How conditional operations work

The circuit for conditional branches is interesting.

The conditional branch instructions RET cc, JP cc, **, CALL cc, ** are designed with structured opcodes where bits 5 and 4 of the opcode select which flag is desired, and bit 3 of the opcode selects whether the flag should be 0 or 1. Specifically, if bit 3 is 0, the flag must be 0 to take the condition. Likewise, if bit 3 is 1, the flag must be 1 to take the condition.

One key part is a multiplexer that selects the desired flag. Specifically, it uses bits 4 and 5 of the instruction to select one of the bits of the register bus, selecting the appropriate bit of the flag register. Instruction bits 00 select regbus bit 6, which is the Z flag. Instruction bits 01 select regbus 0, which is the C flag. Instruction bits 10 select regbus 2, which is the P flag. Instruction bits 11 select regbus 7, which is the S flag.

These bit patterns for RET cc, JP cc, **, CALL cc, **. However, the JR NC, JR C, JR NZ, and JR Z don't fit this pattern because they had to be squeezed into opcodes that the 8080 didn't define, and the available opcodes didn't fit the pattern. Instead, there's a separate gate to clear bit 5 for the JR cc opcodes causing them to work off Z and C, even though their bit patterns match conditionals on P and S.

This is an interesting example of how the constraints of being backwards-compatible with the 8080 led to opcode choices that weren't optimal, but the hardware was able to work around this.

The next step in the instruction handling is to latch the flag bit at time M1T4l as the flags travel from the F register to the flag latches over the register bus. There's a separate conditional latch to hold the flag specified by the instruction. (This happens even if the instruction is not a conditional instruction.)

The flag value is then exclusive-or'd with bit 3 of the instruction to see if the flag value matches. The exclusive-or will be true if the flag value doesn't

matches the instruction bit, and false if it matches.

The PLA decoding is combined with timing to determine if and when to check the condition result. The specific combinations are RET cc in M1, JR cc in M2, or CALL cc or JP cc in M3. That is, if the condition is false, RET cc will terminate at the end of M1, JR cc will terminate at the end of M2, and CALL cc or JP cc will terminate at the end of M3. These timings are determined by the number of bytes in the instruction. JR has one additional byte of relative address, while CALL and JP have two additional bytes of address. It would be faster for the Z80 to skip those additional bytes if the condition is not taken, ending the instruction in M1, but there's no capability for the PC to skip bytes that way.

The result is that if there is a mismatch between the flag value and the desired condition, the end_of_instruction signal gets triggered when the instruction has been read, and the instruction terminates without taking the condition.

This early condition logic is used for other conditional instructions, but without using the flag logic. CPI/CPIR/CPD/CPDR and LDI/LDIR/LDD/LDDR trigger this logic in M3. Note that the non-repeating instructions end after M3 in any case, so the conditional check is redundant. Because these instructions use the 16-bit BC as the loop counter, it is decremented using the incrementer/decrementer, rather than the ALU. The incrementer provides a signal if the current value is 1, and this is used for the condition check. You might expect to check if BC = 0, since that is what terminates these instructions. But instead the Z80 checks if BC is 1 before the decrement. This optimization lets it get the condition result sooner, since it doesn't wait for the decrement to happen.

The third set of conditionals is used for DJNZ (in M2), and INI/INIR/IND/INDR/OUTI/OUTIR/OUTD/OUTDR as well as CPI, CPIR, CPD, CPDR (in M3). These instructions test the zero flag, and terminate the instruction early if zero flag is set. Note that the non-looping instructions terminate in M3 in any case, so the zero test is redundant even if they trigger it. The non-looping instruction will be ignored for the rest of this discussion.

The situation is very different for DJNZ and the looping IN/OUT operations, as opposed to the compare operations. DJNZ and the IN/OUT instructions decrement the 8-bit B register and loop until it is zero. Unlike the previous loop instructions that used the 16-bit BC register, the B register is decremented via the ALU. Thus, when B reaches zero, it will set the Z flag, causing the loop to terminate (for an IN/OUT instruction) or causing the branch to not be taken (for DJNZ).

For CPIR and CPDR, on the other hand, BC is decremented as discussed before. But the comparison will set the Z flag if there is a match. Thus, this circuit will cause the loop to terminate when a match is found. If BC hits zero first, the loop will terminate by the circuit discussed previously.

To summarize, this circuit causes DJNZ/INxR/OUTxR to terminate when B reaches zero, and causes CPxR to terminate if a comparison match is found. In both cases the Z flag is set for the termination condition, but the cause of Z being set is very different.

If the condition is successful, the action depends on the specific instruction.

Now let's look at a relative jump JR cc, *

M2T3l: PC(l) is loaded into ALU's op1 register
M3T1l: offset byte is loaded into ALU's op2 register, and sum (low bits) computed
M3T1h: sum (low bits) is latched in ALU's output register.
M3T2l: half carry is updated
M3T2h: sum (upper bits) computed
M3T3l: sum (upper bits) written to WZ(l)
M3T41: PC(h) is loaded into ALU's op1 register, 0 is loaded into op2 register. Sum (low bits) is computed.
M3T4h: Sum (low bits) is stored in ALU output latch
M3T5l: half carry is updated
M3T5h: Sum (high bits) is computed
M1T1l: Sum is written to WZ(h), WZ is written to incrementer latch (instead of PC)

M1T1h: Latched address is written to address pins for fetch.
M1T2l: incremented address is written to PC

The extra M3 cycle is used to do the address computation. Note that the M3 cycle doesn't do any memory operation, so it is a “fake” M3 cycle.

The key part of the conditional is that M3 doesn't happen if the conditional isn't satisfied. Instead the instruction is finished after M2. But skipping M3 isn't enough to stop the conditional branch. There's one small, but important difference that happens in the next M1.

The WZ register is used to store the sum as it is being computed. A key step is in M1T1l, where WZ is written to the incrementer latch instead of the PC. This is how the PC gets updated - note that the new address is used from WZ for the instruction fetch before the PC is updated.

The circuit to connect the two halves of the register file is fairly complex. Much of it suppresses this connection when it should not happen: during T1 while reading an instruction, during a refresh (M1T2/M1T3). Other parts specify when it may happen.

Specifically, the 824 gate keeps the two halves disconnected during M1T2h to M1T4l. During this time, the left half of the register file is in use for refresh.

The 544 gate branch makes sure the halves are disconnected while the PC is getting written back from the incrementer. This is relevant in a M cycle where a fetch is happening. Specifically gate 568 goes high for instructions of two or three bytes, while 469 goes high for 3-byte instructions. This ensures that the PC and latch are isolated during T1h/T2l of the appropriate M1, M2, and M3 cycles depending on the instruction length.

The next part of the circuit to look at is the 577 gate. Where this outputs a 0, the register halves will be connected, unless inhibited as above. The LAST_T/ entry ensures that the two halves will be connected (subject to the above) except potentially in Tnh, T1l.

The cases that disconnect the register halves in Tnh,T1l are:

578=0: This is M1 for a 2-byte instruction, M2 for a 3-byte instruction, or M3 for ldr/cpr/inr/otr. For a 2-byte instruction, the two halves will be disconnected at the end of M1/beginning of M2, and likewise for the other cases. This is the T state at which the PC gets copied to the latch to fetch the byte. (Disconnection when the latch gets copied back to the PC happens through the other circuit.) Note that because the control signal is delayed half a clock, it must be computed from LAST_T/ in order to take effect in T1l.

So what about M3 in LDR, etc? The purpose of this is these repeating instructions load the PC into the incrementer/decrementer in M4T1l in order to decrement it and stay at the same instruction. Thus, the PC half of the register file needs to be isolated.

1996=0 and 1997=0: The two register halves will be joined at the end of the last M cycle/ beginning of next instruction if there is a transfer of control flow, and otherwise the two halves are separated. That is, if the new PC value is in WZ this lets the value in the WZ get copied to the address latch in M1T1l to fetch from the new address. This happens for JR, RET, JP, CALL (unconditional and successful conditional). It also happens for DJNZ (successful), RST, JP (HL), RETI, RETN

The key part for DNJZ is that the two parts will not be connected in M1T1l if the condition is rejected, and otherwise they will be. Likewise, the PC will be read in M1T1l if the condition fails, while it will not be read if the condition succeeds. WZ is read in M1T1l in either case, but unless the two halves of the register set are connected, being read won't have any function.

The following shows the key steps in the handling of DJNZ, *, which does a relative jump.

M1T5l: B is loaded into ALU's op1 register, 0 is loaded into ALU's op2 register, and and B+~0 (low bits) is computed.

M1T5h: B+~0 (low bits) is latched in the ALU's output register.
M2T1l: half carry is updated
M2T1h: B+~0 (high bits) is computed.
M2T2l: B+~0 is written back to B

M2T3l: PC(l) is loaded into ALU's op1 register
M3T1l: offset byte is loaded into ALU's op2 register, and sum (low bits) computed
M3T1h: sum (low bits) is latched in ALU's output register.
M3T2l: half carry is updated
M3T2h: sum (upper bits) computed
M3T3l: sum (upper bits) written to WZ(l)
M3T41: PC(h) is loaded into ALU's op1 register, 0 is loaded into op2 register. Sum (low bits) is computed.
M3T4h: Sum (low bits) is stored in ALU output latch
M3T5l: half carry is updated
M3T5h: Sum (high bits) is computed
M1T1l: Sum is written to WZ(h), WZ is written to incrementer latch (instead of PC)
M1T1h: Latched address is written to address pins for fetch.
M1T2l: incremented address is written to PC

A lot of operations get crammed into the DJNZ processing. The ALU does the 8-bit B decrement, and the 16-bit address computation. Since the ALU does these operations 4 bits at a time, and it takes two half-clocks for each 4 bit sum, and an half-clock is required in between to update the carry flags, a lot of T states are required.

Most of the instruction processing is the same for DJNZ and JP cc. In particular, the address computation in M3 is the same for both instructions. Both instructions handle unsatisfied conditions in the same way, by ending the instruction after M2 and skipping M3.

The main difference between DJNZ and JP cc is the B register computation between M1T5l and M2T2l. This only adds one T state (T5) to M1. Much of the computation happens at the beginning of M2, when the ALU is not yet

occupied with address computations. Thus, even though the B register computation takes 3 T states, it only adds one T state to the overall instruction timing.

CALL

Now let's take a look at the CALL instruction. The instruction is fetched in M1, M2, and M3.
M3T1l: the low byte of the address goes into Z
M3T4l: the high byte of the address goes into W
M1T1l: WZ goes into the incrementer latch, so it becomes the new fetch address.
M1T2l: The PC is updated with the incremented address.

Meanwhile, on the SP side:
M3T3l: SP goes to decrementer latch
M4T1l: Decremented value written back to SP
M4T1h: Latched SP put on address pins
M4T2l: PC(h) put on data pins and written to memory
M4T3l: Value decremented again and written back to SP
M5T1h: Latched SP put on address pins
M5T2h: PC(l) put on data pins and written to memory

Note that M3 is 4 T states long instead of 3 in order to wait for the incrementer to be freed up after incrementing the PC, so it can be used to decrement the SP.

Also note that the PC doesn't get the destination address until M1T2l of the next instruction, after the fetch of the next instruction has started.

Conditional CALL

If a conditional CALL is taken, the sequence of steps is exactly the same as the unconditional CALL described above.

The condition status is available in M1T4, but it is not used until M3T1 where it is used to decide if the CALL ends with this M cycle or proceeds.

For an unsatisfied conditional CALL, the instruction ends after M3T3h . M3T4 does not take place, and neither do M4 or M5. The other key difference is that in M1T1h, the incrementer latch is loaded from the PC, rather than WZ, so execution takes place in the regular sequence.

The end-of-instruction logic that ends the instruction after M3 has been described. However, why does M3T4 get skipped?