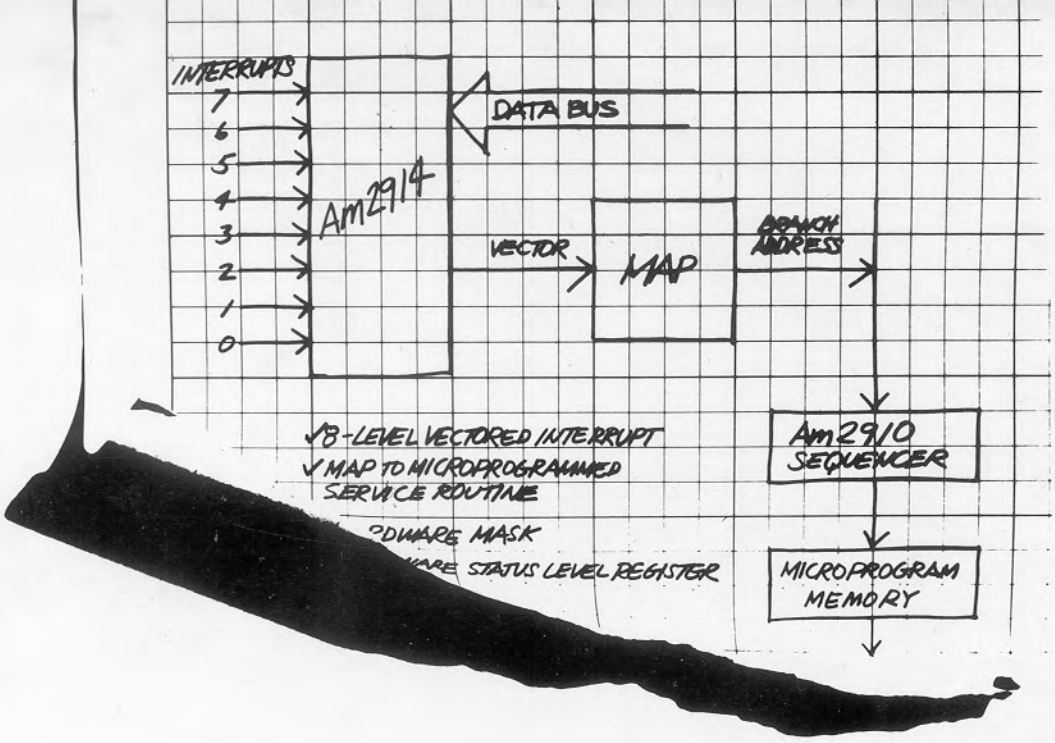



# Build A Microcomputer

## Chapter VI
## Interrupt

## Advanced Micro Devices

AM-PUB073-6

## INTRODUCTION

A digital computer can be viewed as a finite state machine that moves from state to state via the execution of a program. Interrupt mechanisms provide a well-defined way of altering the flow of states in response to outside asynchronous events (interrupts). There is a wide variety of ways of handling interrupts depending upon the system requirements. The choice of a particular interrupt mechanism can have a large impact on the through-put and flexibility of a system. Therefore, time should be spent carefully defining the interrupt mechanism of a new computer design.

## POLLING VS. NON-POLLING

One of the simplest ways to handle asynchronous events is the polling method. With each possible event there is an associated flag that can be accessed by the program. The processor then interrogates each flag in order to determine if service is required. This method trades simple hardware for software. This not only uses memory space but also uses time for polling the flags when no service is required. The polling method has low system through-put, high real time overhead and slow response time.

In non-polling systems, the asynchronous event generates an interrupt request signal which is passed to the processor. The processor in turn suspends the execution of the current process and starts execution of an interrupt service routine. When the interrupt routine is completed, the processor resumes execution of the suspended process. This system is called an interrupt driven system because it executes interrupt service routines that are initiated by interrupt requests.

Although the non-polling method requires more hardware, it has many advantages. Because the execution of interrupt service routines is transparent to the current process, less thought and time is required of the programmer of the current process. The response time is faster because no time is spent interrogating the other non-active interrupts, which in turn increases the system throughput. There is less real time overhead and less memory space required because only the service routine exists in memory and no polling routine is required.

## MACHINE VS. MICROPROGRAM LEVEL INTERRUPTS

There are two levels on which interrupts may be handled. The first and most common is the machine level interrupt. In this method possible interrupt requests are checked for during the machine instruction fetch cycle. This guarantees that an interrupt can only happen when a machine instruction is complete and before a new instruction starts.

The second level of handling interrupts is on the microprogram level. In the machine level interrupt system, the microprogram has complete control of when to recognize an interrupt but in the microprogram level system the microprogram can be interrupted at any time. This method has a smaller response time for servicing interrupt requests but requires that restrictions may be placed on the microprogram and the interrupt mechanism. These restrictions come from setting aside space on the finite microprogram stack in the sequencer for possible interrupt requests. Special consideration may also have to be given to loop counters.

## TYPES OF INTERRUPTS

There are basically four types of interrupts based on the relationship of the source of the interrupt to the processor: within the processor, within the system, between software, and between processors. A multiprocessor has to be able to handle all four levels of interrupts. Therefore, the interrupt structure that is picked will have these design tradeoffs to consider.

A. *Intraprocessor* interrupts are those asynchronous events that happen within the processor during the execution of a machine instruction. This group includes such things as zero divide, overflow, accessing restricted memory, execution of a privileged instruction, machine failure, etc.

B. *Intrasystem* interrupts are interrupts created by system peripherals such as disks, CRT's and printers that require service.

C. *Executive* interrupts are those interrupts caused by the current program that is executing. This provides a way for the current program to make a request of the executive (operating system) program. These requests might include such things as starting new tasks, allocating hardware resources (disks, line printers), communication with other tasks, etc. A good example would be the supervisor call (SVC) in the IBM 360/370 computers.

D. *Interprocessor* interrupts include those interrupts between two intelligent processors. For example, this class of interrupts would be used to initiate data and status transfer between a local processor and a processor at a remote site.

## SEQUENCE OF EVENTS FOR INTERRUPT HANDLING

When an interrupt occurs there is a sequence of six events that happen. These events, which can be implemented in microcode or machine code, integrated together with the hardware comprise the interrupt mechanism. The sequence of events describes the steps that occur to provide for a smooth transfer from the current process environment to an interrupt servicing environment and back again. The sequence ensures that the processor status will be the same immediately after an interrupt is serviced as immediately before the interrupt occurred. The events listed in the next few paragraphs may differ in order or overlap depending upon the machine design and application.

### Interrupt Recognition

This step consists of the recognition of an interrupt request by the processor via an interrupt request line. In this step the processor can determine which device made the request. The method that is used to determine which device to service is directly related to the interrupt structure of the machine. The different types of interrupt structures will be discussed in more detail below.

### Save Status

The goal of this step is to make the interrupt sequence transparent to the interrupted process. Therefore, the processor saves a minimum set of flags and registers that may be changed by the interrupt service routine, so that after the service routine is finished they may be restored.

The minimum set of flags and registers would be those which will be destroyed in the transfer of control from the current process to the interrupt service routine. It is then the responsibility of the service routine to save any other registers which it might change. The minimum set of flags and registers might include the Program Counter, Overflow Flag, Sign Flag, Interrupt Mask, etc. The minimum set also includes any register or flag that needs to be saved that the interrupt service routine cannot access.

### Interrupt Masking

This step can overlap some of the other steps. For the first few steps of the sequence all interrupts are masked out so that no interrupt may occur before the processor status is saved. The mask is then usually set to accept interrupts of higher priority.

Some machines allow the service routine to selectively enable or disable interrupts also. There may be different variations to this step depending upon the application.

### Interrupt Acknowledge

At some point the processor must acknowledge the interrupt being serviced so that the interrupting device knows that it is free to continue its task. The processor can acknowledge several different ways. One of the ways is to have a line devoted to interrupt acknowledge. Another method relies upon the interrupting device recognizing an acknowledge when the cause of the interrupt is serviced.

Some processor designs also use this signal as a request for the interrupting device to send an I.D. down the data bus. This aspect will be discussed in more detail below.

### Interrupt Service Routine

At this point the processor can call the interrupt service routine. The address of the routine can be obtained several ways depending upon the system architecture. The most trivial is when there is only one routine which polls each device to find out which one interrupted. Some designs require that the interrupting device put an address on the data bus so that the processor can store it in its program counter and branch to it. Other designs use an I.D. number derived from the priority of the interrupt and put it through a mapping PROM or look-up table in memory in order to obtain the address of the service routine.

### Restore and Return

After the interrupt service routine has returned via some variation of an Interrupt Return instruction, the processor should restore all the registers and flags that were saved previous to the interrupt routine. If this is done correctly, the processor should have the same status as before the interrupt was recognized.

## INTERRUPT STRUCTURES

There are several interrupt structures that can be implemented. As usual there is a trade-off between hardware and software (or firmware). Listed below are some of the more common structures used. The particular structures vary in the way that the processor determines which device made the interrupt request.

### Single Request, Multiple Poll

In this structure there is one request line which is shared among all interrupting devices. When the processor recognizes an interrupt request it polls all the devices to find the interrupting device (see Figure 1). Priority is introduced via the order in which the devices are polled. This scheme also allows dynamic reallocation of priority.

### Single Request, Daisy Chain Acknowledge

In this structure there is one request line which is shared. When the processor receives an interrupt it sends out a signal acknowledging the interrupt. The acknowledge signal is passed from I/O device to I/O device until the interrupting device receives the signal. At this point the interrupting device identifies itself by putting an I.D. number on the data bus (see Figure 2). This structure requires less software, but has a static priority associated with each interrupting device. There is also a time delay associated with daisy chain acknowledge structure because in each device INTA signal has to pass through several gate delays.

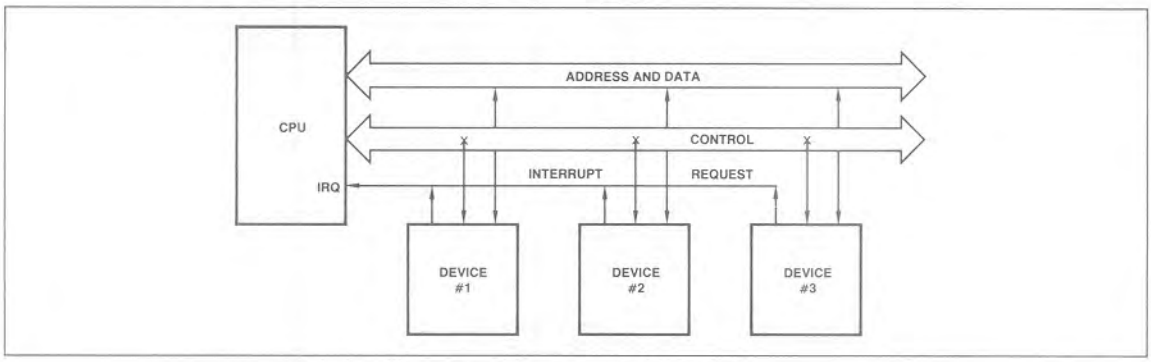



**Figure 1. Single Request, Multiple Poll.**

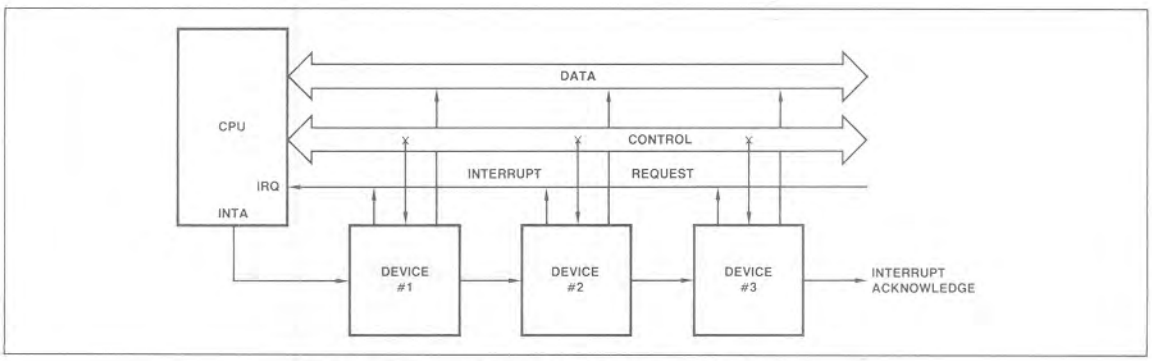



**Figure 2. Single Request, Daisy Chain Acknowledge.**

### Multiple Request

This structure features one line per priority level (see Figure 3). The multiple line structure gives the fastest response time since the interrupting device can be identified immediately. It also results in simpler interfaces in the peripheral units, in general, a single interrupt request flip-flop. This structure allows for the possibility of having a mask bit associated with each priority level (device). The trade-off of this circuit is a wider bus and a limit of one peripheral per priority level.

### Multiple Request, Daisy Chain Acknowledge

This structure combines the Single Request/Daisy Chain Acknowledge with the Multiple Request structure (see Figure 4). For each interrupt request line there is an interrupt acknowledge line which is connected to a string of devices in a daisy chain fashion. When the appropriate device receives the interrupt acknowledge, it puts an I.D. number on the data bus.

The advantage of this structure is that a lot (more than available interrupt levels) of devices may be handled by breaking them up into short daisy chains. This gives a shorter access time than a pure daisy chain with less hardware than an interrupt request line per device. This advantage is that each device must be intelligent to pass on the acknowledge signal which requires more hardware in each device.

## PRIORITY SCHEMES

When handling asynchronous requests one must assume that sometimes two or more requests can happen simultaneously. In order to handle this situation, there must be some sort of priority scheme implemented to pick which request is serviced first.

The two most common priority schemes are the static and the rotating structures. In the static structure, all the interrupt levels are ordered from the lowest priority to the highest priority. This can be fixed in software or hardware and is usually permanent.

In the rotating structure the possible interrupt requests are arranged in a circle. There is a pointer which points to the lowest priority interrupt. The priority of each interrupt increases as one travels around the circle, with the highest priority interrupt being

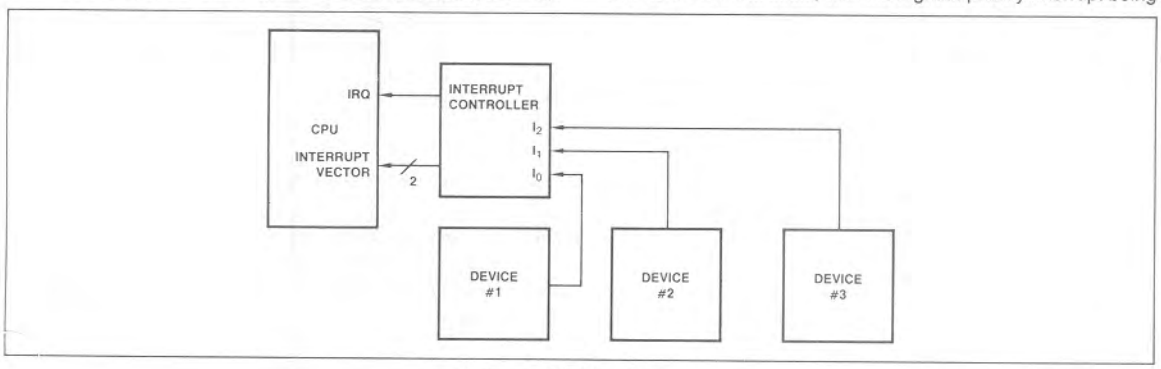



**Figure 3. Multiple Request.**

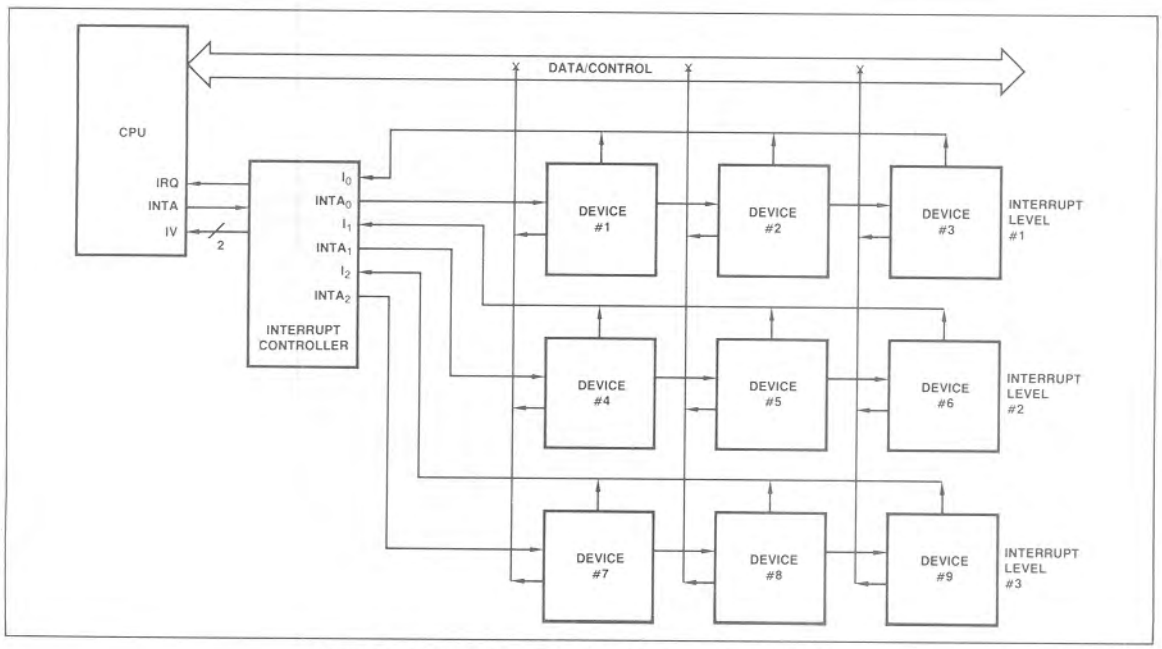



**Figure 4. Multiple Requests, Daisy Chain Acknowledge.**

adjacent to the lowest priority interrupt. The lowest priority interrupt pointer is changed to point at the interrupt that was just serviced. This structure is advantageous when all interrupts have similar priority and service bandwidth requirements.

## NESTING

Nesting allows only higher priority interrupts to interrupt a processing interrupt service routine. Nesting requires fencing off equal and lower level interrupts. Fencing requires that the interrupt structure hold the value of the highest priority interrupt being serviced. This can be implemented with a Status Register that holds the value as a binary encoded number or in other systems as an In-Service Register with a different bit associated with each interrupt.

Whether nesting is performed in microcode or not, all computers must have machine instructions to enable and disable interrupts and set and clear mask bits. With these instructions, interrupt handlers can be written to accomplish nesting of interrupts although less efficiently than when done with microcode and hardware. In low-end computers, the interrupt structure only prioritizes interrupts leaving nesting to the software interrupt handlers.

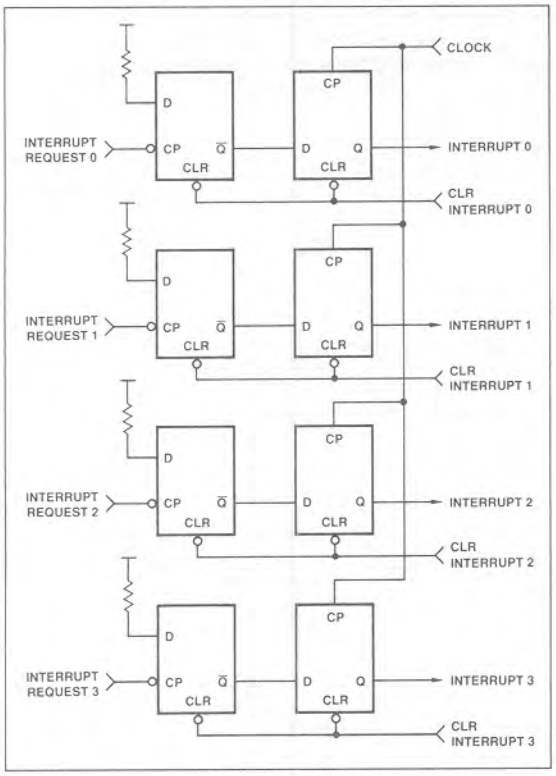



Figure 5.

## A UNIVERSAL HARDWARE INTERRUPT STRUCTURE

While designing a hardware interrupt structure, the designer should consider the specific functions that are to be achieved. This provides for system optimization in not only hardware but also software. In the following paragraphs is a step by step development of a general purpose interrupt structure as related to the design concepts involved.

### Multiple Interrupt Request Handling

Since interrupt requests are generated from a number of sources, the interrupt structures ability to handle interrupt requests from several sources is important.

As implemented in Figure 5, the register configuration allows the hardware to handle interrupt requests from several sources. The first column of registers catches the asynchronous interrupt request. The second column of registers synchronizes the requests with respect to the system. After the interrupt is serviced, one of the CLR lines can be used to selectively clear the interrupt request.

### Interrupt Request Prioritization

Since the processor can service only one interrupt request at a time, the interrupt structure should have the ability to prioritize the requests and determine which has the highest priority. As shown in Figure 6, a priority encoder can be put on the output of the interrupt storage registers. The priority encoder will identify the highest interrupt request as a binary encoded number.

### Dynamic Interrupt Request Masking

The ability to selectively inhibit or "mask" individual interrupt requests under program control is desirable. For example at times it may be important to inhibit all interrupts except Power Failure. As shown in Figure 7 this is realized by ANDing the output of a mask register with the output of the interrupt storage registers. Therefore, the mask register can be used to select which interrupt requests will pass through to the rest of the hardware.

### Interrupt Request Clearing

Flexibility in the method of clearing the interrupt allows different modes of interrupt system operation. Of particular value are the abilities to clear the interrupt currently being serviced or clear all interrupts.

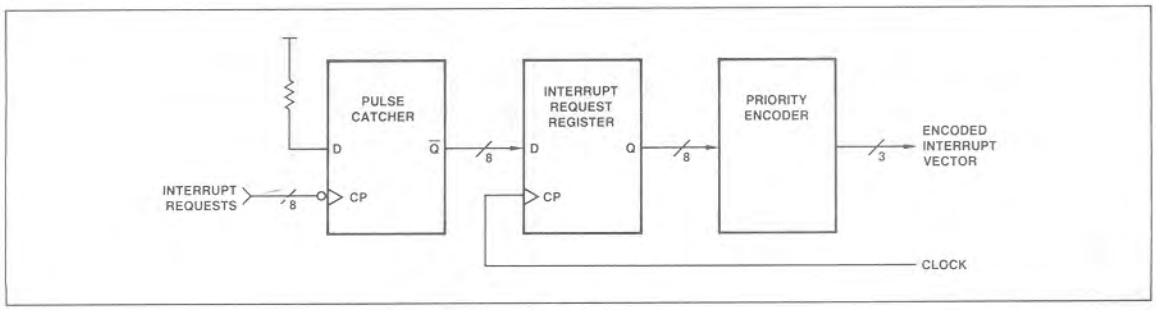



Figure 6.

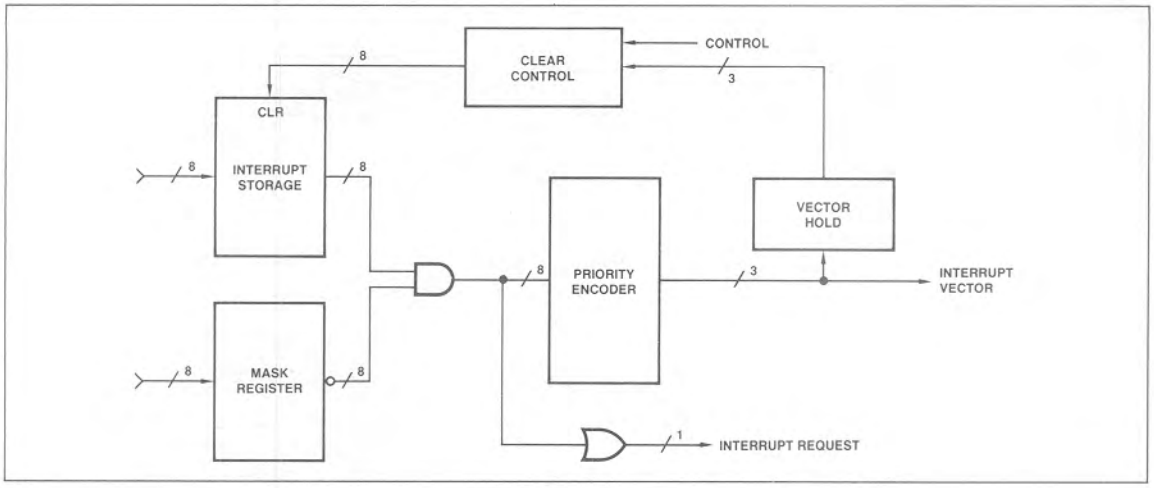



**Figure 7.**

This is implemented in Figure 8 by use of the Vector Hold register on the output of the Priority Encoder. This register holds the latest interrupt request that was recognized. Before another interrupt request is recognized, the output of the Vector Hold register can be fed through some clear control logic to selectively clear the old interrupt.

### Interrupt Request Priority Threshold

The ability to establish a priority threshold is valuable. In this type of operation, only those interrupt requests which have higher priority than a specified threshold priority are accepted. The threshold priority can be defined by microprogram or can be automatically established by hardware at the interrupt currently being serviced plus one. This automatic threshold prevents multiple interrupts from the same source.

This feature is implemented in Figure 8 using an incrementer and status register which is compared with the current request. Each time an interrupt is recognized, the status register is updated with one plus the current level.

### Interrupt Service Routine "Nesting"

This feature allows an interrupt service routine for a given priority request to be interrupted in turn by a higher priority interrupt request. This can be achieved by saving the status register before each interrupt is serviced and restoring it afterwards.

### Microprogrammability and Hardware Modularity

These last two design concepts bring us to the Vectored Priority Interrupt controller, the Am2914. The Am2914 is a modular interrupt system block which is beneficial in two ways. First,

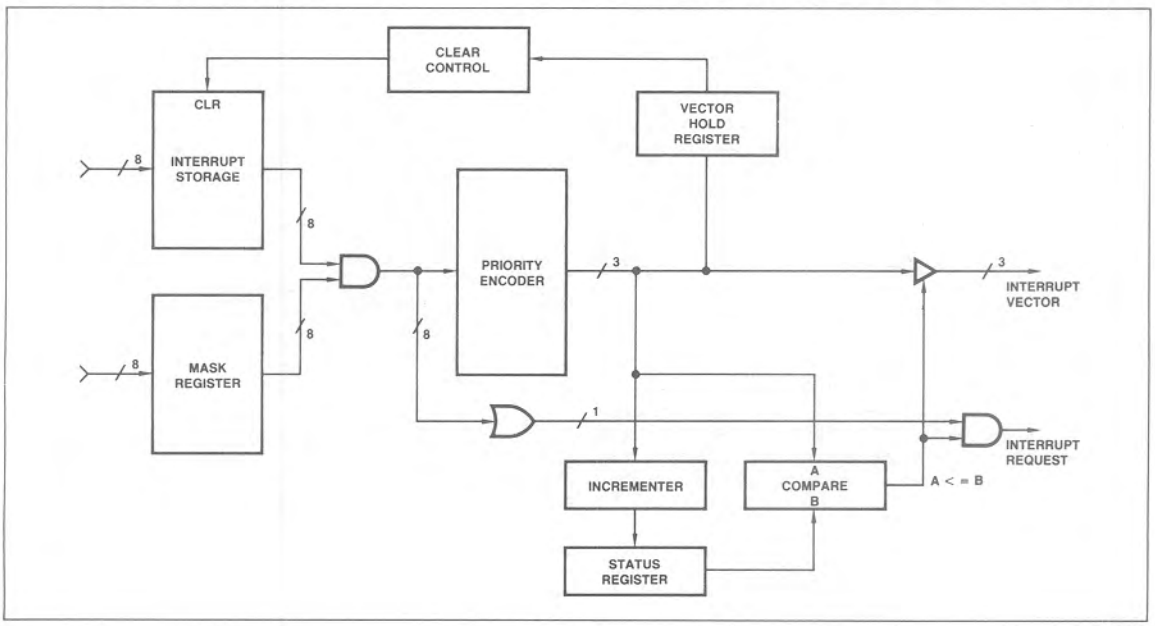



**Figure 8.**

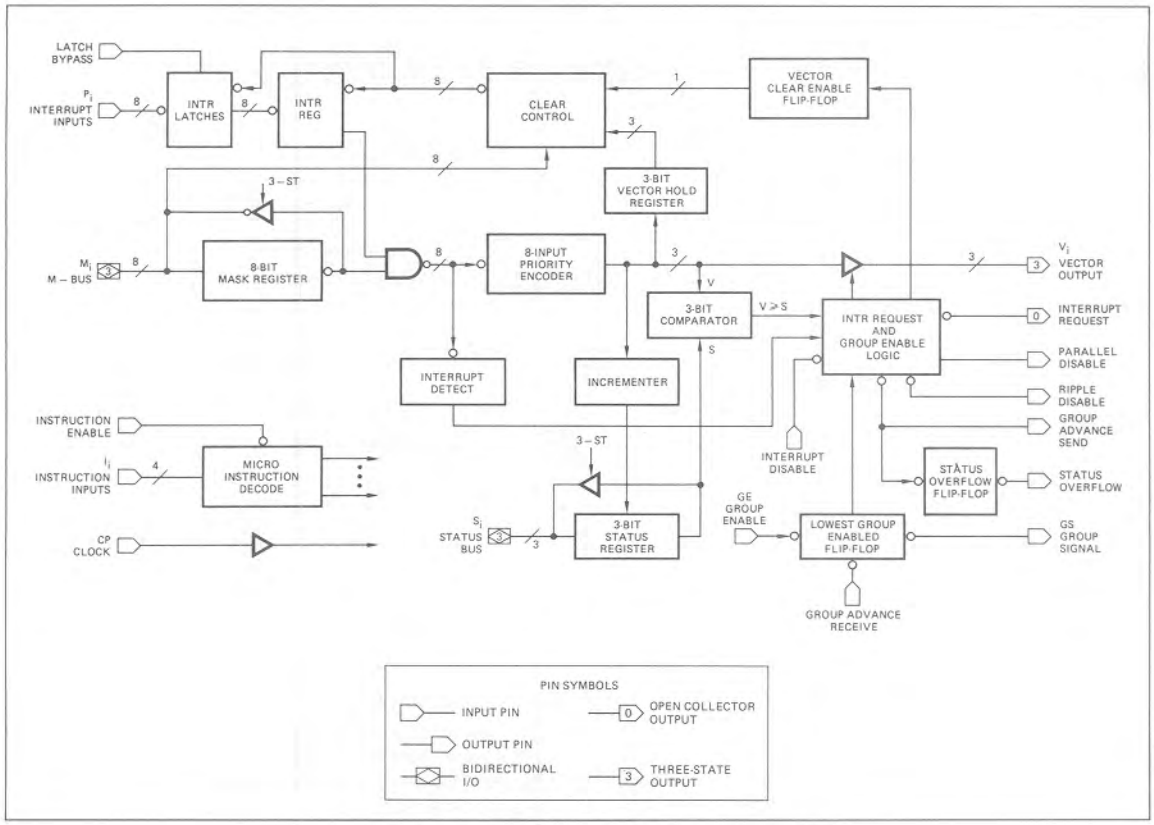



**Figure 9. Am2914 Block Diagram.**

hardware modularity provides expansion capability. Additional modules may be added as the need to service additional requests arises. Secondly, hardware modularity provides a structural regularity which simplifies the system structure and also reduces the number of hardware part numbers.

The Am2914 is microprogrammable, which permits the construction of a general purpose or "universal" interrupt structure which can be microprogrammed to meet a specific application's requirement. The universality of the structure allows standardization of the hardware and amortization of the hardware development costs across a much broader user base. The end result is a flexible, low cost interrupt structure as shown in Figure 9.

## PROGRAMMING THE Am2914

The Am2914 is controlled by a four-bit microinstruction field $I_0$-$I_3$. The microinstruction is executed if $\overline{IE}$ (Instruction Enable) is LOW and is ignored if $\overline{IE}$ is HIGH, allowing the four I bits to be shared with other functions. Sixteen different microinstructions are executed. Figure 11 shows the microinstructions and the microinstruction codes.

In this microinstruction set, the *Master Clear* microinstruction is selected as binary zero so that during a power-up sequence, the microinstruction register in the microprogram control unit of the central processor can be cleared to all zeros. Thus, on the next clock cycle, the Am2914 will execute the *Master Clear* function.

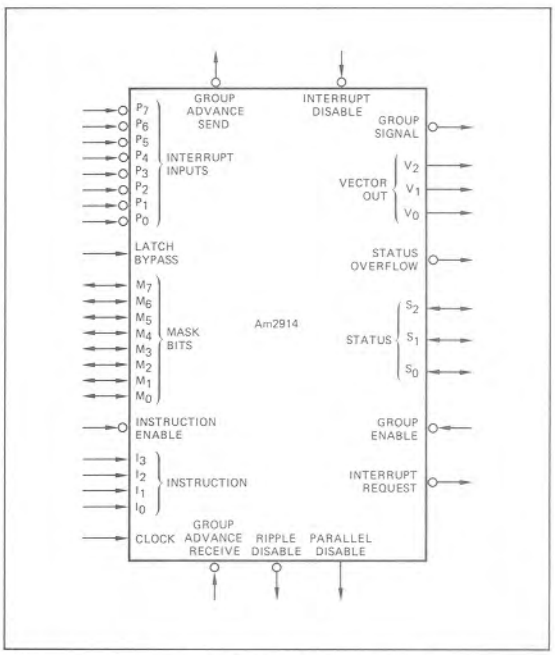



**Figure 10. Am2914 Logic Symbol.**

| MICROINSTRUCTION DESCRIPTION | MICROINSTRUCTION CODE $I_3I_2I_1I_0$ |
|---|---|
| MASTER CLEAR | 0000 |
| CLEAR ALL INTERRUPTS | 0001 |
| CLEAR INTERRUPTS FROM M-BUS | 0010 |
| CLEAR INTERRUPTS FROM MASK REGISTER | 0011 |
| CLEAR INTERRUPT, LAST VECTOR READ | 0100 |
| READ VECTOR | 0101 |
| READ STATUS REGISTER | 0110 |
| READ MASK REGISTER | 0111 |
| SET MASK REGISTER | 1000 |
| LOAD STATUS REGISTER | 1001 |
| BIT CLEAR MASK REGISTER | 1010 |
| BIT SET MASK REGISTER | 1011 |
| CLEAR MASK REGISTER | 1100 |
| DISABLE INTERRUPT REQUEST | 1101 |
| LOAD MASK REGISTER | 1110 |
| ENABLE INTERRUPT REQUEST | 1111 |

**Figure 11. Am2914 Microinstruction Set.**

This includes clearing the Interrupt Latches and Register as well as the Mask Register and Status Register. The LGE flip-flop of the least significant group is set LOW because the Group Advance Receive input is tied LOW. All other Group Advance Receive inputs are tied to Group Advance Send outputs and these are forced HIGH during this instruction. This clear instruction also sets the Interrupt Request Enable flip-flop so that a fully interrupt driven system can be easily initiated from any interrupt.

The *Clear All Interrupts* microinstruction clears the Interrupt Latches and Register.

The *Clear Interrupts from M-Bus* microinstruction clears those Interrupt Latches and Register bits which have corresponding M-Bus bits set equal to one.

The *Clear Interrupts from Mask Register* microinstruction clears those Interrupt Latches and Register bits which have corresponding Mask Register bits set equal to one. The M-Bus is used by the Am2914 during the execution of this microinstruction and must be floating.

The *Clear Interrupt, Last Vector Read* microinstruction clears the Interrupt Latch and Register bit associated with the last vector read.

The *Read Vector* microinstruction is used to read the vector value of the highest priority request causing the interrupt. The vector outputs are three-state drivers that are enabled onto the is instruction. This microinstruction also automatically loads the value "vector plus one" into the Status Register. In addition, this instruction sets the Vector Clear Enable flip-flop and loads the current vector value into the Vector Hold Register so that this value can be used by the *Clear Interrupt, Last Vector Read* microinstruction. This allows the user to read the vector associated with the interrupt, and at some later time clear the Interrupt Latch and Register bit associated with the vector read.

During the *Read Status Register* microinstruction, the Status Register outputs are enabled onto the Status Bus ($S_0$-$S_2$). The Status Bus is a three-bit, bi-directional, three-state bus.

The *Read Mask Register* microinstruction enables the Mask Register outputs onto the bi-directional, three-state M-Bus.

The *Set Mask Register* microinstruction sets all the bits in the Mask Register to one. This results in all interrupts being inhibited.

The *Load Status Register* microinstruction loads S-Bus data into the Status Register and also loads the LGE flip-flop from the Group Enable input.

The *Bit Clear Mask Register* microinstruction may be used to selectively clear individual Mask Register bits. This microinstruction clears those Mask Register bits which have corresponding M-Bus bits equal to one. Mask Register bits with corresponding M-Bus bits equal to zero are not affected.

The *Bit Set Mask Register* microinstruction sets those Mask Register bits which have corresponding M-Bus bits equal to one. Other Mask Register bits are not affected.

The entire Mask Register is cleared by the *Clear Mask Register* microinstruction. This enables all interrupts subject to the Interrupt Enable flip-flop and the Status Register.

All Interrupt Requests may be disabled by execution of the *Disable Interrupt Request* microinstruction. This microinstruction resets an Interrupt Request Enable flip-flop on the chip.

The *Load Mask Register* microinstruction loads data from the three-state, bi-directional M-Bus into the Mask Register.

The *Enable Interrupt Request* microinstruction sets the Interrupt Enable flip-flop. Thus, Interrupt Requests are enabled subject to the contents of the Mask and Status Registers.

## Am2914 BLOCK DIAGRAM DESCRIPTION

The Am2914 block diagram is shown in Figure 9. The Microinstruction Decode circuitry decodes the Interrupt Microinstructions and generates required control signals for the chip.

The Interrupt Register holds the Interrupt Inputs and is an eight-bit, edge-triggered register which is set on the rising edge of the CP Clock signal if the Interrupt Input is LOW.

The Interrupt latches are set/reset latches. When the Latch Bypass signal is LOW, the latches are enabled and act as negative pulse catchers on the inputs to the Interrupt Register. When the Latch Bypass signal is HIGH, the Interrupt latches are transparent.

The Mask Register holds the eight mask bits associated with the eight interrupt levels. The register may be loaded from or read to the M-Bus. Also, the entire register or individual mask bits may be set or cleared.

The Interrupt Detect circuitry detects the presence of any unmasked Interrupt Input. The eight-input Priority Encoder determines the highest priority, non-masked Interrupt Input and forms a binary coded interrupt vector. Following a Vector Read, the three-bit Vector Hold Register holds the binary coded interrupt vector. This stored vector can be used later for clearing interrupts.

The three-bit Status Register holds the status bits and may be loaded from or read to the S-Bus. During a Vector Read, the Incrementer increments the interrupt vector by one, and the result is clocked into the Status Register. Thus, the Status Register points to a level one greater than the vector just read.

The three-bit Comparator compares the Interrupt Vector with the contents of the Status Register and indicates if the Interrupt Vector is greater than or equal to the contents of the Status Register.

The Lowest Group Enabled Flip-Flop is used when a number of Am2914's are cascaded. In a cascaded system, only one Lowest Group Enabled Flip-Flop is LOW at a time. It indicates the eight interrupt group, which contains the lowest priority interrupt level which will be accepted and is used to form the higher order status bits.

The Interrupt Request and Group Enable logic contain various gating to generate the Interrupt Request, Parallel Disable, Ripple Disable, and Group Advance Send signals.

The Status Overflow signal is used to disable all interrupts. It indicates the highest priority interrupt vector has been read and the Status Register has overflowed.

The Clear Control logic generates the eight individual clear signals for the bits in the Interrupt Latches and Register. The Vector Clear Enable Flip-Flop indicates if the last vector read was from this chip. When it is set it enables the Clear Control Logic.

The CP clock signal is used to clock the Interrupt Register, Mask Register, Status Register, Vector Hold Register, and the Lowest Group Enabled, Vector Clear Enable and Status Overflow Flip-Flops, all on the clock LOW-to-HIGH transition.

## CASCADING THE Am2914

A number of input/output signals are provided for cascading the Am2914 Vectored Priority Interrupt Encoder. A definition of these I/O signals and their required connections follows:

Group Signal ($\overline{GS}$) – This signal is the output of the Lowest Group Enabled flip-flop and during a Read Status microinstruction is used to generate the high order bits of the Status word.

Group Enable ($\overline{GE}$) – This signal is one of the inputs to the Lowest Group Enable flip-flop and is used to load the flip-flop during the Load Status microinstruction.

Group Advance Send ($\overline{GAS}$) – During a Read Vector microinstruction, this output signal is LOW when the highest priority vector (vector seven) of the group is being read. In a cascaded system Group Advance Send must be tied to the Group Advance Receive input of the next higher group in order to transfer status information.

Group Advance Receive ($\overline{GAR}$) – During a Master Clear or Read Vector microinstruction, this input signal is used with other internal signals to load the Lowest Group Enabled flip-flop. The Group Advance Receive input of the lowest priority group must be tied to ground.

Status Overflow ($\overline{SV}$) – This output signal becomes LOW after the highest priority vector (vector seven) of the group has been read and indicates the Status Register has overflowed. It stays LOW until a Master Clear or Load Status microinstruction is executed. The Status Overflow output of the highest priority group should be connected to the Interrupt Disable input of the same group and serves to disable all interrupts until new status is loaded or the system is master cleared. The Status Overflow outputs of lower priority groups should be left open (see Figure 14).

Interrupt Disable ($\overline{ID}$) – When LOW, this input signal inhibits the Interrupt Request output from the chip and also generates a Ripple Disable output.

Ripple Disable ($\overline{RD}$) – This output signal is used only in the Ripple Cascade Mode (see below). The Ripple Disable output is LOW when the Interrupt Disable input is LOW, the Lowest Group Enabled flip-flop is LOW, or an Interrupt Request is generated in the group. In the ripple cascade mode, the Ripple Disable output is tied to the Interrupt Disable input of the next lower priority group (see Figure 13).

Parallel Disable (PD) – This output is used only in the parallel cascade mode (see below). It is LOW when the Lowest Group Enabled flip-flop is LOW or an Interrupt Request is generated in the group. It is not affected by the Interrupt Disable input.

## CASCADING CONFIGURATIONS

A single Am2914 chip may be used to prioritize and encode up to eight interrupt inputs. Figure 12 shows how the above cascade lines should be connected in such a single chip system.

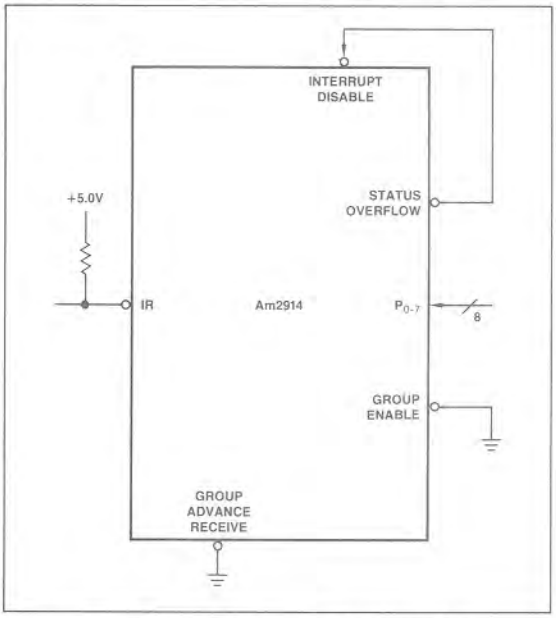



**Figure 12. Cascade Lines Connection for Single Chip System.**

The Group Advance Receive and Group Enable inputs should be connected to ground so that the Lowest Group Enabled flip-flop is forced LOW during a *Master Clear* or *Load Status* microinstruction. Status Overflow should be connected to Interrupt Disable in order to disable interrupts when vector seven is read. The Group Advance Send, Ripple Disable, Group Signal and Parallel Disable pins should be left open.

The Am2914 may be cascaded in either a Ripple Cascade Mode or a Parallel Cascade Mode. In the Ripple Cascade Mode, the Interrupt Disable signal, which disables lower priority interrupts, is allowed to ripple through lower priority groups. Figures 13, 16, and 17 show the cascade connections required for a ripple cascade 32 input interrupt system.

In the parallel cascade mode, a parallel lookahead scheme is employed using the high-speed Am2902 Lookahead Carry Generator. Figures 14, 15, and 17 show the cascade connections required for a parallel cascade 32-input interrupt system. For this application, the Am2902 is used as a lookahead interrupt disable

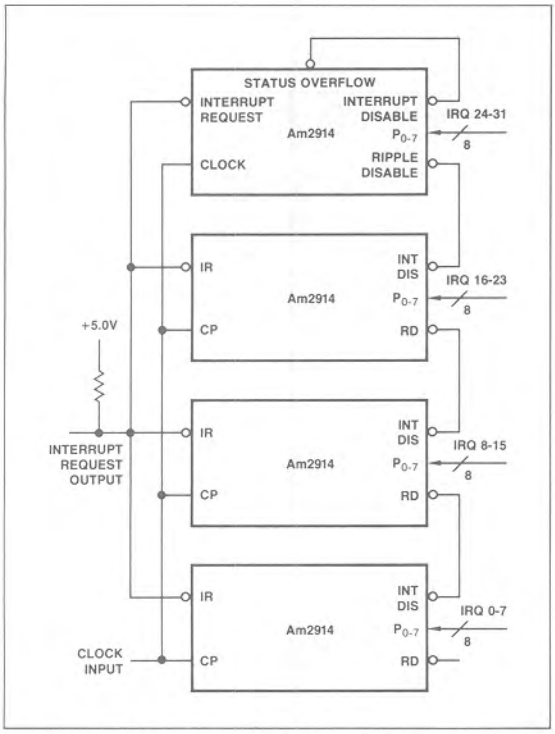



**Figure 13. Interrupt Disable Connections for Ripple Cascade Mode.**

generator. A Parallel Disable output from any group results in the disabling of all lower priority groups in parallel. Figure 15 shows the Am2902 logic diagram and equations.

In Figures 16 and 17 the Am2913 Priority Interrupt Expander is shown forming the high order bits of the vector and status, respectively. The Am2913 is an eight-line to three-line priority encoder with three-state outputs which are enabled by the five output control signals G1, G2, $\overline{G3}$, $\overline{G4}$, and $\overline{G5}$. In Figure 16, the Am2913 is connected so that its outputs are enabled during a Read Vector instruction, and in Figure 17 the Am2913 is connected to microinstruction bits so that its outputs are enabled during a Read Status Instruction. The Am2913 logic diagram and truth table are shown in Figure 18.

The Am25LS138 three-line to eight-line Decoder also is shown in Figure 17. It is used to decode the three high order status bits during a Load Status instruction. The Am25LS138 logic diagram and truth table are shown in Figure 19.

### Am2914 IN THE Am2900 SYSTEM

The block diagram of Figure 20 shows a typical 16-bit minicomputer architecture. The Am2914 is the heart of the Interrupt Control Unit as shown at the bottom of the block diagram. It receives its microinstructions from the Computer Control Unit. The mask, Status and Interrupt vector information are passed on the data bus. The interrupt request line from the Am2914 input into the next microprogram Address Control unit where it can be tested to determine if an interrupt request has been made.

Figures 21 and 22 show the detailed hardware design of two example interrupt control units (ICU's) for an Am2900 Computer

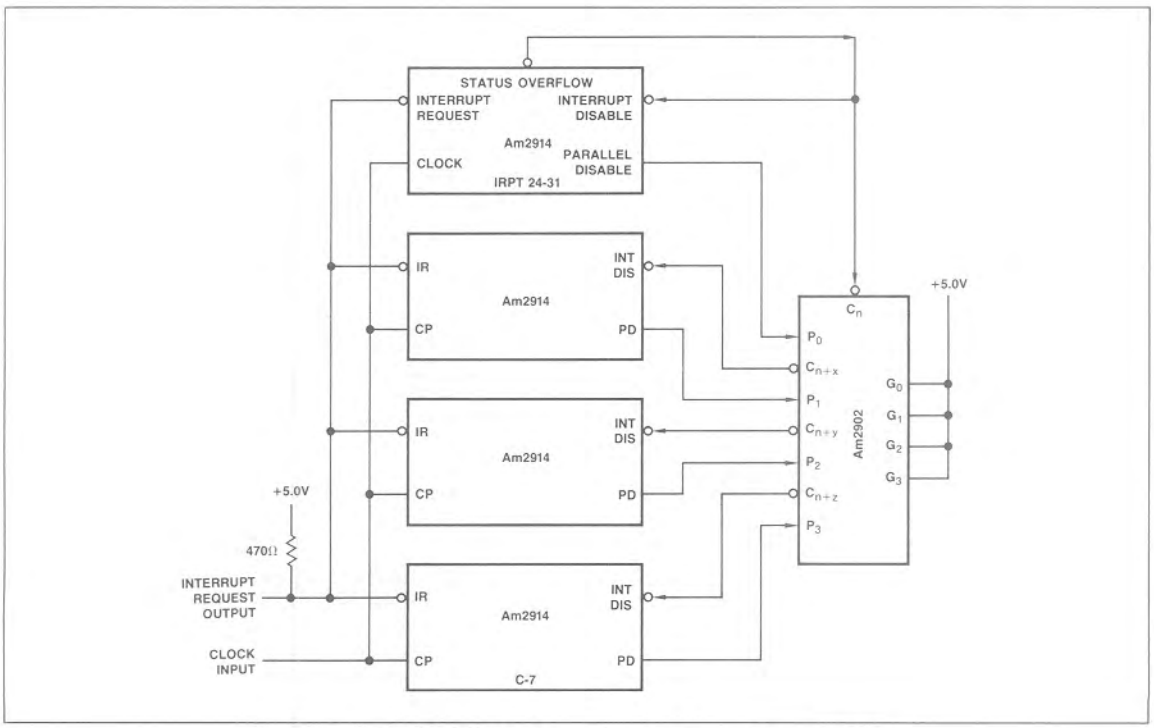
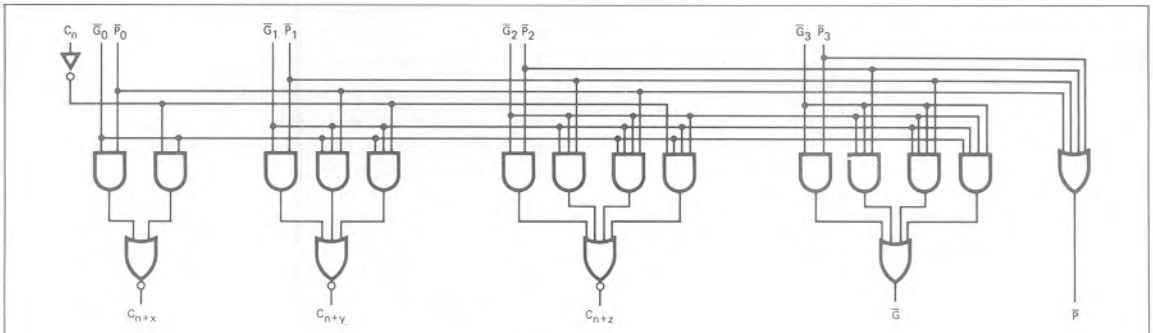



**Figure 14. Interrupt Disable Connections for Parallel Cascade Mode.**




$$C_{n+x} = G_0 + P_0C_n$$
$$C_{n+y} = G_1 + P_1G_0 + P_1P_0C_n$$
$$C_{n+z} = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_n$$
$$G = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$
$$P = P_3P_2P_1P_0$$

**Figure 15. Am2902 Carry Look-Ahead Generator Logic Diagram and Equations.**

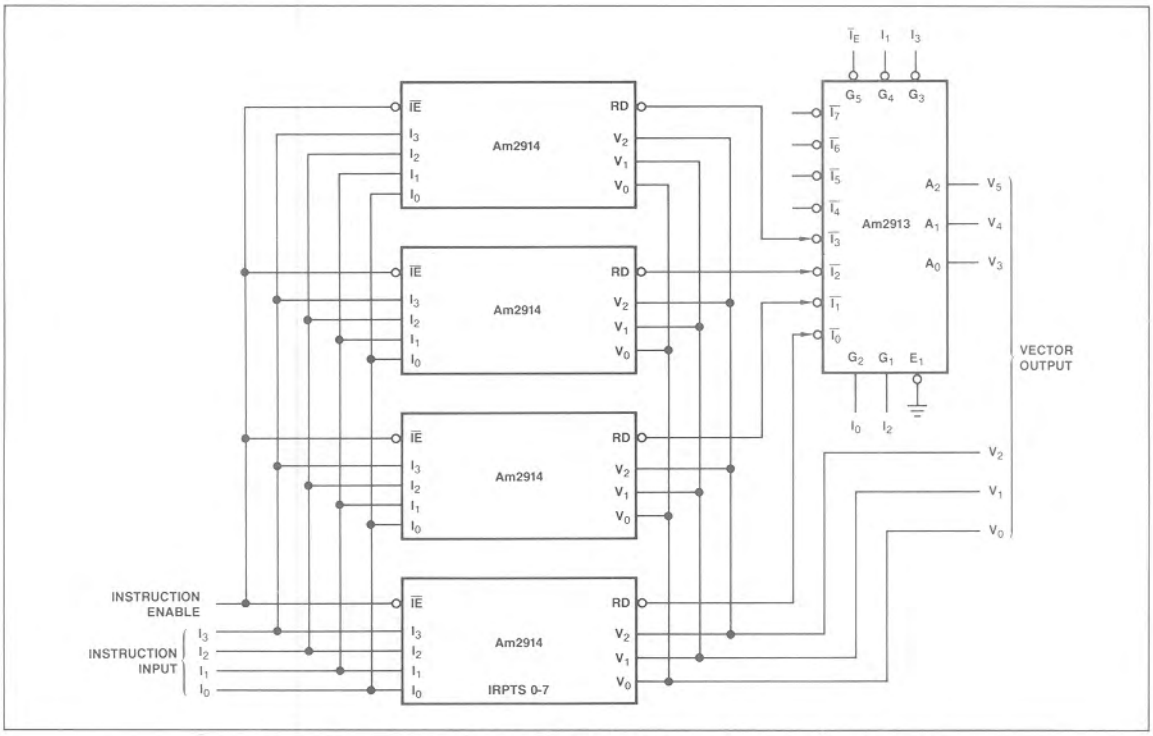



**Figure 16. Vector Connections for both the Parallel and Ripple Cascade Modes.**

System. Figure 21 shows an eight interrupt level ICU, and Figure 22 shows an ICU which has sixteen levels. In both designs, the Am2914 Instruction inputs and Instruction Enable input are driven by the $I_{0\text{-}3}$ field and $\overline{IE}$ bit, respectively, of the Microinstruction Register. Note that Am2914 Instruction inputs are enabled only when the $\overline{IE}$ bit is LOW. Therefore, the $I_{0\text{-}3}$ field of the Microinstruction Register may be shared with another functional unit of the computer such as the ALU.

The Latch Bypass input is shown connected to ground so that a Low-going pulse will be detected at any of the Interrupt Inputs. The designer has the option of connecting the Latch Bypass input to a pull up resistor connected to +5 volts. This makes the inputs low level sensitive. They are clocked in by each system clock. It is therefore implied that the processor will have to acknowledge the interrupt so that the interrupting device will know when to release the interrupt request line.

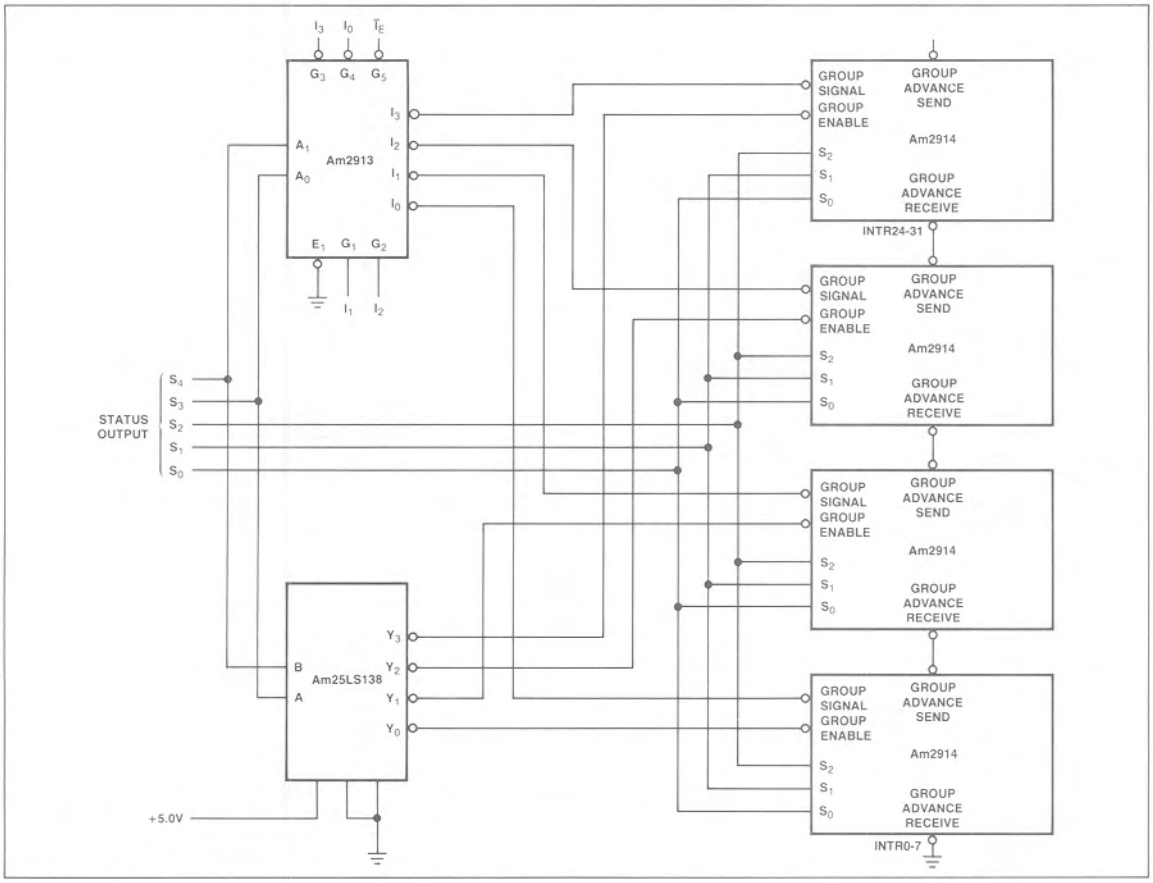
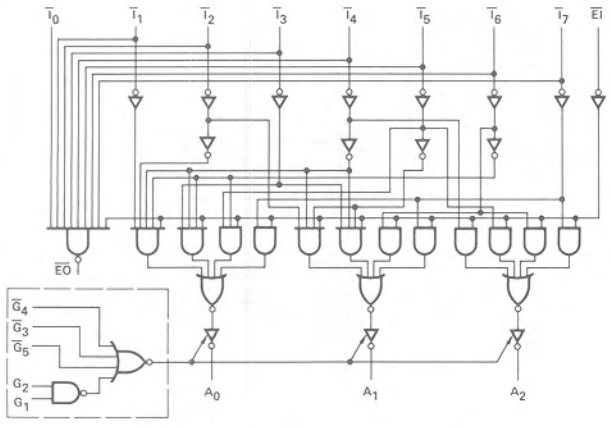



**Figure 17. Group Signal, Group Enable, Group Advance Send, Group Advance Receive and Status Connections for Both the Parallel and Ripple Cascade Modes.**



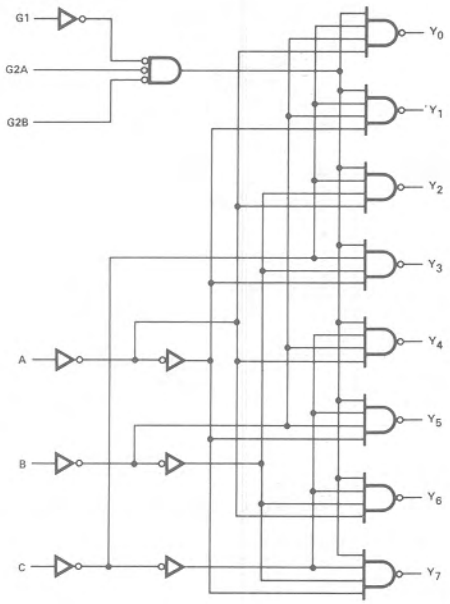

| Inputs | | | | | | | | | Outputs | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\overline{EI}$ | $\overline{I}_0$ | $\overline{I}_1$ | $\overline{I}_2$ | $\overline{I}_3$ | $\overline{I}_4$ | $\overline{I}_5$ | $\overline{I}_6$ | $\overline{I}_7$ | $A_0$ | $A_1$ | $A_2$ | $\overline{EO}$ |
| H | X | X | X | X | X | X | X | X | L | L | L | H |
| L | H | H | H | H | H | H | H | H | L | L | L | L |
| L | X | X | X | X | X | X | X | L | H | H | H | H |
| L | X | X | X | X | X | X | L | H | L | H | H | H |
| L | X | X | X | X | X | L | H | H | H | L | H | H |
| L | X | X | X | X | L | H | H | H | L | L | H | H |
| L | X | X | X | L | H | H | H | H | H | H | L | H |
| L | X | X | L | H | H | H | H | H | L | H | L | H |
| L | X | L | H | H | H | H | H | H | H | L | L | H |
| L | L | H | H | H | H | H | H | H | L | L | L | H |

H = HIGH Voltage Level
L = LOW Voltage Level
X = Don't Care
For $G_1$ = H, $G_2$ = H, $G_3$ = L, $G_4$ = L, $G_5$ = L

| G1 | G2 | $\overline{G3}$ | $\overline{G4}$ | $\overline{G5}$ | $A_0$ | $A_1$ | $A_2$ |
|---|---|---|---|---|---|---|---|
| H | H | L | L | L | Enabled | | |
| L | X | X | X | X | Z | Z | Z |
| X | L | X | X | X | Z | Z | Z |
| X | X | H | X | X | Z | Z | Z |
| X | X | X | H | X | Z | Z | Z |
| X | X | X | X | H | Z | Z | Z |

Z = HIGH Impedance

**Figure 18. Am2913 Priority Interrupt Expander Logic Diagram and Truth Table.**




| Inputs | | | | | | Outputs | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable | | | Select | | | | | | | | | | |
| G1 | G2A | G2B | C | B | A | $Y_0$ | $Y_1$ | $Y_2$ | $Y_3$ | $Y_4$ | $Y_5$ | $Y_6$ | $Y_7$ |
| L | X | X | X | X | X | H | H | H | H | H | H | H | H |
| X | H | X | X | X | X | H | H | H | H | H | H | H | H |
| X | X | H | X | X | X | H | H | H | H | H | H | H | H |
| H | L | L | L | L | L | L | H | H | H | H | H | H | H |
| H | L | L | L | L | H | H | L | H | H | H | H | H | H |
| H | L | L | L | H | L | H | H | L | H | H | H | H | H |
| H | L | L | L | H | H | H | H | H | L | H | H | H | H |
| H | L | L | H | L | L | H | H | H | H | L | H | H | H |
| H | L | L | H | L | H | H | H | H | H | H | L | H | H |
| H | L | L | H | H | L | H | H | H | H | H | H | L | H |
| H | L | L | H | H | H | H | H | H | H | H | H | H | L |

H = HIGH
L = LOW
X = Don't Care

Figure 19. Am25LS138 3 to 8 Line Decoder Logic Diagram and Truth Table.

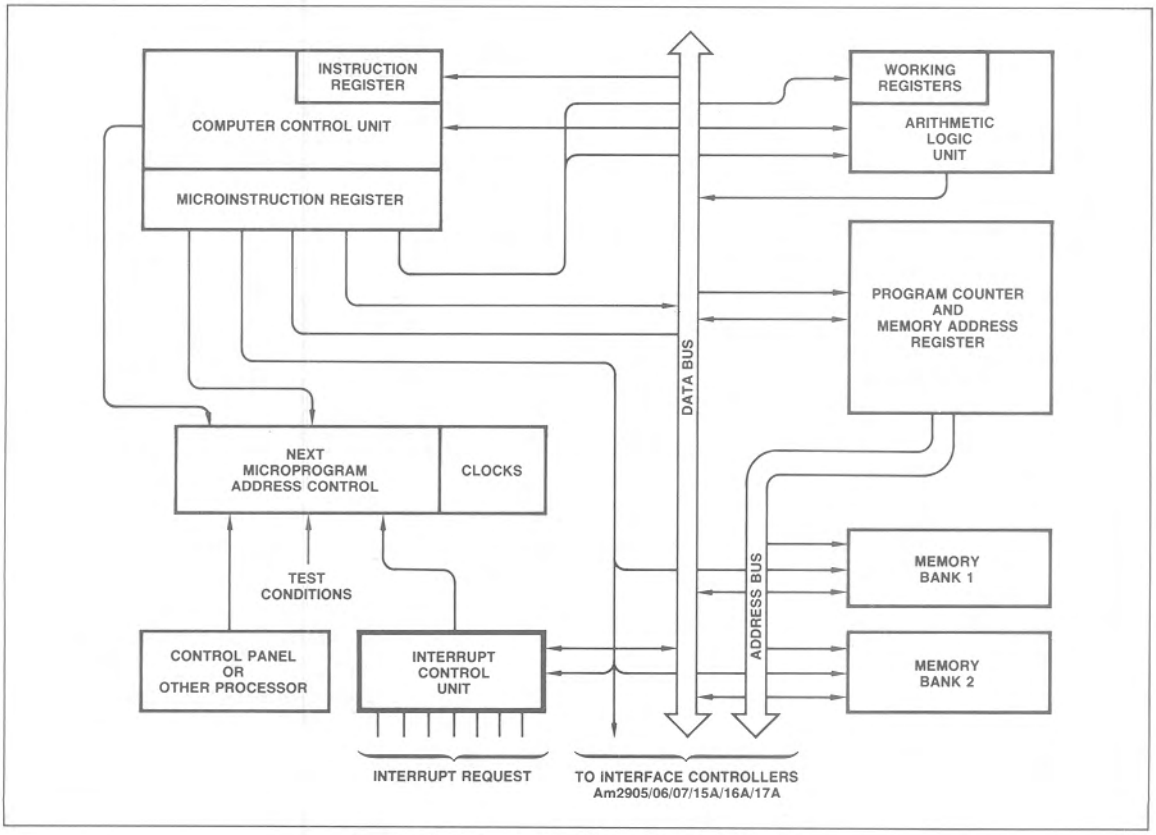



Figure 20. A Generalized Computer Architecture.

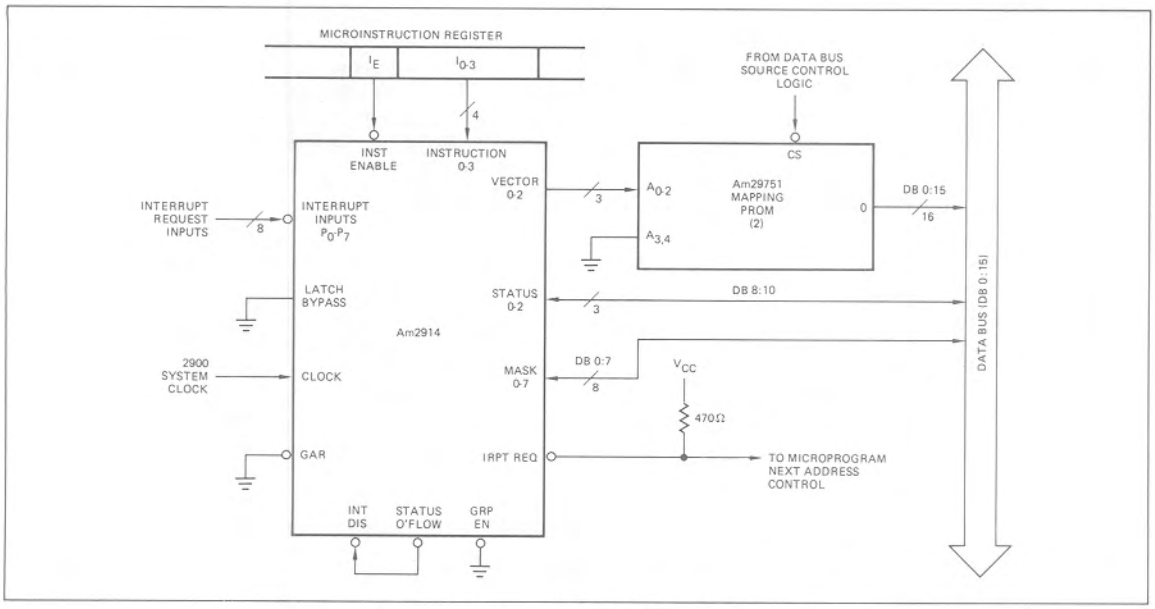



**Figure 21. 8 Level Interrupt Control Unit for Am2900 System.**

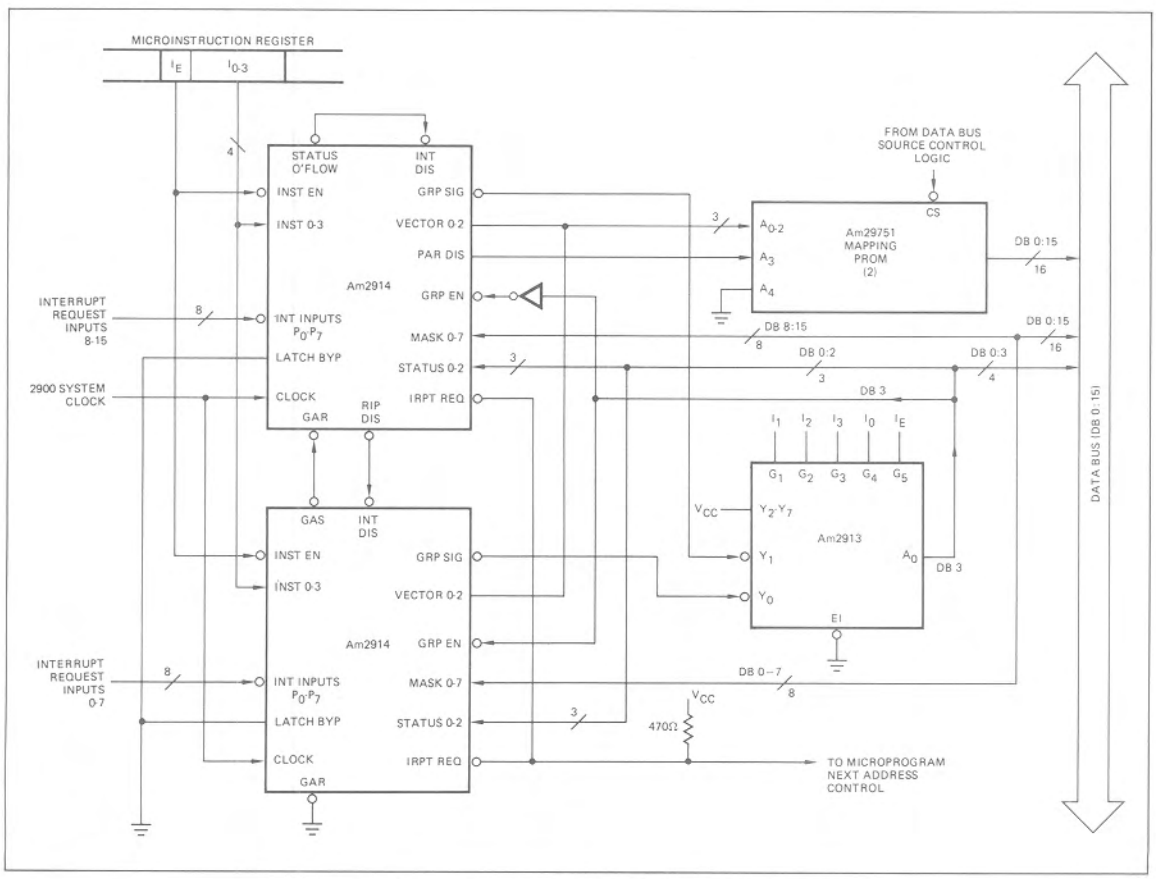



**Figure 22. 16 Level Interrupt Control Unit for Am2900 System.**

In Figures 21 and 22, the Status and Mask inputs/outputs are connected to the data bus in a bi-directional configuration so that Status and Mask Registers may be loaded from or read to the data bus with appropriate Am2914 instructions. This gives the designer two possibilities which could be very advantageous.

Number one is the ability to store the Status and Mask information on a stack in memory. This is very advantageous when doing nested interrupts. Secondly, it allows the designer to construct machine instruction that can modify these two registers. This is very important to the system programmer who is involved in writing software to manage the interrupts.

For the eight level ICU of Figure 21, the Status Overflow output is connected to the Interrupt Disable input, and the Group Advance Receive and Group Enable inputs are connected to ground, as previously described.

For the 16 interrupt level ICU of Figure 22, the Parallel Disable output of the higher priority group serves as the high order vector bit. An Am2913 Priority Interrupt Expander is gated by the Am2914 instruction lines so that its output is enabled only during a Read Status instruction, and is used to encode the high order bit of the status. An inverter suffices to decode the high order bit of the status bit during a Load Status instruction. As described previously for a ripple cascade system, the Group Advance Receive input of the next higher priority group; the Ripple Disable output is connected to the Interrupt Disable input of the next lower priority group; the Status Overflow output of the highest priority group is connected to the Interrupt Disable input of the same group, and the Group Advance Receive input of the lowest priority group is connected to ground.

In both designs, two Am29751 32-word by 8-bit PROM's with three-state outputs are used to map the Am2914 Vector outputs into a 16-bit address vector. The PROM outputs are connected to the data bus. When a Read Vector Instruction (Am2914) is executed, the address vector is available to be used either as the address of the next instruction or a location to find the address of the next instruction to execute.

Figure 23 shows a design where the address vector from the mapping PROM can be clocked into a register in the Am2903's. The registers in the Am2903's would be split between general purpose, scratch, stack pointers and Program Counter registers.

The address vector also may be gated directly to the "D" inputs of the Am2911 Microprogram Sequencer as shown in Figure 24, and used as the start PROM address of a microinstruction interrupt service routine. This method would be most useful in a controller application. This method would trade faster service for a bigger microprogram that accommodates all the code to service each individual interrupt.

## FIRMWARE EXAMPLE FOR Am2914 INTERRUPT SYSTEM

The software for handling interrupt requests is on two levels. The first level to come into play is the microprogram level. This is the level at which the request is recognized and the program counter is manipulated to start execution of a machine level interrupt service routine which is the second level. When the machine level interrupt service routine is finished, some form of a Return Interrupt instruction is executed. The microcode for the return instruction manipulates the program counter so that execution of the current machine program previous to the request is restored as shown in Figure 25.

This example is concerned with the microprogram level. This microcode goes along with the hardware shown in Figure 23. In this example the code is shown in the form of Flow Charts because the actual microprogram format will vary from machine to machine.

The important features to notice that have a direct relevance to the firmware are the Latch Bypass and where the Mask, Status and Vector busses go. For this example, the Latch Bypass is LOW making the Interrupt Latches latch up on a negative going pulse. The Mask and Status busses go to the data bus allowing the Status and Mask data to be transferred to and from memory. The Vector bus passes through a mapping PROM to the data bus where it can be read into the Program Counter contained in the Am2903's. The PROM contains addresses of service routines which correspond to the different interrupt levels.

Another relevant fact, important to understanding the firmware is that the interrupt mechanism is limited to handle interrupts on the machine level.

As shown in Figure 26a, the first thing that happens in the fetch routine (written in microcode) is a conditional subroutine call that will be taken if an interrupt request is present. This happens before the current machine instruction is fetched and the program counter is incremented.

In the Interrupt routine (shown in Figure 26b) a microprogram subroutine is first called to push the program counter onto the system stack. This is done so that the program counter can be restored in order to resume execution of the machine program after the interrupt service routine is done. The next thing that is saved on the system stack is the contents of the Am2914 Status Register. This is done because the status register which contains the priority level that would be serviced prior to the interrupt, will be restored after the interrupt is serviced. This maintains a nested interrupt structure (fence).

After saving the program counter and status register, the vector is read out of the Am2914 through the mapping PROM to obtain the address of the machine interrupt service routine. The address is then read into the program counter which resides in the Am2903's. When the Vector is read, the interrupt request priority plus one is automatically put into the status register by the Am2914 so that all interrupt requests of lower priority than the one being serviced are ignored. This is often referred to as moving the fence up. Since the vector has been read and the new address is in the program counter, the interrupt request can be cleared from the interrupt register via the Clear Interrupt/Last Vector Read instruction. At this point a jump is made to the Fetch routine which will now fetch the first instruction of the machine Interrupt Service routine.

The last instruction that the machine level interrupt service executes is an Interrupt Return. This will in turn call Return Interrupt microprogram. The status is first popped off the system stack and loaded back into the status register. This restores the Interrupt Fence. The program counter is then popped off the system stack and loaded into the program counter register. This restores the program counter to point to the instruction that was going to be executed when the interrupt request occurred.

## TIME DELAY WHEN USING THE Am2914

An aspect that should be covered when using any part is how it will fit into the system timing; because the cycle time of the system will be as long as the longest delay path in the machine. Shown in Figure 27 is the longest delay path through the Am2914 for the previous 16-bit computer example. The calculations were using both typical and worst case values at 25°C and 5.0V.

The longest delay path for the system where the vector from the mapping PROM feeds into the "D" inputs of the Am2910 is

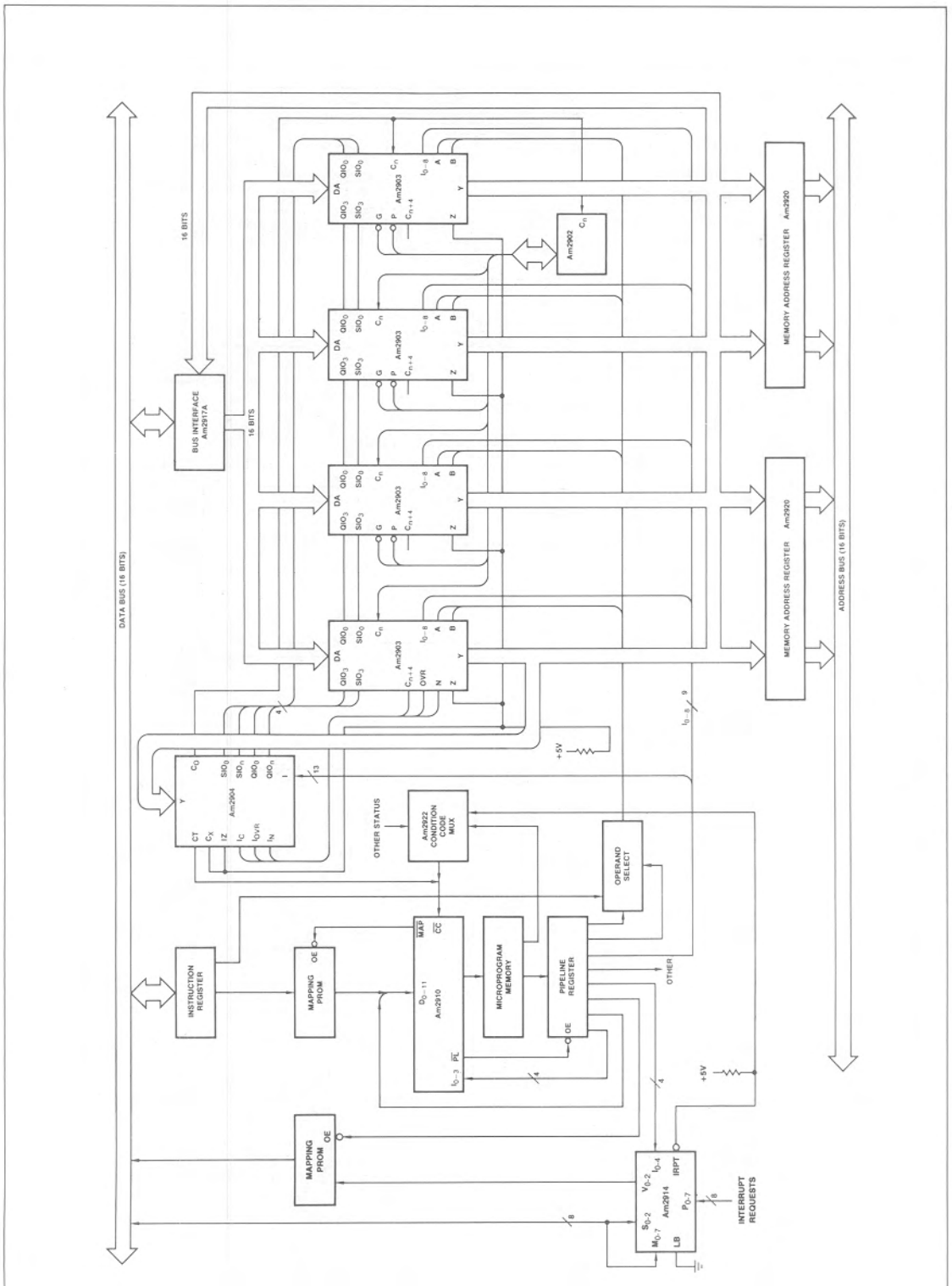



Figure 23. Example of a 16-Bit Computer #1.

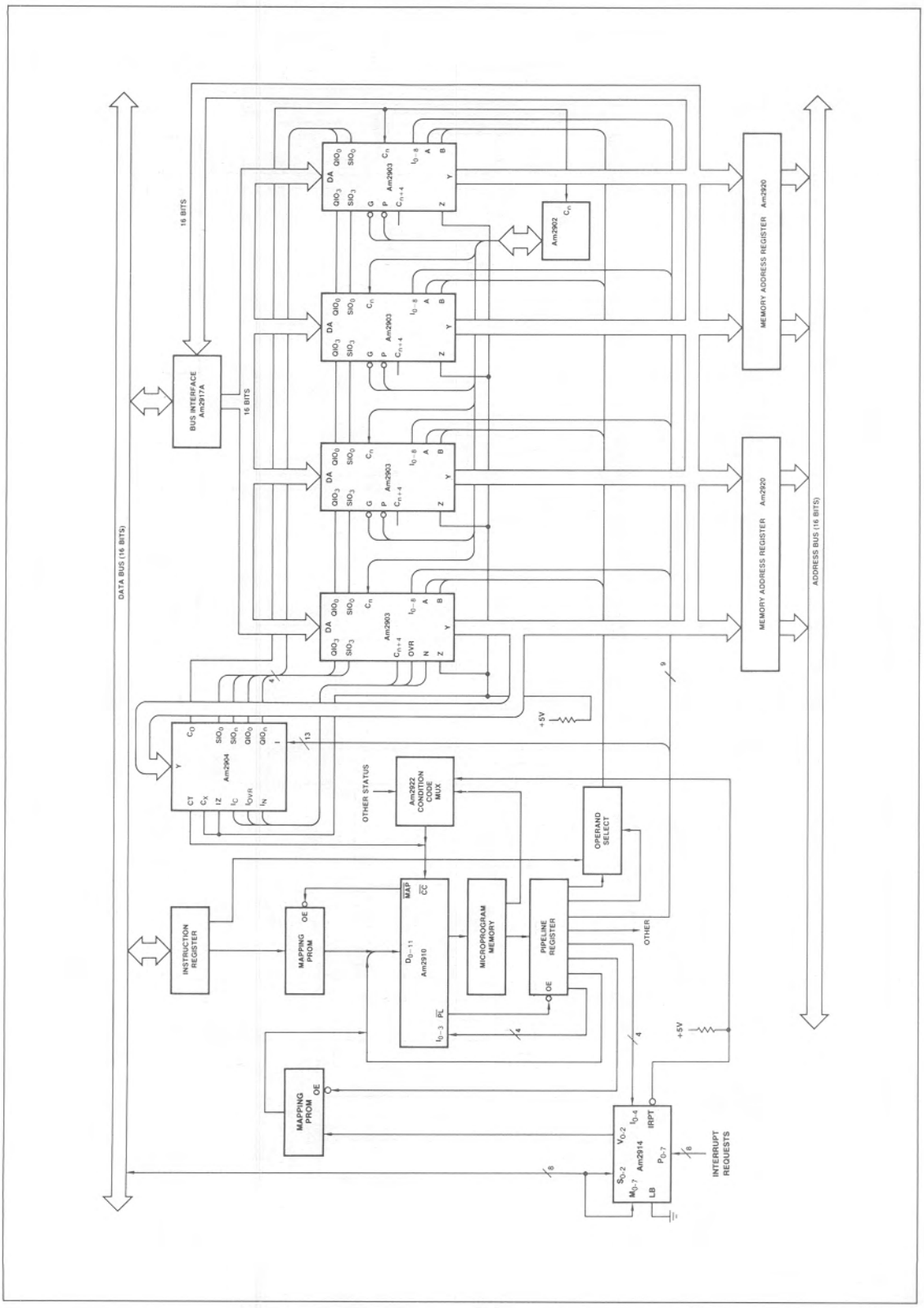

DATA BUS (16 BITS)
16 BITS
BUS INTERFACE Am2917A
16 BITS
Am2903
Am2902
MEMORY ADDRESS REGISTER Am2920
MEMORY ADDRESS REGISTER Am2920
ADDRESS BUS (16 BITS)
+5V
Am2904
OTHER STATUS
Am2922 CONDITION CODE MUX
OPERAND SELECT
INSTRUCTION REGISTER
MAPPING PROM
Am2910
MICROPROGRAM MEMORY
PIPELINE REGISTER
OTHER
MAPPING PROM
+5V
Am2914
INTERRUPT REQUESTS

Figure 24.

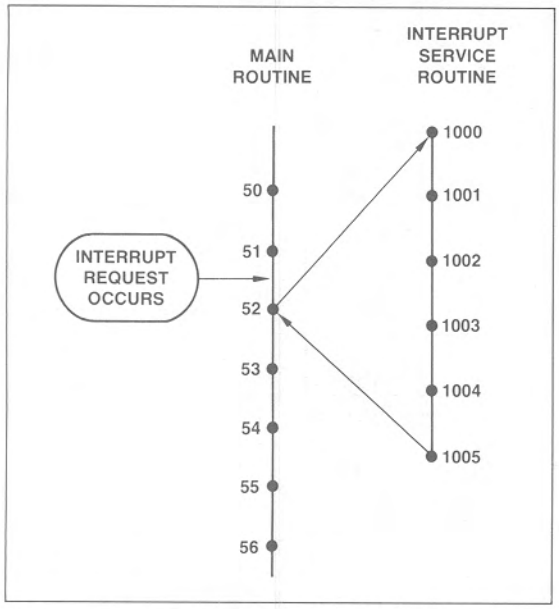



Figure 25. Machine Level Instruction Flow During Interrupt Request.

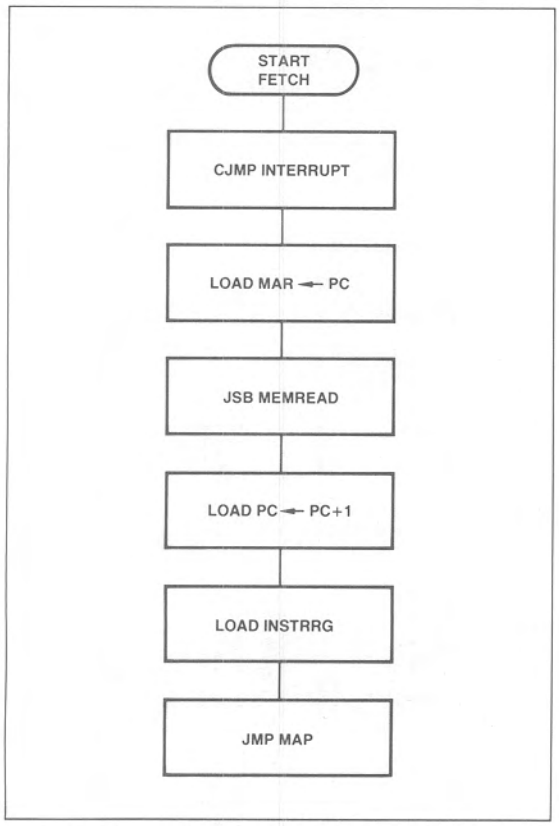



Figure 26a. Flow Chart for a Simplified Microprogram Fetch Routine.

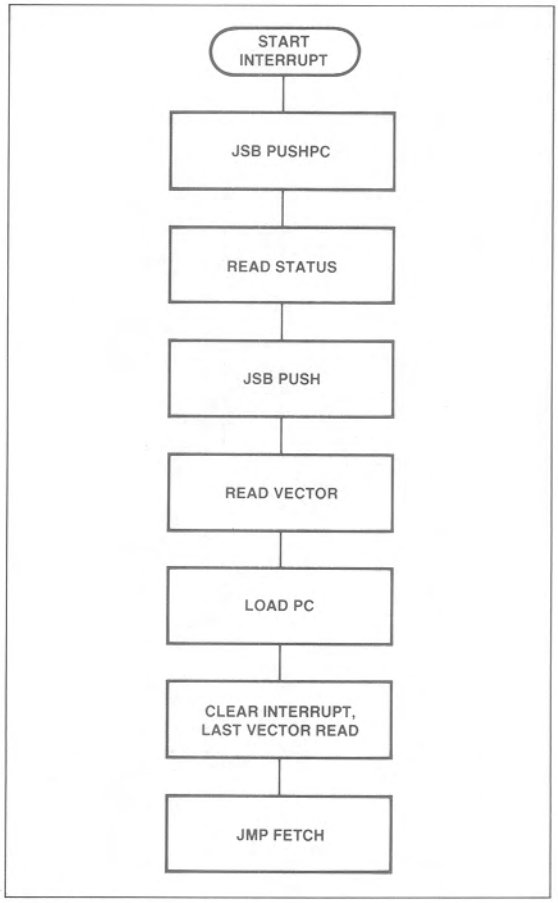



Figure 26b. Call Interrupt Service Routine Microprogram Flow Chart.

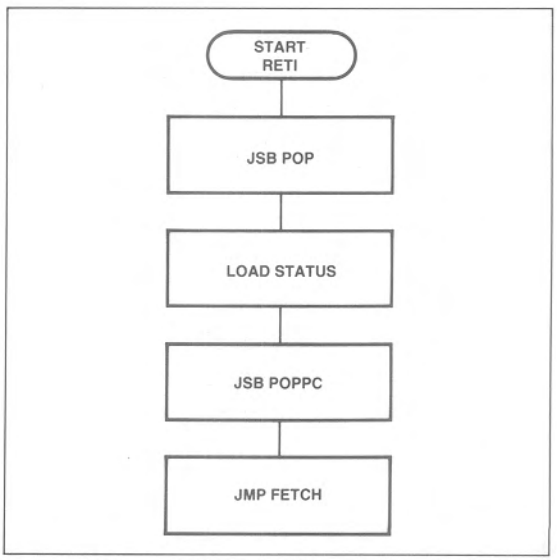



Figure 26c. Return Interrupt Microprogram Flow Chart.

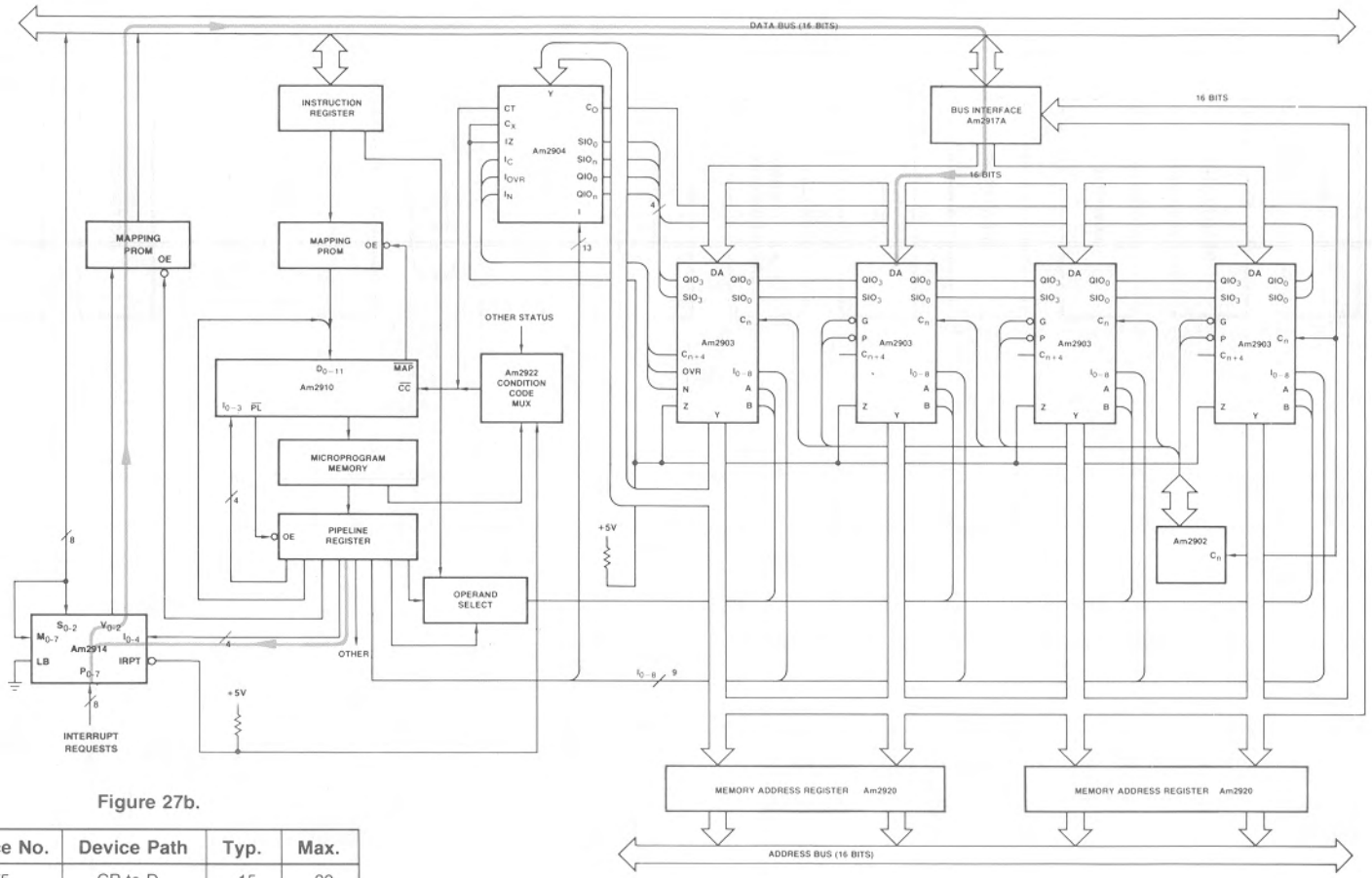



Figure 27b.

| Device No. | Device Path | Typ. | Max. |
|---|---|---|---|
| 29775 | CP to D | 15 | 20 |
| 2914 | I to $V_{012}$ | 40 | 55 |
| 27S19 | A to O | 25 | 40 |
| 2917A | Bus to R | 18 | 30 |
| 2903 | $t_S$ to (Y) | 10 | 16 |
| Total-ns | | 108 | 161 |

Figure 27a. AC Calculations.

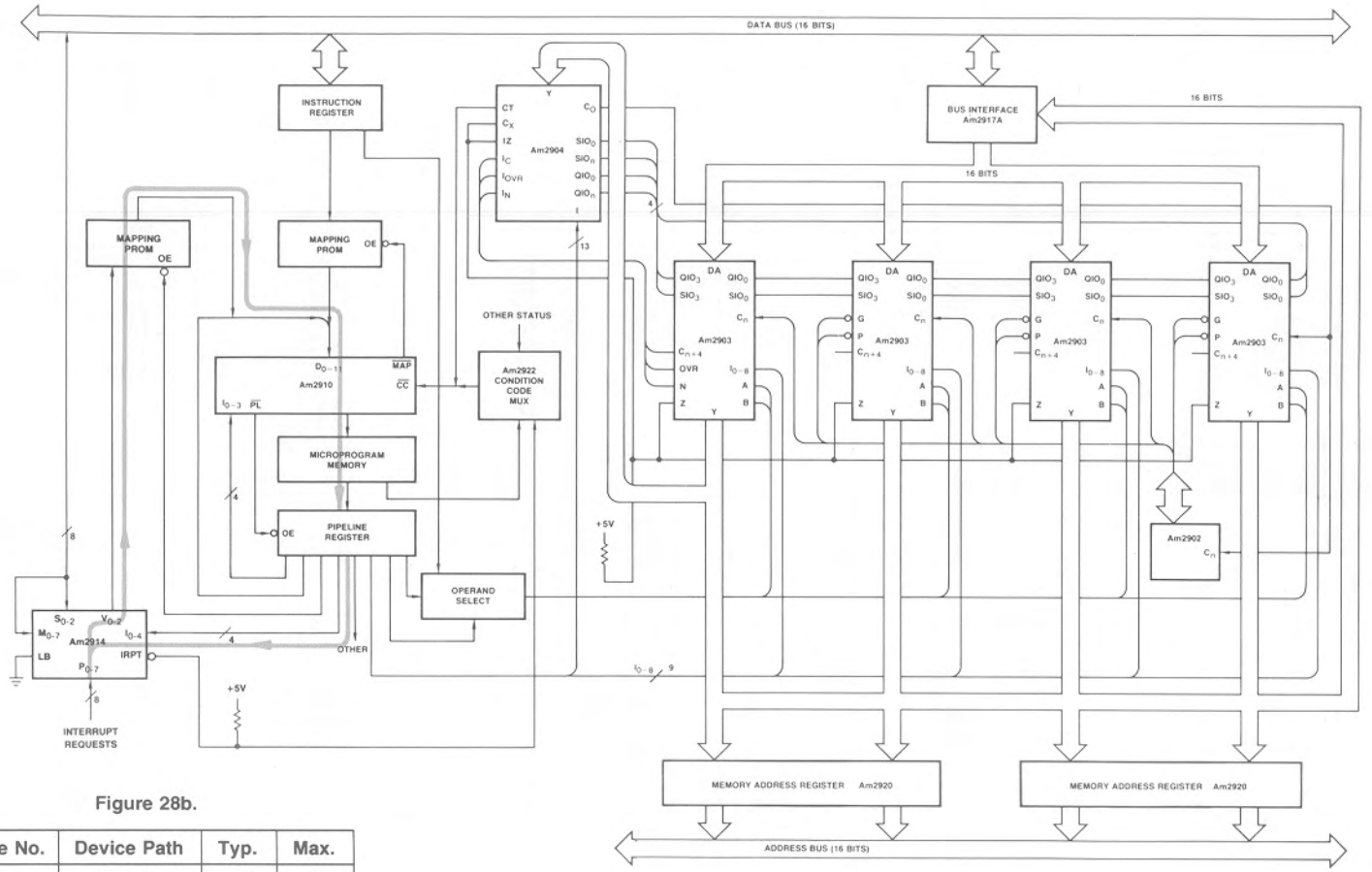



Figure 28b.

| Device No. | Device Path | Typ. | Max. |
|---|---|---|---|
| 29775 | CP to D | 15 | 20 |
| 2914 | I to V | 40 | 55 |
| 27S19 | A to O | 25 | 40 |
| 2910 | D to Y | 14 | 22 |
| 29775 | $t_S$ (A) | 40 | 50 |
| Total-ns | | 134 | 187 |

DELAY PATH

Figure 28a.

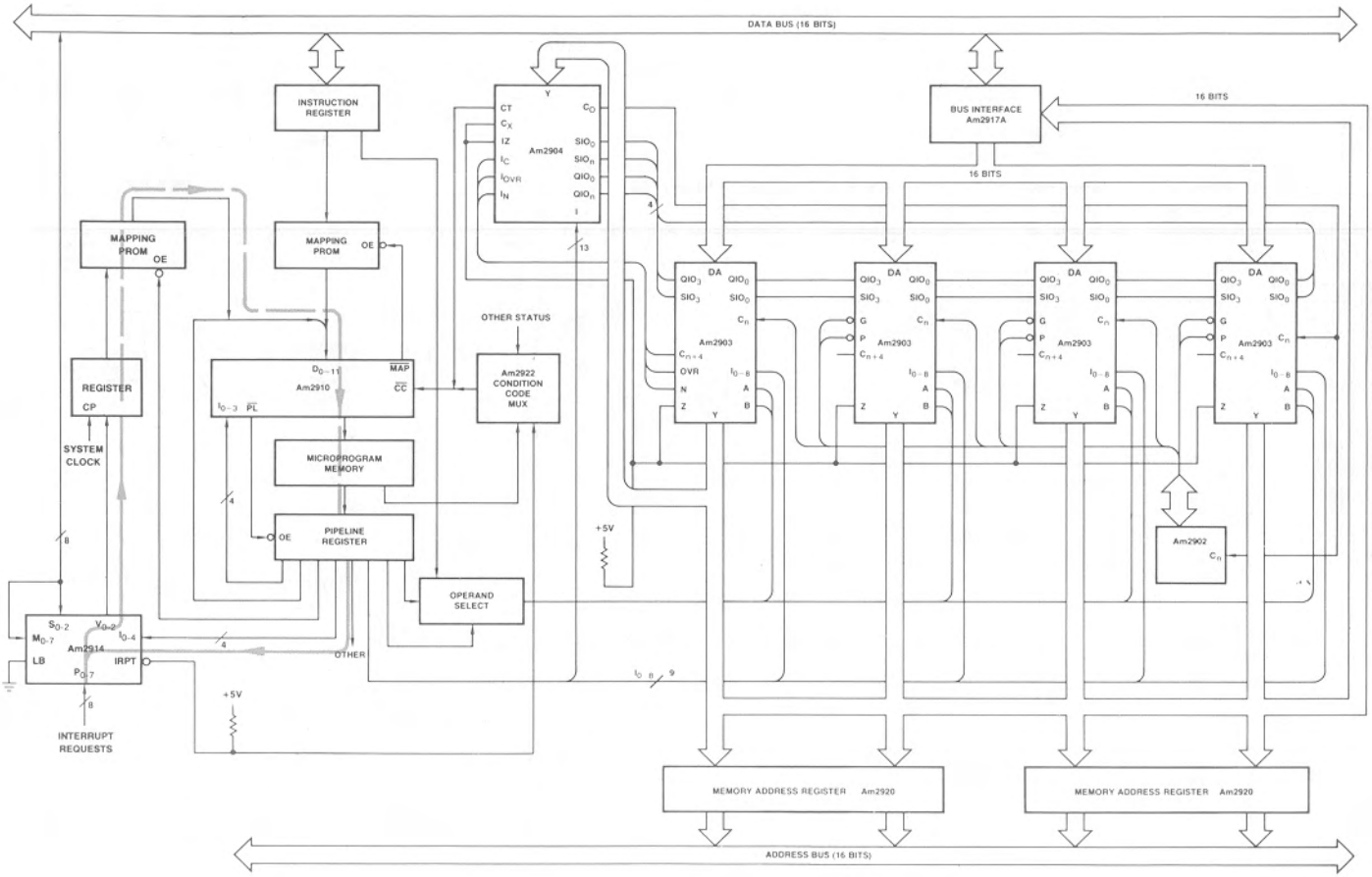

DATA BUS (16 BITS)
INSTRUCTION REGISTER
MAPPING PROM
OE
MAPPING PROM
OE
REGISTER
CP
SYSTEM CLOCK
Am2910
MICROPROGRAM MEMORY
PIPELINE REGISTER
OE
Am2914
INTERRUPT REQUESTS
+5V
Am2904
OTHER STATUS
Am2922 CONDITION CODE MUX
OPERAND SELECT
OTHER
BUS INTERFACE Am2917A
16 BITS
16 BITS
Am2903
Am2902
MEMORY ADDRESS REGISTER Am2920
MEMORY ADDRESS REGISTER Am2920
ADDRESS BUS (16 BITS)
DELAY PATH, CYCLE n
DELAY PATH, CYCLE n+1

Figure 28c.

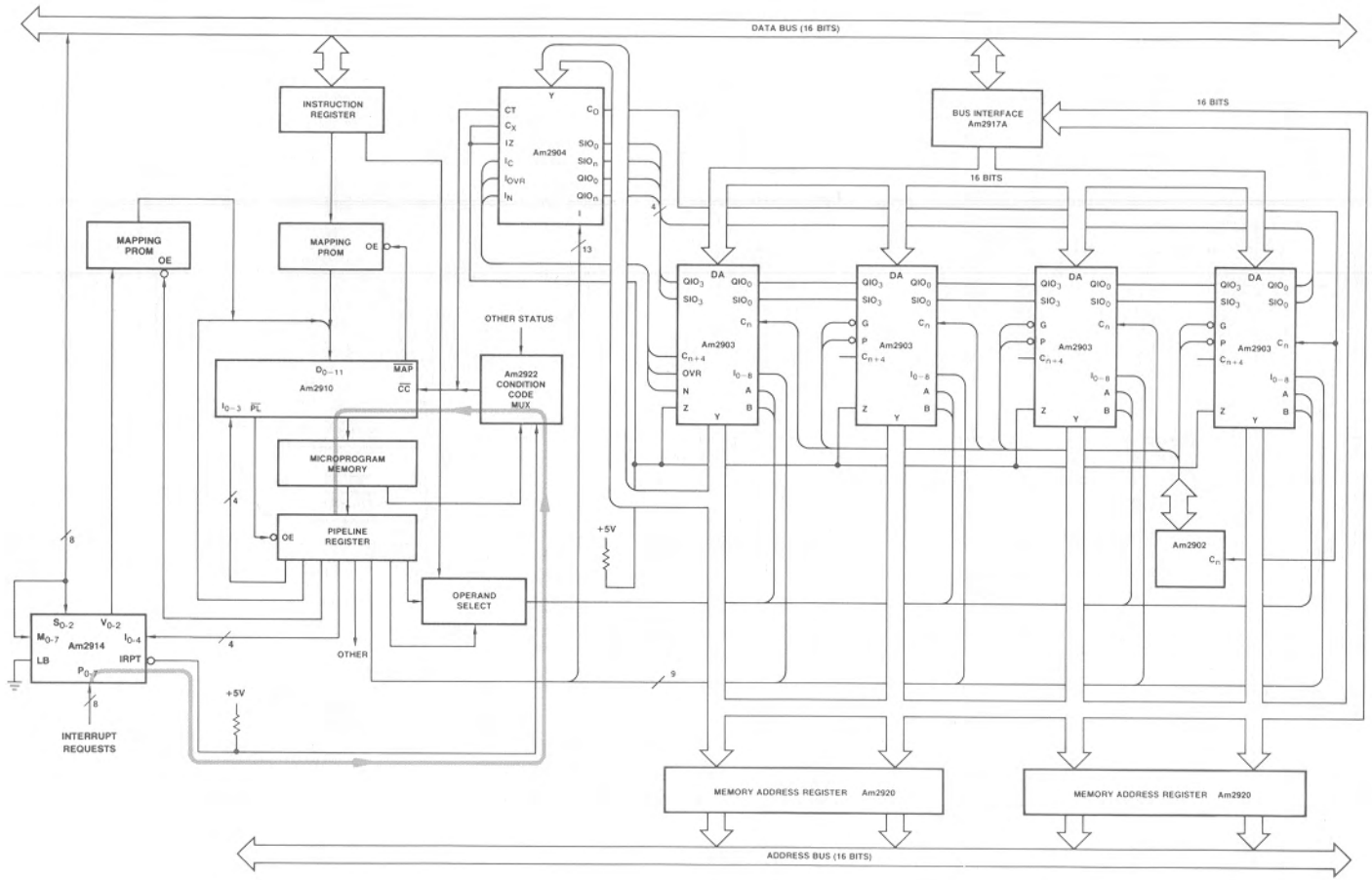

DATA BUS (16 BITS)
INSTRUCTION REGISTER
BUS INTERFACE Am2917A
16 BITS
16 BITS
Am2904
MAPPING PROM
MAPPING PROM
OTHER STATUS
Am2910
Am2922 CONDITION CODE MUX
MICROPROGRAM MEMORY
PIPELINE REGISTER
OPERAND SELECT
Am2903
Am2902
Am2914
INTERRUPT REQUESTS
OTHER
+5V
+5V
MEMORY ADDRESS REGISTER Am2920
MEMORY ADDRESS REGISTER Am2920
ADDRESS BUS (16 BITS)
DELAY PATH

Figure 28d.

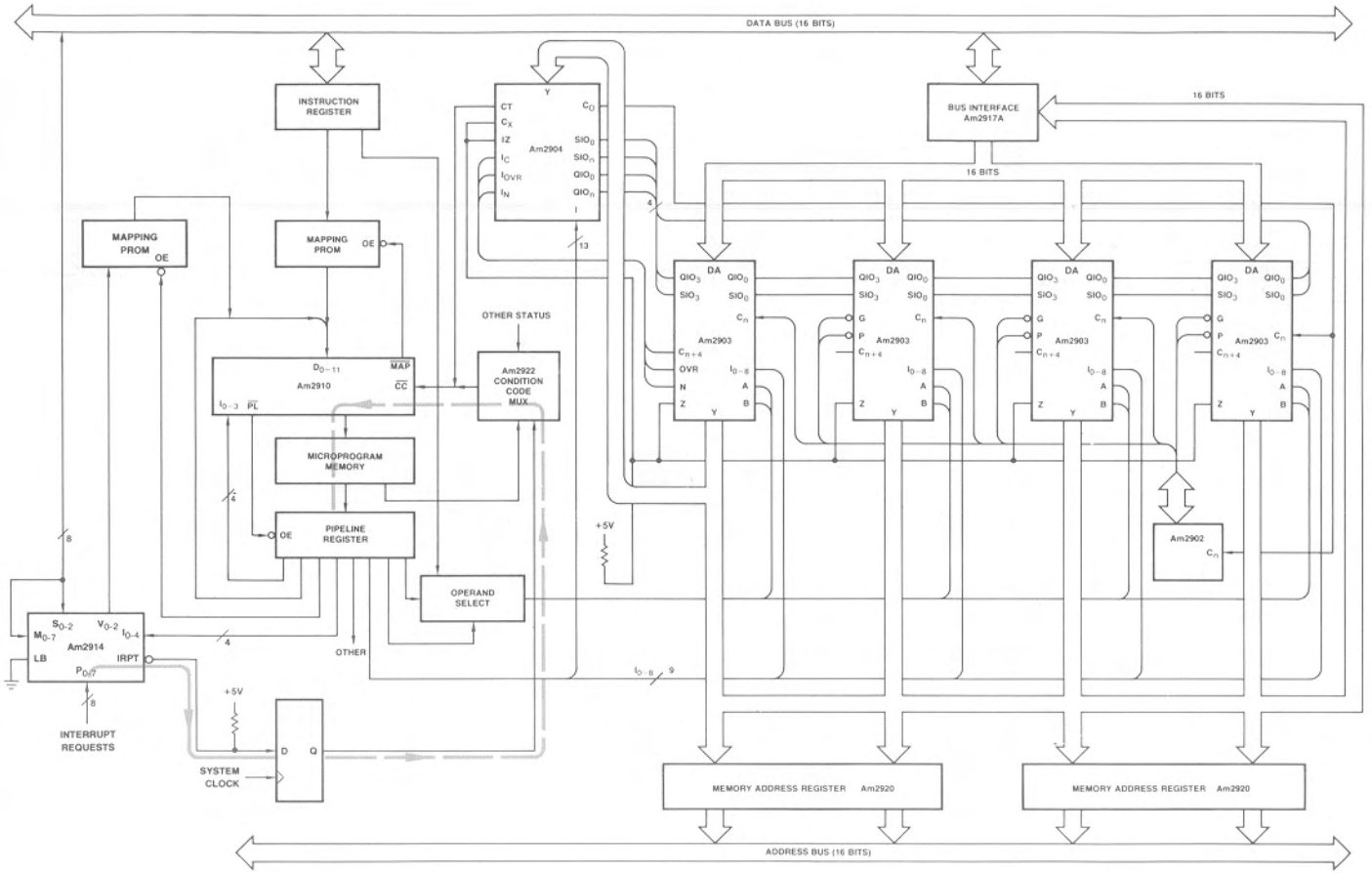

DATA BUS (16 BITS)
INSTRUCTION REGISTER
MAPPING PROM
MAPPING PROM
Am2904
BUS INTERFACE Am2917A
16 BITS
16 BITS
OTHER STATUS
Am2910
Am2922 CONDITION CODE MUX
MICROPROGRAM MEMORY
PIPELINE REGISTER
OPERAND SELECT
Am2903
Am2902
Am2914
INTERRUPT REQUESTS
SYSTEM CLOCK
+5V
OTHER
MEMORY ADDRESS REGISTER Am2920
MEMORY ADDRESS REGISTER Am2920
ADDRESS BUS (16 BITS)
DELAY PATH, CYCLE n
DELAY PATH, CYCLE n+1

Figure 28e.

| Device No. | Device Path | Typ. | Max. |
|---|---|---|---|
| 29775 | CP to D | 15 | 20 |
| 2914 | I to V | 40 | 55 |
| 2918 | $t_s$ (Data) | 5 | 5 |
| Cycle n Total-ns | | 60 | 80 |
| 2918 | CP to Q | 8.5 | 13 |
| 27S19 | A to O | 25 | 40 |
| 2910 | D to Y | 14 | 22 |
| 29775 | $t_s$ (A) | 40 | 50 |
| Cycle n+1 Total-ns | | 97.5 | 125 |

**Figure 28f.**

| Device No. | Device Path | Typ. | Max. |
|---|---|---|---|
| 2914 | CP to IRQ | 65 | 82 |
| 2922 | $D_n$ to Y | 13 | 19 |
| 2910 | $\overline{CC}$ to Y | 27 | 44 |
| 29775 | $t_s$ (A) | 40 | 50 |
| Total-ns | | 145 | 195 |

**Figure 28g.**

| Device No. | Device Path | Typ. | Max. |
|---|---|---|---|
| 2914 | CP to IRQ | 65 | 82 |
| 74S74 | $t_s$ (Data) | 3 | 3 |
| Cycle n Total-ns | | 68 | 85 |
| 74S74 | CP to Q | 6 | 9 |
| 2922 | $D_n$ to Y | 13 | 19 |
| 2910 | $\overline{CC}$ to Y | 27 | 44 |
| 29775 | $t_s$ (A) | 40 | 50 |
| Cycle n+1 Total-ns | | 86 | 122 |

**Figure 28h.**

shown in Figure 28. This path is much longer because of the two PROM's that have to be accessed. Therefore, there may be a trade-off of slightly longer system cycle time for faster service of interrupts via service routines in microcode.

For some systems the delay time shown in Figure 28b may be too long. Therefore, the designer can split the delay time into parts by putting a register between the Am2914 and the mapping PROM as shown in Figure 28c. When done in two system clock cycles, the delay time will be as shown in Figure 28f.

Figure 28d shows the delay path from the Interrupt Request Register through the Condition Code MUX to the Am2910. The time calculations are shown in Figure 28g. Again, for some systems, this path may be too long. Therefore, as shown above, this path may be broken in two, which is shown in Figure 28e. This will result in two system clock cycles. The delay involved in each cycle is shown in Figure 28h.

## ANOTHER EXAMPLE OF Am2900 SYSTEM USING THE Am2914

As shown in Figure 29, this example varies in the way that the interrupt request is recognized by the microprogrammed machine. In this example the interrupt request line for the Am2914 enables or disables the $\overline{MAP}$ signal going to the mapping PROM. When an interrupt request is present and a Jump Map instruction is executed, the output of the mapping PROM remains tri-stated; and the bus connected to the "D" inputs of the Am2910 is HIGH because of the pull-up resistors. Therefore, the microprogram will start executing at the highest location in microprogram memory when an interrupt request is present. At this location a Jump Instruction to the microprogram interrupt service routine could be placed. The microcode is written so that the only time a Jump Map instruction is executed is at the end of the Fetch microprogram routine as shown in Figure 30a.

In the previous example the interrupt request was recognized before the program counter is incremented after which the Jump Map instruction is executed. When the Jump Map is executed, either the instruction is executed or an interrupt request is serviced. Therefore, when the Return Interrupt machine instruction is executed, the program counter needs to be backed up via microcode, as shown in Figure 30b, in order to refetch the machine instruction which was lost. This also dictates that the program counter have a path to an incrementer/decrementer or ALU, which in this example is handled by putting the program counter in the Am2903's.

## MICROPROGRAM LEVEL INTERRUPT EXAMPLE

Some high-speed control applications require extremely fast interrupt response. While it may ordinarily be desirable to complete an entire processing sequence (such as executing a microprogram for a macroinstruction) prior to testing for the interrupt and allowing it to occur, it is not always possible to achieve the required interrupt response time desired. If this is the case, microinstruction level interrupt handling must be employed. The technique described below has a maximum latency of three microcycles which can be 450-600ns total. Implementation is straightforward using the Am2910 Microsequencer, a 40-pin LSI device that can control 4096 words of microprogram at a 150ns cycle time, and a few extra MSI and SSI packages. In this application, the Am2910 is configured in its standard architecture. The additional logic does not influence the normal system cycle time.

If microlevel interrupt handling is to be employed, logic must be provided to generate a substitute microprogram address corresponding to the location of the interrupt service routine. In the event of a microlevel interrupt, the sequencer address outputs are tri-stated and the substitute address is placed on the microprogram address bus, causing the next microinstruction fetch to be determined by the interrupt control vector generator. While this is happening, steps must be taken with the Am2910 to insure that the interrupted routine can be properly restored. To understand this procedure, it will be necessary to examine the Am2910 in more detail.

Referring to Figure 31, the microprogram address bus is driven by the Y outputs of the Am2910 through a tri-state buffer than can be disabled by means of the $\overline{OE}$ input. The address is selected in a multiplexer from a direct input, from a register/counter, from a push/pop stack, or from a microprogram counter register. The microprogram counter register is commonly used as the address source when executing the next microinstruction in sequence. Whenever an address appears at the multiplexer outputs, it is incremented and presented to the microprogram counters inputs. At the rising edge of the clock, this new address that is current address-plus-1 is loaded into the microprogram counter and a microprogram access begins at this address.

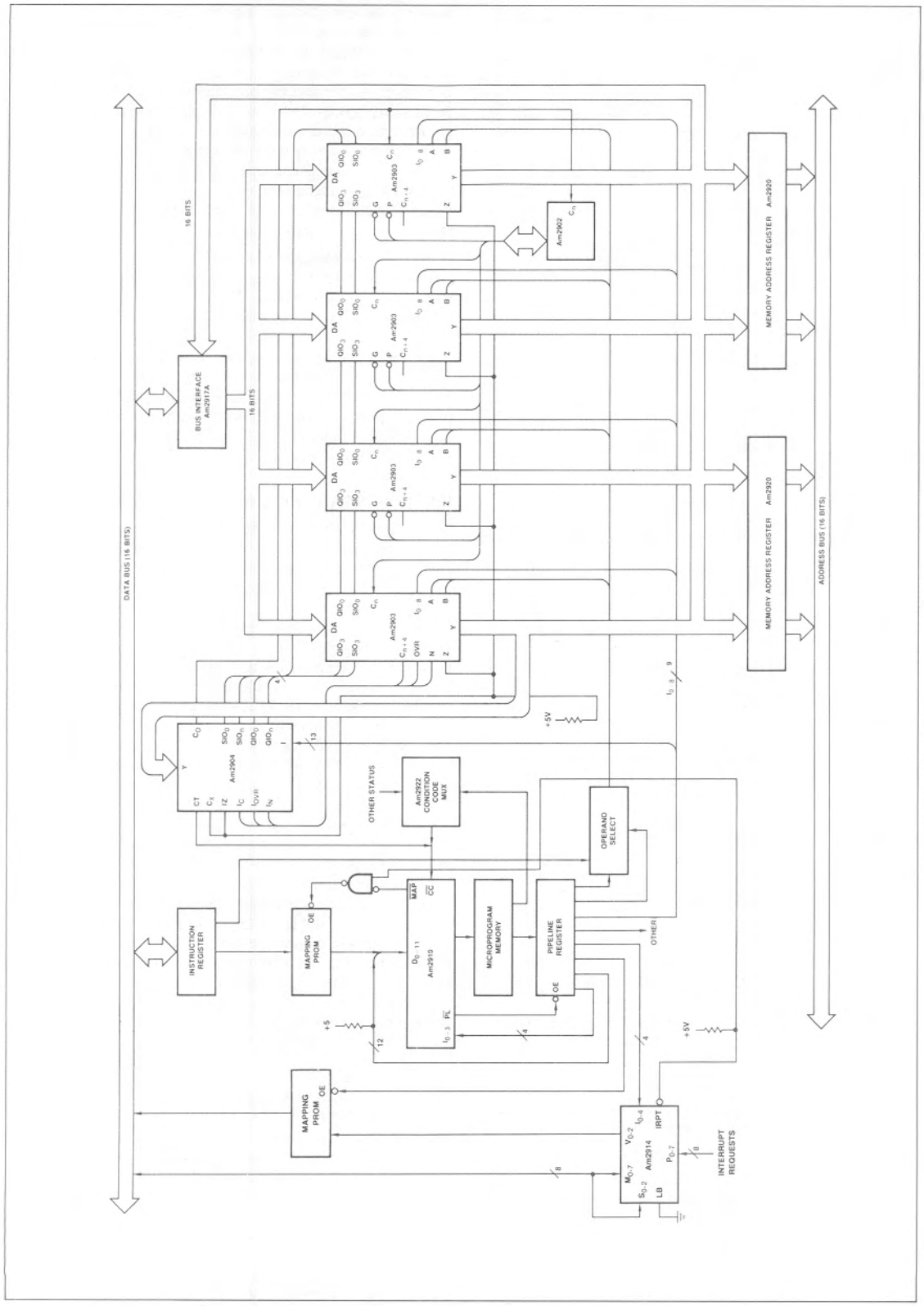



Figure 29. Example of a 16-Bit Computer #2.

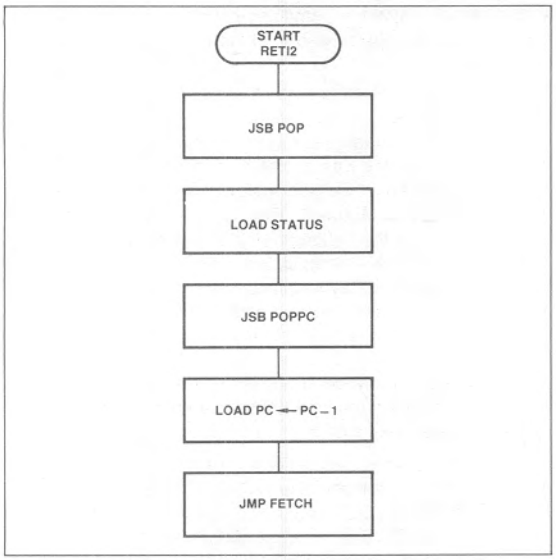



Figure 30a. Return Interrupt Microprogram for Second Example.

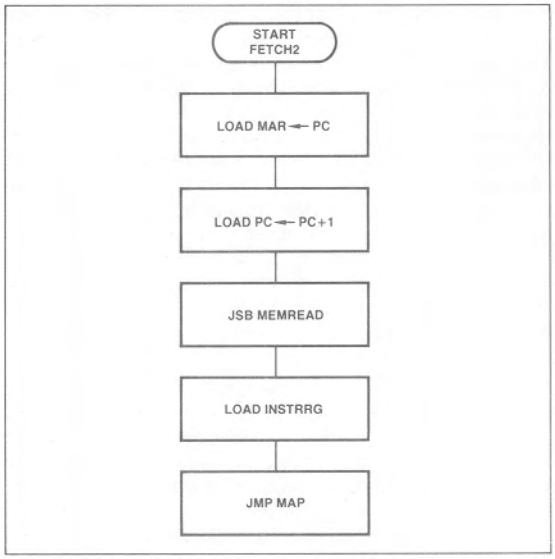



Figure 30b. Fetch Microprogram for the Second Example.

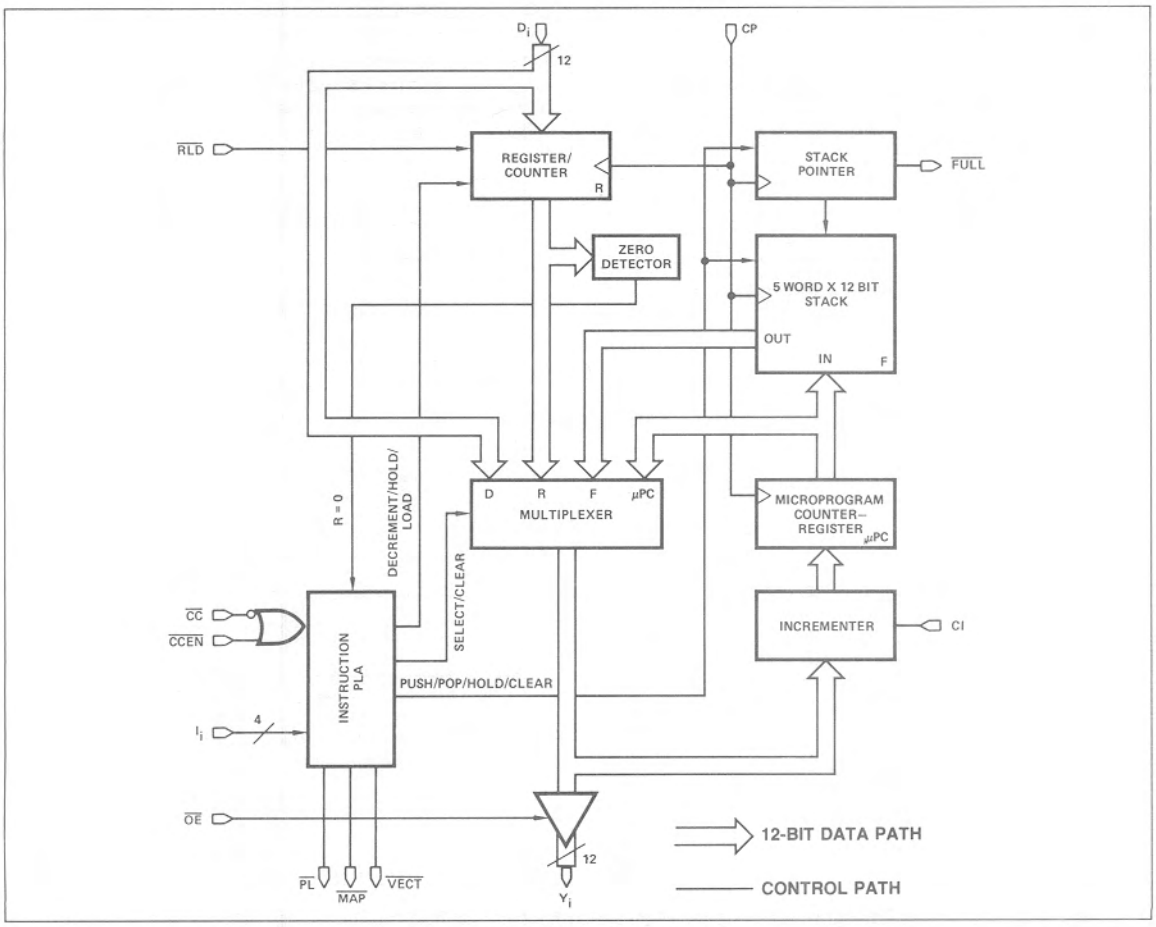



Figure 31. Am2910 Block Diagram.

Note that at this time, whatever was fetched at the previous address was loaded into the microword register for execution. Thus, the microprogram sequencer is always looking for the address of the next microinstruction to be executed (while a previously fetched microinstruction is residing in the microword register). Subroutine and microprogram loops may be accomplished by using the stack and the register counter. *Regardless of what is selected as source of next address, the selected address will be incremented and presented to the microprogram counter.* So to accomplish a microprogram branch, one would simply select the D inputs for a branch address for one cycle, then the next address source could be switched back to the program counter on the next cycle which would then contain the branch address plus 1.

This is a carry input to the incrementer which is normally tied HIGH. In the case of a microlevel interrupt, the microprogram sequencer will not determine the address of the next microinstruction to be executed. Instead the sequencer output will be tri-stated and a substitute address will be placed on the bus. The sequencer continues to operate in a normal fashion with its multiplexer output being incremented and presented to the microprogram counter register. It must now be noted that the instruction located at the address then coming out of the multiplexer outputs *will not be executed* but rather the next microinstruction to be executed will be determined by the interrupt vector generator. It would therefore, be wrong to increment this microprogram address but rather it must be saved intact in order to push it onto the stack for access during interrupt return. This is easily accomplished in the Am2910 by grounding the carry input to the incrementer simultaneously with three-stating the sequencer output. Then the multiplexer output will be stored in the microprogram counter register and on the next microcycle the Am2910 must be told to push in order to preserve this address on the stack.

This carry-in input is all important and exists on all Advanced Micro Devices' microprogram sequencers. Unless the carry-in is grounded, whatever address was in the multiplexer output when the sequencer output was tri-stated is incremented and an instruction is missed in the interrupted routine. This, of course, would likely be disastrous. The key to this microinterrupt technique is that the address of the unexecuted instruction (when the Am2910 was tri-stated and a substitute address supplied) is preserved by inhibiting the increment via the carry input, so the address is passed on intact to the microprogram counter. If the microinterrupt is to be more than one cycle long, the microprogram counter must be pushed so as to save the return address. Otherwise, a "continue" may be used to return from the interrupt on the very next cycle. In this event the microinterrupt effectively inserts one instruction in the stream.

Figure 32 is the block diagram of a hardware design that implements the above concept. The SYNC/CONTROL and INTERRUPT CONTROL/VECTOR GENERATOR logic are shown in detail in Figure 33. Part of the Am2918 and both 'LS74 Flip-Flops are used to synchronize the recognition of the asynchronous interrupt request as shown in Figure 34. The interrupt request arrives at the interrupt input. On the next clock cycle it is clocked into the Am2918. In the following clock cycle a pulse that is one system clock cycle long is put out by the flip-flop pair FF1 and FF2. The pulse is used to disable the carry input of the Am2910, tri-state the output of the Am2910, and enable the jump vector onto the input of the PROM. The vector indexes into a table in microprogram memory that contains "JUMP SUBROUTINE" instructions to different interrupt service routines.

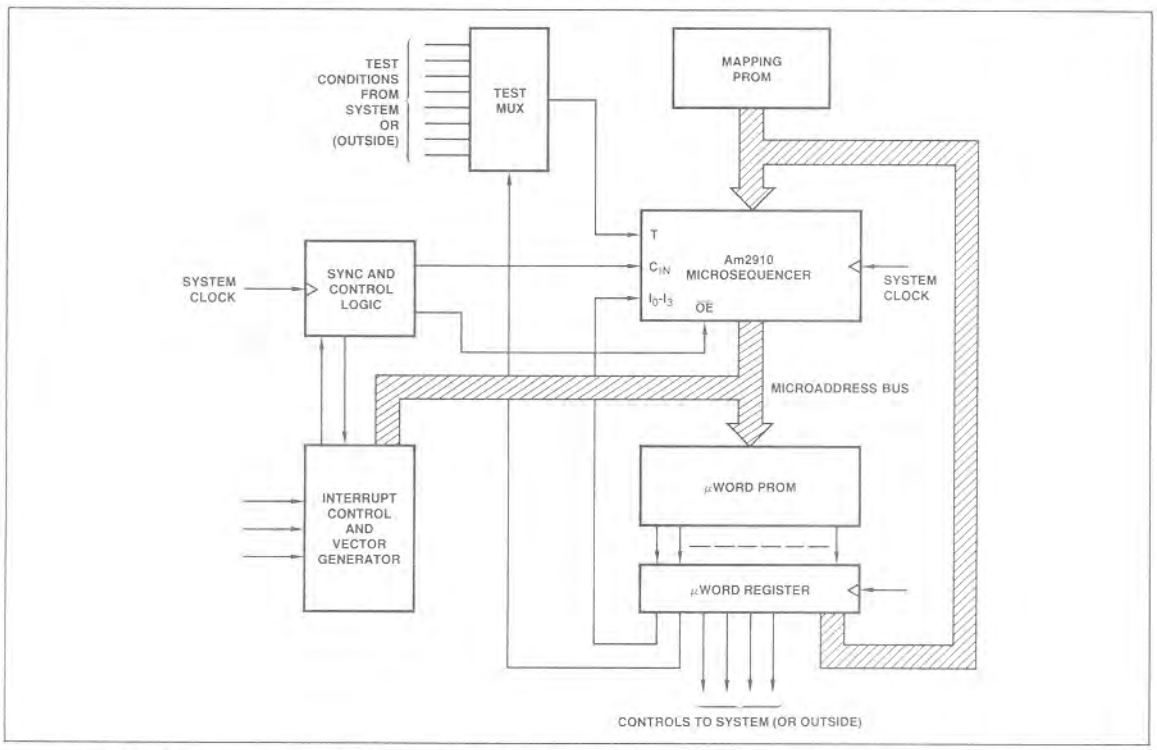



**Figure 32. Computer Control Unit Set-up for High-Speed Micro-Level Interrupt Handling. Latency is a Maximum of Two Microcycles (i.e., about 300 to 500ns).**

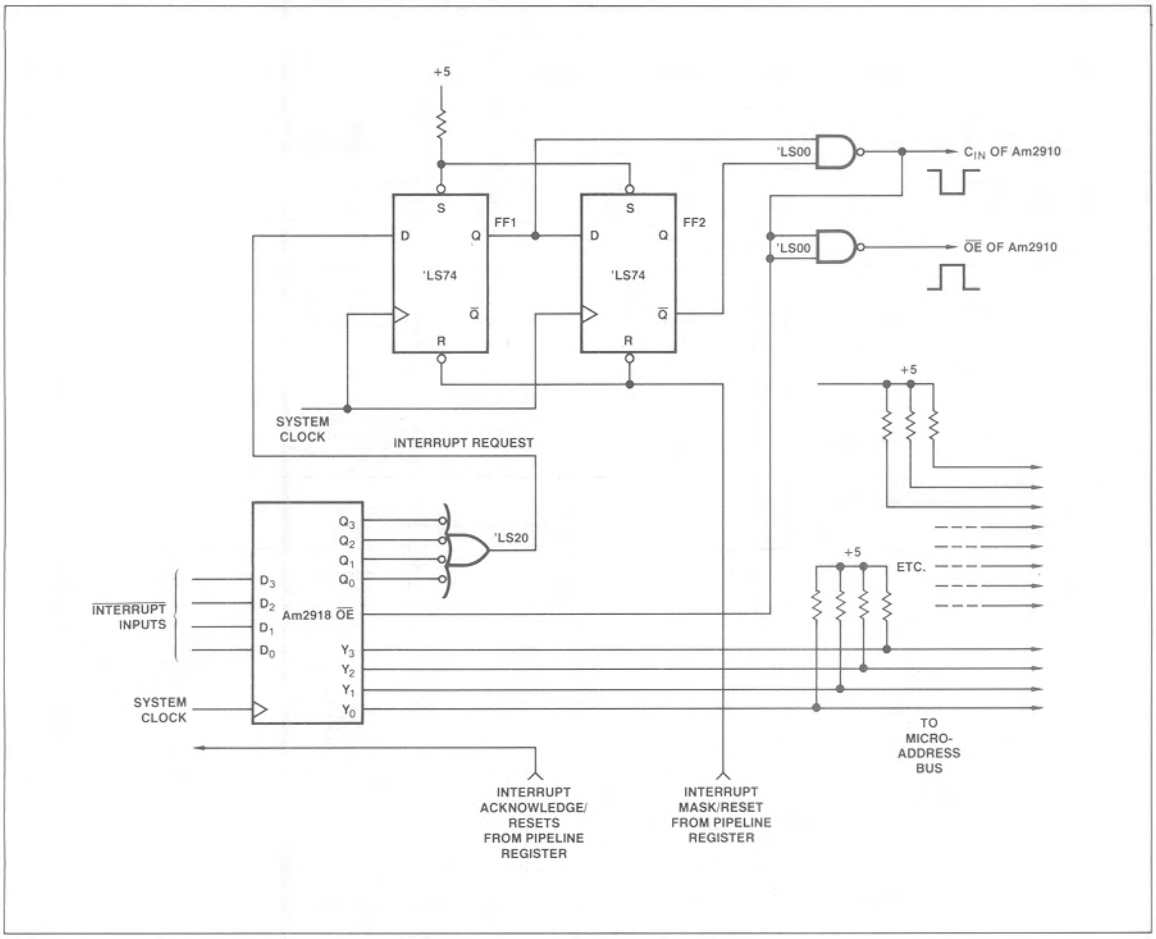



**Figure 33. Example of Sync Control Logic and Vector Generator.**

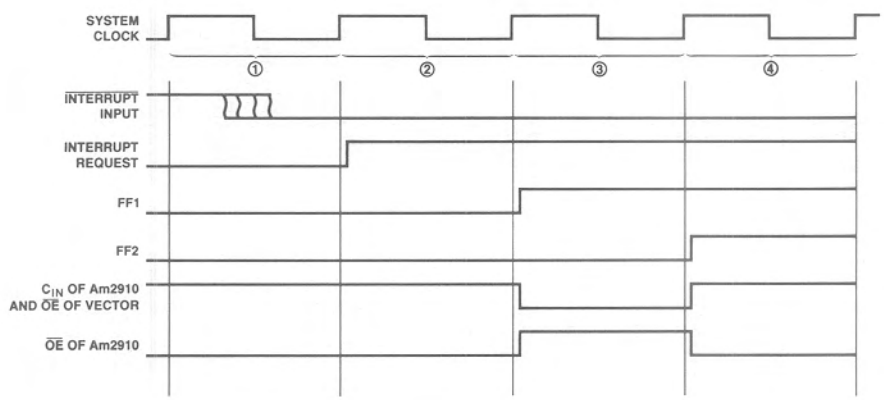
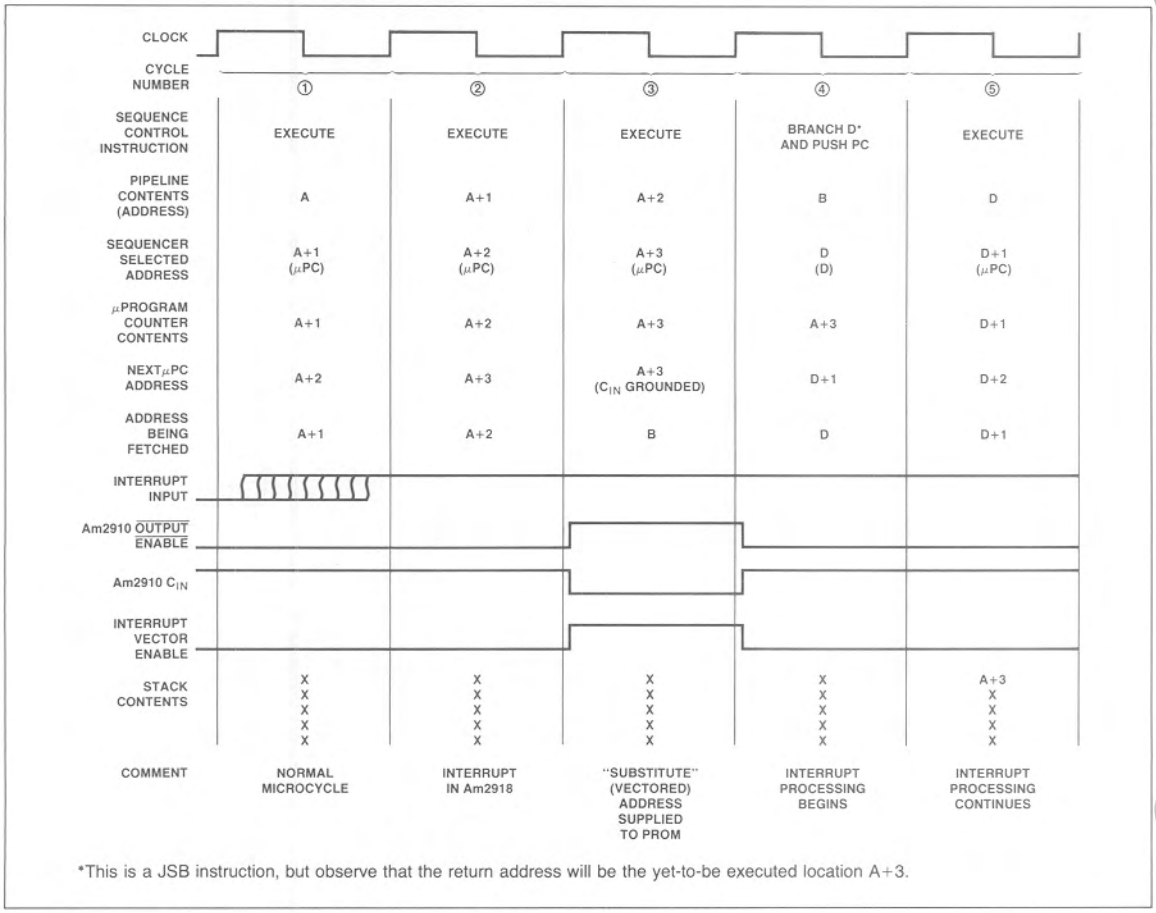



**Figure 34. Timing of Vector Generator and Sync Control Logic.**




*This is a JSB instruction, but observe that the return address will be the yet-to-be executed location A+3.

**Figure 35. Interrupt Sequence Timing.**

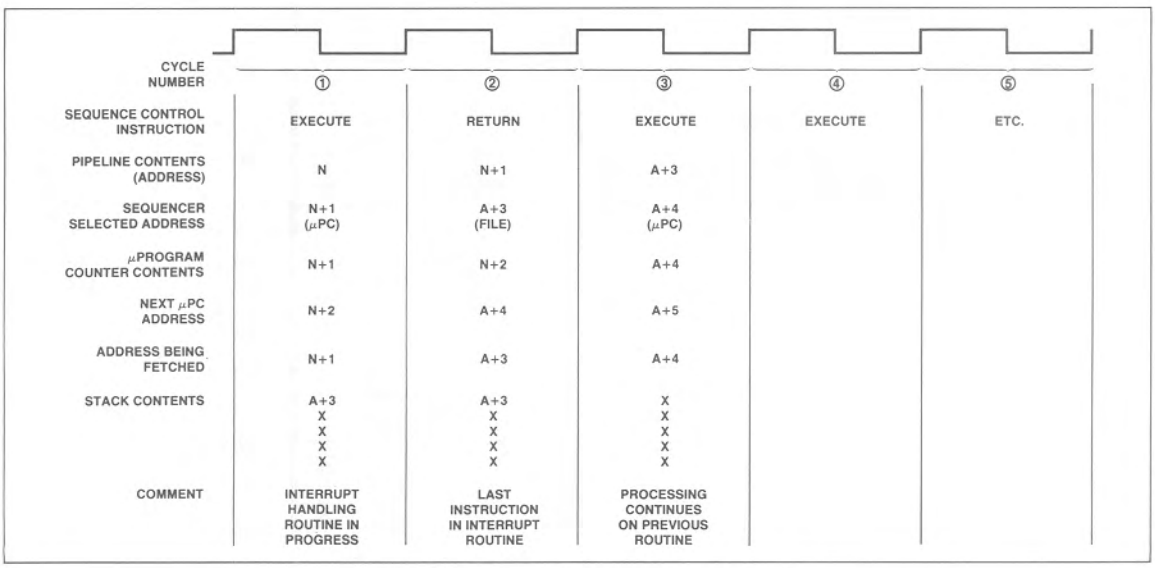



**Figure 36. Return-From-Interrupt Sequence Timing.**

Figure 35 shows how the interrupt sequence timing fits into the normal flow of microprogram address in the Am2910. Note how the stack is used. This demonstrates the need for always reserving room on the stack to allow for interrupts. This applies to any room that the interrupt service routine may require as well as the return address. This limitation may require that only one interrupt request be serviced at a time.

Figure 36 shows how the return from the interrupt service routine fits into the microprogram flow. Notice that a Return instruction is used to accomplish this.

## SUMMARY

In this chapter, Interrupts were discussed beginning with a definition of the Interrupt Mechanism and proceeding to a classification of different interrupts and how they are handled. A discussion of the concepts that go into designing the "Universal Interrupt" hardware was given which culminated with the Am2914. The chapter ends with several Interrupt Mechanism applications using the Am2914 and Am2910.

In this chapter it was shown how interrupts can be handled using parts from the Am2900 family. Because of their hardware modularity and universal architecture, they may be used in a variety of applications. Since the Am2900 Family parts are microprogrammable, they allow the user's system to grow with time as system requirements change. Together these attributes make the Am2900 Family the flexible cost effective family that it is.

***ADVANCED***
***MICRO***
***DEVICES, INC.***
*901 Thompson Place*
*Sunnyvale*
*California 94086*
*(408) 732-2400*
*TWX: 910-339-9280*
*TELEX: 34-6306*
*TOLL FREE*
*(800) 538-8450*