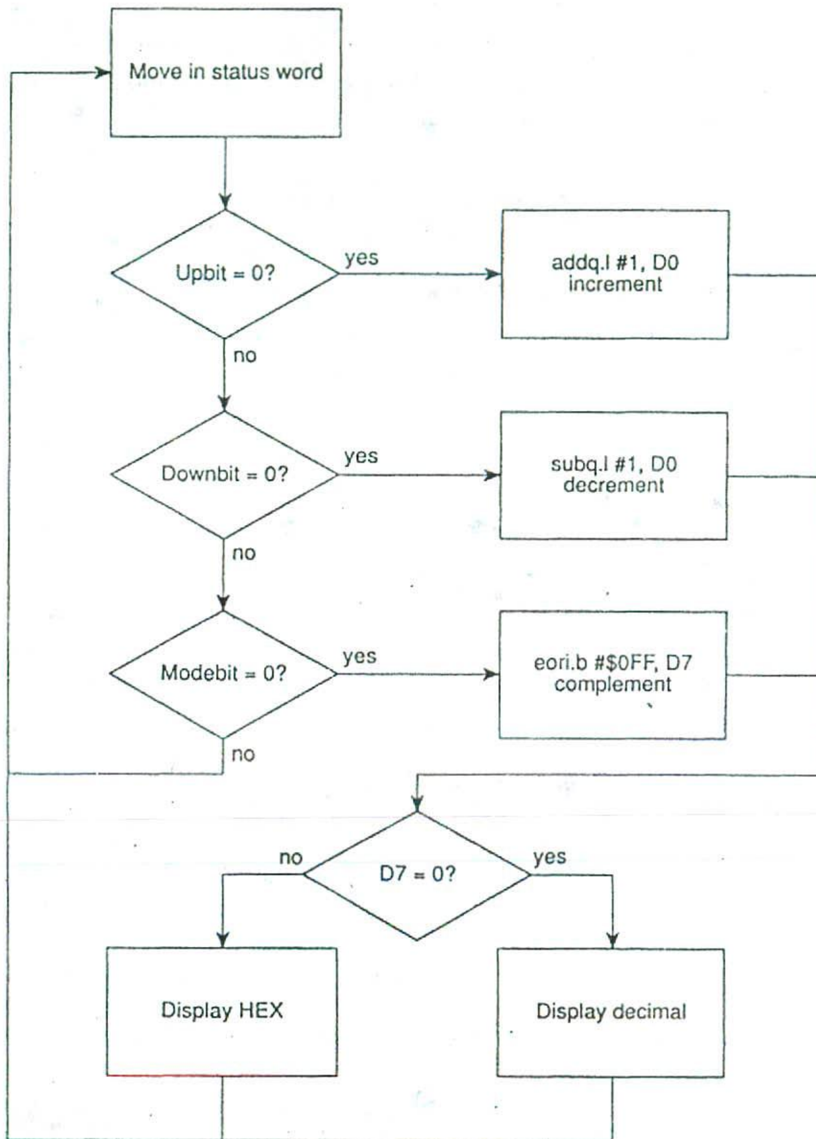


FIGURE 10.12 *continued.*

Questions and Problems

Design and develop the software and hardware for the following using a particular microprocessor (unless mentioned) and its support chips with a microcomputer development system of your choice.

10.1 Design and develop the hardware and software for a microprocessor-based system that would measure, compute, and display the Root-Mean-Square (RMS) value of a sinusoidal voltage. The system is required to:

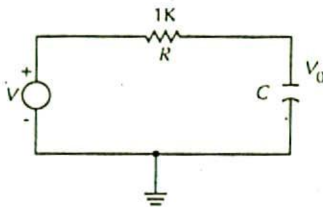
1. Sample a 60-Hz sinusoidal voltage 128 times.
2. Digitize the sampled value through a microprocessor-controlled analog-to-digital converter.

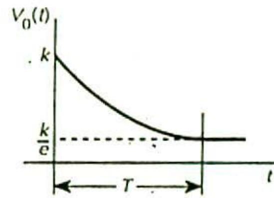
3. Input the digitized value to the microprocessor using an interrupt.
4. Compute the RMS value of the waveform using the equation

$$\text{RMS value} = \sqrt{\frac{\sum Xi^2}{N}}$$

where Xi's are the samples and N is the total number of samples.
5. Display the RMS value using two digits.

10.2 Design a microcomputer-based capacitance meter using the following RC circuit:





The voltage across the capacitor is $V0(t) = ke^{-t/RC}$. In one time constant RC, this voltage is discharged to the value k/e. For a specific value of R, the value of the capacitor C = t/R, where t is the time constant that can be counted by the microcomputer. Design the hardware and software for a microprocessor to charge a capacitor by using a pulse to a voltage of up to 10 V peak voltage via an amplifier. The microcomputer will then stop charging the capacitor, measure the discharge time for one time constant, and compute the capacitor value.

10.3 Design and develop the hardware and software for a microprocessor-based system to drive a four-digit seven-segment display for displaying a number from 0000H to FFFFH.

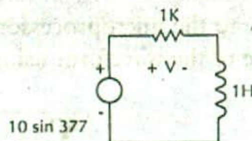
10.4 Design a microprocessor-based digital clock to display time in hours, minutes, and seconds on six-digit seven-segment displays in decimal.

10.5 Design a microcomputer-based temperature sensor. The microcomputer will measure the temperature of a thermistor. The thermistor controls the timing pulse duration of a monostable multivibrator. By using a counter to convert the timing pulse to a decimal count, the microcomputer will display the temperature in degrees Celsius.

10.6 Design a microprocessor-based system to test five different types of IC, namely, OR, NOR, AND, NAND, and XOR. The system will apply inputs to each chip and read the output. It will then compare the output with the truth table stored inside the memory. If the comparison passes, a red LED will be turned OFF. If the comparison fails, the red LED will be turned ON.

10.7 Design a microprocessor-based system that reads a thermistor via an A/D converter and then displays the temperature in degrees Celsius on three seven-segment displays.

10.8 Design a microprocessor-based system to measure the power absorbed by a 1K resistor. The system will input the voltage V across the 1K resistor and then compute the power using V^2/R.



10.9 It is desired to design a priority vectored interrupt system using a daisy-chain structure for a microcomputer. Assume that the system includes four interrupt devices DEV0, DEV1, ..., DEV3, which, during the interrupt sequence, place the respective instructions RST0, RST1, ..., RST3 on the data bus. Also assume that DEV0, ..., DEV3 are Teledyne 8703 A/D converters (DEV3 highest, DEV0 lowest priority) or equivalent.

i) Flowchart the problem to provide service routines for inputting the A/D converters' outputs.
ii) Design and develop the hardware and software.

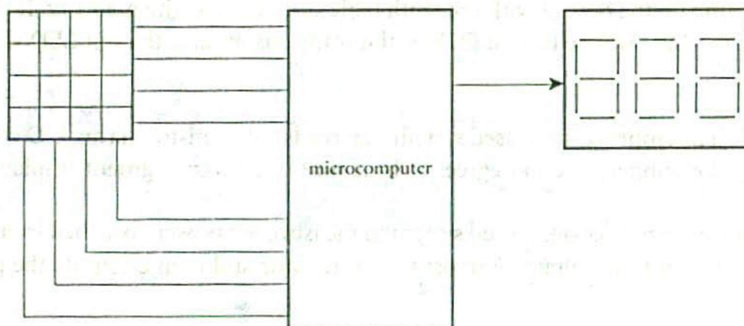
10.10 It is desired to drive a six-digit display through six output lines of a microcomputer system. Use six Texas Instruments TIL311, 14-pin MSI hexadecimal displays or equivalent:

i) Design the interface with minimum hardware.
ii) Flowchart the software.
iii) Convert the flowchart to the assembly language program.
iv) Implement the hardware and software.

10.11 Design a microcomputer-based combinational lock which has a combination of five digits. The five digits are entered from a hexadecimal keyboard and they are to be entered within 10 s. If the right combination is entered within the same limit, the lock will open. If after 10 s either all five digits are not entered or a wrong combination is entered, then the display will show an error signal by displaying "E". The system will then allow 5 s for the first digit to be entered the second time. If after this time the digit is not entered, the system will turn ON the alarm. If the second try fails, the alarm is also turned ON. When the alarm is ON, in order to reset the system, power has to be turned OFF.

10.12 Design a microcomputer-based stopwatch. The stopwatch will operate in the following way: the operator enters three digits (two digits for minutes and one digit for tenths of minutes) from a keyboard and then presses the GO key. The system counts down the remaining time on three seven-segment LED displays.

10.13 Design a microcomputer-based system as shown in the following diagram. The system scans a 16-key keyboard and drives three seven-segment displays. The keyboard is a 4 × 4 matrix. The system will take each key pressed and scroll them in from the right side of the displays and keep scrolling as each key is pressed. The leftmost digit is just discarded. The system continues indefinitely. Do not use any keyboard encoder chip. Use the 68000 micropreccessor.



10.14 Design a microcomputer-based smart scale. The scale will measure the weight of an object in the range of 0–5 lb. The scale will use a load cell as a sensor such as the one manufactured by transducer, Inc. (Model # c462-10#-10pl strain-gage load cell). This load cell converts a weight in the range of 0–10 lb to an analog electrical voltage in the range 0–20 mV. The weight in lbs. and tenths should be displayed onto two BCD displays.

10.15 Design a microcomputer-based EPROM programmer to program a 2716.

10.16 Design a microcomputer-based system to control a stepper motor.

10.17 Design a microcomputer-based sprinkler control system.

10.18 Design a phone call controller. The controller will allow the user to pass only ten random phone numbers chosen by the user. The controller will use the touch-tone frequencies to encode the user information code numbers. A device will be used to decode the touch-tone signals and convert each into a seven-bit word. A microprocessor will then interpret this word and see if it is a match with one of the ten different numbers chosen by the user. The ten numbers are inputted by the user via the * button from the touch-tone system. The controller will have a manual override via the # button from the touch-tone system.

10.19 Design a microprocessor-based appointment reminder system with a clock. The system will alert the user before the present appointment time. The user has to set the appointments into fixed slots; for example: 9 AM or 2 PM. The system will deliver a voice message such as "Your next appointment is five minutes away" five minutes before the appointment time. A real time clock is to be included in the system to display the current time and will show the appointment time slots. You may use the Radio Shack SP0256 narrator speech processor.

10.20 Design a microcomputer-based autoranged ohmmeter with a range of 1 ohm to 999 kohm as follows: the microcomputer generates a pulse to charge a capacitor up to 10 V peak voltage through an amplifier and then stops charging the capacitor. The microcomputer measures the discharge time of the capacitor for one time constant and then computes the value of the resistor.

10.21 Interface two microcomputers to a pair 2K × 8 dual-ported RAMs (IDT7132) without using any bus locking mechanism. Two seven-segment displays will serve as an indicator. A program will be written to verify the dual-ported RAM contents. One processor will write some known data to the dual-ported RAM and the other processor will read and verify this data against the known data.

10.22 Design a microcomputer-based low frequency (1 Hz to 10 kHz) sine waveform generator. One cycle of a sine wave will be divided into a certain number of equal intervals. Each interval is defined as a phase increment. The precalculated sine values corresponding to the intervals are stored in ROM. The frequency of the signal will be set up by switches. When the system is started, the microprocessor will read the switches and will determine the time delay corresponding to the phase increment. The microprocessor will follow the time increments to send data to a D/A converter to convert the digital signal to an analog signal.

10.23 Design a microcomputer-based automobile alarm system. The purpose of this system is to prevent intruders from stealing a car or having enough time to steal a stereo or other valuable items in a car.

10.24 Design a microcomputer-based three-axis robot arm controller. The microcomputer will perform the calculations and the I/O to control movement of the arm. The microcomputer

will receive destination data from an external source and perform coordinate transformations and boundary checking on the external data. It will then provide motor commands to the motor controllers to move the arm to the desired position.

10.25 Design a microcomputer-based home controller system. The system will simultaneously control six sprinkler stations, a heater, an air conditioner, and a burglar alarm. The system will contain a 12-hour clock and a temperature sensor. The user will program the system through a keypad. The time and temperature will be entered to control the sprinklers, the heater, and the A/C. The alarm will be armed or disarmed by entering a 4-digit code.

10.26 Design a microcomputer-based FM modulator. The microcomputer will read an analog input, convert the signal to digital, and perform several data manipulations to generate a digital representation of the FM signal. Finally, the microcomputer will convert the FM value to an analog signal.

10.27 Design and develop a microcomputer-based system for FFT (Fast Fourier Transform) computation. The microcomputer will sample eight data points using an A/D converter and compute the time-decimation FFT. After computation of the FFT, the result will be stored in system RAM where it can be used by another program for signal processing.