M cycle control

For the most part, M cycles follow each other in sequence. The logic to end an M cycle was discussed earlier. This is the generation of the LAST_T_STATE/ control, which forces the start of the next M cycle.

There's one important control line to explain: LAST_M_CYCLE/. This control goes to 0 for the last M cycle of an instruction. In this case, the instruction ends when indicated by LAST_T_STATE/ and the next instruction starts at M1. (Unless there's an interrupt, which is only processed after the end of an instruction.)

The logic for LAST_M_CYCLE is fairly complex, since instructions can last from 1 to 5 M cycles, and there isn't any overriding logical system. Thus, the control is computed by random logic, which combines the PLA outputs as necessary. The PLA outputs normally correspond to similar instructions, so this provides some degree of compression, rather than handing every single instruction separately.

The main gate generating the LAST_M_CYCLE/ signal is NOR gate 232.
Any inputs of 1 will force the end of the instruction.
The first input is M1 AND SINGLE_M. If SINGLE_M is active, the instruction will end after M1.

SINGLE_M is generated by a NOR gate with 4 inputs. A 1 input will force the instruction to have multiple M states; if all inputs are 0, the instruction will end after M1.
Input 254: This forces multiple M states for pla37, pla63, pla64, pla26, pla47, pla48

Input 1690: This forces multiple M states for pla23, pla27, pla37, pla68, pla69, pla45, pla46, pla35, pla36, pla50, or an int/RST. This corresponds to an operation that has M4 but no M3.

Input 286: This forces multiple M states for pla52, pla53, pla54, pla55, pla58, pla49, pla50, pla59. This corresponds to operations using HL, which have

M3 and M4.

Input 1748: This forces multiple M states for pla20, pla21, pla26, pla43, pla47, pla48, pla18, pla19, pla24, pla7, pla10, pla15, pla29, pla30, pla31, pla38, pla42, or interrupts using mode 2.

Together, these PLA outputs cover all the instructions that go beyond M1, so what's left is the instructions that end with M1. Note that some PLA outputs such as pla47 are used in multiple combinations.

These PLA combinations are probably used because they were already generated for other purposes. It's not totally clear which instructions go into which category. 1748 appears to be instructions that have M3, but the other inputs don't map cleanly onto other M cycles.

The result of this circuit is an output indicating if an instruction terminates after M1. This is combined in gate 232 with M1 as one of the signals forcing LAST_M_CYCLE active.

The second input to gate 232 is BRANCH_NOT_TAKEN. This will cause the instruction to end in the current M cycle if it is a conditional instruction that was not taken, or a repeat instruction that is not repeated.

The final input to gate 232 is 1547. This handles the end of all the instructions that go beyond M1, so it has a lot of cases and the circuit is fairly complex. NOR gate 249 triggers LAST_M_CYCLE if it is 0, so any 1 input to NOR gate 249 will cause the instruction to end.

The inputs are as follows:

rst: a reset signal will cause the instruction to end after the current M cycle. Maybe this is to prevent peripherals from getting confused by resetting the bus in the middle of a M cycle.

The next input combines M2, 613, and 254. If both 1613 and 254 are 1, the instruction will end after M2.

1613 uses the 1690 PLA input discussed above, as well as a complex combination of signals including the mysterious 273. Signal 1613 is mostly 1, but it goes 0 for some instructions. 1613 must remain 1 for everything terminating in M2.
254 was also discussed above. Signal 254 must be activated for everything that ends in M2, but also for some other instructions that continue beyond M2.

Let's jump to M4.
An instruction will end in M4 if 266 is 1. This line is controlled by a bunch of PLAs:

266: pla15, pla18, pla19, pla20, pla21, pla27, pla36, pla37, pla38, pla50, pla59, pla52, pla58, pla72

266 is active for all the instructions that end in M4. However, there are a few exceptions: LDI/CPI/INI/OUTI/LDD/CPD/IND/OUTD activate 266 even though they end in M3. (The corresponding -R instructions will be ended in M3 by BRANCH_NOT_TAKEN when the repeat ends.)
The BIT test instructions active 266, even though they all end in M1, except for BIT n, (HL) which ends in M4. Probably it made the circuit simpler to activate on all the BIT instructions rather than specifically BIT n, (HL) here. Note that it doesn't matter if 266 is active for instructions ending sooner than M4, since they will never reach the M4 cycle, so the circuit here is redundant.

Moving back to M3, there are several inputs.
pla0 will terminate in M3. This is activated by ldx/cpx/inx/outx, which end in M3. (This is not activated by the repeating -R forms of these instructions, which go to M4 when the repeat.)

The next M3 case is 305 and (1630 or 288)
305 is zero for any of pla10, pla20, pla21, pla49, pla15, pla24, pla30, pla31, pla38, pla42, pla53, pla54, int_im2. Thus, all these instruction will not terminate in M3. (Except ini/oti/ind/otd, which were forced to terminate by pla0 as described above.) These instructions generally go to M4 or M5, but

it's unclear exactly what they have in common.

Except as blocked by 305, 1630 will force an instruction to end in M3. (As you can see, this circuit consists of exceptions on top of exceptions.)

The subcircuit computes 288 OR (266 NOR 288). This is strangely redundant, since it equals 288 OR NOT 266. I thought maybe the motivation for this was routing, perhaps it was easier have 288 cut across the other gate, but looking at the die, that's not the explanation. And wire 1630 isn't used anywhere else. So it remains a mystery why the circuit was built this way.

As discussed above, 266 is instructions ending in M4 (mostly), so this adds the instructions not ending in M4 (according to 266) to the ones generated by 288.
288: pla2, pla7, pla20, pla29, pla30, pla31, pla38, pla43, int_im2

To summarize the M3 path, instructions matching 288's PLA entries or ones not matching 266's PLA entries will end in M3, except the ones matching 305's PLA entries are filtered out. The underlying logic of this remains opaque.

For example, 288 is triggered by LD HL, ** which ends in M3.
288 is not triggered by DJNZ, which ends in M3, but 266 will add it.
288 is triggered by LD (**), HL, which ends in M5, but 305 subtracts it.

266 adds many instructions that end in M1 and M2, but this doesn't matter. It adds instructions ending in M5, but 305 doesn't subtract all of them. What about ADD HL, BC - this instruction looks like it should be forced to end in M3. 305 and 1630 are 1, so 249 should go low in M3. The trick is this instruction doesn't have a M3 - it goes directly from M1to M4, so the M3 circuit doesn't have a chance to fire.

The final input is M5. This forces an instruction to end after M5 if it hasn't ended already. This input is why there is no M6.

Putting this all together, each M cycle will run until LAST_T_STATE/ is active. Then the next M cycle will start unless LAST_M_CYCLE/ causes the instruction to end.

Another interesting thing is w148, which causes the instruction to skip to M4. One example is LD (HL), B, which has M1 and M4. OUT (*), A has M1, M2, and M4. PUSH rr has M1, M4, and M5.

This uses many of the same PLA entries that are used in computing LAST_M_CYCLE.

The first half of the NAND gate computes after which M cycle the skip to M4 happens. It will happen after M1 unless blocked by 253's PLAs, in which case it will happen after M2. 253 has several PLA lines that don't matter, but the relevant ones are OUT (*), A, IN A, (*), LD (HL), *.

The second half of the NAND gate determines which instructions skip to M4. It is set up as multiplexer on w286 (HL memory op). For a HL memory op, it is triggered unless the IX/IY prefix is active (since that uses M3 before M4). For a non-HL-memory op, it is triggered by w293 discussed above, which indicates a M4 op that skips M3.