Course Ware reproduction for COMP ENG 4DL4

Microcomputer Systems

Liu and Gibson

Prentice Hall

Chapter 11 pgs 450 to 477

11 Multiprocessor Configurations

In Chap. 9 it was seen that a DMA controller could improve the system throughput by concurrently performing I/O as the CPU continued its processing. This is possible because the CPU does not utilize all of the bus cycles. An 8086 typically uses only 50 to 80 percent of the available bus time, depending on the application. A DMA controller can steal bus cycles to transfer data while only minimally affecting the processing. Also, it releases the CPU from performing the relatively slow I/O functions. Such a system is a simple case of having more than one processor in the system, with both processors operating in parallel to improve the system performance.

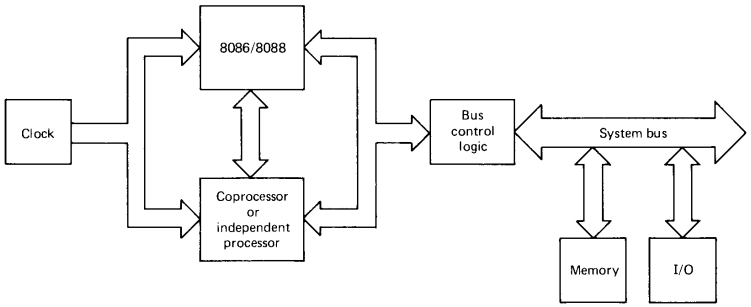
Although the capability of a DMA controller is rather limited, the concept of concurrent operations, which has been proved to be an effective way of improving a system's operation, can be extended to components that are somewhat more complex. If a system includes two or more components that can execute instructions simultaneously, it is called a *multiprocessing system*. The added processors could be special purpose processors which are specifically designed to perform certain tasks efficiently, or other general purpose processors. For example, due to the 8086's limited data width and its lack of floating point arithmetic instructions, it requires many instructions to perform a single floating point operation. For a system requiring a lot of computations, it would be desirable to perform such calculations with a supporting numeric data processor that is specifically designed to quickly operate on floating point numbers and numbers having larger than usual data widths. Also, it is sometimes advantageous to include in a system an I/O processor with greater capabilities than a DMA controller, one that can perform string manipulations, code conversion, character searching, and bit testing as well as the normal DMA operations. This would permit the CPU to concentrate on higher-level functions.

As the cost/performance ratio of single-chip microprocessors declines, it becomes more cost effective to use multiple processors than to use a single complex multiple-chip processor in what has come to be known as the centralized approach. In addition to improving the overall cost/performance ratio of a system, a multiprocessor configuration offers several desirable features not found in a one-processor design. First, several processors may be combined to fit the needs of an application while avoiding the expense of the unneeded capabilities of a centralized system. Yet the modularity of a multiprocessor system provides room for expansion because it is easy to add more processors as the need arises. Second, in a multiprocessor system tasks are divided among the modules. Should a failure occur, it is easier and cheaper to find and replace the malfunctioning processor than it is to find and replace the failing part in a complex processor.

Two problems must be considered in designing a multiprocessor system, bus contention and interprocessor communication. Because more than one processor shares the system memory and I/O devices through a common system bus, extra logic must be included to ensure that only one processor has access to the system bus at a time. In order for one processor to dispatch a task or return a result to another processor, an unambiguous way for the two processors to interact must be provided. How the bus contention and processor communication problems are resolved dictates the connections between the processors.

The maximum mode of the 8086 and 8088 is specifically designed to implement multiprocessor systems. Multiprocessing features are provided in maximum mode to accommodate three basic configurations. They are the coprocessor, closely coupled, and loosely coupled configurations. The first two of these configurations are very similar in that both the CPU and the supporting, or external, processor share not only the entire memory and I/O subsystem, but as shown in Fig. 11-1, they also share the same bus control logic and clock generator. In both of these con-

Figure 11-1 Closely coupled configuration.



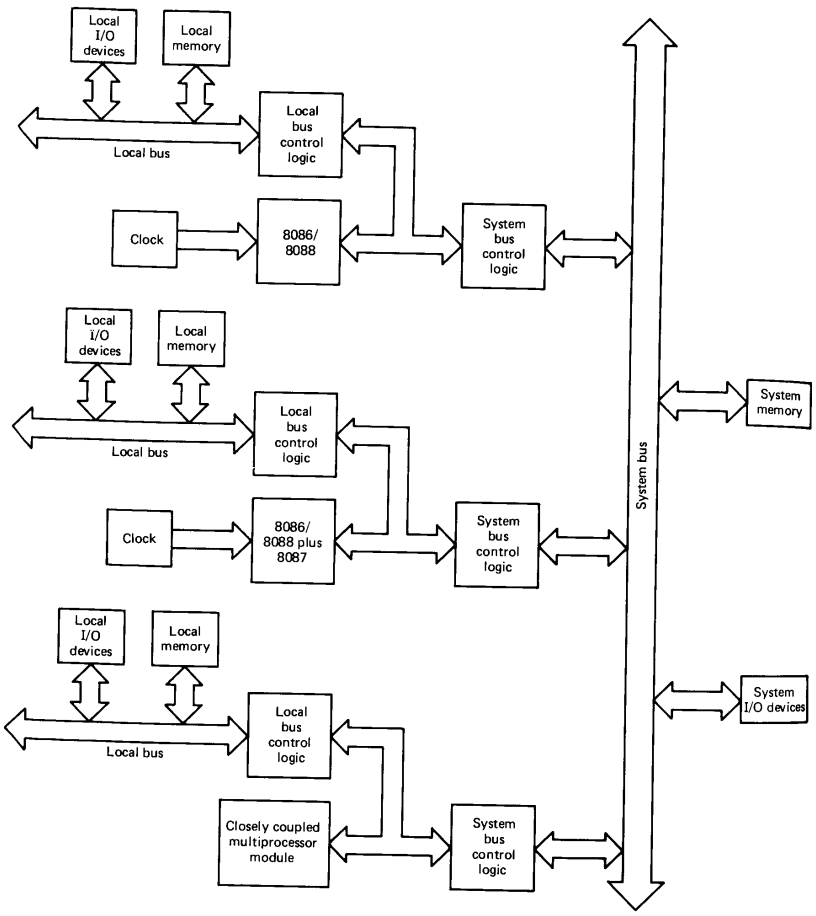


Figure 11-2 Loosely coupled configuration.

figurations, the 8086/8088 is the master, or host, and the supporting processor is the slave. The bus access control is provided by the CPU; therefore, the bus request signal from the supporting processor is connected to the CPU. In a closely coupled configuration the supporting processor may act independently, but in a coprocessor design it is dependent and must interact directly with the CPU. In a coprocessor arrangement there are more direct lines between the processing elements. Since the 8086/8088 always acts as the host in a coprocessor or closely coupled design, two 8086/8088s cannot appear in these configurations.

Loosely coupled configurations are used for medium-size to large systems. Each module in the system may be the system bus master and may consist of an 8086, an 8088, another processor capable of being a bus master, or a coprocessor or closely coupled configuration. Several modules may share the system resources, and system bus control logic must resolve the bus contention problem. As shown in Fig. 11-2, each potential bus master runs independently and there are no direct connections between them. Interprocessor communication is made possible through the shared resources. In addition to the shared resources, each module may include its own memory and I/O devices. The processors in the separate modules can simultaneously access their private subsystems through their local buses and perform their local data references and instruction fetches independently, thus improving the degree of concurrent processing.

This chapter first discusses the multiprocessing features of 8086/8088 processors, including their special instructions and control signals. Implementation of the various 8086/8088 multiprocessor configurations are examined in Sec. 11-2. The last two sections describe Intel's 8087 numeric data processor and 8089 integrated I/O processor, respectively.

11-1 QUEUE STATUS AND THE LOCK FACILITY

Although the maximum mode and the 8288 bus controller were introduced in Chap. 8, their multiprocessing features were not considered at that point. It is the purpose of this section to extend that discussion to multiprocessing situations, even though detailed discussions of their applications is left to later sections.

Because the 8086 has a 6-byte instruction queue and the 8088 has a 4-byte queue, the instruction that has just been fetched may not be executed immediately. In order to allow external logic to monitor the execution sequence, a maximum mode 8086/8088 outputs the queue status through its QS1 and QS0 pins. During each clock cycle the queue status is examined and the QS1 and QS0 bits are encoded as follows:

00—No instruction was taken from the queue.

01—The first byte of the current instruction was taken from the queue.

10—The queue was flushed because of a transfer instruction.

11—A byte other than the first byte of an instruction was taken from the queue.

By monitoring the bus and the queue status, external logic can simulate the CPU's instruction execution sequence and determine which instruction is currently being executed. This facility is necessary so that the 8086/8088 can indicate when an instruction is to be executed by a coprocessor.

As pointed out in Chap. 7, it is necessary to control the accesses to the shared resources in a multiprogramming environment. Normally, semaphores are used to ensure that at any given time only one process may enter its critical section of code in which a shared resource is accessed. Let us now reconsider the semaphore implementation:

```
            MOV     AL,0
TRYAGAIN:   XCHG    SEMAPHORE,AL
            TEST    AL,AL
            JZ      TRYAGAIN
             .      Critical section in which a process
             .      accesses a shared resource
             .
            MOV     SEMAPHORE,1
```

This implementation works fine for a system in which all of the processes are executed by the same processor, because the processor cannot switch from one process to another in the middle of an instruction. However, if competing processes are running on different processors, the situation is more complex.

Suppose that processor A is concurrently ready to update a record in memory while processor B is ready to sort the same record. Since both processors are running independently, they might test the semaphore at the same time. Note that the XCHG instruction requires two bus cycles, one which inputs the old semaphore and one which outputs the new semaphore. It is possible that after processor A fetches the semaphore, processor B gains control of the next bus cycle and fetches the same semaphore.

Suppose that the location SEMAPHORE contains a 1 and both processors A and B are executing

```
TRYAGAIN:   XCHG    SEMAPHORE,AL
```

and

1. Processor A uses the first available bus cycle to get the contents of SEMAPHORE.
2. Processor B uses the next bus cycle to get the contents of SEMAPHORE.
3. Processor A clears SEMAPHORE during the next bus cycle, thus completing its XCHG instruction.
4. Processor B clears SEMAPHORE during the next bus cycle, thus completing its XCHG instruction.

After this sequence is through, the AL registers in both processors will contain 1 and the

```
TEST    AL,AL
```


instructions will cause the JZ instructions to fail. Therefore, both processors will enter their critical sections of code.

To avoid this problem, the processor that starts executing its XCHG instruction first (which in this example is processor A) must have exclusive use of the bus until the XCHG instruction is completed. On the 8086/8088 this exclusive use is guaranteed by a LOCK prefix:

1	1	1	1	0	0	0	0

which for a maximum mode CPU, activates the $\overline{\text{LOCK}}$ output pin during the execution of the instruction that follows the prefix. The $\overline{\text{LOCK}}$ signal indicates to the bus control logic that no other processors may gain control of the system bus until the locked instruction is completed. To get around the problem encountered in the above example the XCHG instructions could be replaced with:

```
TRYAGAIN:   LOCK  XCHG  SEMAPHORE,AL
```

This would ensure that each exchange will be completed in two *consecutive* bus cycles.

Physically, in a loosely coupled system each processing module includes a bus arbiter and the bus arbiters are connected together by special control lines in the system bus. One of these lines is a busy line which is active whenever the bus is in use. When a processing module's arbiter is given control of the bus it will activate the busy line, which will prevent other arbiters from seizing the bus until after the next bus cycle. If a $\overline{\text{LOCK}}$ signal is sent to the arbiter controlling the bus, then that arbiter will retain control of the system by holding the busy line active until the $\overline{\text{LOCK}}$ signal is dropped. Thus, if a processor applies a $\overline{\text{LOCK}}$ signal throughout the execution of an entire instruction, its arbiter will not relinquish the system bus until the instruction is complete. Bus arbiters are discussed further in Sec. 11-2-3, where loosely coupled configurations are considered in detail.

In a module including a coprocessor or closely coupled configuration, if a bus request is made through one of the 8086/8088's $\overline{\text{RQ}}/\text{GT}$ pins while the $\overline{\text{LOCK}}$ pin is active, then the request will be held until the $\overline{\text{LOCK}}$ signal is dropped. Then the 8086/8088 will respond to the request by returning a grant. In addition to being activated by a LOCK prefix, an interrupt request on a CPU's INTR pin will cause the $\overline{\text{LOCK}}$ pin to be held low from the beginning of the first $\overline{\text{INTA}}$ pulse until after the second $\overline{\text{INTA}}$ pulse. This guarantees availability of the bus until the completion of an interrupt cycle.

Another possible application of the bus lock capability is to allow fast execution of an instruction which requires several bus cycles. For example, in a multiprocessor system a block of data can be transferred at a higher speed by using the LOCK prefix as follows:

```
LOCK    REP    MOVS    DEST,SRC
```

During the execution of this instruction the system bus will be reserved for the sole use of the processor executing the instruction.

A prefix is considered an extension of the instruction following the prefix therefore, interrupts that occur during the execution of an instruction having a LOCK prefix are not honored until the entire instruction is completed. However if LOCK is combined with a REP prefix as in the above example, interrupts may be acknowledged at the end of each move operation. Furthermore, upon return from the interrupt, only the prefix immediately preceding the MOVS instruction is restored. Therefore, it is usually best to disable interrupts before executing a LOCK REP combination such as the one indicated above.

In addition to the multiprocessing capabilities of the 8086/8088, the 8288 bus controller also provides control functions for supporting loosely coupled systems in which an 8288 is used in conjunction with a bus arbiter. These control functions are made possible by the presence of the address enable ($\overline{AEN}$), I/O bus mode (IOB), command enable (CEN), and master cascade enable/peripheral data enable (MCE/$\overline{PDEN}$) pins. There applications are examined in the loosely coupled configurations discussion in Sec. 11-2-3. In the previous discussions the $\overline{AEN}$, IOB, CEN, and MCE/$\overline{PDEN}$ pins were connected for using the 8288 in a single CPU design.

11-2 8086/8088-BASED MULTIPROCESSING SYSTEMS

Let us now consider the three fundamental multiprocessor configurations that the 8086 and 8088 are designed to support. This section gives a general description of how the 8086 and 8088 general purpose microprocessors are used as the dominant processors in these configurations, and the succeeding sections provide specific examples based on other Intel devices with processing capability.

11-2-1 Coprocessor Configurations

Although the 8086 and 8088 are powerful single-chip microprocessors, their instruction set is not sufficient to effectively satisfy some complex applications. For such applications, the 8086/8088 must be supplemented with coprocessors that extend the instruction set in directions that will allow the necessary special computations to be accomplished more efficiently. For example, the 8086/8088 has no instructions for performing floating point arithmetic, but by using an Intel 8087 numeric data processor as a coprocessor, an application that heavily involves floating point calculations can be readily satisfied.

It will be seen that, except for the coprocessor itself, a coprocessor design does not require any extra logic other than that normally needed for a maximum mode system. Both the CPU and coprocessor execute their instructions from the same program, which is written in a superset of the 8086/8088 instruction set. Other than possibly fetching an operand for the coprocessor, the CPU does not need to take any further action if the instruction is executed by the coprocessor.

The interaction between the CPU and the coprocessor when an instruction is executed by the coprocessor is depicted in Fig. 11-3. An instruction to be executed

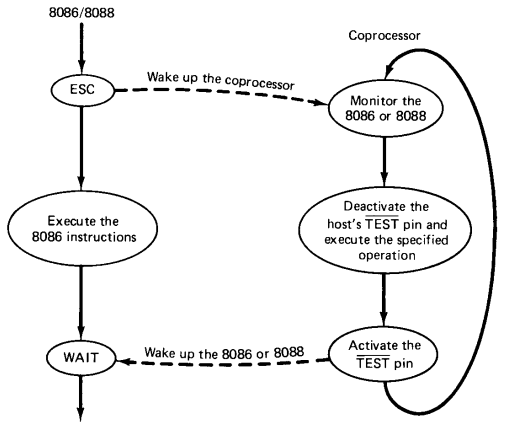


Figure 11-3 Synchronization between the 8086 and its coprocessor.

by the coprocessor is indicated when an escape (ESC) instruction appears in the program sequence. Only the host CPU can fetch instructions, but the coprocessor also receives all instructions and monitors the instruction sequencing of the host. An ESC instruction contains an external operation code, indicating what the coprocessor is to do and is simultaneously decoded by both the coprocessor and the host. At this point the host may simply go on to the next instruction or it may fetch the first word of a memory operand for the coprocessor and then go on to the next instruction. If the CPU fetches the first word of an operand, the coprocessor will capture the data word and its 20-bit physical address. For a source operand that is longer than one word, the coprocessor can obtain the remaining words by stealing bus cycles. If the memory operand specified in the ESC instruction is a destination, the coprocessor ignores the data word fetched by the host and later the coprocessor will store the result into the captured address. In either case, the coprocessor will send a busy (high) signal to the host's $\overline{TEST}$ pin and, as the host continues processing the instruction stream, the coprocessor will perform the operation indicated by the code in the ESC instruction. This parallel operation continues until the host needs the coprocessor to perform another operation or must have the results from the current operation. At that time the host should execute a WAIT instruction and wait until its $\overline{TEST}$ pin is activated by the coprocessor. The WAIT instruction repeatedly checks the $\overline{TEST}$ pin until it becomes activated and then executes the next instruction in sequence.

An ESC assembler language instruction has two operands. The first of two operands indicates the external op code, which determines the action to be taken by the coprocessor. If the second operand specifies a memory location, then as

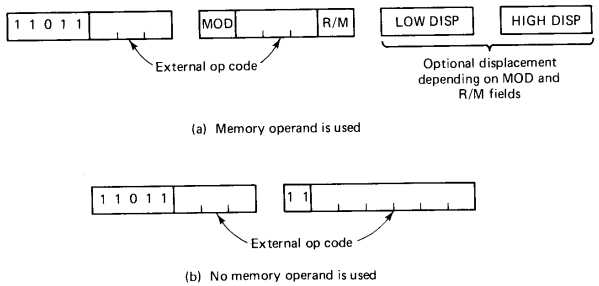
explained above, the 8086/8088 will fetch a word from this location for the coprocessor and may pass the coprocessor an address for storing a result. If the second operand is a register, the register address is treated as part of the external op code and the CPU does nothing.

The machine code for the ESC instruction may have either of the two forms given in Fig. 11-4. In both cases the first byte consists of 11011 followed by 3 bits of the external op code. For the form in Fig. 11-4(a) the second byte specifies a memory addressing mode and 3 more bits of the external op code, thus permitting up to 64 distinct external op codes. If the addressing mode calls for a displacement, this form is extended to include 1 or 2 bytes for holding the displacement. The second byte of the form in Fig. 11-4(b) consists of 11 followed by 6 additional external op code bits. This gives a total of 9 external op code bits and permits up to 512 of these op codes.

The interfacing of a coprocessor to a host CPU is shown in Fig. 11-5. Both the host and the coprocessor share the same clock generator and bus control logic. It is possible to have two coprocessors connected to the same host CPU. When this is done, the coprocessors must be assigned distinct subsets of the set of external op codes and each coprocessor must be able to recognize and execute the members of its subset. For the most part parallel lines can be used to connect the host to its coprocessors. For two coprocessors, one could use the $\overline{RQ}/\overline{GT0}$ pin on the 8086/8088 and the other could use the $\overline{RQ}/\overline{GT1}$ pin. The two coprocessors would be connected to separate 8259A interrupt request pins.

In order for a coprocessor to determine when the host is executing an ESC instruction, it must monitor the host CPU's status on the $\overline{S2}$–$\overline{S0}$ lines and the AD15–AD0 lines for fetched instructions. Because instructions are prefetched by the CPU, an ESC instruction might not be executed immediately or, if it is preceded by a branch instruction, it might not be executed at all. The coprocessor must track the instruction stream by monitoring the queue status bits QS1 and QS0 and maintaining an instruction queue identical to that of the host. If the queue's status is 00, the coprocessor does nothing, but if it is 01, it will compare the five MSBs of the first

Figure 11-4 Machine code formats for the ESC instruction.



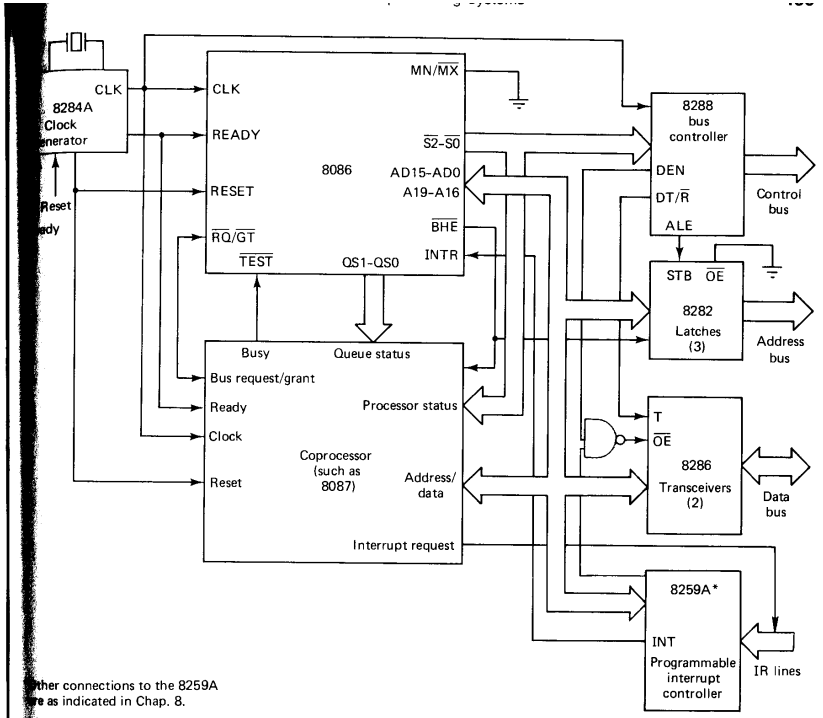


Figure 11-5 Coprocessor configuration.

byte in the queue to 11011. If there is a match, then an ESC instruction is ready to be executed and, assuming that the coprocessor recognizes the external op code, it will perform the indicated operation; otherwise, this byte is ignored and is deleted from the queue. The queue status 10 indicates that the queue in the host is being flushed and, therefore, causes the queue in the coprocessor to also be emptied. The 11 status combination indicates that the first byte in the queue is not the first byte of an instruction and this byte is looked at by the coprocessor only if it is known to be part of an ESC instruction. If not part of an ESC instruction, this byte is ignored.

The coprocessor should be designed so that when an error occurs during the decoding or execution of an ESC instruction, it will send out an interrupt request (which is normally sent to an 8259A). The coprocessor should also be designed so that it can steal bus cycles by making bus requests through one of the host's $\overline{RQ}$/

$\overline{GT}$ pins when additional data must be read from or stored in memory. Last, the coprocessor must be able to apply a high signal to the host's $\overline{TEST}$ pin while it is busy.

It is necessary for a coprocessor to know whether it is working with an 8086 or an 8088 because they have different data bus widths and instruction queue lengths. The type of host can be determined each time the system is brought up by having the coprocessor examine the host's pin 34 immediately following the RESET. This pin is always high on a maximum mode 8088 and on an 8086 it is the $\overline{BHE}$/S7 pin. Recall that the first instruction following a reset is taken from address FFFF0. Therefore, the first address on the bus is always even and $\overline{BHE}$ is always initially low. In order for the coprocessor to know when to check pin 34 it must be connected to the same reset line as the 8086 or 8088.

When both a coprocessor and an independent processor which fetches its own instructions are connected to a CPU, the coprocessor must be able to determine whether an instruction is being fetched by the independent processor or the CPU; otherwise, the instruction queue in the coprocessor may be erroneously updated. This can be done by monitoring the S6 status bit, which is always low for the 8086 and 8088 and always high for the 8089.

Although at the present time the only Intel device which can be used as a coprocessor with the 8086/8088 is the 8087, one could design a customized coprocessor to facilitate any particular application. However, any coprocessor design must be compatible with a maximum mode 8086/8088 system. It must be able to read the CPU status and queue status, make bus and interrupt requests, receive reset and ready signals, receive bus grants, maintain an instruction queue, and decode the external op codes in the ESC instructions.

11-2-2 Closely Coupled Configurations

The 8086/8088 also supports another type of external processor, referred to as an independent processor, which, unlike a coprocessor, executes its own instruction stream. To minimize costs, the independent processor can be tied to the CPU to form a closely coupled multiprocessor system in which both share the same clock and bus control logic. For instruction fetches and data references, the independent processor accesses the bus through the CPU's $\overline{RQ}/\overline{GT}$ lines.

Instead of using special instructions such as ESC and WAIT, communication between the host and the independent processor is accomplished through shared memory space. As illustrated in Fig. 11-6, the host sets up a message in memory and then wakes up the independent processor by sending a command to one of the independent processor's ports. The independent processor then accesses the shared memory to get the assigned task and executes the task in parallel with the host. After the task is completed, the external processor notifies its host of the completion by using either a status bit or an interrupt request. The message format varies depending on the design of the independent processor and the application. Typically, a message should specify which operation is to be performed, the input parameters, and the addresses of the locations in which to store the results. An

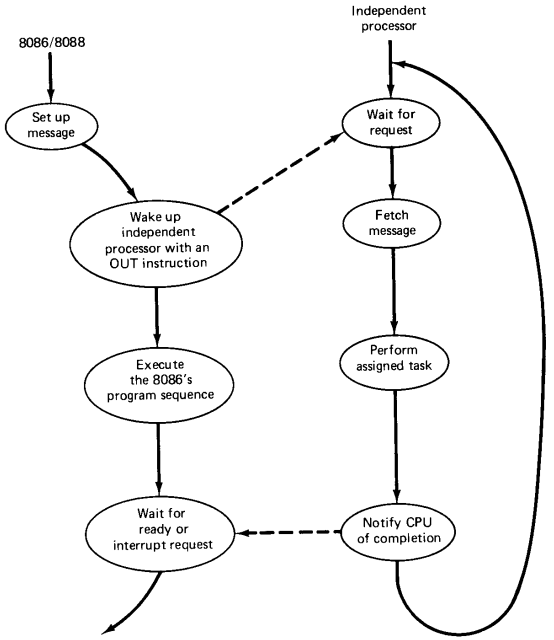


Figure 11-6 Interprocessor communication through shared memory.

example of an independent processor is the Intel 8089 I/O processor and the message layout for the 8089 is described in Sec. 11-4.

A typical connection between the 8086 and an independent processor in a closely coupled configuration is given in Fig. 11-7. Since an independent processor executes its own program, the host's queue status bits are not monitored. The independent processor requests bus accesses via a request/grant line. When one processor is using the bus, the other processor forces its status and address/data outputs to their high-impedance state so that they are logically disconnected from the bus. To wake up the independent processor, the host executes an OUT instruction to output to a port that is assigned to the independent processor. Upon completion of a given task, the independent processor sets a status bit in the shared memory space and may also generate an interrupt. Because the 8086 and 8088 have two request/grant pins, more than one independent processor and/or coprocessor can be connected to the host, thus providing multiprocessing capabilities at a

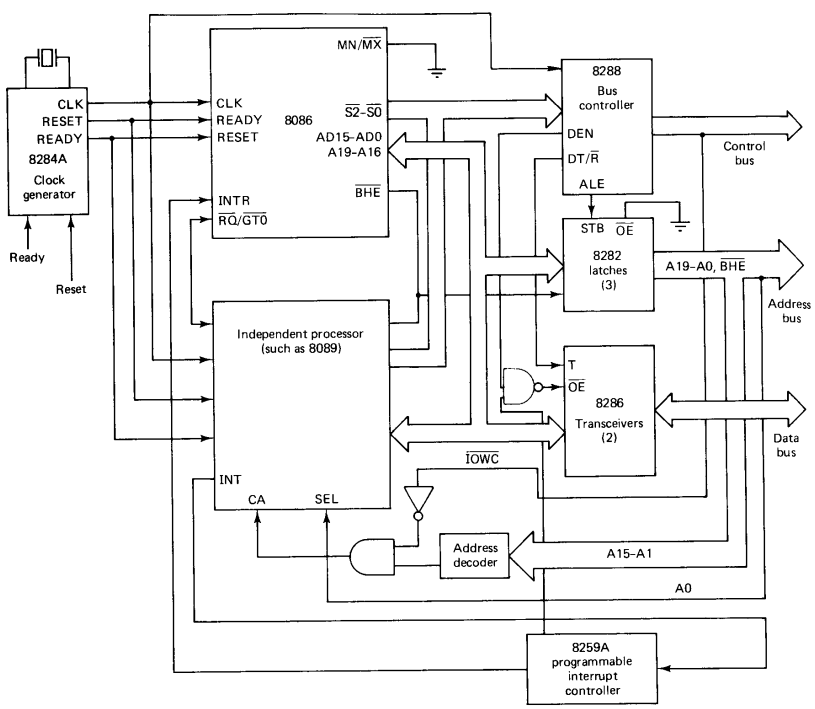
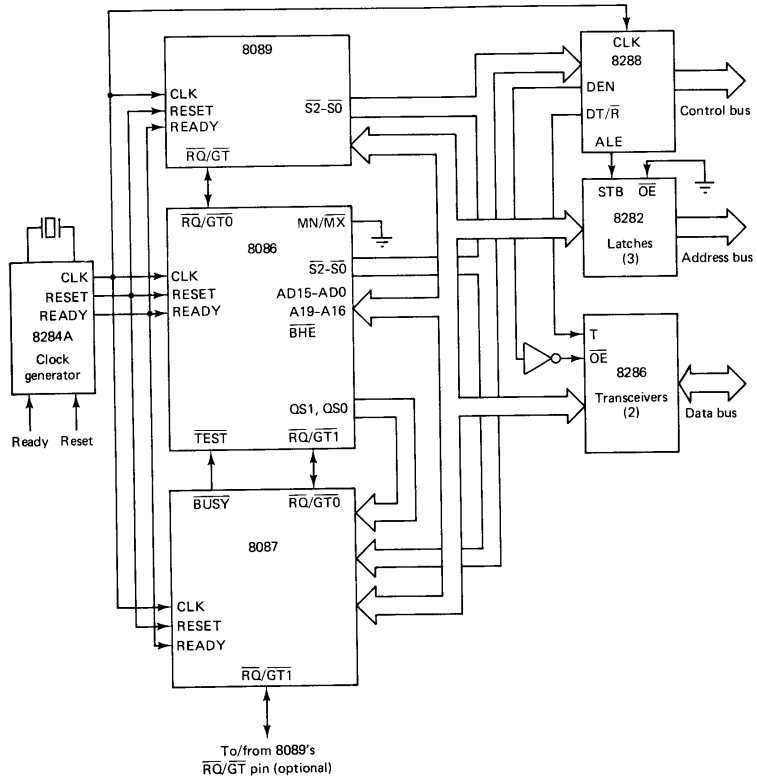


Figure 11-7 8086 connections in a closely coupled configuration.

minimal cost. Closely coupled configurations typically comprise small to medium-size multiprocessor systems. Figure 11-8 illustrates how the coprocessor and closely coupled configurations may be combined. In the example, the 8087 and 8089 share the bus control logic with the host 8086, which provides the bus arbitration function. Because $\overline{RQ}/\overline{GT0}$ is connected to the 8089, for simultaneous requests the 8086 always gives higher priority to the 8089. However, when the 8087 is using the bus, an 8089's request will not be honored until the 8087 releases the bus. To reduce the maximum wait time, the 8089's $\overline{RQ}/\overline{GT}$ line could be connected to the $\overline{RQ}/\overline{GT1}$ pin of the 8087. When this is done, a bus request from the 8089 will force the 8087 to release the bus after the current bus cycle even though it may need several more bus cycles to complete its current instruction.



NOTE: Interrupt system and other details are not shown.

Figure 11-8 Configuration involving both a coprocessor and an independent processor.

11-2-3 Loosely Coupled Configurations

In a multiprocessor system, two 8086s or 8088s cannot be tied directly together. In a loosely coupled multiprocessor system, each CPU has its own bus control logic and bus arbitration is resolved by extending this logic and adding external logic that is common to all master modules. Therefore, several CPUs can form a very

large system and each CPU may have independent processors and/or a coprocessor attached to it. A loosely coupled configuration provides the following advantages:

1. High system throughput can be achieved by having more than one CPU.
2. The system can be expanded in a modular form. Each bus master module is an independent unit and normally resides on a separate PC board. Therefore, a bus master module can be added or removed without affecting the other modules in the system.
3. A failure in one module normally does not cause a breakdown of the entire system and the faulty module can be easily detected and replaced.
4. Each bus master may have a local bus to access dedicated memory or I/O devices so that a greater degree of parallel processing can be achieved.

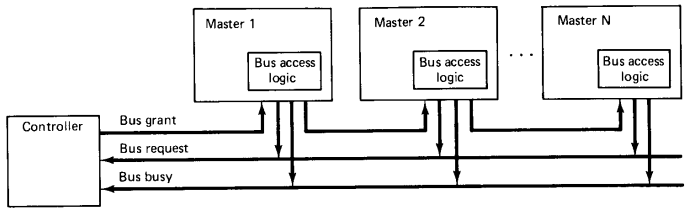
In a loosely coupled multiprocessor system, more than one bus master module may have access to the shared system bus. Since each master is running independently, extra bus control logic must be provided to resolve the bus arbitration problem. This extra logic is called *bus access logic* and it is its responsibility to make sure that only one bus master at a time has control of the bus. Simultaneous bus requests are resolved on a priority basis. There are three schemes for establishing priority:

1. Daisy chaining.
2. Polling.
3. Independent requesting.

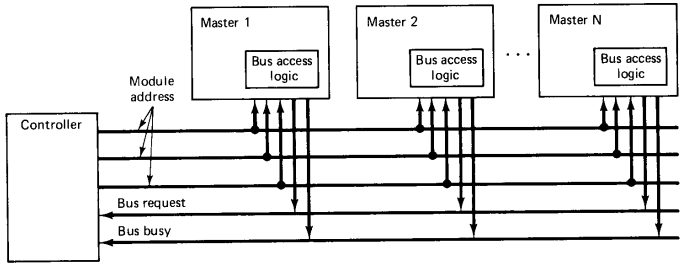
Figure 11-9 illustrates the general concepts underlying these schemes. Implementation of the control logic may vary depending on the complexity of the bus access logic included in each module.

The daisy chain method is characterized by its simplicity and low cost. All masters use the same line for making bus requests. To respond to a bus request signal, the controller sends out a bus grant signal if the bus busy signal is inactive. The grant signal serially propagates through each master until it encounters the first one that is requesting access to the bus. This module blocks the propagation of the bus grant signal, activates the busy line, and gains control of the bus. Therefore, any other requesting module will not receive the grant signal and the priority is determined by the physical locations of the modules. The one located closest to the controller has the highest priority.

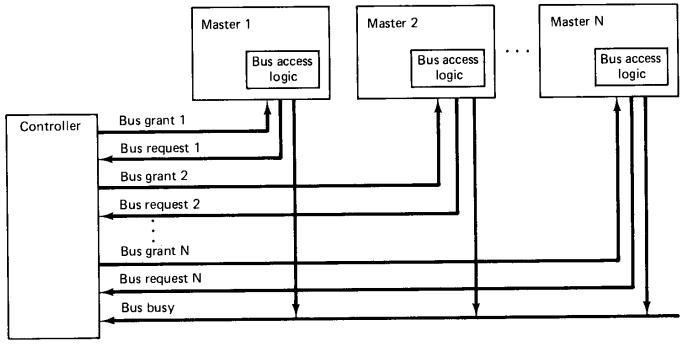
Compared to the other two methods, the daisy chain scheme requires the least number of control lines and this number is independent of the number of modules in the system. However, the arbitration time is slow due to the propagation delay of the bus grant signal. This delay is proportional to the number of modules and, therefore, a daisy chain-based system is limited to only a few modules. Furthermore, the priority of each module is fixed by its physical location and the failure of a module causes the whole system to fail.



(a) Daisy chain method



(b) Polling method



(c) Independent requests method

Figure 11-9 Bus allocation schemes.

The polling scheme uses a set of lines sufficient to address each module. In response to a bus request, the controller generates a sequence of module addresses. When a requesting module recognizes its address, it activates the busy line and begins to use the bus. The major advantage of polling is that the priority can be dynamically changed by altering the polling sequence stored in the controller.

The independent requests scheme resolves priority in a parallel fashion. Each module has a separate pair of bus request and bus grant lines and each pair has a priority assigned to it. The controller includes a priority decoder, which selects the request with the highest priority and returns the corresponding bus grant signal. Arbitration is fast and is independent of the number of modules in the system. Compared to the other two methods, the independent requests design is the fastest; however, it requires more bus request and bus grant lines ($2n$ lines for n modules).

A module's host 8086 or 8088 lacks the capability of requesting bus access and recognizing bus grants. Therefore, as noted above, it is necessary for each module containing a bus master to have extra logic for sending and receiving bus access signals. The Intel 8289 bus arbiter is specifically designed to provide the necessary bus access handshaking. Operating in conjunction with the bus controller, the bus arbiter controls the access of its associated master to the bus by using either the daisy chain or the independent requests scheme.

Figure 11-10 shows the connections between the bus master and the bus arbiter, assuming that all memory and I/O devices are global and are shared by the bus masters through the system bus. Other situations in which a bus master may have access to its private memory or I/O devices through a local bus will be considered shortly.

As shown in the figure, the 8289 inputs and monitors the CPU's status bits $\overline{S2}$–$\overline{S0}$ to determine when to request or release the bus. The 8288 also monitors the status of the CPU to detect the beginning of a bus cycle. After the CPU initiates a bus cycle, a signal on the ALE line causes the address to be latched into the 8283s. If the bus arbiter is currently in control of the bus, it enables the outputs of the 8288 bus controller and the 8283 address latches, and the DEN output of the 8288 enables the data transceivers. Thereafter, the bus cycle proceeds in the normal way. If the bus arbiter does not presently control the bus, then it raises its $\overline{AEN}$ pin, forcing the command outputs of the 8288 and the address outputs of the 8283s into their high-impedance state. Since the DEN output of the 8288 is also controlled by the $\overline{AEN}$ line, the data transceivers are disabled. In addition, a high on the $\overline{AEN}$ line prevents the clock generator from sending a ready signal to the CPU, so that the CPU enters a wait state immediately after the T3 state of the current cycle.

Once the bus arbiter is granted the bus access, it activates the $\overline{BUSY}$ and $\overline{AEN}$ pins. The $\overline{AEN}$ signal causes the address followed by the command signals (after a delay for setup time) to be placed on the system bus, and enables the data transceivers by way of the DEN line. After the data transfer is complete, the addressed location returns an acknowledge signal via the ready line, which causes the 8284A to activate the READY pin. The CPU then exits the wait state and completes its current bus cycle.

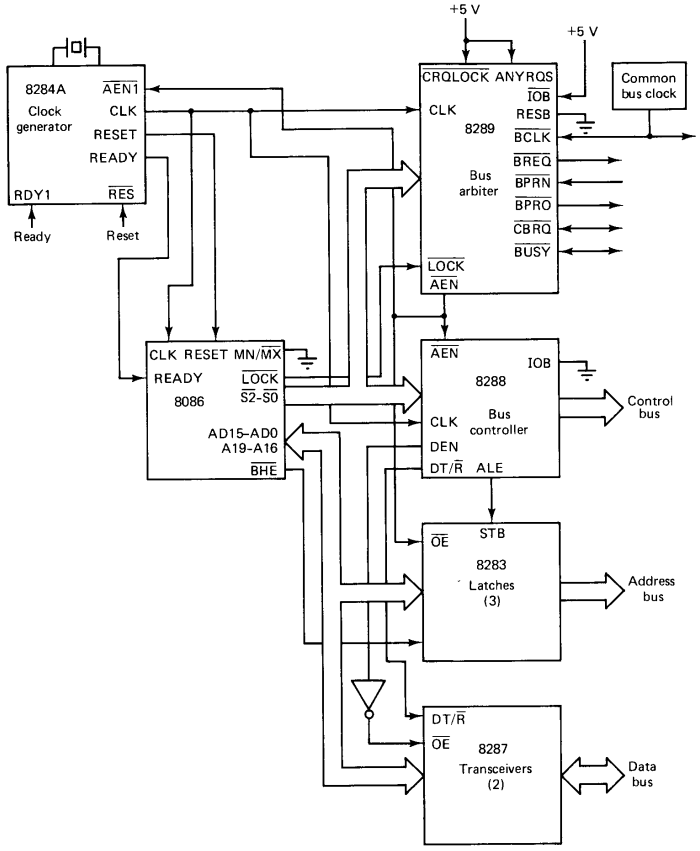


Figure 11-10 Single-processor module.

The $\overline{LOCK}$ output of the CPU is directly connected to the bus arbiter. A low on the $\overline{LOCK}$ line prevents the 8289 from relinquishing the bus. The RESB (resident bus mode) and $\overline{IOB}$ (I/O bus mode) pins are strapping options which permit the processor to access local memory and I/O devices. Because the design in Fig. 11-10 assumes that all resources are shared, both of these modes are disabled.

Four of the other 8289 pins are for bus request, grant, and release lines. The 8289 provides these pins to support the independent requests (parallel) and the daisy chain (serial) accessing schemes. Figure 11-11 shows how bus arbiters are connected to resolve bus priority using the independent requests method. For a system using this design the $\overline{\text{BREQ}}$ (bus request) line from each arbiter is fed into a priority resolving controller. The priority encoder generates a binary number corresponding to the bus request input with the highest priority. This number is decoded to return a bus grant signal to the selected arbiter through its $\overline{\text{BPRN}}$ (bus priority in) pin. Because the timing of each module is controlled by an independent clock, it is necessary to have a separate common clock to synchronize the bus requests.

The $\overline{\text{CBRQ}}$ (common bus request) and $\overline{\text{BUSY}}$ (busy) pins provide a means for the current bus master to release the bus. Both pins are bidirectional with open collector outputs. Their connections are also shown in Fig. 11-11. When a module needs to frequently access the system bus, it is better for its arbiter to retain the bus (except when its processor is inactive) until it is forced off by another module. Otherwise, each bus cycle would incur an overhead due to the request and release steps. A bus master of higher priority can acquire the bus via its $\overline{\text{BREQ}}$ line and the priority selection logic. After the current bus master completes its bus cycle, it releases the bus and deactivates $\overline{\text{BUSY}}$. The requesting master which accepts the $\overline{\text{BPRN}}$ signal then activates $\overline{\text{BUSY}}$ and begins to use the bus.

A bus arbiter of lower priority may use the $\overline{\text{CBRQ}}$ line to acquire the bus. This, of course, would require that the $\overline{\text{CBRQ}}$ pins be connected together as shown in Fig. 11-11. When the $\overline{\text{CBRQ}}$ is low, the current master will surrender the bus if it is in a TI state. The priority resolving logic then allocates the bus to the requesting arbiter which has the next higher priority by putting a signal on the corresponding $\overline{\text{BPRN}}$ line.

In connection with allowing lower-priority devices to take control of the bus, the 8289 provides two inputs, ANYRQST (any request) and $\overline{\text{CRQLCK}}$ (common bus request lock). The ANYRQST pin is a strapping option. When ANYRQST is tied to +5 V, an assertion of $\overline{\text{CBRQ}}$ forces the bus arbiter to release the bus at the end of the current bus cycle regardless of the priority of the requesting bus master. The $\overline{\text{CRQLCK}}$ input allows an 8289 to ignore a $\overline{\text{CBRQ}}$ signal. Therefore, activating $\overline{\text{CRQLCK}}$ prevents an 8289 from releasing the bus to any other bus master having a lower priority.

The need for having an external priority controller is eliminated by using the daisy chain bus allocation scheme illustrated in Fig. 11-12. The $\overline{\text{BPRO}}$ (bus priority out) of each module is connected to the $\overline{\text{BPRN}}$ (bus priority in) of the next lower priority bus master in the chain. For any processing module in the chain, a low input on $\overline{\text{BPRN}}$ indicates that it has the highest priority and may acquire the bus. If its $\overline{\text{BPRN}}$ pin is high, it will not have received the grant signal and will not be able to pass on a grant signal to its $\overline{\text{BPRO}}$. The $\overline{\text{BPRN}}$ signal of the highest-priority arbiter is grounded; therefore, the $\overline{\text{BPRN}}$ pin of the requesting arbiter with the highest priority will be low and, for arbiters farther down the chain, it will be high.

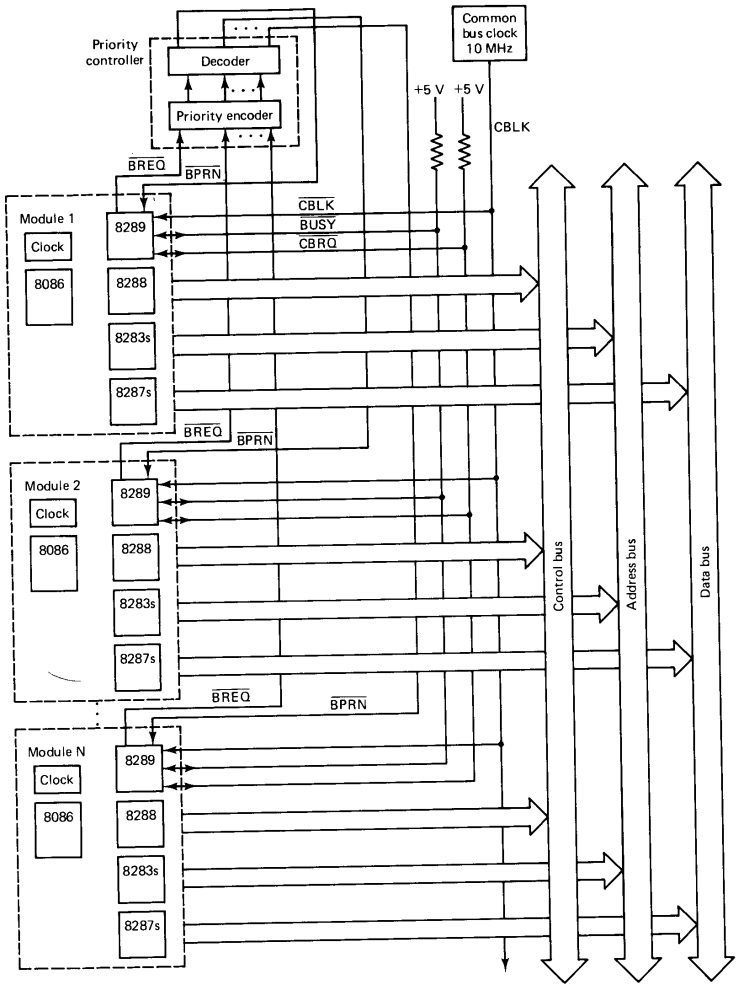


Figure 11-11 8289 connections to a priority controller.

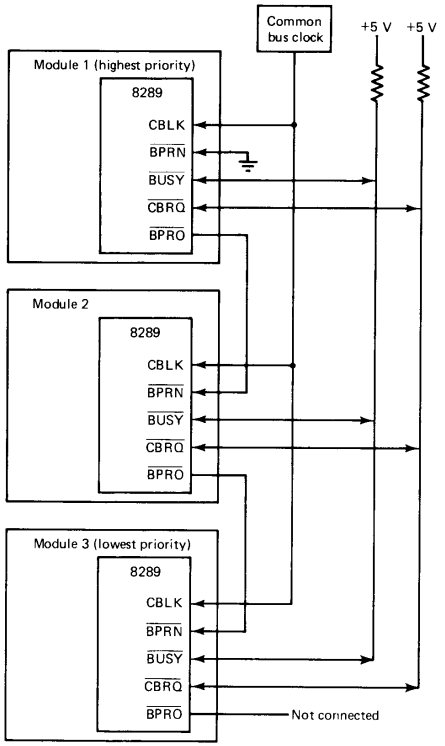


Figure 11-12 Daisy chain bus allocation using 8289s.

Since the bus allocation must be resolved in a $\overline{\text{BCLK}}$ clock period, the timing is normally such that a chain cannot include more than three processing modules.

As more processing modules are tied to the system bus, the bus utilization will soon become saturated. Bus congestion seriously degrades the performance of a system and tends to negate the speed advantage of multiprocessing. A rough analysis illustrating the influence of the number of processors on system performance can be obtained by considering k identical modules. It is reasonable to say that the instruction execution rate of each processor is proportional to the rate at which that processor accesses the bus. Assume that the bus bandwidth is N cycles per second and that without interference each module in the system uses M bus cycles per second. The bus utilization by a module is defined as the average fraction

of bus cycles used by the module assuming no interference and is $B_P = M/N$. This ratio must be less than 1 and will depend on the instructions being executed. For the 8086, it will typically range from 0.5 to 0.8. Let I_S be the system instruction execution rate (i.e., the number of instructions executed by the system per second) and I_P be the instruction execution rate of each module assuming no interference. Then

$$I_S \simeq kI_P \qquad \text{if} \quad k \leq \frac{1}{B_P}$$

and

$$I_S \simeq \frac{I_P}{B_P} \qquad \text{if} \quad k > \frac{1}{B_P}$$

A representative graph of I_S versus the number of processing modules for $B_P = 1/2$ is shown in Fig. 11-13(a). Although I_S is approximately equal to kI_P for an underutilized system, I_S is actually less than kI_P because a module will enter a wait state if it needs to access the system bus and the bus is being used by another processor. It is important to note from the figure that once the bus is saturated, adding more modules will no longer improve the system performance. The bus becomes the bottleneck of the system and the system is said to be *bus bound*.

The utilization of each module, which is defined as the ratio of the instruction execution rate (in the presence of other modules) to I_P, is somewhat degraded when bus congestion becomes serious, i.e., kM approaches N. This utilization is

$$U_m \leqq \frac{N/k}{M} = \frac{N}{kM} = \frac{1}{kB_P}$$

Figure 11-13(b) illustrates the influence of the number of modules on the performance of each module for the case $B_P = 1/2$.

The most effective solution to the bus congestion problem is for each processing module to include a local memory dedicated to its CPU. Each module would use its local memory to store its program and as a working area, and the shared memory would be primarily used for intermodule communications. This would allow several CPUs to fetch instructions or reference operands simultaneously, thus reducing the traffic on the system bus and increasing the degree of concurrent processing.

As illustrated in Fig. 11-14, if a processing module has a local bus, then it must include two sets of bus control logic, one for accessing the system bus and the other for the local bus. Each set would need to contain a bus controller, address latches, and data transceivers. Since the local bus is dedicated to only one CPU, it does not require a bus arbiter and its address latches are always enabled (assuming there is no local DMA controller). During a bus cycle, the address is decoded to determine if it is in the local space or shared space. If the address is in the local space, the local bus controller is enabled and a bus cycle proceeds immediately. Meanwhile, the output of the decoder disables the system bus controller (and causes the bus arbiter to release the system bus if the arbiter has control of it). If the

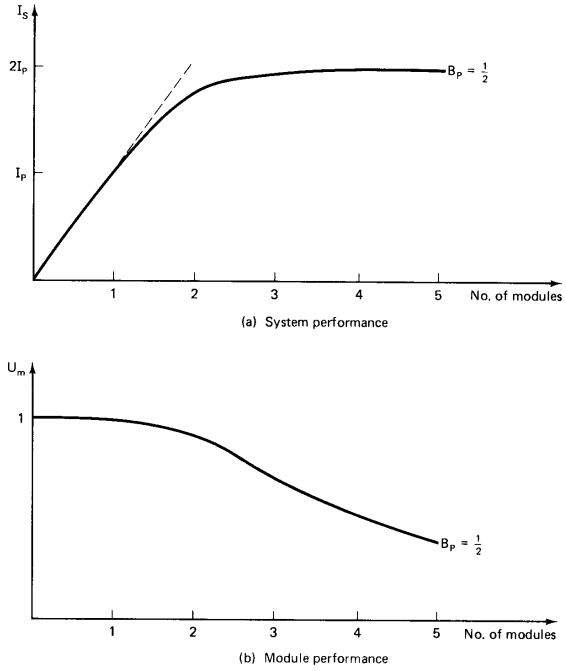


Figure 11-13 Multiprocessing system performance criteria.

address is in the address space of the system bus, the address decoder activates the SYSB/$\overline{\text{RESB}}$ pin on the 8289 and a system bus access is attempted. In this event, the local bus controller is disabled, preventing it from sending out a bus command, and the system bus interface is enabled. The bus arbiter then requests the use of the system bus and initiates a bus cycle when it becomes the arbiter having the highest priority.

As shown in Fig. 11-14, the RESB and $\overline{\text{IOB}}$ lines are tied to +5 V. With local resources, each processing module would normally access the system bus only occasionally. Therefore, it is desirable to enable ANYRQST so that the 8289 may release the bus to lower-priority processing modules as soon as the current bus cycle is complete.

Figure 11-15 gives another configuration that reduces the traffic on the system bus, but results in a simpler design than the one shown in Fig. 11-14. This config-

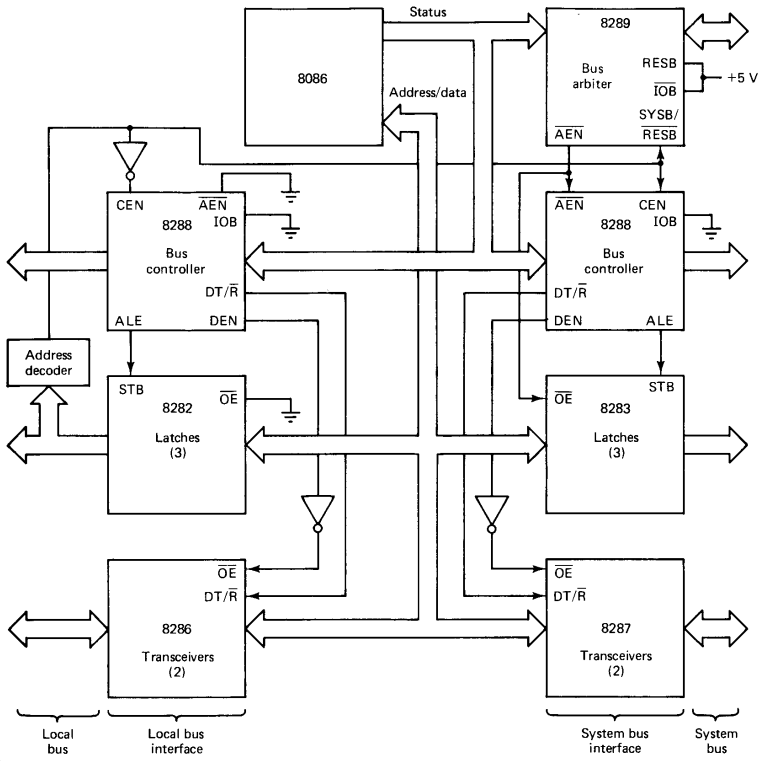


Figure 11-14 Configuration with both a local bus and a system bus.

uration locates all the I/O on a local bus and all of the memory on the system bus. The separation of the I/O and memory address spaces permits special facilities of the 8289 to direct the address and control signals flow and eliminates the need for the address decoder and local 8288 bus controller. By strapping the RESB pin on the 8289 low the SYSB/$\overline{\text{RESB}}$ is ignored, and by strapping the $\overline{\text{IOB}}$ pin low on the 8289 and high on the 8288 these devices are put into their I/O bus mode. In this mode the system bus is requested and surrendered according to the $\overline{\text{S2}}$ line, which indicates whether an I/O (low) or memory (high) transfer is to be conducted. The $\overline{\text{AEN}}$ output of the 8289 is still used to control the address/data and memory

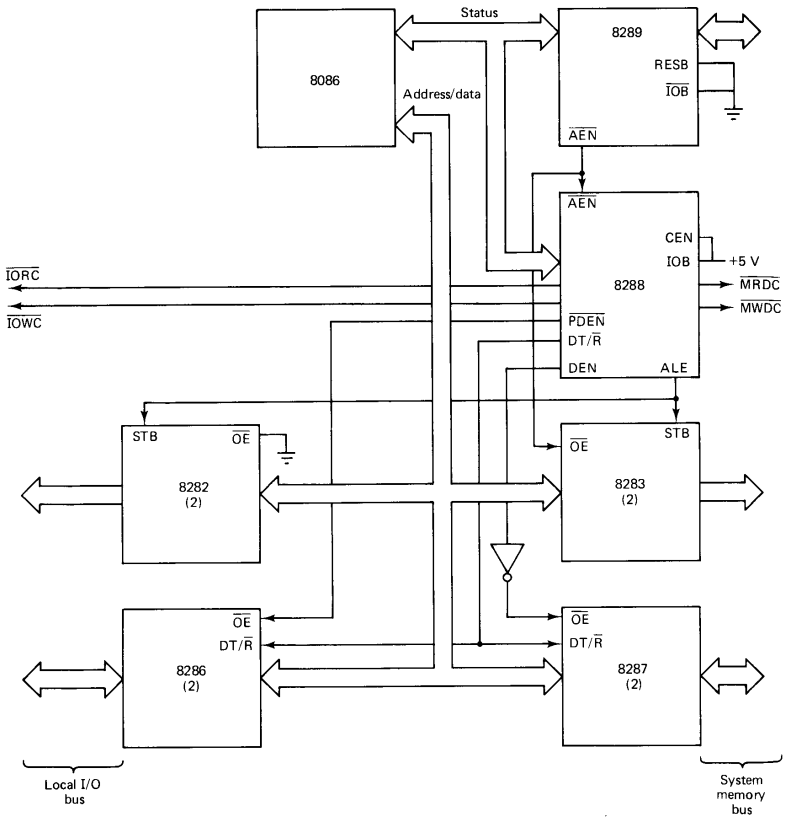


Figure 11-15 Separate local I/O bus configuration.

command lines but has no effect on the I/O control lines $\overline{\text{IORC}}$, $\overline{\text{IOWC}}$, $\overline{\text{AIOWC}}$, and $\overline{\text{INTA}}$ (whose outputs are always enabled). With IOB on the 8288 strapped high the MCE/$\overline{\text{PDEN}}$ pin becomes the peripheral data enable pin, which serves to enable the I/O bus data transceivers. When an I/O transfer is being made $\overline{\text{PDEN}}$ is active and DEN is inactive. For a memory transfer DEN is active and $\overline{\text{PDEN}}$ is inactive. This configuration is particularly useful when implemented in conjunction

with an 8089 I/O processor. It is also possible to connect I/O interfaces to the system bus, but their port addresses must be addressed as memory (i.e., memory-mapped I/O devices).

The local bus mode and the I/O bus mode may be combined by strapping the RESB and $\overline{\text{IOB}}$ pins of the 8289 to high and low, respectively. The resulting configuration allows part of the memory to be placed on the local bus along with the I/O devices.

It has been pointed out that local memory reduces the use of the system bus in a multiprocessor system. However, a processing module may still need to initially load the control program from the shared system memory to its local memory, and move results from the local memory to the system memory if the results are needed by other modules. A dual-port memory can be used to further reduce the traffic on the system bus by handling the interprocessor communications. This is a hardware concept that is similar to the software concept of using a common area to provide the communication between a calling program and a subprogram. Such a memory has two ports, to allow references from both the local bus and the system bus. As shown in Fig. 11-16, the processor of the master module in which the dual-port memory resides accesses the memory through the local bus. But other proc-

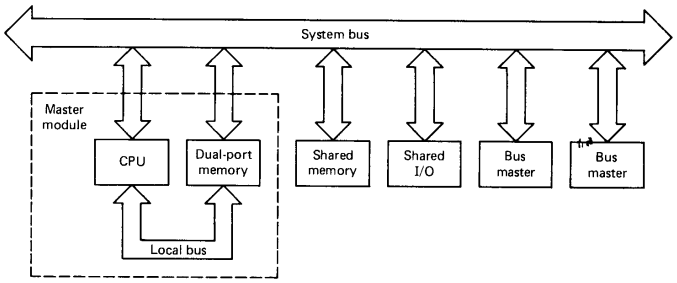


Figure 11-16 Dual-port memory.

essing modules may also access the same memory, although they can communicate with it only through the system bus. This configuration is particularly useful in inputting or outputting large blocks of data through a processing module which is dedicated to I/O processing. In this design, locking the system bus does not prevent the master module from accessing the dual-port memory, and vice versa. In order to use the scheme for avoiding the critical section problems described in Sec. 11-1, the semaphore should be placed in the shared system memory; otherwise, the implementation becomes more complicated.

In summary, processing modules of different configurations may be combined

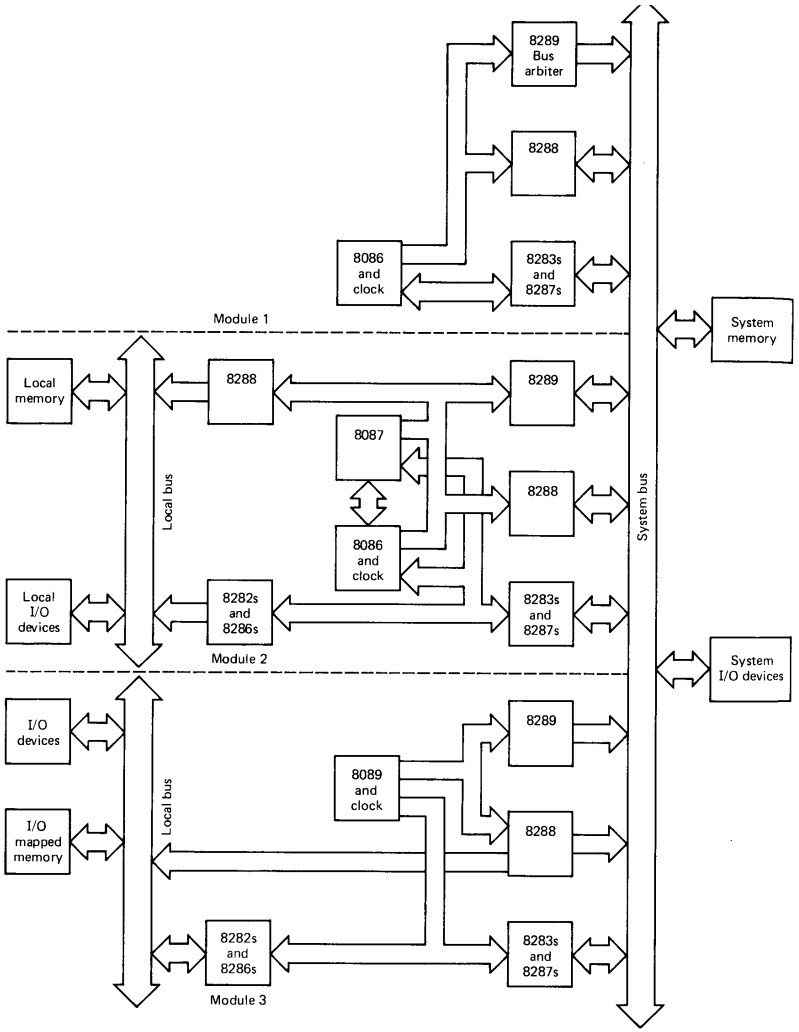


Figure 11-17 Complex multiprocessing system.

to form a complex, loosely coupled multiprocessor system. Each module in such a system may be:

1. A single 8086 or 8088 or an independent processor such as an 8089.
2. A cluster of processors consisting of an 8086 or 8088 and a coprocessor (such as an 8087) and/or independent processors.
3. A cluster of independent processors (such as two 8089s).

In addition, each module may include a local bus or a dedicated I/O bus. Figure 11-17 illustrates one of the many possible multiprocessing designs—designs that may include several complex processing modules.

11-2-4 Microcomputer Networks

The previous multiprocessor configurations have a common characteristic and that is that all processors share the same system bus. Thus, the interprocessor communications are through shared memory and processors must be physically located close to each other.

By using serial links, many microcomputer systems can communicate with each other and share some of the same hardware and software resources. Large systems of this type are called *computer networks*. Normally, synchronous serial transfer is employed to send messages between the systems. Commands and data are transmitted as message packets. Each packet normally includes several fields containing such things as synchronization characters, the sender's address, the receiver's address, the text, error detection characters, and termination characters. The controlling information and its arrangement is referred to as the system's *communication protocol*. Each microcomputer (or multiprocessor element) in the network is relatively independent of the others, and the microcomputers could be miles, or even thousands of miles, apart. A failure in one microcomputer element of the network could be easily isolated without affecting the remainder of the system.

11-3 THE 8087 NUMERIC DATA PROCESSOR

The 8087 numeric data processor (NDP) is specially designed to perform arithmetic operations efficiently. It can operate on data of the integer, decimal, and real types, with lengths ranging from 2 to 10 bytes. The instruction set not only includes various forms of addition, subtraction, multiplication, and division, but also provides many useful functions, such as taking the square root, exponentiation, taking the tangent, and so on. As an example of its computing power, the 8087 can multiply two 64-bit real numbers in about 27 μs and calculate a square root in about 36 μs. If performed by the 8086 through emulation, the same operations would require approximately 2 ms and 20 ms, respectively. The 8087 provides a simple and