Undocumented instructions and features in the Z80

The Z80 has many byte codes that don't correspond to an instruction, at least an instruction that is described in the documentation. Many people are curious about what happens if the processor tries to execute one of these undocumented instructions.

One motivation for investigating undocumented instructions is to build a Z80 emulator that is as accurate as possible and will have the correct behavior even if running instructions that aren't documented.

Other people look at undocumented instructions to find useful instructions that can be used in code, to make code shorter. This is not recommended for a couple reasons. First, there is no guarantee that future versions of the chip will have the same behavior, so the code could fail on different chips. This could be an NMOS vs CMOS version of the chip, a Z80 from a different manufacturer, or even just a revision of the chip. Second, undocumented instructions can have actions that are undefined electrically, for instance outputting two signals to the bus at the same time. In this case, the behavior could vary from chip to chip based on minor manufacturing differences. The same chip could even have different behavior based on temperature or power supply voltage.

Why do undocumented instructions have actions? As was seen in the chapter on instruction decoding, the PLA matches against bit patterns in the opcode. If these bit patterns are matched, various control lines will be activated and actions will take place in the chip. An undocumented instruction can match the PLA's bit patterns by chance and thus will have some action. The remainder of this chapter will give details of how this happens.

Another reason, which I saw on the 8085, is that the manufacturer implemented instructions, but decided not to support them. I believe this was because after designing the chip they decided not to implement the instructions on later chips. Since they wanted the later chips to be backwards-compatible with the 8085, the easiest solution was to not document the unwanted instructions, even though the 8085 supported them.

A third reason is if the hardware has bugs and an instruction doesn't work properly, the manufacturer can leave the instruction out of the documentation until it is fixed. I believe this happened with the ROR instruction on the 6502, which was buggy at first. (There's a myth that the 6502 worked perfectly the first time it came back from fabrication, but this is untrue, as the ROR instruction was faulty.)

Why don't chips avoid undocumented instructions entirely? The main reason is it takes extra circuitry to force every instruction to have a documented action. In a chip such as the Z80, it wasn't worth the extra transistors. Modern chips, such as the x86, are designed to trigger an exception if an undocumented instruction is executed. This prevents code from depending on undocumented instructions, which ensures that later changes to the chip won't break old code.

One very detailed investigation of undocumented instructions is Sean Young's “The Undocumented Z80 Documented” (http://www.z80.info/zip/z80-documented.pdf). This chapter looks at the discussion of undocumented instructions in that document, and examines these instructions from the hardware perspective.

Flags

The two undocumented flag bits are bits 3 and bits 5, which are called XF and YF in various sources. These flag bits copy the values of bits 3 and 5 in the result. The reason can be seen from the flag hardware. The flags are copied between the register file and the active flag latches. Since there are not active flag latches for bits 3 and 5, whatever values are on the bus in bits 3 and 5 will remain there (due to capacitance?) and will get copied into the register file.

Power on defaults

The state of the chip after power-on is not fully documented. Investigations have found that AF and SP are always set to 0xFFFF, and the PC is set to

0x0000 while other registers are undefined. Interrupt mode is set to 0 and IFF1 and IFF2 are cleared. This can be seen in the hardware by looking at the reset signal. It clears … but doesn't affect...

The Z80 defines all the one-byte opcodes, so undocumented opcodes are all prefix instructions.

CB prefix

The opcodes with CB (bit) prefix are all defined except for 8 opcodes which are missing. From their position in the opcode table, you'd expect them to be left shift instructions, and that turns out to be the case. These instructions perform a left shift, putting a 1 into the lower bit.

Looking at the hardware shows why these opcodes have this effect. They trigger the shift PLA and the shift left control. The logic that selects the new lowest bit generates a 1 for these opcodes. (This presumably isn't a deliberate choice, but just what the logic happens to generate in this case.) The result is a 1 is shifted into the lower bit.

This instruction is likely not documented because it isn't particularly useful. And there's no obvious shift instruction to put in its place. For right shifts, it makes sense to distinguish between SRA (shift right arithmetic) and SRL (shift right logical), but SLA (shift left arithmetic) has the action you'd expect from a logical shift left.

An arithmetic shift right corresponds to dividing a number by two. For a twos-complement number, the top bit must be preserved, which the SRA does. For instance -10 shifted right becomes -5, or in binary 11110110 becomes 11111011; note that the top 1 bit remains 1, and the bottom 0 bit is shifted out. Going the other way (multiplying by 2), is just a plain shift left. As can be seen from -5 and -510, a 0 is put into the lowest bit and the top bit doesn't need any special treatment. Thus, an arithmetic shift right is different from a logical shift right, but an arithmetic shift left is the same as a logical shift left.

The DD (IX) prefix

The DD prefix does not have any effect on the PLA or early stages of instruction decoding. Instead, the DD prefix causes instructions that use the HL register to use the IX register instead. As a result, if the instruction does not use the HL register, the DD-prefixed opcode is an undocumented duplicate of the corresponding unprefixed opcode. (Check this is true, DD03 seems to be DE+1 → BC).

There are some instructions that use HL that aren't documented, but still work with a prefix. These instructions work as expected. In particular, operations that act on H or L will act on the corresponding half of the IX register.

This can be understood by looking at the hardware. The DD prefix affects selection of the register as follows:

Note that the EX DE, HL and EXX instructions only affect the register relabeling flip-flops, and don't actually move the contents of the HL register. Thus, these instructions don't use the IX register when prefixed.

The FD (IY) prefix

The undocumented instructions are the same as with DD, except use the IY register.

The ED prefix

The ED prefix triggers totally different PLA entries from regular opcodes. Many undocumented ED-prefixed opcodes don't match any PLA entries, so they are just NOP instructions. But some undocumented ED-prefixed opcodes do match the PLA entries from documented ED opcodes and do have an action. In many cases, the undocumented instruction duplicates the documented instruction, since the PLA matches and no other bits of the instruction matter. <Insert example of PLA matching>

A few cases where the instruction is different:

ED70 IN (C), IN F, (C)
ED71 OUT(C), 0

The cause for these actions is: (things not getting enabled)

The DDCB (IX bit) prefix
There are only 31 documented instructions with this prefix.
Because the DD prefix does not affect the PLA decoding, the remaining instructions are similar to the corresponding CB instructions, except first the ALU (or specified register?) is loaded indirect through IX. The result is then stored to the specified register and stored to memory indirect through IX. This happens for the instructions that change a register.

The reason this happens can be seen from the hardware. The IX indirection path is triggered for loading the ALU. But the store path uses the register specified in the opcode. Hardware...

Because the BIT operations don't store a value, they are the same as the documented IX instructions.

The FDCB prefix
This has the same behavior as DDCB, except with IY.

Prefix combinations

Prefixes can be combined in undocumented ways. For instance, if multiple DD (IX) and FD (IY) prefixes are used, only the last one has an effect. This can be understood by looking at the hardware. Because there is a latch to control IX vs IY, the latch will have the value of the last prefix.

Note that a prefix cannot follow a CB prefix. This is clear from the documentation, since after a CB prefix, all potential prefix bytes correspond to instructions. It's unclear from the documentation what happens if a prefix follows an ED (EXTD) prefix, but it turns out that a prefix cannot follow ED. In particular, the prefix will be treated as a NOP instruction.

The reason from the hardware is that the PLA rows XXX trigger a XXX prefix only if there is no prefix already. After a CB prefix, an opcode triggers the appropriate CD path. After an ED prefix, any potential prefix bytes don't match any PLA row (verify), so the byte is treated as a NOP.

Undocumented effects

BIT instructions
Documentation says S and P/V are unknown.
Explain the effects on flags, why is it like AND except carry is unchanged (for AND, carry is reset).

Explain why XF and YF (flag bits 3 and 5) are set from IX+d high byte.
Explain why XF and YF come from mystery register (WZ?) for BIT n, (HL).

Memory block instructions
LDI: Explain how XF and YF are set (based on A+n?)

CPI:

Explain XF and YF.

I/O block

According to the documentation, S, H and P/V flags are unknown.
Explain strange values, including XF and YF.
Explain why out different from in.

16-bit I/O ports
explain why 16 bits are put on address lines.

Explain why 16-bit additions affect flags. Do these use ALU or incrementer?

DAA instruction

Explain exactly what happens with the DAA instruction.

Interrupts (fold into interrupt chapter)

See document for the number of T-states taken to accept interrupts.

Mode 2: lower-bit is not forced to 0: explain this in the hardware

Details of IFF1 to IFF2 copying with nested NMI.

Are undocumented ED RET instructions RETI or RETN? The only difference between RETI and RETN is the opcode, which peripheral chips can detect RETI?

Timing and R register
Why does DDCB increase R by two, not three?
R register reset by power-on, reset?
R register increased by interrupt?

Memptr register http://www.grimware.org/lib/exe/fetch.php/documentations/devices/z80/z80.memptr.eng.txt

The WZ register pair is used internally and isn't visible to the programmer. However, investigators discovered that bits 11 and 13 of WZ are copied to the flag bits 3 and 5 by the BIT n, (HL) instruction. In addition, it was discovered that CPI increments WZ by 1 and CPD decrements WZ by 1. Using these two facts, investigators were able to laboriously determine the exact values of the WZ register after particular instructions, and reverse-engineered how the instructions affect WZ. With the hardware operation revealed, it is now possible to verify these reverse-engineered explanations and show why WZ is affected in these ways. Values get stored into WZ for two reasons. First as a genuine use of WZ as a temporary register. Second, because the WZ register doesn't need to be preserved, it often gets written unnecessarily. That is, the WZ register may be modified by a larger set of instructions than necessary, because it's simpler to implement the logic for the larger set than restrict it exactly.

For example:

```
LD A,(addr)
       WZ = addr + 1
```

The WZ register is used for temporary storage of the address as it is read in, during M2 and M3, which is expected. In M4T1, WZ is written to the address latch. In M4T3, the incremented value is stored back to WZ (for unknown reasons). This is how WZ ends up with addr+1, but it doesn't explain why.

LD (addr), A
As before, WZ is used for temporary storage, and in M4T3 the incremented value is stored back to WZ. But then in M1T1 (of the next instruction), when the A value gets written to memory, the W register also gets loaded from the bus (for unknown reasons). So W contains A, while Z still contains the incremented address.