Z80 interrupt handling

Normally the Z80 executes instructions sequentially. However, there are situations when it is useful to interrupt the regular execution of code and do something else. This is accomplished by a mechanism called interrupts.

One use of interrupts is interfacing with the outside world to handle an external activity. For instance, an interrupt can occur when a key is pressed on the keyboard. The interrupt will make the Z80 stop its current execution and read the key and then continue the interrupted program. Other similar uses of interrupts are for peripherals such as disk drives, modems, or printers, where an interrupt can happen when the device has data ready to read or is ready to receive more data.

Interrupts can also occur at a fixed rate. For instance, an external real time clock chip could cause an interrupt once a second, and the Z80 could update an internal clock when the interrupt is received.

A third use of interrupts is for unusual or erroneous conditions. For instance, the Z80 could receive an interrupt if the power is failing, at which point it could shut down cleanly while power is still available.

Finally, if the CPU is halted with a HALT instruction, it will execute NOPs until an interrupt is received.

The Z80 provides two types of interrupts. The first interrupt is controlled by the $\overline{\text{INT}}$ pin and the second is controlled by the $\overline{\text{NMI}}$ pin (Non-Maskable Interrupt). The main difference between the two is that $\overline{\text{INT}}$ can be enabled and disabled while $\overline{\text{NMI}}$ cannot be disabled (i.e. masked). Each type of interrupt has its uses. There are many cases when you temporarily don't want to handle an interrupt. For instance in time-critical code it may be better to wait to handle the interrupt. Another example is in a “critical section” when modifying memory that is shared between two tasks, an interrupt that switches tasks may result in the second task accessing half-updated values, so delaying the interrupt preserves correctness. Finally, inside an interrupt handler interrupts are often blocked to avoid complications from handling

two interrupts at once. On the other hand, some interrupts need to be handled immediately or need to be handled even if a bug leaves interrupts turned off. For these interrupts, non-maskable interrupts make more sense.
When a regular interrupt occurs, the CPU reads a byte from the data bus, which allows an I/O device to provide data.

When a non-maskable interrupt occurs, the CPU jumps to address 0x0066, where code should be provided to handle the interrupt.

A maskable interrupt can be handled in one of three modes.

Mode 0 provides compatibility with interrupts as handled in the 8080 microprocessor. In this mode, external hardware can put any instruction on the data bus and the CPU will execute it. Typically a restart instruction is used, but it can be any single or multi-byte instruction, such as a jump to an instruction handling routine. This allows the Z80 to have different handlers for different interrupts, even though it has only a single interrupt pin. The M cycle is extended with two wait states in case the external circuitry needs additional time to select the interrupt. Mode 0 is the default interrupt state after a reset.

In Mode 1, the CPU executes a restart to fixed location 0x0038.

In Mode 2, the CPU can jump to one of 128 addresses through an interrupt address table. Specifically, when the interrupt happens, the CPU reads one byte from the data bus. This byte is combined with the contents of the I register to form a 16-bit address. The two bytes at this address are used as the starting address of the interrupt handler. This mode is powerful since it allows a large number of handlers to be selected based on a single data byte, and also allows multiple jump tables to be selected based on the I register.

Typically an external chip such as the Z80 peripheral chip (?) will handle and prioritize interrupts from multiple devices and generate the appropriate byte on the data bus. The RETI and RETN instructions have the same action inside the Z80, but the peripheral chip can tell which one was executed and will act accordingly.

The interrupt modes are selected with the IM 0, IM 1 or IM 2 instructions. These instructions operate as follows.

The hardware implementation of the three modes is as follows.

Two latches hold the IM state: a high latch, which latches bit 4 of the instruction, and a low latch, which latches bit 3 of the instruction.

IM 0 has opcode ED 46, so bits 4 and 3 are 00. IM 1 has opcode ED 56, so bits 4 and 3 are 10. IM 2 has opcode ED 5E, so bits 4 and 3 are 11. This latch is triggered by PLA 96 (`01xxx110`) during M1T1 (of the next instruction). The latches are cleared by the reset signal, ensuring the processor starts in mode 0.

When an interrupt comes in, these latches are decoded into multiple control signals

The interrupt control flip-flops

Maskable interrupts are controlled by two interrupt enable flip-flops called IFF1 and IFF2. The first flip-flip IFF1 is set by the Enable Interrupts instruction (EI) and cleared by the Disable Interrupts instruction (DI). The second flip-flop IFF2 is used for temporary storage of IFF1 when a non-maskable interrupt happens (user manual page 23). This ensures that a second interrupt won't happen until the programmer explicitly allows it with an EI instruction.

These flip-flops are found on the chip <here>. The EI and DI instructions control these flip-flops by <explanation>.

The state of the IFF2 register can be read with the LD A, I or LD A, R instructions, which copy the IFF2 register to the parity flag (which is a somewhat unexpected action). The interrupt handler code can then restore the original IFF1 state by enabling interrupts or not as appropriate. The hardware to implement this is <explanation>.

The IFF1 state is also restored from the IFF2 copy when a RETN (return from non-maskable interrupt) instruction is executed. The hardware to implement this is <explanation>.

Interrupts - including non-maskable interrupts - can only be processed between instructions, as it would cause chaos if execution switched while an interrupt were half-completed. In particular, an interrupt cannot happen between a prefix byte and the rest of the instruction; you wouldn't want the first instruction of the interrupt handler to unexpectedly be treated as a prefixed instruction! People studying undocumented instructions have found that an arbitrary number of prefixes can be strung together, and interrupts are blocked the whole time.

The hardware that forces interrupts to be handled between instructions is <here>. It works as follows. <explanation>. The signal X is asserted when an instruction is finished, but is blocked by a prefix. Internally, prefixes are processed as separate instructions, but this logic ensures that prefixes cannot be interrupted.

When maskable interrupts are re-enabled with the EI instruction, any pending interrupt request will be handled. However, the new interrupt will be disabled until the instruction after the EI has completed (which typically will be a return instruction to return from the interrupt). (User manual page 23.) The motivation is to ensure the previous interrupt handler has completed to avoid the stack filling up. The hardware to handle this is <explanation>.

The interrupt pins

The Z80's INT and NMI pins have slightly different logic. The INT pin is active-low, so an interrupt will be triggered whenever it is low. This allows a wired-OR configuration, where multiple devices are connected to the pin. If two devices trigger interrupts at once, the higher priority one (as decided by the external circuitry) will have its interrupt handled first (i.e. it will send its data on the data bus). When the interrupt handler for this device completes, interrupts will be re-enabled and another interrupt will be triggered since the INT pin will still be pulled low.

The circuitry for this interrupt pin is shown below. Like other input pins, the INT/ pin has a diode to protect the chip if the input drops too low. Two inverters buffer the input. The interrupt input then goes through two latches to synchronize it with the clock. The two latches form a master-slave flip flop that will latch the state of the interrupt pin on the rising clock edge.

The NMI pin is somewhat different - it is negative edge triggered, so a transition from high to low signals a non-maskable interrupt. Note that if it were implemented like the INT pin and triggered whenever low, a NMI would be re-triggered on every instruction and the interrupt handler wouldn't be able to run. This motivates the edge-triggered implementation of this pin.

The circuit for the NMI pin is shown below. A short pulse is triggered when NMI goes low because of the extra two inverters in one path. This is why the NMI pin is edge-sensitive. The NMI-generated pulse is used to clock a latch, which then feeds two latches set up as a master-slave flip flop, similar to the INT/ circuit. This flip flop stores the output of the NMI latch on the rising clock edge.

Then the interrupt circuitry gets more complex.

The interrupt page address register I is implemented as part of the register set, specifically the upper half of the I/R pair. In other words, it is implemented as a normal register, even though very few instructions can access it.

The incrementer (discussed <here>) has special logic to increment only the R register without incrementing the I register.

One of the motivations for two register sets in the Z80 is to allow the main program to use one register set while the interrupt handler uses the second register set. This makes interrupt handling faster since the interrupt code doesn't need to save registers to the stack (as on the 8080), which is slow.

The following circuit adds two wait states to the M cycle when handling an interrupt. This gives external circuitry time to determine which interrupt should take place. In particular, a common configuration for external interrupt hardware is a “daisy chain”, where each device is connected in sequence, and a higher-priority device will block a lower-priority device. If there are many devices, the interrupt signal must propagate through all the devices, which may be relatively slow. The wait states provide additional time for this to happen.