Application note

THE Z80 FAMILY PROGRAM INTERRUPT STRUCTURE

TABLE OF CONTENTS

Note: The following Application Note assumes a basic knowledge of the Z80 Microprocessor family. The Zilog literature listed below is recommended as reference material:

- ◆ Z80-CPU Technical Manual

- ◆ Z80-Assembly Language Programming Manual

- ◆ Z80-PIO Technical Manual

- ◆ Z80-CTC Technical Manual

1 INTRODUCTION

The goal of an efficient interrupt system is to handle all peripheral devices with a minimum amount of response time and overhead. This overhead is composed of recognizing which device is requesting service, saving the current status of the CPU, and branching to the service routine. Having serviced the device, it is necessary to restore the pre-interrupted conditions and to return to the interrupted program. To service multiple devices, a priority interrupt structure will be necessary to optimize overall system performance.

In order to meet these goals, the Z80 Family incorporates one non-maskable and three software maskable interrupt modes including a vectored interrupt mechanism that offers very fast interrupt response by an indirect call to the start of the device service routine and the use of an alternate register bank to achieve very fast context switching.

2 INTERRUPT PROCESSING

The acceptance and service of any interrupt is related to the status of the Bus Request line (/BUSRQ) and the occurrence of the interrupt within the instruction cycle. A Bus Request is used to request the CPU to put address, data, and control lines into a high impedance state usually in response to a pending Direct Memory Access transfer (DMA).

Figure 1 is a flow diagram detailing the standard interrupt processing sequence.

The Z80 CPU will honor an external event according to the following priority:

1. Bus Request
2. Non Maskable (/NMI)
3. Maskable Interrupt (/INT)

The bus request line (/BUSRQ) will be sampled (Internal CPU Bus Request Flip Flop set if active (low)) and if active, will be acknowledged at the end of a machine cycle. If the last cycle is also the end of the instruction, the CPU will recognize an interrupt (non-maskable or maskable) by setting appropriate internal flip flops, but will first service any pending Bus Request. At completion of a Bus Request operation, the CPU checks the status of the /NMI flip flop. If set, it executes the /NMI interrupt routine as indicated in Figure 2. If the /NMI flip flop is not set, the CPU checks the status of the maskable interrupt flip flop (/INT F/F). If set, it executes any one of the three interrupt modes as indicated in Figure 3. If the /INT F/F is reset, the CPU returns to main line program processing.
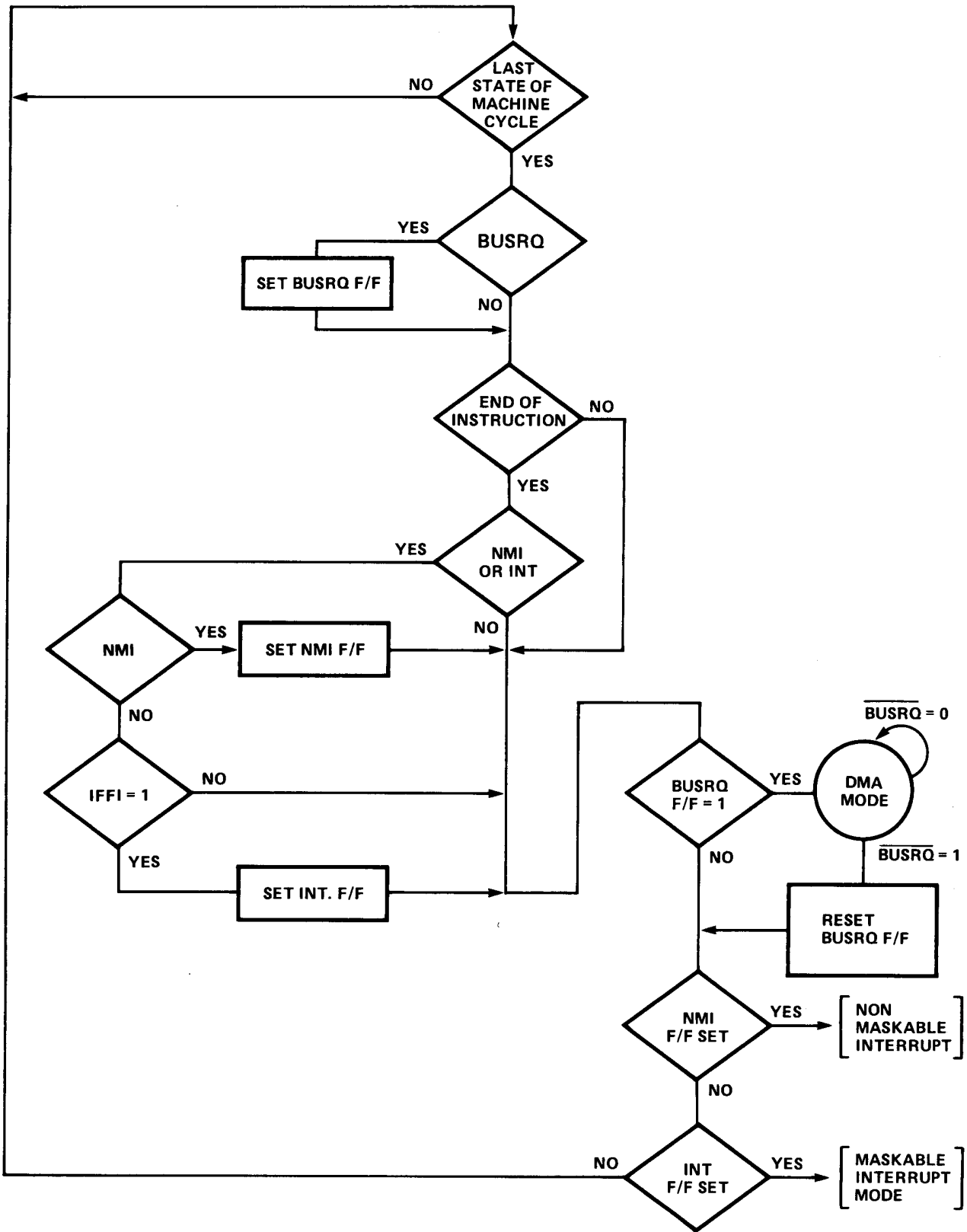
Figure 1: Z80 Flow Diagram: Interrupt Sequence

3 NON-MASKABLE INTERRUPT

This negative edge triggered interrupt cannot be disabled under program control and will be accepted at any time by the CPU to be honored at the completion of the current instruction (if Bus Request not pending). If an /NMI occurs during a /BUSRQ true condition, it will be latched and processed after /BUSRQ goes false. If an /NMI occurs during a reset true condition, the CPU will execute one mainline program step after reset is taken away and then process the /NMI.

The non-maskable interrupt (/NMI) has priority over any maskable interrupt and generates an automatic restart to location 0066H. It is normally dedicated to functions requiring a rapid response (such as power failure) or is used with peripherals that must be serviced immediately.

When /NMI is accepted, the flip flop used to enable maskable interrupts (IFF1) is reset to prevent interrupts during the /NMI service routine. However, the state of IFF1 just prior to the /NMI is saved in the temporary storage flip flop IFF2. (IFF2 can be copied into the parity flag by loading the accumulator with either the I or R register (LD A,I or LD A,R).) Figure 4 details the /NMI request operation. The CPU response to an /NMI is similar to a normal memory read operation except that the data bus is ignored on the next /M1 cycle. The CPU automatically pushes the Program Counter (PC) onto the external stack in the following cycle. At the completion of the /NMI routine, the execution of a Return from Non-Maskable Interrupt (RETN) will copy the contents of IFF2 back into IFF1 thus automatically restoring maskable interrupt enable status. (See Figure 2.)
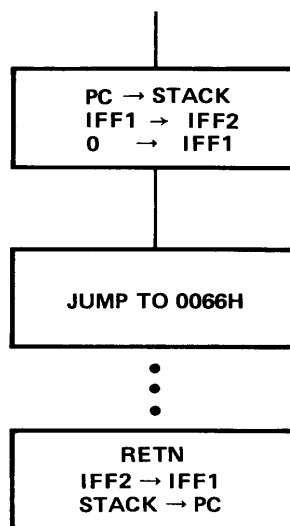
```
                    ┌─────────────────────┐
                    │   PC → STACK        │
                    │   IFF1 → IFF2       │
                    │   0   →  IFF1       │
                    └─────────────────────┘
                    ┌─────────────────────┐
                    │                     │
                    │   JUMP TO 0066H     │
                    │                     │
                    └─────────────────────┘
                              •
                              •
                              •
                    ┌─────────────────────┐
                    │      RETN           │
                    │   IFF2 → IFF1       │
                    │   STACK → PC        │
                    └─────────────────────┘
```

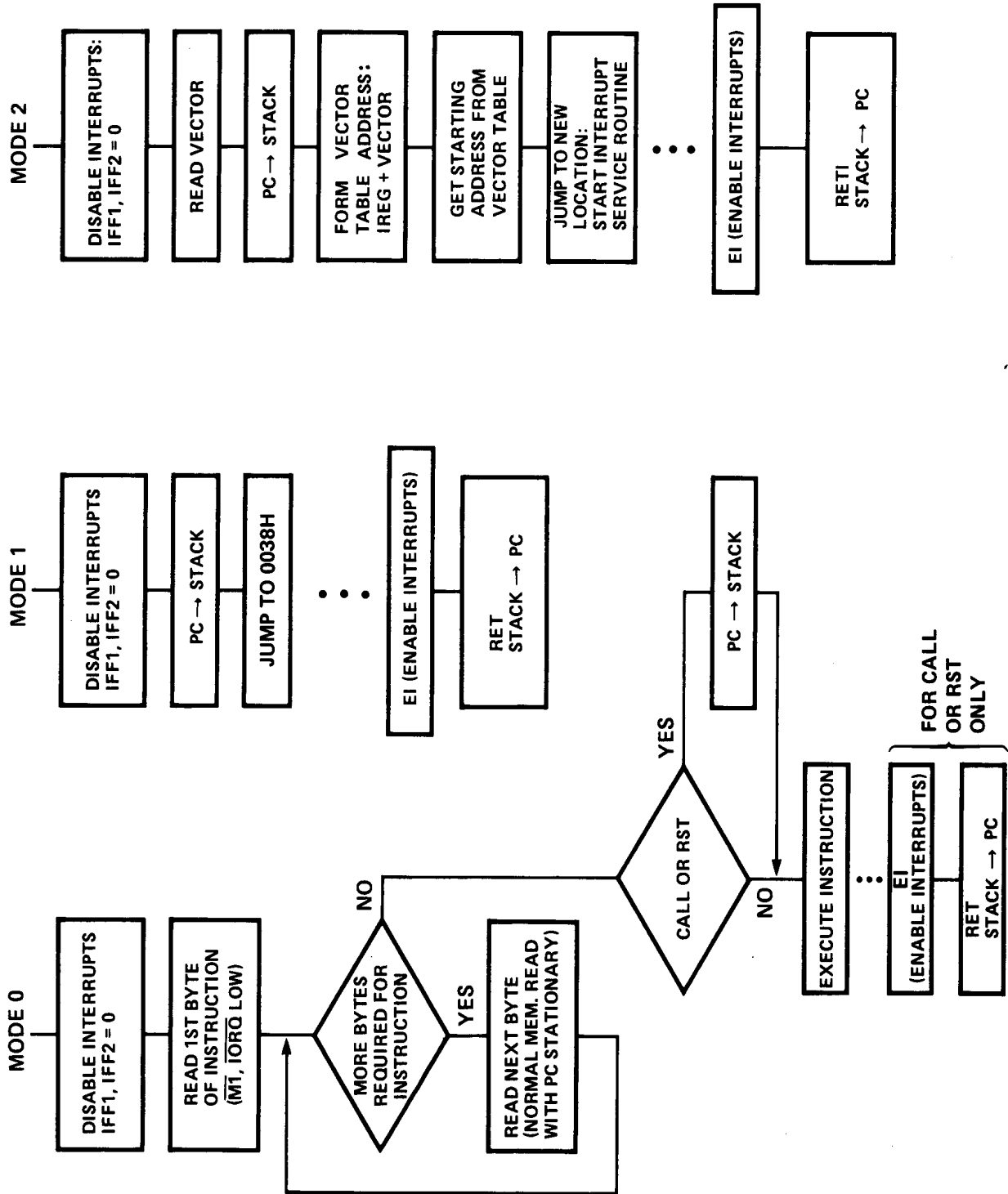*Figure 2: Non-maskable Interrupt Sequence*

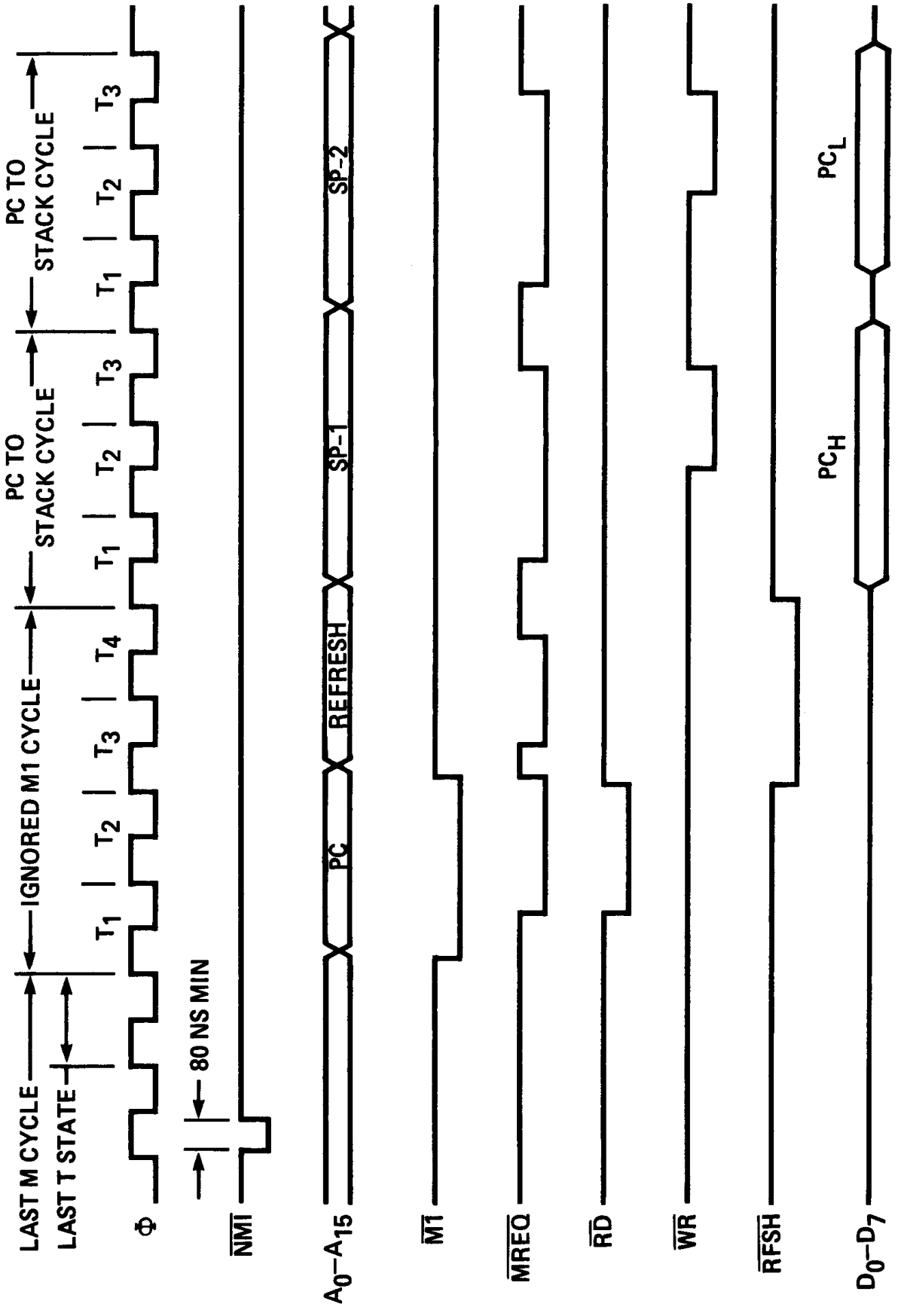Figure 3: Maskable Interrupt Sequences

6

Figure 4: Non-maskable Interrupt Request Operation

7

4 MASKABLE INTERRUPTS

The maskable interrupt (/INT) can be selectively enabled or disabled by the programmer. The enable interrupt instruction (EI) will set both IFF1 and IFF2 to a logic 'one' allowing recognition of any maskable interrupts at the completion of the instruction following the EI instruction. The Disable Interrupt Instruction (DI) will reset both IFF1 and IFF2 to a logic 'zero' and interrupts will not be recognized. Note, that during the execution of both EI and DI, maskable interrupts will be disabled.

MASKABLE INTERRUPT MODE 0

In the maskable interrupt mode 0 the interrupting device places an instruction on the data bus for execution by the Z80-CPU (identical to the 8080A interrupt response mode). The instruction is normally a Restart (RST) instruction since this is an efficient one byte call to any one of eight subroutines located in the first 64 bytes of memory. (Each subroutine is 8 byte long.) However, any instruction may be given to the Z80-CPU. The first byte of a multi-byte instruction is read during the interrupt acknowledge cycle. Subsequent bytes are read in by a normal memory read sequence (The PC, however, remains at its pre-interrupt state and the user must insure that memory will not respond to these read sequences.). (See Figure 3.)

When the interrupt is recognized, further interrupts are automatically disabled (IFF1 and IFF2=0). Any time after the interrupt sequence begins an EI instruction can be executed so that this subroutine itself can be interrupted. This process may continue to any level as long as all pertinent data are saved and restored.

A CPU reset will automatically set interrupt mode 0.

MASKABLE INTERRUPT MODE 1

This maskable mode allows peripherals of minimal complexity interrupt access. In this respect, it is similar to the /NMI interrupt except that the CPU does an automatic CALL to location 0038H instead of 0066H. As in the /NMI, the CPU automatically pushes the PC onto the Stack (See Figure 3). Note, that when doing programmed I/O the CPU will ignore any data put onto the data bus during the interrupt acknowledge cycle.

MASKABLE INTERRUPT MODE 2 (VECTORED INTERRUPTS)

The Z80-CPU supports an interrupt vectoring structure that allows the peripheral device to identify the starting location of the interrupt service routine.

Mode 2 is the most powerful of the three maskable interrupt modes allowing an indirect call to any memory location by a single 8 bit vector supplied from the peripheral. In this mode the peripheral generating the interrupt places the vector on the data bus in response to an interrupt acknowledge. This vector then becomes the least significant 8 bits of he indirect pointer while the I register in the CPU provides the most significant 8 bits. This address in turn points to an address in a vector table which is the starting address of the interrupt

routine. Interrupt processing thus starts at an arbitrary 16 bit address allowing any location in memory to be the start of the service routine. Notice that since the vector is used to identify two adjacent bytes to form a 16 bit address, only 7 bits are required for the vector and the least significant bis is zero. Figure 5 shows the sequence of events for processing vectored interrupts.
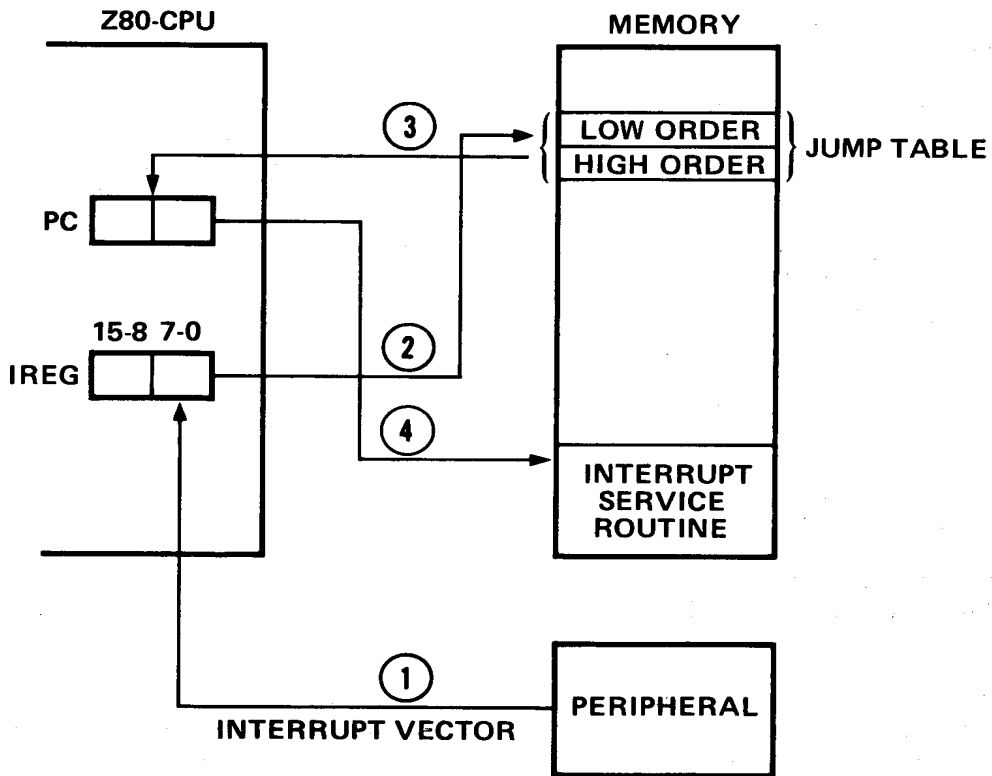

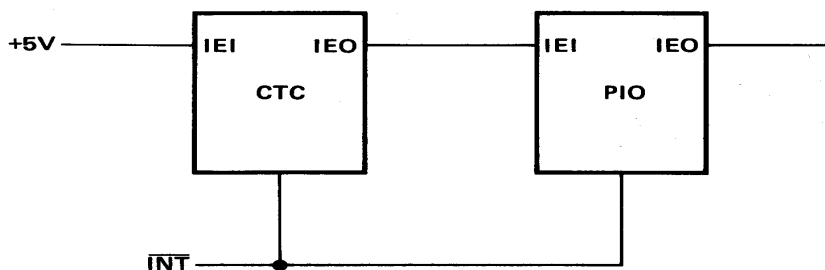Figure 5: Vector Processing Sequence

Figure 6: CTC/PIO Priority Chain

5 INTERRUPT PRIORITY

Real time overhead in handling multi-level interrupts is significantly reduced in Z80 systems. Without requiring additional logic between the Z80 CPU and peripheral devices the interrupt mechanism is able to select the prioritized device from among several requesting simultaneous service.

Priority is set by the location of the device in a daisy chain configuration with each device tied to the /INT line. Two lines, Interrupt Enable In (IEI) and Interrupt Enable Out (IEO) are provided in each peripheral. Figure 6 shows the Z80-CTC and Z80-PIO peripherals connected in a typical configuration. The IEI of the CTC is tied to +5 volts to indicate that it has the highest priority. The PIO is the second highest priority device with its IEI tied to the IEO of the CTC. The priority string insures that a device with a higher priority will be serviced before a lower priority device when two or more /INT requests occur at the same time.

For a device to have priority, its IEI must be high. When a device needs service, it will prevent downstream devices from interrupting by pulling low on its IEO line. The next device in the chain sensing a low at the IEI input will pass this 'priority signal' on to the next device by pulling low on its IEO line. The IEO line, therefore, of any given peripheral will satisfy the following relationship:

$$IEO = IEI \cdot /HELP$$

Where /HELP is an internal peripheral signal indicating that the device needs service.

For example, if the CTC in Figure 6 needs service, it will generate an interrupt. It pulls low on the IEO line blocking any interrupts from downstream devices. When it receives an interrupt acknowledge (/M1 low and /IORQ low), it places the service vector address on the bus.

Before the fall of /M1 a device is free to pull low on the /INT line and contend for interrupt priority. However, during the interrupt acknowledge time frame, Z80 peripherals will be inhibited from changing their interrupt request status. This time, between the fall of /M1 and the fall of /IORQ, is used to resolve interrupt priority. (See Figure 10.) It should also be pointed out that the priority structure works for any of the maskable interrupt modes since an interrupt acknowledge cycle is common to each.
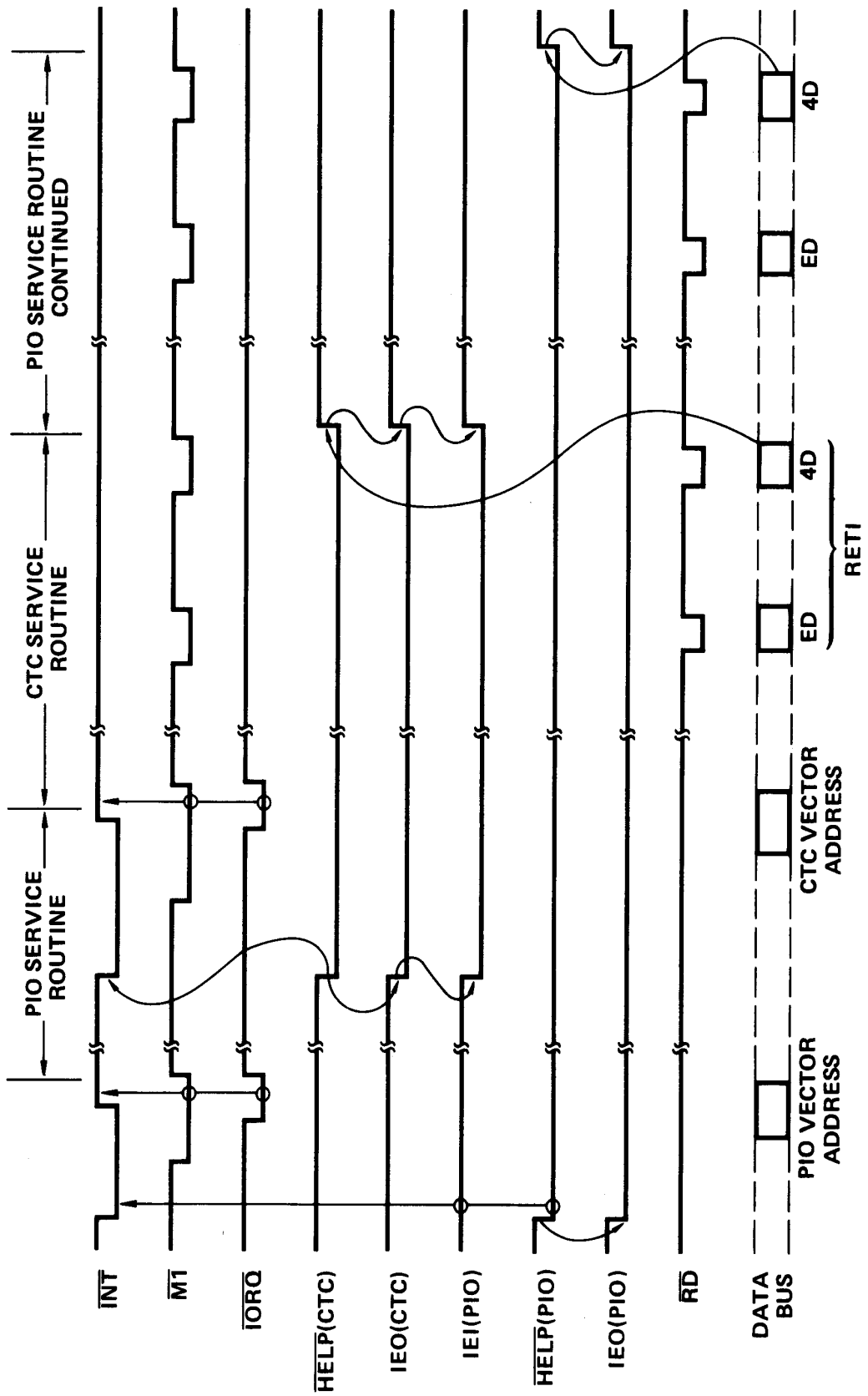
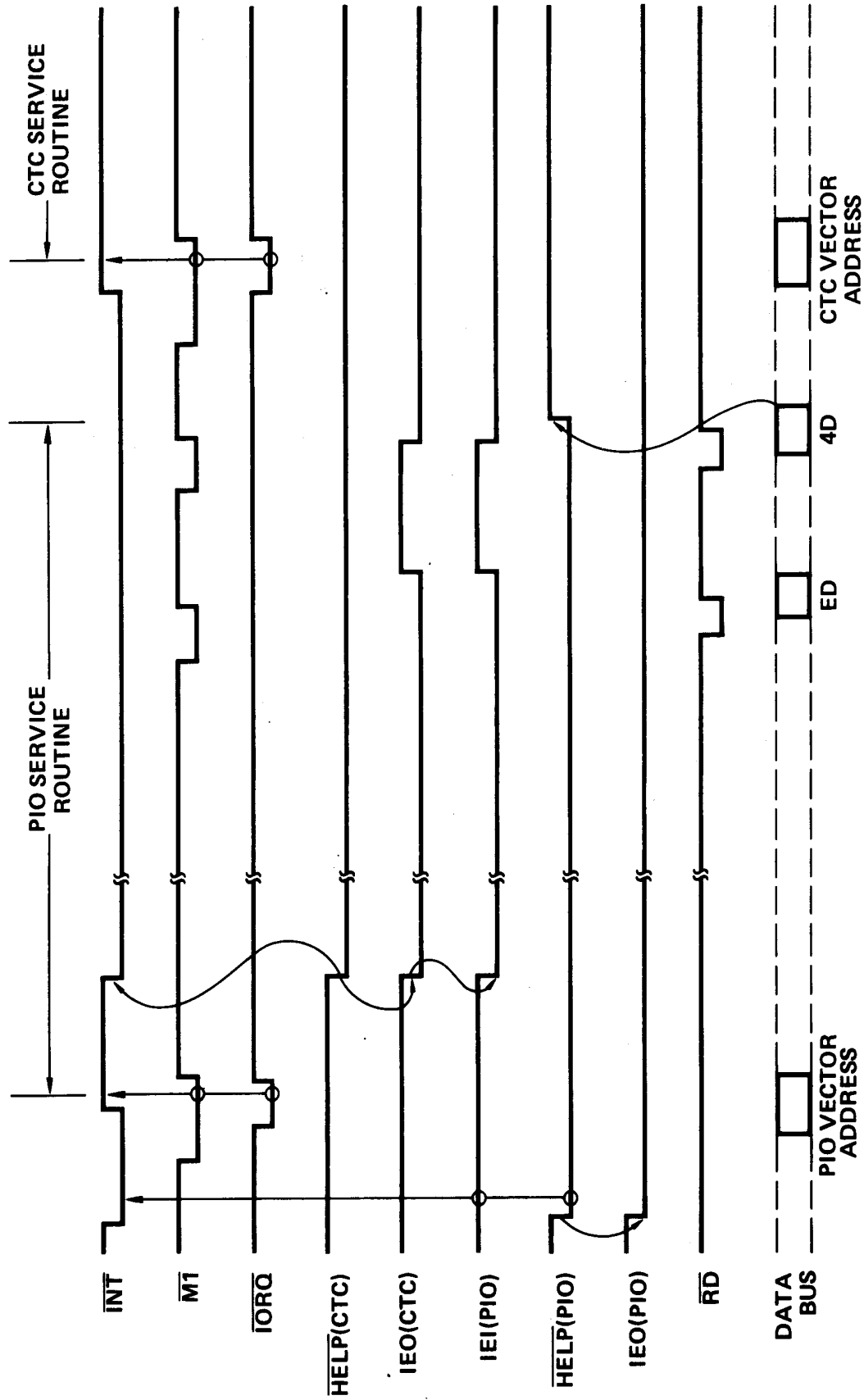*Figure 7: CTC/PIO Interrupt Timing with Interrupts Enabled*

11

*Figure 8: CTC/PIO Interrupt Timing with Interrupts Disabled*

6 INTERRUPT NESTING

By enabling interrupts during a particular device service routine, a particular device service routine, a priority structure is established which allows higher priority devices to interrupt the current service routine. This nesting mechanism will enable a high speed device interrupt access and to temporarily suspend interrupt service of a slower device. An automatic return to the slower speed device service routine is performed at the completion of the higher speed device service routine.

To illustrate the nesting mechanism, a typical interrupt processing routine is detailed on the configuration in Figure 6 and the timing relationships are shown in Figure 7 and Figure 8.

The PIO requests interrupt service by setting its /HELP logic and pulling the /INT and its IOE line low. Assuming interrupts have been enabled, the Z80-CPU finishes the current instruction and responds with an interrupt acknowledge (/M1 and /IORQ low). The interrupt vector is read and the contents of the PC are stored in the external stack. Interrupts are automatically disabled (IFF1 and IFF2=0), whenever an interrupt is acknowledged and must be subsequently re-enabled (EI) to permit detection and handing of future higher priority interrupts.

While in the service routine for the PIO (with interrupts having been enabled (IFF1=1)), the CTC generates an interrupt (/INT, /HELP and /IEO all go low). The IEI of the PIO is now low blocking any downstream devices. The Z80-CPU finishes the current instruction of the PIO service routine, responds with an interrupt acknowledge and reads in the interrupt vector. The contents of the PC are stored in the external stack and IFF1 and IFF2 are once again reset, disabling interrupts. The external stack now looks like:
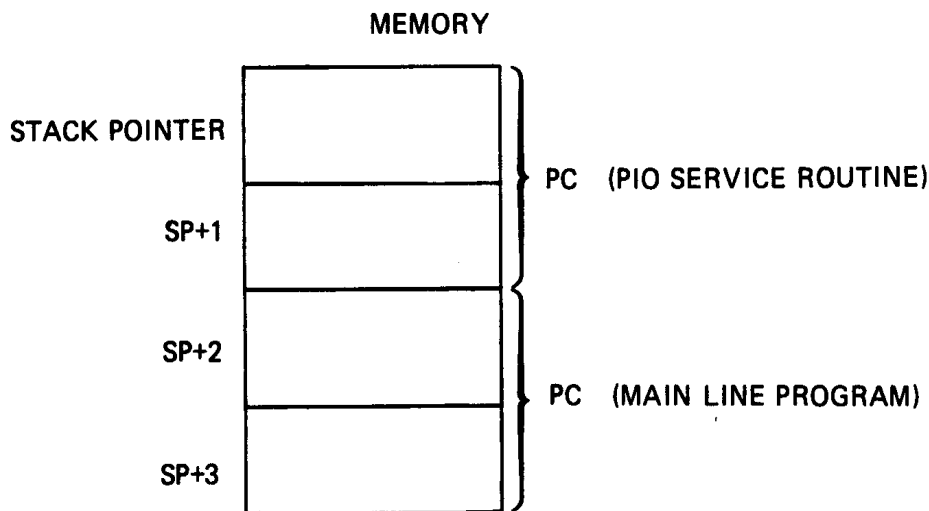

At the completion of the CTC service routine, the RETI (Return from Interrupt) instruction is used to restore the contents of the Program Counter (PC) to that of the PIO service routine and to reset the CTC /HELP logic. The CTC IEO line goes high and the PIO IEI now being high indicates that the PIO can finish its service routine.

Note that, as indicated previously, prior to the RETI instruction, an EI instruction can be executed to enable interrupts for the remaining segment of the PIO service routine and EI will not take effect until after the RETI has been executed.

The PIO service routine is completed and another RETI instruction (ED 4D) resets the /HELP logic and the IEO line. The stack is popped restoring the PC to the main line program.

A variation to the above example results when interrupts are disabled during the PIO service routine. Now if the CTC needs service (/HELP logic low with PIO low), the /INT line will be pulled low but the CPU will not respond (IFF1=0). However, when an RETI is issued for the PIO routine, the CTC /HELP logic must not be reset. Therefore, the CTC will allow its IEO line to go high for one /M1 cycle during the RETI instruction (see Figure 8.) In other words, if an interrupt acknowledge is not given to a device requesting service, its IEO line will be forced high for one /M1 cycle after decoding ED (first byte of RETI) to allow downstream devices to decode RETI. For systems with more than four peripherals, the propagation delay of this signal rippling from the highest priority device with a pending interrupt to the lowest priority device needing to be reset may exceed the time interval between the decoding of ED and the decoding of 4D. The downstream device could potentially miss the RETI instruction. To prevent this, the IEO line could be inhibited from going low (with a pending interrupt) until the beginning of the interrupt acknowledge cycle. (See the section on Interrupt Acknowledge and Daisy Chain Propagation.) Figure 9 identifies the transition of the IEI an IEO lines within the RETI instruction and the relative propagation from IEI to IEO.

Figure 9: IEI/IEO TIMING WITHIN RETI INSTRUCTION AND IEI TO IEO PROPAGATION DELAY
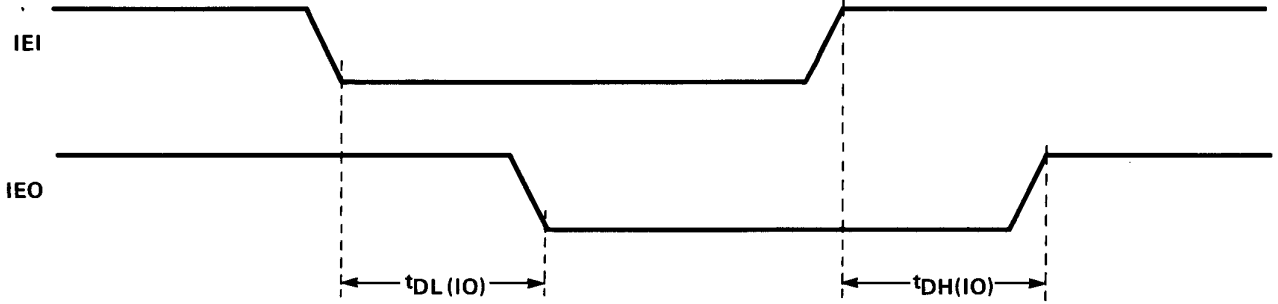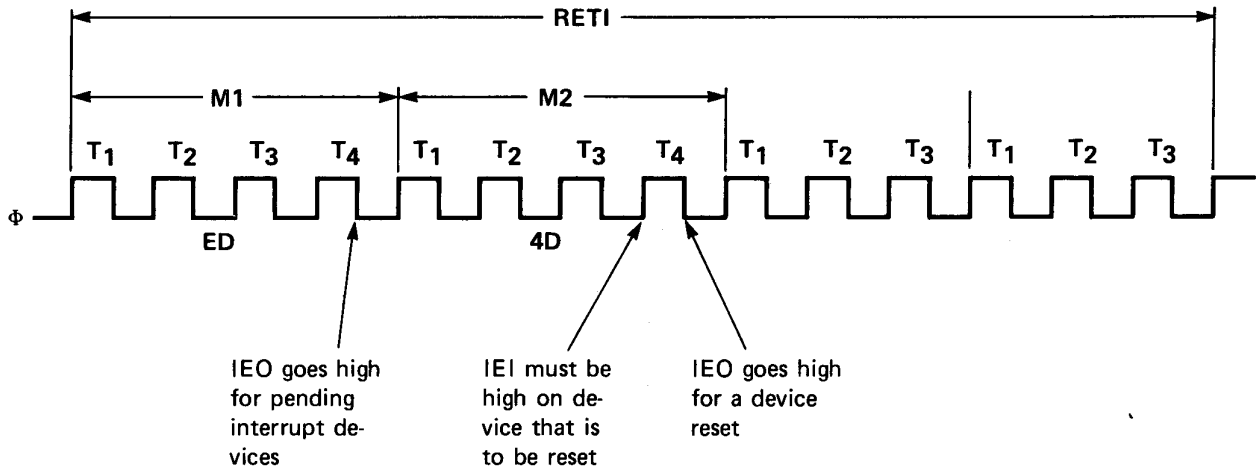
4 MHz PART	tDL(IO)	tDH(IO)
Z80-PIO	130 ns max	160 ns max
Z80-CTC	130 ns max	160 ns max
Z80-DMA	130 ns max	160 ns max
Z80-SIO	100 ns max	120 ns max

7 INTERRUPT OVERHEAD

The time and bytes required in saving and restoring CPU working registers and status conditions is an important consideration during interrupt processing. The Z80 with a dual register bank is uniquely designed to minimize status and data saving operations because for many interrupt conditions the contents of the CPU working registers can be simply switched from one register bank to another. Two 1 byte instructions exchange the contents of the entire working register bank:

      EXX   EXCHANGE THE CONTENTS OF BC, DE, HL WITH

                               CONTENTS OF BC', DE', HL' RESPECTIVELY

      EX AF,AF'   EXCHANGE THE CONTENTS OF AF AND AF'

After servicing a device, the service routine restores the pre-interrupt status of the CPU, enables interrupts and returns control to the interrupted program. The enable interrupt (EI) instruction however has no effect on IFF1 until the next instruction has begun. Thus, after the EI instruction, the CPU always executes one more instruction (usually a return instruction) before another interrupt can start.

The Z80 also utilizes an external memory pushdown stack from the storage of all pertinent data and addresses during interrupt processing. The stack is essentially a last in first out buffer with the top of the stack as the entry point. The address of this entry is stored in the 16 bit Stack Pointer register (SP). Storage and retrieval from the stack is performed using the Z80 PUSH and POP instructions. The following instructions store the contents of the working registers and status onto the stack:

      PUSH   BC
      PUSH   DE
      PUSH   HL
      PUSH   AF
      PUSH   IX
      PUSH   IY

The following instructions restore the registers and status:

POP	IY
POP	IX
POP	AF
POP	HL
POP	DE
POP	BC

During an NMI and mode 1 and 2 maskable interrupts, the Program Counter is automatically pushed onto the stack. An interrupt routine is ended with the return from interrupt instruction (RETN (non-maskable) and RETI (maskable)) which automatically pops the stack, restoring the PC. Notice that for nesting routines, the correct return address will always be at the current top of the stack.

8 INTERRUPT ACKNOWLEDGE AND DAISY CHAIN PROPAGATION

The /INT line is sampled at the beginning of the last T state of the last M cycle of an instruction. Upon acceptance (If IFF1=1 and interrupts are enabled) an interrupt acknowledge will be performed. This consists of a special /M1 cycle with an /IORQ signal instead of the normal memory request signal (/MREQ). At the leading edge of /M1 and before the leading edge of /IORQ (see Figure 11) peripherals are free to contend for priority.

The CPU will automatically insert one more wait state (for a total of two) to allow additional ripple time for this priority determination. Up to four Z80 peripheral devices can be serviced without added logic.

When designing systems with more than four I/O devices, the USER can either extend interrupt acknowledge cycle time or reduce priority ripple time.

A possible technique for extending interrupt acknowledge time is to gate /IORQ to the peripheral devices. Figure 10 details the relatively simple logic required. When /M1 goes low the /WAIT line is activated and /IORQ' (to the peripherals) is maintained high until a time out determined by the LS191 synchronous counter. This counter (strapped to count up) can be user-programmed to add from none up to 5 wait states. Figure 10 Shows the counter strapped to add one wait state, which requires that it be clocked four times (counter clocks on the falling edge of the T clock after /M1 goes low, see Figure 10). When the D output goes high, the wait line is deactivated and /IORQ' goes true (low). (It should be pointed out, however, that while in the wait state, the CPU will not recognize a non-maskable interrupt.)

If no additional wait states are necessary, counter inputs A, B and C are tied high and the D input is tied to ground. The counter will now set the D output high on the next clock edge after /M1 goes low resetting the D flip flop and thus allowing a normal /IORQ.

To reduce the ripple time, a configuration as detailed in Figure 11 can be used. The 74S182 is a look-ahead carry generator that can be used with a group of up to 4 peripherals. This logic anticipates an IEO low condition at any device and generates a look-ahead signal to the propagate output. The logic can be cascaded to allow additional peripherals and results in a 25 ns maximum ripple for any IEO to P out. Note, however, that this configuration only hinders the RETI ripple function. This ripple propagation occurs when the chain is re-estabished after a higher priority device has its service routine completed and allows its IEO line to go high. The look ahead logic now adds to this propagation delay.

For systems with more than four peripheral devices, it is a good design practice to inhibit IEO from going low until the beginning of the interrupt acknowledge cycle. This eliminates the RETI ripple propagation with interrupts disabled, as previously discussed. Figure 13 incorporates a D flip flop between the two banks of peripherals as a possible implementation of this technique. This flip flop is clocked when a valid interrupt acknowledge cycle is detected. This involves detection of /M1 without an /RD signal. /RD, however, does not go true until after the fall of the clock edge after /M1 goes true. If the next clock edge is used to indicate a valid cycle, approximately one-third of the cycle has already elapsed. This leaves about 400 ns which does not allow sufficient propagation delay for systems with more than three peripherals.
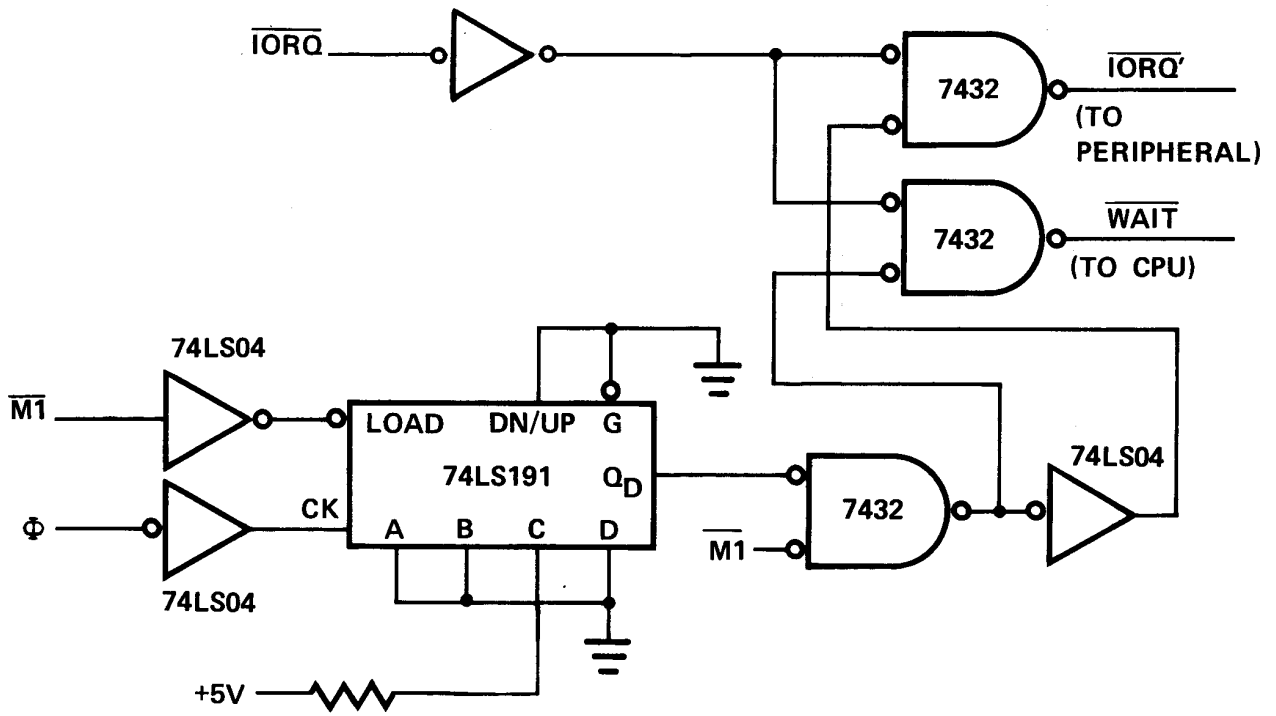
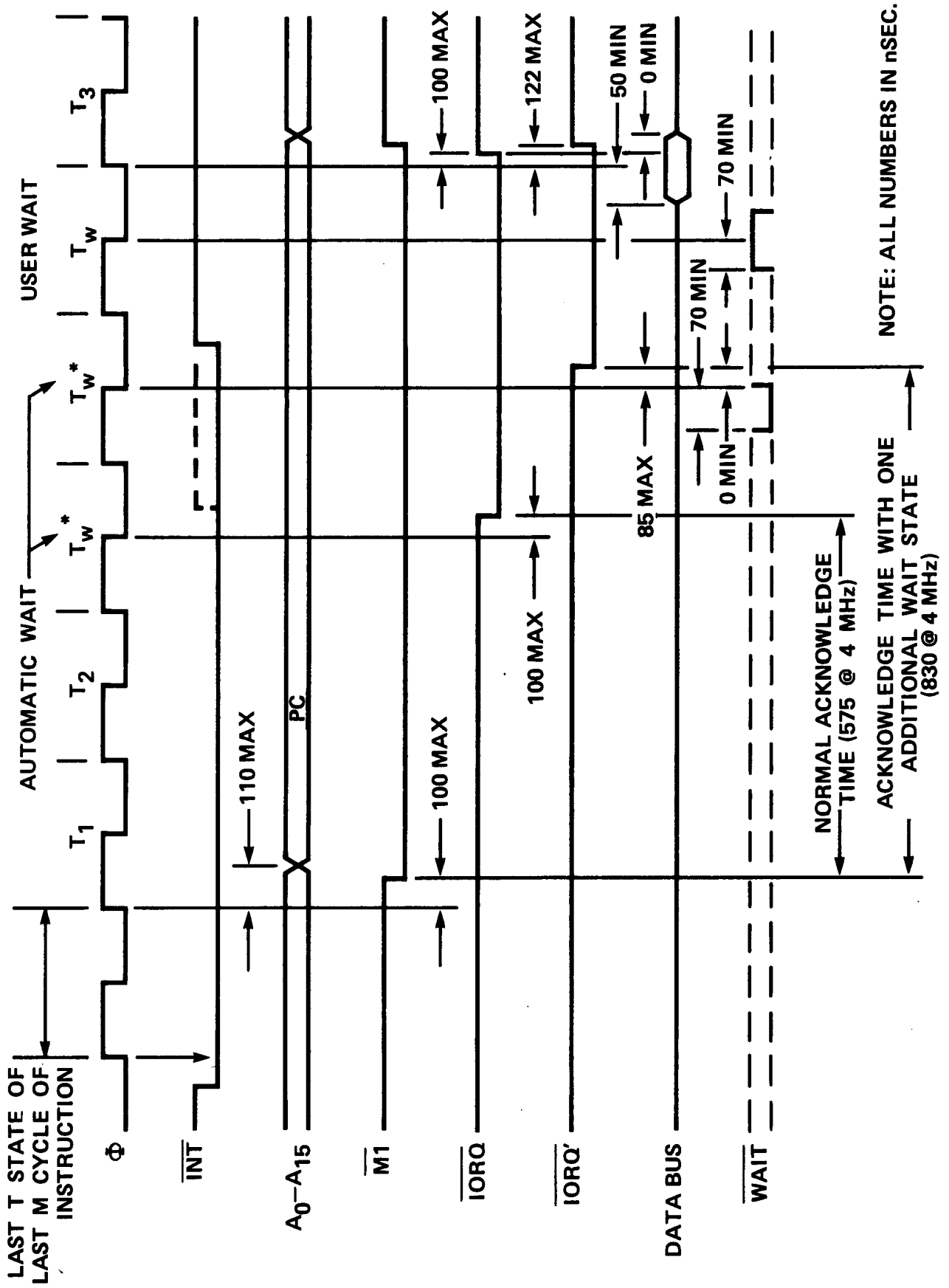*Figure 10: Extending Interrupt Acknowledge Time with Wait States*
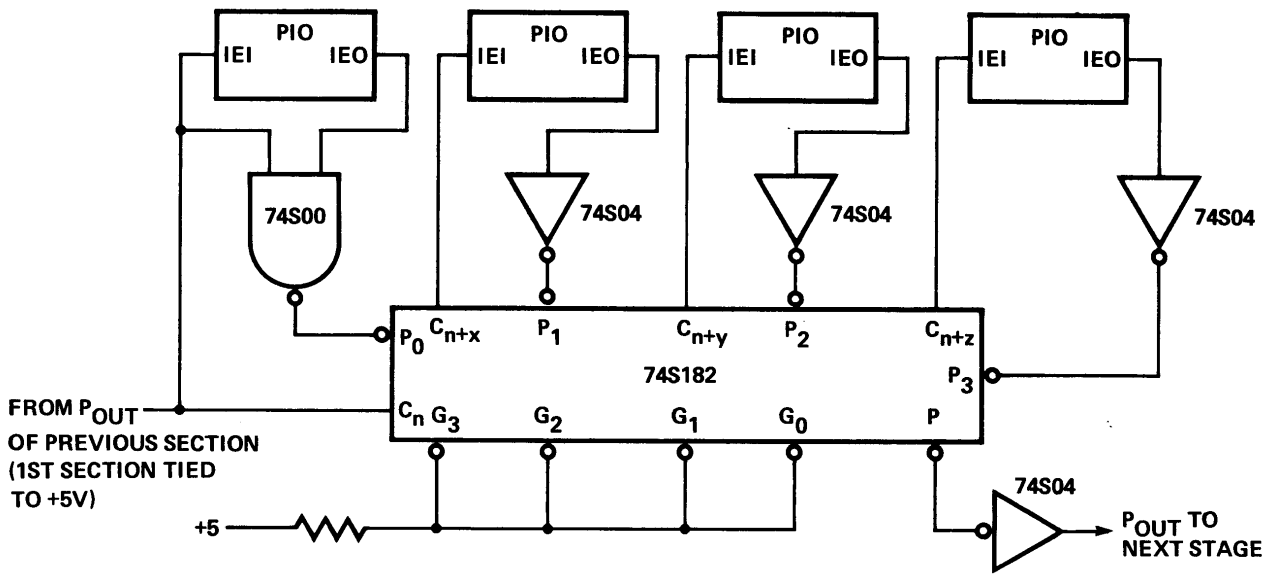
Figure 11: Interrupt Acknowledge Timing

20

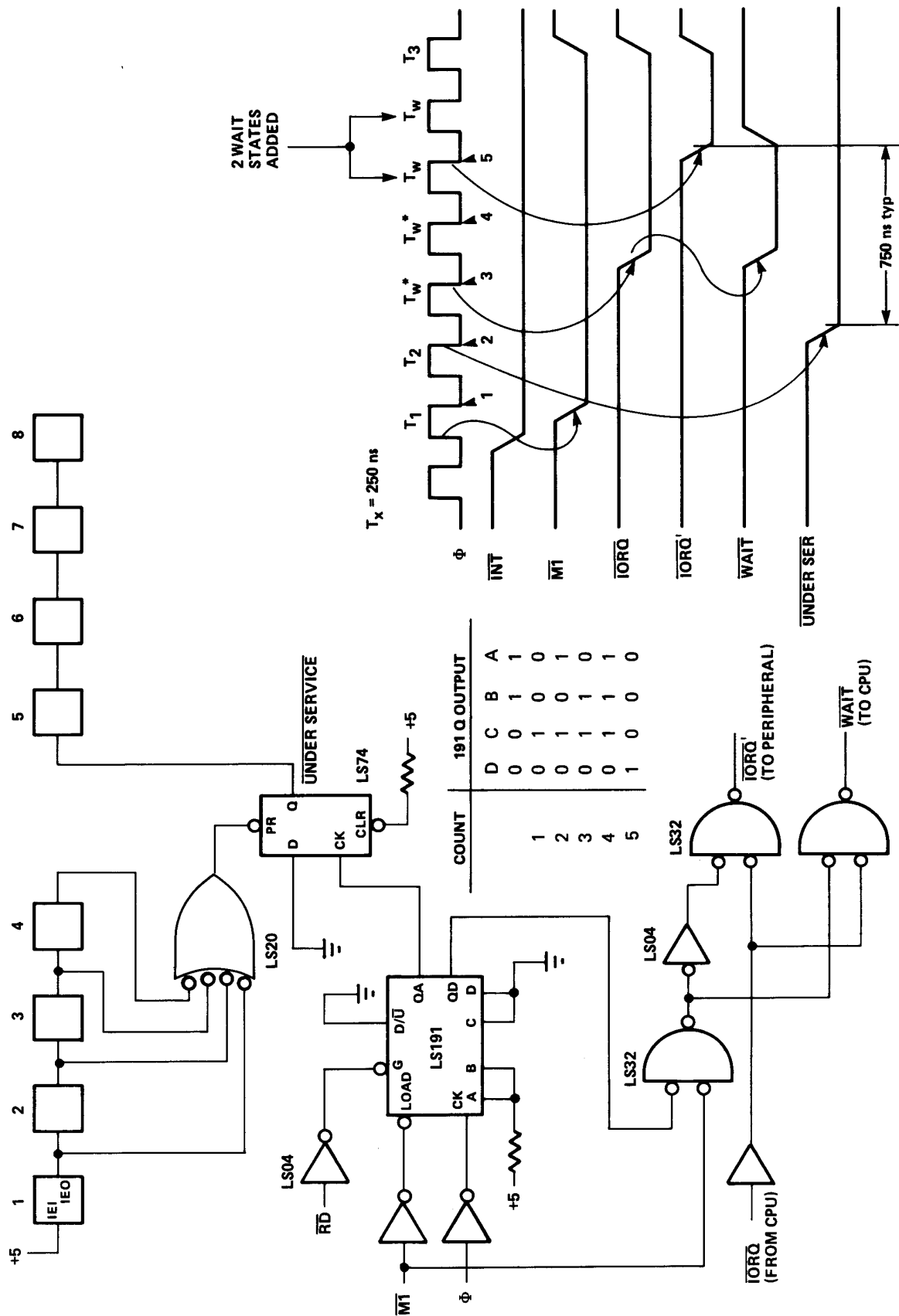*Figure 12: Reducing Ripple Time Using Look-ahead Logic*

21

Figure 13: CONFIGURATION FOR EIGHT Z80 PERIPHERALS INCORPORATING WAIT STATE, LOOK-AHEAD AND IEO INHIBIT LOGIC

22

ERRATA

Application Note: The Z80 Family Program Interrupt Structure, May 1978.

The logic in Figure 13, page 22 of the Interrupt AP Note has several flaws that need to be considered in a practical application. The LS191 enable input, to which /RD is connected, should change only when the clock input is high. However, if a Read Cycle is in progress, /RD will go active during the low transition of the clock. Also, when /M1 goes false at the end of a Read Cycle, the Qa output of the 191 is cleared, erroneously clocking the LS74 flip flop.

The following schematic using the LS161 synchronous counter represents the necessary logic for configuring a system with eight peripherals. Note, that no additional IC's are required and that the timing remains the same.
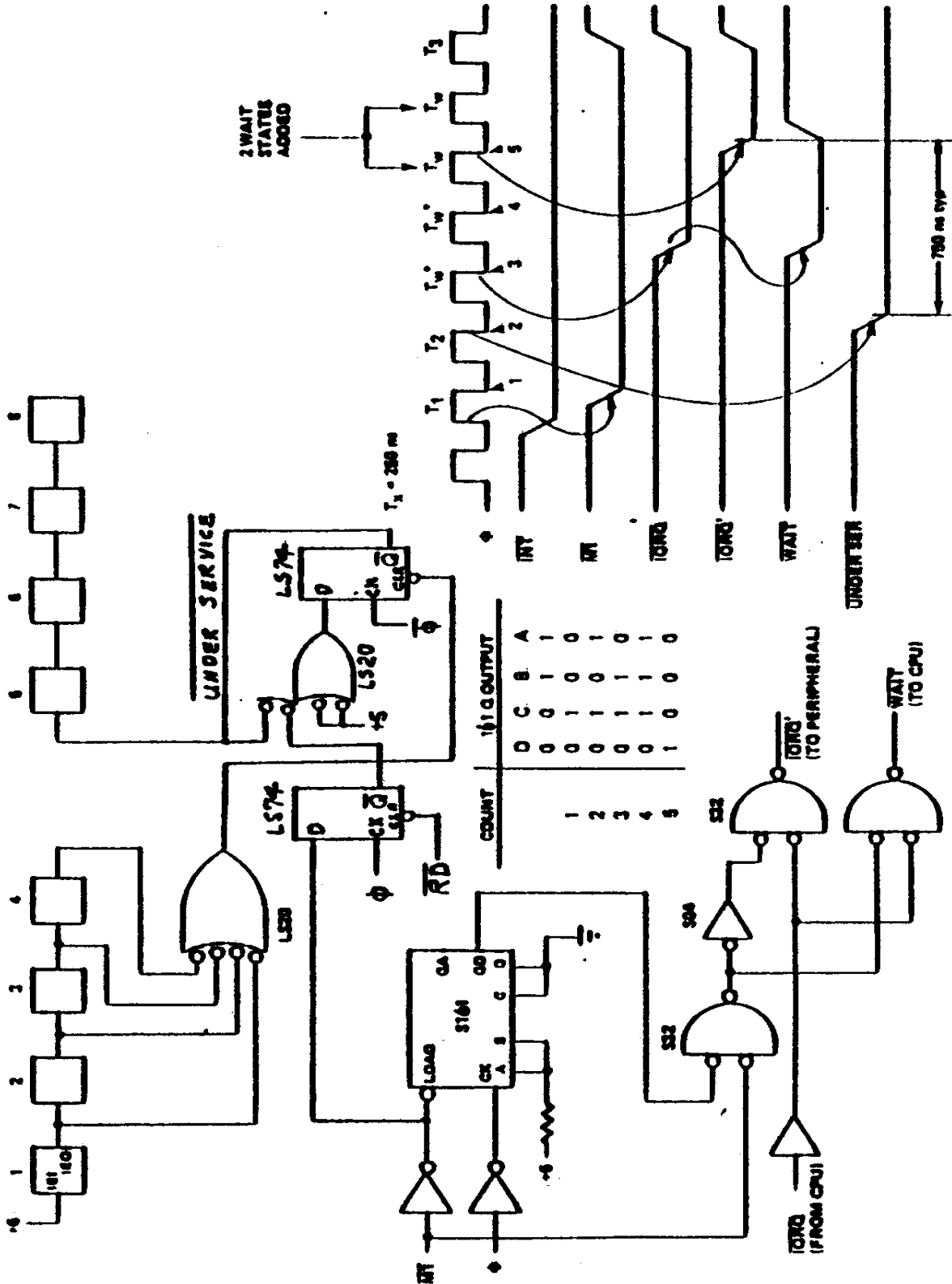
The approach taken in Figure 13 is to use a combination of look-ahead logic and insertion of wait states to configure a system of eight Z80 peripherals. The LS191 counter performs a dual function:

1. It clocks the D flip-flop to indicate the valid recognition of an interrupt acknowledge cycle. The counter is clocked twice before the B output goes high. During this time, the counter can be disabled if a /RD is detected indicating that the current cycle is an op code fetch cycle.

2. It allows insertion of wait states. The configuration in Figure 13 is set up for two additional wait states and the counter is clocked five times before the D output goes high to inhibit further wait states and also to send /IORQ true.

It should be noted, however, that the use of the logic in Figure 13 requires certain constraints. For proper chain operation, an EI instruction must be executed just before the RETI instruction to prevent the possibility of a higher priority device with a pending interrupt from blocking the RETI to a lower priority device. For example, in Figure 13, suppose device No. 5 interrupts and is granted priority. During the service routine for No. 5 with interrupts enabled, device No. 2 interrupts and is recognized, temporarily suspending service of No. 5. While in the service routine for No. 2, device No. 3 needs service. However, No. 3 does not have priority and its interrupt remains pending. If No. 2 finishes its routine without issuing an EI, then with No. 3 preventing the resetting of the D flop, No. 5 can never see its RETI.

9 HALT EXIT USING INTERRUPTS

Whenever a software halt instruction is recognized by the CPU it will enter the Halt state by executing NOP's until an interrupt is received. Each NOP consists of one /M1 cycle with four T states. The CPU will sample the state of the /NMI and /INT line on the rising edge of each T4 clock (see Figure 14). When an interrupt exists on either line the following cycle will be either a memory read operation (/NMI) or an interrupt acknowledge (/INT). Figure 14 shows a typical maskable interrupt causing the CPU to exit the halt state.

10   CONTROLLER DESING TECHNIQUES

This section describes device interface design techniques for compatibility with the vectored interrupt structure of the Z80 Family. It adds emphasis to the previous discussion and can be used to supply useful guidelines for controller design.

Figure 15 represents a conceptual configuration. When the device needs service, it will set the interrupt pending flip flop A (FFA) pull IEO low (via 7408 AND gate) and pull /INT low. /INT should stay low until acknowledged by the CPU. This will insure recognition and service of the device in a priority structure system.

An interrupt acknowledge (INTA) will reset FFA and set interrupt under service flip flop B (FFB) (IEO is maintained low via FFB). The internal peripheral signal (/HELP) discussed in previous sections can be equated to FFA and FFB as follows:

$$/HELP \ = \ /FFA \ + \ /FFB$$

INTA also enables the vector address to be placed on the data bus if IEI is high and FFB is set.

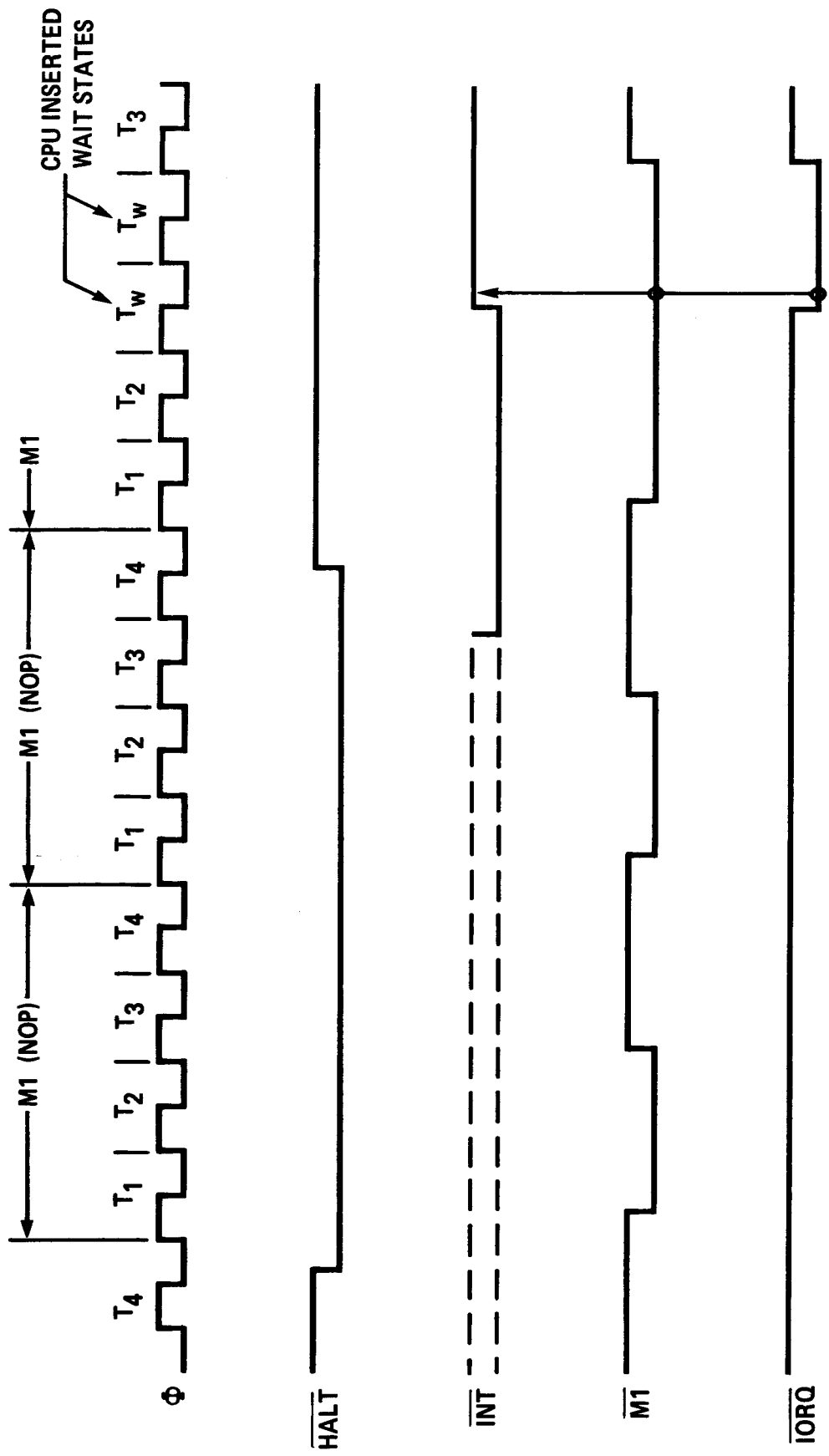Figure 14: Exit Halt State with Maskable Interrupt

IEI

+5V

NEED
SERVICE

7404

7401   0C   → INT

D Ⓐ Q   INTR. PENDING   C R

D Ⓑ Q   INTR. UNDER SERVICE   C R Q̄

7427   → IEO

"ED"

7408

7408   7408

7408

IEI

RESET

ĪORQ̄

M̄1

7427   ĪNTA   INTA

DECODED RETI

7410

R̄D

7432   M̄R̄D̄

7410

D Ⓒ Q   "ED" DECODER   C R Q̄

D Ⓓ Q   "ED" DECODER   C R

'ED'

7408

"ED"

DATA BUS   8   256x4 ROM

"4D"

"CB"
"DD"   +
"ED"
"FD"

D Ⓔ   SPECIAL OPCODE DECODER   C R Q̄

D Ⓖ Q   "4D" DECODER   C R

'4D'

RESET

VECTOR ADDRESS

DATA BUS

Figure 15: Z80: Interrupt Control Logic

The multi-byte instruction, RETI, is used to reset FFB and must be distinguished as a sequence of ED and 4D. Illegal recognition of ED and/or 4D with other op codes will cause improper resetting of FFB. For example, the following two instructions could cause the improper resetting:

```
SET   5,L          CB
                   ED
LD    C,L          4D
```

In the preceding sequence, an ED followed by a 4D is not an RETI instruction. The interrupt controller, therefore, must recognize the CB ED combination and prevent resetting of FFB. In Figure 13 FFE will be set blocking the AND gate if a CB is detected. Now if a ED follows, it is not allowed to be stored in FFC.

For an RETI instruction, the first byte (ED) is clocked into FFC at /M1 and /RD time. The second bye (4D) is clocked into FFG and ED is clocked into FFD for decoding by the 3 input NAND gate.

If a device needs service (FFA set) but has not received an INTA, FFC will be set when the first byte of an RETI is recognized. With FFC set, IEO will be forced high until the next /M1 cycle allowing downstream devices to decode RETI.