THE AMSTRAD CPC CRTC COMPENDIUM V1.8

LOGON SYSTEM

This document is licensed under a CC BY-NC-ND 4.0 license
Attribution-Non Commercial-NoDerivatives 4.0 International
https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode

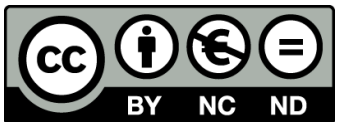


TABLE OF CONTENTS

TABLE OF CONTENTS 2
1 PREFACE 11
2 VERSION HISTORY 12
3 GENERAL 13
3.1 TERMINOLOGY 13
3.2 ACRONYMS 14
4 CRTC & CPC 15
4.1 GENERAL 15
4.2 CRTC TYPES 17
4.3 GENERAL VIEW OF THE REGISTERS 21
4.4 CRTC ACCESS 21
4.4.1 GENERAL 21
4.4.2 Z80A INSTRUCTIONS 23
4.4.3 ACCESS DELAYS 25
4.4.4 OUTs DISSECTION 26
5 OTHER CIRCUITS 30
5.1 ACCESS 30
5.2 CPC + / GX 4000 31
6 BUILDING A FRAME 32
6.1 GENERAL LOGIC 32
6.1.1 CHARACTER COUNTING 32
6.1.2 SYNCHRONIZATIONS 32
6.1.3 CHARACTER DISPLAY 32
6.1.4 VIDEO POINTER 33
6.1.5 DIAGRAMS 33
7 SYNCHRONIZATION 36
7.1 PRINCIPLES 36
7.2 VSYNC SYNCHRONIZATION 37
7.3 FAKE VSYNC 40
8 DISPLAY, Z80A & GATE ARRAY 42
8.1 INSTRUCTION LD(HL),reg8 (2 µsec) 42
8.2 INSTRUCTION LD (aaaa),HL (5 µsec) 42
8.3 INSTRUCTION PUSH reg16 (4 µsec) 43
9 GATE ARRAY 44
9.1 PIXELS 45
9.2 COLOURS 47

9.2.1 BORDER AND GRAPHIC MODE 2 47
9.2.2 SPEED OF PROCESSING 47
9.3 GRAPHIC MODE 49
9.3.1 GENERAL 49
9.3.2 CRTC 0, 1, 2 50
9.3.3 CRTC 3, 4 50
9.3.4 MODE SPLITTING 51
10 COUNTER : REGISTER R9 71
10.1 GENERAL 71
10.2 TIME LIMITS 72
10.3 COUNTING RULES 72
10.3.1 CRTC 0 73
10.3.2 CRTC 1 74
10.3.3 CRTC 2 75
10.3.4 CRTC 3, 4 75
11 COUNTER : REGISTER R5 78
11.1 GENERAL 78
11.2 COUNTING IN VERTICAL ADJUSTEMENT 79
11.2.1 GENERAL 79
11.2.2 CRTC 0 79
11.2.3 CRTC 1, 2 81
11.2.4 CRTC 1 81
11.2.5 CRTC 2 82
11.2.6 CRTC 3, 4 82
11.3 UPDATING R5 DURING AN ADJUSTMENT 82
11.3.1 CRTC's 0, 2 83
11.3.2 CRTC 1 83
11.3.3 CRTC's 3, 4 83
11.4 R5 UPDATE BEFORE AN ADJUSTMENT 84
11.4.1 CRTC's 1, 2, 3, 4 84
11.4.2 CRTC 0 84
11.5 VSYNC DURING VERTICAL ADJUSTMENT 84
11.5.1 CRTC's 0, 1, 2 84
11.5.2 CRTC 0 84
11.5.3 CRTC 2 84
11.5.4 CRTC's 3, 4 84
11.6 RUPTURE FOR DUMMIES (R.F.D.) ON CRTC 1 85
11.6.1 RFD AND FRAME PARITY 86

11.6.2 IVM ON/OFF 86
11.6.3 R.F.D. IN SUMMARY. 88
11.6.4 R.F.D. AND OTHERS CRTC 88
11.7 R6 AND VERTICAL ADJUSTMENT 88
11.8 ADJUSTMENT DURING INTERLACE 89
11.9 INTERLACE ADJUSTMENT LINE 89
12 COUNTING : REGISTER R4 90
12.1 GENERAL 90
12.2 CRTC 0 90
12.2.1 CASE STUDY : LINE-TO-LINE RUPTURE (R.L.A.L.) 91
12.3 CRTC 1 92
12.4 CRTC 2 93
12.4.1 LAST LINE CONCEPT 93
12.4.2 CASE STUDY: LINE-TO-LINE RUPTURE (R.L.A.L.) 94
12.5 CRTC 3, 4 97
13 COUNTING : REGISTER R0 98
13.1 GENERAL 98
13.2 CRTC 0 99
13.2.1 THE FIRST 3 MICROSECONDS 99
13.2.2 FREEZE OF VSYNC 101
13.2.3 FREEZE OF ADDITIONAL ADJUSTMENT LINE 101
13.2.4 FREEZE OF C9 101
13.2.5 CASE STUDY: R0=1 103
13.2.6 CASE STUDY: R0=0 104
13.2.7 CASE STUDY: VERTICAL RUPTURE LAST LINE (R.V.L.L.) 105
13.3 CRTC 1 107
13.3.1 CASE STUDY : VERTICAL INVISIBLE RUPTURE (R.V.I.) 108
13.4 CRTC 2 112
13.4.1 CASE STUDY: VERTICAL RUPTURE LAST LINE (R.V.L.L.) 113
13.5 CRTC 3, 4 115
13.6 R0 UPDATE 116
13.6.1 CRTC 0, 2 : CHRONOGRAM 116
13.6.2 CRTC 1 : CHRONOGRAM 116
13.6.3 CRTC 3, 4 : CHRONOGRAM 117
13.7 SPECIAL CASES 118
13.7.1 CRTC 1 118
13.7.2 CRTC 0 118
13.8 OFFSET ACCORDING TO C0 120

13.8.1 4 µsec FRAMES (R0=3) 121
13.8.2 2 µsec FRAMES (R0=1) 122
13.8.3 1 µsec FRAMES (R0=0) 123
14 SYNCHRONISATION : REGISTER R3 124
14.1 GENERAL 124
14.2 VSYNC LENGTH 125
14.3 HSYNC GATE ARRAY VERSUS CRTC 126
14.4 HSYNC AND FRAME POSITION 126
14.5 UPDATING R3 DURING HSYNC 128
14.5.1 CRTC 0, 2 129
14.5.2 CRTC 1 130
14.5.3 CRTC 3, 4 131
14.5.4 ZOOM ON R3.JIT 132
14.6 ABSENCE OF HSYNC 135
14.7 HSYNC START-UP 135
14.7.1 CRTC 0, 1, 2 135
14.7.2 CRTC 3, 4 136
14.8 HSYNC AND INTERRUPTIONS 136
14.9 HSYNC SCHEMATICS 137
15 SYNCHRONIZATION : REGISTER R2 139
15.1 GENERAL 139
15.2 HSYNC WHEN R2 IS PREDEFINED 141
15.2.1 CRTC's 0, 1, 2 141
15.2.2 CRTC's 3, 4 141
15.3 UPDATING R2 DURING HSYNC 142
15.3.1 GENERAL 142
15.3.2 INFINITE HSYNC 142
15.3.3 CRTC 0 143
15.3.4 CRTC's 1,2 144
15.3.5 CRTC's 3,4 145
15.4 VSYNC CONSIDERATION DURING HSYNC 146
15.4.1 GENERAL 146
15.4.2 CRTC 0, 1 146
15.4.3 CRTC 3, 4 147
15.4.4 CRTC 2 147
15.5 BORDER AND HSYNC 149
15.5.1 CRTC 0, 1, 3, 4 149
15.5.2 CRTC 2 149

15.6 CRTC 2 AND HSYNC 149
15.7 THE RIGHT MOMENT 150
15.7.1 GO FROM R2=46 TO R2=50 ON 64 µSEC LINES 151
15.7.2 GO FROM R2=50 TO R2=46 ON LINES OF 64 µsSEC 151
16 SYNCHRONIZATION : REGISTER R7 152
16.1 GENERAL 152
16.2 VSYNC-CRTC VERSUS VSYNC-GATE ARRAY 153
16.2.1 VSYNC AREA DISPLAY 153
16.2.2 MONITOR C-SYNC SIGNAL 156
16.2.3 C-SYNC ALGORITHM 159
16.2.4 TOLERANCES 159
16.2.5 CRTC AND GATE ARRAY INTERACTIONS 160
16.3 VSYNC PROTECTION 161
16.4 CONDITIONS TO CONSIDER 162
16.4.1 CRTC 0 162
16.4.2 CRTC 1 163
16.4.3 CRTC 2 163
16.4.4 CRTC 3, 4 164
16.5 DELAYED VSYNC 165
16.5.1 CRTC 0 165
16.5.2 CRTC 1 165
16.5.3 CRTC 2 165
16.5.4 CRTC' s 3 & 4 165
16.6 LIMITLESS VSYNC ! 166
16.7 THE RIGHT MOMENT 168
17 DISPLAY : REGISTER R1 169
17.1 GENERAL 169
17.2 DISPLAYS ACCORDING TO R1 171
17.2.1 DISPLAY WITH R1 <= R0 171
17.2.2 DISPLAY WITH R1 > R0 172
17.3 DYNAMIC R1 UPDATE 173
17.4 VMA'/VMA WHEN C4=0 176
17.4.1 CRTC 0, 3, 4 176
17.4.2 CRTC 1 176
17.4.3 CRTC 2 177
17.5 ACKNOWLEDGMENT R1=0 179
17.5.1 CRTC 0, 1, 2 179
17.5.2 CRTC 3, 4 179

17.6 INTERLINE BORDER 179
17.6.1 R1=R0 AND C0=R0 179
17.6.2 R1>R0 AND C0=R0 180
18 DISPLAY : REGISTER R6 182
18.1 GENERAL 182
18.2 BORDER R6 DEADLINES AND PRIORITIES 182
18.2.1 GENERAL 182
18.2.2 CRTC 0, 2 182
18.2.3 CRTC 1 183
18.2.4 CRTC 3, 4 183
18.3 R6 CONFLICTS 183
18.3.1 GENERAL 183
18.3.2 CRTC 0, 2 184
18.3.3 CRTC 1 185
18.3.4 CRTC 3, 4 185
19 DISPLAY : REGISTER R8 186
19.1 GENERAL 186
19.2 FUNCTIONS « SKEW-DISPTMG » 187
19.2.1 BORDER ON 187
19.2.2 BORDER OFF 187
19.2.3 BORDER DELAY +1 / +2 187
19.2.4 NO CONDITION C0=R1 188
19.2.5 DISINTEGRATION OF THE BORDER ON CRTC 0 190
19.3 INTERLACE FUNCTIONS 192
19.3.1 GENERAL 192
19.3.2 THE TWO INTERLACE MODES 193
19.3.3 LIMITATIONS 195
19.3.4 UNLOVED FEATURE 195
19.4 VERTICAL INTERLACE PROGRAMMING 197
19.4.1 CRTC 0 197
19.4.2 CRTC 1 197
19.4.3 CRTC 2 197
19.4.4 CRTC 3 & 4 198
19.5 PARITY 199
19.5.1 GENERAL 199
19.5.2 CRTC 0 199
19.5.3 CRTC 1 202
19.5.4 CRTC 2 206

19.5.5 CRTC 3 & 4 207
19.6 ADDITIONAL INTERLACE LINE 210
19.6.1 CRTC 0 210
19.6.2 CRTC 1 210
19.6.3 CRTC 2 210
19.6.4 CRTC'S 3 & 4 211
19.7 MID-VSYNC 212
19.7.1 GENERALITIES 212
19.7.2 CRTC 0, 1, 2 212
19.7.3 CRTC's 3, 4 212
19.8 COUNTING IN INTERLACE VIDEOMODE 213
19.8.1 CRTC 0 213
19.8.2 CRTC 1 219
19.8.3 CRTC 2 220
19.8.4 CRTC 3, 4 228
20 VIDEO POINTER:REGISTERS R12/R13 234
20.1 GENERAL 234
20.2 VIDEO POINTER CALCULATION 234
20.3 UPDATE CONDITIONS 235
20.3.1 CRTC 0 235
20.3.2 CRTC 1 235
20.3.3 CRTC 2 236
20.3.4 CRTC 3 & 4 236
20.4 DEADLINES 236
20.5 OVERSCAN-BITS 237
21 READ REGISTERS. 238
21.1 GENERAL 238
21.2 READING THE CONTENTS OF THE REGISTERS 238
21.2.1 CRTC 0 238
21.2.2 CRTC 1, 2 238
21.2.3 CRTC 3, 4 239
21.3 READING STATUS 239
21.3.1 GENERAL 239
21.3.2 CRTC 0, 2 240
21.3.3 CRTC 1 240
21.3.4 CRTC 3, 4 241
21.4 DUMMY REGISTER 242
22 FULLSCREEN & CENTERING 243

22.1 FOREWORD 243
22.2 HORIZONTAL FULLSCREEN 244
22.3 VERTICAL FULLSCREEN 244
23 TIPS AND TRICKS 245
23.1 R12/R13 UPDATES 245
23.2 COMMON USE OF REGISTER(S) 246
23.3 WAITING VSYNC 247
23.4 ZERO VALUE 247
23.5 OUTI/OUTD AND STATUS REGISTER 247
23.6 SELF-MODIFIED CODE 247
23.7 ITERATIONS AND UNROLLED CODE 250
23.8 UNCONDITIONAL BRANCHING 252
23.9 PAGES TREATMENTS 253
23.10 BULK TIPS 256
23.10.1 LOOPS 256
23.10.2 CALCULATION OF THE VIDEO POINTER 257
23.10.3 FLAG SETTINGS 260
23.10.4 RATHER THAN... 260
24 A BRIEF HISTORY OF FIXED TIME 261
24.1 INTRODUCTION 261
24.2 METHODS 262
24.3 FIXED TIME AND INTERRUPTIONS 265
24.4 COMPENSATORY TOOLS 267
24.5 FREE FIXED TIME! 270
24.6 FROM FREE TIME TO FIXED TIME... 270
24.7 WASTING TIME... 271
25 DURATION OF INSTR. ON THE CPC 272
26 INTERRUPTS 274
26.1 GENERAL 274
26.2 MANAGEMENT OF R52 COUNTER 274
26.3 TRIGGER CONDITIONS 275
26.3.1 TRIGGER WHEN R52 LOOPS TO 0 275
26.3.2 TRIGGERING ON VSYNC 275
26.3.3 Z80A AND INTERRUPTIONS 275
26.4 INTERRUPT MODE 1 276
26.5 INTERRUPT MODE 2 276
26.6 CRTC & INTERRUPTS 277
26.6.1 GENERAL 277

26.6.2 CRTC 0, 1, 2 278
26.6.3 CRTC 0, 1 278
26.6.4 CRTC 2 278
26.6.5 CRTC 3, 4 278
26.6.6 PERSPECTIVE 279
26.7 THREESOME 279
26.7.1 R52 IN TIME 279
26.7.2 RELIABILITY OF INTERRUPTIONS 280
27 CRTC IDENTIFICATION 283
27.1.1 VIA C4 AND/OR C9 OVERFLOW 283
27.1.2 VIA VSYNC MANAGEMENT DURING HSYNC 283
27.1.3 VIA CONSIDERATION OF VSYNC ACTIVATION 283
27.1.4 VIA VSYNC LENGTH 283
27.1.5 VIA HSYNC LENGTH 283
27.1.6 VIA THE BORDER, VISUALY 284
27.1.7 VIA THE INTERLACE MODE 284
27.1.8 VIA STATUS REGISTER &BE00 284
27.1.9 VIA READ REGISTER &BF00 284
27.1.10 VIA R10/R11 STATUS REGISTERS 284
28 CPC IDENTIFICATION 285
28.1 IDENTIFICATION METHODS 285
28.1.1 ENABLING EXTENDED FEATURES 285
28.1.2 BUG PPI PORT C 285
28.1.3 BUG PPI PORT B 285

1 PREFACE

The objective of this document is to provide detailed information on the operation of the various CRTC 6845 circuits implemented in the CPC's created by AMSTRAD. The CRTC is a controller circuit capable of providing an interface between microcomputers and cathode ray screens that manage video scanning.

The document also discusses the operation of some circuits associated with the CRTC's, especially the GATE ARRAY.

Game or demo programmers still working on these machines designed in the 1980s and the 1990s may find the information presented here useful.
It can also serve as a reference for anyone wishing to create an emulator by adapting code ad hoc for specific programs.

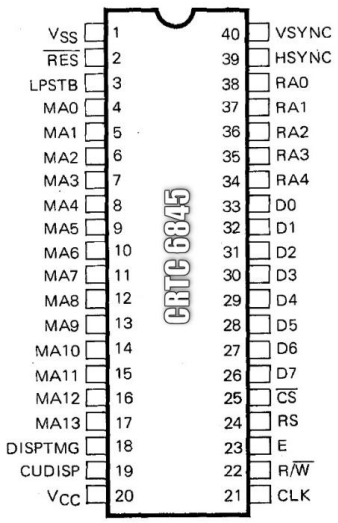
Finally, this document may provide useful information to users of other machines equipped with similar CRTC's. However, it must be noted that the timing between the processor, the CRTC and the associated video circuit of another machine can significantly modify the behaviours described here. Furthermore, the document only describes the CRTC functions used on the CPC; "text" mode and the "cursor" are therefore not discussed here.

Part of the information presented in this document has been checked using a benchmarking program called SHAKER. It was tested for each CRTC on several machines. A document for each CRTC exists containing the photos from the results of each test for versions 1.8 and 1.9 of SHAKER. From version 2.2 onwards, a portal is available to facilitate access, comparisons and updates for these results: https://shaker.logonsystem.eu

Truth takes up little space, but error occupies an infinite number of places.
This document is subject to revision and change.

Serge Querné
Longshot / Logon System
serge.querne@logonsystem.eu

Acknowledgements



David MANUEL (*DManu78*), author of the excellent Amspirit emulator, for our discussions and constructive tests to improve this document.

Claire DUPAS (*Cheshirecat*) for her tests, feedback on HSYNCs and her ideas that are sometimes not so far-fetched ;-)

Stéphane SIKORA (*Siko*) for the help provided on some GATE ARRAY tests and the creation of the SHAKERLAND portal.

Marc MAC PHAIDIN (*Lmimmfn*), for the English correction of this document (up to version 1.3).

Arnaud STORQ (*NoRecess*) for sending SHAKER results when my CPC's were out of reach for the very first version of this document.

2 VERSION HISTORY

Version	Date	Update
1.0	01/01/2021	Document created. Proofreading: Arnaud Storq, Claire Dupas, Sébastien Broudin
1.1	15/12/2021	Diagrams reimported. Corrections made in §4.3, 4.5, 7.2. Statuses grouped together in § 21 with addition of read registers and definition of CRTC 3 & 4 statuses. CPI/CPD/CPIR/CPDR instruction duration corrected.
1.2	01/01/2022	Major update. Precisions, corrections and clarifications on some chapters (especially on CRTC 0: VSync, C9 freeze on R0=0, Interlace Modes). New chapters on the GATE ARRAY (horizontal splitting mode, hsync, interrupts). Description of the RFD. Vertical scroll at 1/64th of a pixel. Proofreading: David Manuel, Claire Dupas, Olivier Antoine
1.3	14/09/2022	Addition of details 4.4.2, 7.2, 10.3.1, 11.1, 11.2.1, 11.2.3, 11.2.4 ,11.5.2, 11.6, 12.2, 14.2, 15.2, 19.4.1, 19.4.3, 19.6.4 Corrections 7.2, 10.3.3, 11.4, 11.7 12.4, 11.3, 12.4.2 ,13.4, 13.4.1, 15.2.3, 16.4.4, 19.5.3, 19.6.2, 19.6.3, 20.2, 20.5, 25.1 New chapters 16.6, 25.7.2
1.4	11/01/2023	Corrections chapter 25 (thanks to Salvaren). Added clarification in chapters 11.7, 16.1, 19.4.1, 19.5.1, 19.5.5, 19.6.4, 27.1.9, 11.3.1, 18.2.3. Important additions in chapter 23. Creation of new chapters 11.8, 24.
1.5	03/03/2023	Correction on CRTC statutes 3 & 4 (21.3.4). Major developments on the chapters relating to synchronization. Thanks to Frédéric Mantegazza for his help and advice on signal analysis
1.6	25/07/2023	Corrections diagrams 14.5.4.4 and 10.3. Case R1=0 CRTC 2, chapters 13.4, 17.4.3 and 20.3.3. Correction chapters 13.3, 13.6, 13.7, 13.7.1.2, on R0 updates via OUTC/OUTI + RFD 2. Corrections and clarifications on Last Line for Crtc 0.
1.7	10/10/2023	Additions in 11.1, 11.3.2 and 17.4.2, Correction schema 16.2.2. 19/02/2024: correction Tw in 4.4.4 (thanks to SerErris)
1.8	09/05/2024	Correction IFF state 42.3.3, Crtc 2 Meow Bug 17.4.3, Rev chapters 11.1 and 13.2 (counting bugs & R5 logic). Correction 4.4.4 (Tks to SerErris)

This document is licensed under a CC BY-NC-ND 4.0 license
Attribution-Non Commercial-NoDerivatives 4.0 International

3 GENERAL

3.1TERMINOLOGY

The CRTC is a circuit that spends its time counting and comparing.

It therefore mainly consists of counters, the limit value of which is generally defined in the programmable registers of the circuit.

With the goal of constructing frames (image for CRTC) made of several vertical characters, in turn made of several vertical lines, which in turn is made of several horizontal characters, certain terms are associated with these registers and counters.

We therefore speak of "*horizontal total number of characters*", "*vertical total number of characters*" and "*Maximum Raster*" to define the value of certain registers, with some differences on occasion depending on the technical documentation under consideration. Some emulators use "unofficial" acronyms to define the names of associated counters: HCC, VTAC, VLC, VCC, ... to name a few.

These terms are no longer appropriate when working at a level other than that intended for the construction of a "standard" frame.

Indeed, at a certain level, the logic of “horizontal” and “vertical” counting practically disappears. It then becomes difficult to speak of “characters” without tripping on one’s shoelaces, as depending on the circuit's programming, a vertical character can also be a horizontal character.

This notion exists at most in relation to the synchronization logic of the constructed frame.

Since it is possible to reduce the size of a line to 1 µsec, to reduce the size of a vertical character to 1 line and to reduce the size of a frame to 1 character, the horizontal and vertical qualifiers somewhat lose their meaning.

For these reasons, I will generally denote the CRTC registers in this document as **Rn** and the associated counters as **Cn**. "Characters" will denote the words processed by the GATE ARRAY from the address provided by the CRTC. I have named the CRTC’s two internal pointers VMA and VMA’.

I invite the authors of emulators based on this document to adopt these notations.
Here are equivalents for some of the terms that I have encountered:

Names identified in certain emulators	Counters
HCC (Horizontal Char Counter)	C0
VLC (Vertical Line Counter)	C9
VCC (Vertical Character Counter)	C4
VSC (Vertical Sync Counter)	C3h
HSC (Horizontal Sync Counter)	C3l
VTAC (Vertical Total Adjust Counter)	C5 or C9 (on CRTC’s 0,3,4)
VMA (byte pointer)	VMA or VMA’ (word pointer)

3.2ACRONYMS

ASIC : Application Specific Integrated Circuit : Integrated circuit which groups on the same chip all the functions necessary for a specific application. The GATE ARRAY is an ASIC.

CRTC : Cathode Ray Tube Controller: Circuit used to interface a computer with a cathode ray monitor capable of handling raster lines.

C-HSYNC: Composite signal generated by the GATE ARRAY for horizontal synchronization of the image on the monitor.

C-VSYNC: Composite signal generated by the GATE ARRAY for vertical synchronization of the image on the monitor.

HBL : Horizontal BLank Line: Refers to the period during which the monitor's electron beam, which has reached the right side of the monitor, turns off the beam to return to the left of the screen.

HSYNC : Horizontal SYNC: Refers to the signal emitted by the CRTC which allows the monitor (via the C-HSYNC signal emitted by the GATE ARRAY) to synchronize the frame horizontally on the sreen.

IVM : Interlace Video Mode.

IS : Interlace Sync.

PIXEL-M2 : Definition of a pixel displayed in graphics mode 2 (0.0625 µsec).

RFD : Rupture for Dummies on CRTC 1.

RLAL : Rupture Ligne à Ligne (from French). Definition of a rupture where the video address can be updated on each each raster line of the frame.

RV : Rupture Verticale (from French): Definition of a rupture whose axis is not "horizontal", and where horizontal zones can be created.

RVI : Rupture Verticale Invisible (from French) : Refers to vertical ruptures occurring during the Hsync and not visible.

RVLL : Rupture Verticale Last Line (from Frenglish): Refers to vertical ruptures using the C4 « last line » processing on CRTC’s 0 and 2.

VBL : Vertical BLank Line : Refers to the period during which the monitor electron beam, which has reached the bottom of the screen, deactivates the beam to return to the top of the screen.

VMA : Video Memory Address : CRTC pointer to the words addressed in memory and supplied to the GATE ARRAY for display.

VSYNC : Vertical SYNC : Signal emitted by the CRTC which allows the monitor (via the C-VSYNC signal emited by the GATE ARRAY) to synchronize the frame vertically on the screen.

4 CRTC & CPC...

4.1GENERAL

On the AMSTRAD CPC, the duration of a CRTC character is 1 µsec..

This CRTC character represents 2 bytes in memory.

The memory pointer is **communicated by the CRTC to the GATE ARRAY, which will always read the "central ram" of 64k**. The GATE ARRAY cannot read the data in ROM or in the additional RAM of 6128s (or memory extensions).

The CRTC is programmed so that the image thus created is supported by a monitor.
A language shortcut is to use the term NOP instead of µsec (microsecond) because it is the time taken by this instruction in Z80A on CPC.

On CPC, the Z80A instructions are aligned by rounding the M cycles of an instruction to a multiple of 4 T cycles. This alignment is linked to the need for the GATE ARRAY to interrupt the Z80A to access the ram whose address is provided by the CRTC. This operation slows down some instructions relative to the clock frequency. Achieving accurate code requires knowing the exact time taken by each instruction. See Chapter 25, page 272, for details of these durations for each instruction.

A bit of history

Old Cathode ray monitors and televisions lined up their display frequency with the electricity distribution network of the country of marketing.

In Europe in particular, televisions built to support SECAM and PAL formats operate with a horizontal frequency of 15625 Hz, and a vertical frequency of 50 Hz.

Note that the American (and Japanese) format was the NTSC (National Television System Committee) with a horizontal frequency of 15734 Hz and vertical of 60 Hz.

The horizontal frequency of 15625 Hz comes from the use of 64 µs delay line, invented for the design of the Secam format by the engineer Henri de France. As a reminder, 64µs = 0.000064 seconds (just enough time to have a strong coffee).

The CRTC follows this logic and processes about 1 million characters per second because its frequency is 1 MHz.

Without diabolical intervention on the position of the HSYNC, it is in principle programmed to generate complete lines of 64 µsec.

To get closer to the desired frequency, the CRTC is generally programmed to display 312 lines of 64µs, or exactly 19968 µsec (0.019968 seconds).

This frequency being linked to a clock, we can see drifts between 2 machines after a certain time.

The standard format initialized by the CPC ROM in a European country is :

- Lines of 64 horizontal "CRTC" characters of 1 µs each (composed of 40 characters displayed and 24 not displayed).
- 312 vertical lines, subdivided into vertical characters of 8 lines, or 39 characters.
- Formula : ((R4+1) x (R9+1)) + R5
- Of these 39 character lines, 25 are displayed.
- CRTC table ROM address: **&5C5**

The standard format initialized by the CPC ROM in the United States is:

- Lines of 64 horizontal "CRTC" characters of 1 µs each (composed of 40 characters displayed and 24 not displayed).
- 262 vertical lines, sub divided into vertical characters of 8 lines, or 32 characters + 6 lines of adjustment
- Formula : ((R4+1) x (R9+1)) + R5
- Of these 32.75 character lines, 25 are displayed.
- CRTC table ROM address: **&5D5**

Note 1 : It is the bit 4 of port B of the PPI (LK4) that allows you to test which table to use.

Note 2 : Initializing the CRTC is one of the first things the CPC's LOW ROM does when the machine is turned on and reset. Register initialization begins 64 µsec after the Z80A reads the first ROM instruction. Registers are updated from register 15 to register 0.

The CRTC has registers to manage a cursor and read data sent by an "optical pen". Cursor registers are not used on CPC, which does not handle a hardware cursor, usually provided when a text mode is handled.

However, they are a point of interest, because actions on other registers in or out of a synchronization period, with small values, could lead to consequences on other registers. Otherwise, it is always possible to store a value there, such as the type of CRTC.

The operation of these registers is not covered (at the moment) in this document.

4.2 CRTC TYPES

AMSTRAD had the brilliant idea of using CRTC 6845 circuits manufactured by different manufacturers in its machines. They even designed ASIC's that could emulate its operation.

A bit of history

While Amstrad wanted to attack the US market with the CPC 6128, a problem was identified in ROM with registry value 5 (vertical adjustment).

This value was set at 6 in accordance with the desired frequency of 60Hz in the USA (262 lines), but (wrongly) ignored the interrupt system managed by the GATE ARRAY. Indeed, on a European CPC, interruptions start 2 lines (we will consider that we have 1 HSYNC per line) after the occurrence of the VSYNC signal by the CRTC.

These interruptions have a period of 52 "HSYNC lines", which gives exactly 6 periods during the 312 lines (See Chapter 26.6 on INTERRUPTS). To stall 5 periods of 52 lines, it would have been necessary to program 260 lines and not 262 as was done, and therefore that R5 is programmed with 4 instead of 6.

These 2 more lines cause the interruption to arrive on not the same line on which the CRTC reports the start of VSYNC, BUT before it.

That is, on a US CPC, a program whose main code tests the VSYNC wait via the PPI can be interrupted during this wait. If the interruption lasts too long, when it returns, the VSYNC waiting loop has missed the signal (the bit has returned to 0), and a risk of "deadlock" exists.

This could lead to compatibility issues for programs produced in Europe. AMSTRAD then decided to remedy the problem by ....changing the 60 Hz table to ROM....
No... I'm kidding... too simple.

To avoid this "deadlock" without modifying the ROM, Amstrad engineers thought they could "limit the problem" by increasing the duration of the VSYNC. This was perfectly possible using the CRTC's Registry 3, which had been programmed with 8 lines ..... in the ROM.

It is likely that they thought that there were CRTC models without the function used to set the number of VSYNC lines (these models set the number of lines at 16) and thus decided that the American CPCs would only be equipped with CRTC's 1 and 2, without this function.

If AMSTRAD engineers were aware of the existence of these differences, it can be assumed that the first CRTC's used were type 0s, since this function is used and programmed by the ROM.[this was confirmed when Roland Perry showed the 464 prototype in September 2023 at the RetroAlicant meeting, in Spain].

From there to be able to say that without a bug related to a 4 instead of a 6 in the ROM, there would be only one type of CRTC used in all CPC's, it is to ignore the commercial considerations in the component market compared to the success of the machine in Europe.

Several companies have created different versions of the circuit, implementing additional functions, such as programming the number of lines of the VSYNC that I have just mentioned.

It can be deduced, however, that the designers of the BASIC ROM:

- originally worked with a CRTC with the ability to program the number of lines for VSYNC.

Beyond the functional and technical differences documented in circuit manufacturers' guides (called "datasheets" with 2 "e"'s), these CRTC's tend to behave differently when starting to modify registers:

- multiple times during a frame.
- during or outside HSYNC/VSYNC periods.
- with a value of 0, which is a special case for managing multiple registers.

These differences impact the compatibility of programs, especially when the address of the video pointer is updated.
This technique is still commonly called "*Rupture*" because... it is simpler and more generic than to say "*Offset Split Screen*".

Differences in counter management (which can overflow in a few situations) usually lead to a horizontal (R2) and/or vertical (R7) synchronization defect.

If a Z80A code "waits" for the VSYNC signal or uses interrupts (which depend on HSYNC), the mess is accentuated.

A bit of history

To my knowledge, the first program to have carried out a CRTC test was the game "Get Dexter" ("Crafton & Xunk" in French), written by Rémi Herbulot and Michel RHO in 1986. In this game, the image update takes place through horizontal scrolling, which uses register 2 (positioning the HSYNC on another character).

This method, on a MOTOROLA CRTC, causes a loss of vertical synchronisation when VSYNC occurs during a HSYNC (Ghost Vsync). In other words, in this situation, the CRTC falsely believes it is generating a VSYNC signal for the monitor.

I guess Rémi Herbulot had to have access, at Ere Informatique game company, to a CPC with a HITACHI CRTC (like mine at the time) and another with a MOTOROLA CRTC. Having made this observation, he created a test based on the reading in &BF00 of register 12, which makes it possible to distinguish the CRTC HITACHI from the CRTC MOTOROLA.

And so, he managed a display with and without scrolling on the screen.

That is why there is no scrolling in this game on CPC's equipped with a UMC CRTC, while this CRTC does allow it without issue, because as with the MOTOROLA CRTC, its register 12 is not readable.

A bit of history (yet !)

When the first "rupture" techniques began to be used massively in demos, the differences between CRTC soon began to be a problem. And especially when independent programmers (often high school or university students) began to form the "demo" scene and their demos began to circulate in a less discretionary way. Initially, these first demos were mostly introductions for cracked games that circulated in schoolyards.
The first demo programmers had not yet built up a "network" (the demoscene) and usually had only one machine on hand. The first intros and demos circulated in a small and very regional circle. The "network" consisted, in the 80s, of postal exchanges, Minitel's (a French ancestor of the internet, like Prestel in the UK) and ruinous telephone exchanges, with very little or no relations with other countries.

Communication with people from other regions first went through the consultation of classified ads in the few computer magazines of the time, because it was difficult to find contact details in the introduction preceding a cracked game. Programmers could hardly see the result produced by their code on other machines, and the inertia was enormous. It was difficult to adapt the code via postal exchanges, without having the machine (or even wanting to do it, simply). The problems encountered could range from image synchronization-loss to outright crashing of the machine.

It was difficult to be categorical about the actual origin of these problems, and especially about their extent. Nevertheless, with the grouping of the demomakers and the enlargement of the demoscene, it became possible to compare the code on different machines.

The presence of some electronics engineers in the ranks of the demomakers made it possible to identify the culprits... (Mr. SUGAR is still at large).

Some "universal" rules resulting from empirical approaches have been described in secret documentation which did not remain so for long ("You should not do this operation here in order for it to work everywhere", for example).

It should be noted that some techniques are still currently considered impossible to port from one CRTC to another (until this document...).

CRTC 2 (MOTOROLA) quickly proved to be the problematic one for one of the most widely used techniques, which is to place R4=R9=0.

The code needed to ensure CRTC compatibility can be much more complex than just knowing when to modify register 9 or 4, we'll see...

In order to exploit the rules allowing compatibility, it was first necessary to identify the different CRTC's.

Many methods exist today. See Chapters 27, page 283, and 28, page 285.

The numbering has remained fixed to this day from the order where I discovered these circuits with the help of some members of the group of demomakers "Logon System".

We discovered the CRTC emulated "Pre ASIC" after the release of the AMSTRAD PLUS, and that's why its number is greater than 3, although it came out earlier chronologically.

Type	Brand	Model
0	HITACHI	HD6845S(P)
0	U.M.C. (United Microcircuits Corporation)	UM6845
1-A	U.M.C. (United Microcircuits Corporation)	UM6845R
1-B	U.M.C. (United Microcircuits Corporation)	UM6845R
2	MOTOROLA	MC6845(P)
3	AMSTRAD	ASIC 40489
4	AMSTRAD	ASIC 40226

Note 1 : CRTC's 3 and 4 are CRTC's emulated by ASIC's, but are nevertheless CRTC! Without enabling the complementary functions of ASIC 40489, the behaviour of these two CRTC's cannot be differentiated at present.

Note 2 : The "Pre-ASIC" (CRTC 4) was most certainly designed with the AMSTRAD PLUS in mind, as its C9 counter is intended to be interrupted on any line. This is why positioning R9 at 0 can be done on the last line of a character (CRTC 0 compatibility) or on the first line of a character (CRTC 1 compatibility).

Note 3 : It cannot be excluded even at present, that some series of CPC's may be equipped with different CRTC models, available from the 3 manufacturers. (Please let me know if you discover another model, and you are sure that it was not your little sister who replaced it for a laugh). Additionally, HITACHI's CRTC HD6845S, which is identified as type 0, behaves exactly like UMC's CRTC UM6845. Probably a business deal between the firms where it is just the manufacturer marking that has changed... The UMC documentation specifies this in its comparison table with other circuits. There is currently no test that allows the distinction of these two circuits, but it may be possible via the "Interlace" mode or the special management of C0 by the HD6845S.

Note 4 : Programming R3 with &8x is a very bad habit, since CRTC's 1 and 2 also exist in European machines, and they do not respect this value...

Note 5 : Scheduling a specific update to a CRTC register in an interrupt routine that does not interrupt perfectly synchronized code is a very bad idea. (And may suggest (wrongly) a CRTC difference).

Note 6 : The difference observed between two CRTC's UM6845R may not be linked to the CRTC. Nevertheless, there is a noticeable difference. (See Chapter 0)

4.3 GENERAL VIEW OF THE REGISTERS

			CRTC 0									CRTC 1, 2									CRTC 3, 4								
Register	**Definition**	**Unit**	**r/w**	7	6	5	4	3	2	1	0	**r/w**	7	6	5	4	3	2	1	0	**r/w**	7	6	5	4	3	2	1	0
R0	Horizontal total character number	Char	w									w									w								
R1	Horizontal displayed character number	Char	w									w									w								
R2	Position of horizontal sync. pulse	Char	w									w									w								
R3	Pulse width of horizontal sync. pulse	Function	w	v	v	v	v	h	h	h	h	w					h	h	h	h	w	v	v	v	v	h	h	h	h
R4	Vertical total character number	Char Row	w									w									w								
R5	Total raster adjust	Scan Line	w									w									w								
R6	Vertical displayed character number	Char Row	w									w									w								
R7	Position of vertical sync. pulse	Char Row	w									w									w								
R8	Interlace Mode and Skew	Function	w	c	c	d	d			i	i	w							i	i	w	c	c	d	d			i	i
R9	Max Scan Line Address	Scan Line	w									w									w								
R10	Cursor start	Scan Line	w		b	p						w		b	p						r	s	1	s	s	s	s	s
R11	Cursor end	Scan Line	w									w									r	s	0	s	1	s	s	s	s
R12	Display start address (High)	Pointer	r/w									w									r/w								
R13	Display start address (Low)	Pointer	r/w									w									r/w								
R14	Cursor address (High)	Pointer	r/w									r/w									r/w								
R15	Cursor address (Low)	Pointer	r/w									r/w									r/w								
R16	Light Pen (High)	Pointer	r									r									r								
R17	Light Pen (Low)	Pointer	r									r									r								
Access ports to the CRTC on CPC			**r/w**	7	6	5	4	3	2	1	0	**r/w**	7	6	5	4	3	2	1	0	**r/w**	7	6	5	4	3	2	1	0
&BC00	Register selection	Number	w									w									w								
&BD00	Register write	Value	w									w									w								
&BE00	Register status	Function										r		L	b						r								
&BF00	Register read	Value	r									r									r								

See Chapters 21, 27.1.8 and 27.1.9 for more information on the content of the status register and reading this register according to the different CRTC's.

4.4 CRTC ACCESS

4.4.1 GENERAL

Access to I/O with a Z80A usually requires the use of specific instructions.

These instructions (OUT, OUTI, INI, IND...) are in principle intended to use devices whose addresses are defined on the least significant 8 bits of the 16-bit address bus.

The 16-bit address bus is specified in the **BC** register, but some instructions (OUTD, OUTI, INI, IND, ...) also use **B** as counter.

From a computer design perspective, it is not advisable to place the address of devices that can use these instructions on the most significant byte of the 16-bits address bus.

This wise advice from Mr. ZILOG was not listened to by Mr. SUGAR.

Also access to devices on CPC mainly goes through **A8..A15 bits of the address bus** (FDC 765 being an exception in part).

Device selection bits should therefore be set to B.

Welcome to the CPC!

Examples		
Selection register 12		
	BASIC	OUT &BC00,12
	Z80A	LD BC,&BC00+12:OUT(C),C
Sending valeur &30 in register 12 (previously selected)		
	BASIC	OUT &BD00,&30
	Z80A	LD BC,&BD30:OUT (C),C
Reading register 12 (previously selected / updated)		
	BASIC	PRINT INP(&BF00)
	Z80A	LD BC,&BF00 : IN A,(C)
Reading status register		
	BASIC	PRINT INP(&BE00)
	Z80A	LD BC,&BE00 : IN A,(C)

The use of register B prevents the use of interesting instructions to send or read a series of successive values on a port, such as OTIR, OTDR, INIR, INDR.

Indeed, these instructions use B as the counter of the number of values to be read in a table and decrement this counter until it reaches the value 0.

Digressive Remark :

These repetitive instructions can be used experimentally, to process more than one value, on a device whose selection bits do not participate in the counter (most significant bits), such as the GATE ARRAY. It is possible to issue one of these instructions and savagely interrupt it by positioning it judiciously before an interruption occurs. This interruption will nevertheless have to "forget" the return address placed on the stack before re-authorizing other interruptions. This interesting aspect is however very limited (and especially playful) and can activate other devices depending on the number of values read in the table.

However, it is still possible to use

- OUTI or OUTD instructions to send one by one the data from a table (pointed by HL) to the CRTC.
- INI or IND instructions to read one by one the data from a table (pointed by HL) of the CRTC.

The advantage is that these instructions are "fast" according to the number of operations performed.

The use of the OUTI/OUTD statement is possible by first incrementing B between each instruction for OUTI/OUTD because B is decremented before accessing the port. For the CRTC, which uses bits 0 and 1 of B as an index, this implies that B is pre-loaded with the port address + 1 for the OUTI/OUTD instructions to work.

For reading instructions, the addressed device is defined by BC before B is decremented, so it must contain the normal address of the port before reading.

Example in Z80A :

```
                LD BC,&BC02         ; Hsync position register selection
                OUT (C),C
                LD HL,TABHSYNC      ; Pointer to Hsync position table
                LD B,&BD+1          ; IO Address sending data+ 1 : &BE
                OUTI                ; OUTI decrease B by 1, send TABHSYNC[0] on the &BD IO Addr
                                    ; and then increase the pointer by 1
                INC B               ; B is set again to &BE
                OUTI                ; OUTI decrease B by 1,  send TABHSYNC[1] on the &BD IO Addr
                                    ; and then increase the pointer by 1
TABHSYNC        DB 50, 10
```

4.4.2 Z80A INSTRUCTIONS

Instructions in Z80A for processing Input/Output :

INSTR.	DURATION	DESCRIPTION
OUT (C),r8	4 µsec	r8=[A or B or C or D or E or H or L] Writing to devices defined in BC the value contained in the r8 register.
OUT (n),A	3 µsec	The input-output address is defined by the couple An, and the data sent to the device is A. This double constraint drastically limits the number of values available to one device without causing a collateral effect on another device (selection + value).
OUT (C),0	4 µsec	Writing value 0 to the device(s) defined in BC. An interesting instruction for a demomaker! But not only that...
OUTI	5 µsec	Decrement B, then read the value pointed by HL, incrementing HL and sending the value read on the port sent by BC.
OUTD	5 µsec	Decrement B, then read the value pointed by HL, decrementing HL and sending the value read to the port sent by BC
IN r8,(C)	4 µsec	r8=[A or B or C or D or E or H or L] Read in the r8 register the value sent by the device defined in BC. If several devices are selected, there is no doubt that a game of musical chairs will take place...
IN A,(n)	3 µsec	The input-output address is defined by the couple An, and the data read will modify A.
IN (C)	4 µsec	It is an "unofficial" instruction. The value present on the data bus is read and its evaluation affects F.
INI	5 µsec	Reading the value on the port addressed by BC and writing this value to the address pointed by HL, HL decrement, B decrement
IND	5 µsec	Reading the value on the port addressed by BC and writing this value to the address pointed by HL, HL decrement, B decrement

When an I/O takes place different signals of the Z80A can be activated.

The MREQ signal is set high and IORQ low. The RD/WR pins are basically used to indicate whether the Z80A should read or write data.

The GATE ARRAY is write-only, and the RD pin is in the inactive state, which implies that a read on this circuit is not considered. At best, a high impedance state available on the data bus is recovered.

However, the CRTCs are not connected to the Z80A's RD and WR pins, so there is no detection of the I/O direction. Consequently, if a **read instruction** is used on a **write register of the CRTC**, then **a data is sent to the CRTC**.

With an **IN A,(C)** instruction, the data on the data bus is sent to the CRTC.

But with the **IN A,(n)** instruction, it is possible to send the contents of A to the An port, just as it can be done with the OUT (n),A instruction. It is therefore possible to send a value to the CRTC while modifying A. However, it would be risky to trust the returned value.

Example in Z80A:

```
LD A,%00011001   ; Bits 0 and 1 select the Write port of the CRTC
IN A,(#FF)       ; Sends the value of A to the last selected CRTC register
```

4.4.3 ACCESS DELAYS

The following table indicates for some I/O write instructions the timing of the register update in the CRTC and for some I/O read instructions the timing when the value from the circuit is updated in the Z80A register or RAM.

INSTRUCTIONS	DURATION	I/O CONSIDERATION	
		CRTC 0, 1, 2	CRTC 3, 4
OUT (C),r8	4 µsec	**3rd µsec**	**4th µsec**
OUT (C),0	4 µsec	**3rd µsec**	**4th µsec**
OUT (n),A	3 µsec	3rd µsec	3rd µsec
OUTI	5 µsec	5th µsec	5th µsec
OUTD	5 µsec	5th µsec	5th µsec
IN r8,(C)	4 µsec	4th µsec	4th µsec
INI	5 µsec	4th µsec	4th µsec
IND	5 µsec	4th µsec	4th µsec
IN A,(n)	3 µsec	3rd µsec	3rd µsec

It is important to note however, that consideration of a write during the update microsecond does not take place at the same "time" according to the instructions used, and this can therefore affect the management of the value by the circuit.

One way to measure this difference is to use processes that are not "slowed down" by the CRTC and the GATE ARRAY, such as the HSYNC display for example. See Chapter 14.9, page 137.

We can also measure this difference, for example, between what happens with an I/O on the 3rd NOP of an OUT(C),R8 and the 5th NOP of an OUTI on a CRTC 1. See Chapter 13.7, page 118.

It should also be noted that on CRTC's 3 and 4, the CRTC misses the I/O on the 3rd µsec of OUT(C),R8 instruction and retrieves it on the 4th µsec of the instruction. Which delays register updates by 1 µsec if this instruction is used. **This shift does not occur if the OUTI/OUTD instructions are used**. See next Chapter.

The majority of the diagrams that refer to Input-Outputs in this document are performed on the basis of the OUT(C),R8 instruction. For CRTC's 3 and 4, the Input-Output is positioned on the 4th µsecond.

In practice, for code produced for CRTC's 3 or 4, it is necessary to position a write I/O instruction 1 µsec before the one which would have been placed for CRTC's 0, 1 or 2.

This is established!

4.4.4 OUTs DISSECTION

This chapter aims to try to explain why an output entry with an "OUT(C),R8" occurs on the 3rd NOP for a CRTC equipped with a GATE ARRAY, and on the 4th NOP for an ASIC that emulates a CRTC (CRTC's 3 and 4), but also why there is a difference (whether on a CRTC or ASIC) between an input/output performed with an "OUT(C),R8" and that performed with an "OUTI".

The GATE ARRAY, within the CPC, is THE conductor for many components. It is clocked at the staggering speed of 16 MHz (which allows it to display pixels Mode 2 at 0.0625 µSec). It gives a 1 MHz rate for the AY-3-8912 (sound generator), the CRTC, and clocks the Z80A at 4 MHz.

One of the objectives of the gate array designers was to use Z80A's ability to slow down its execution to retrieve the RAM access priority for addresses pointed to by the CRTC.

The instructions of a z80a consist of periods of execution (called cycles M) in which several sub-periods occur (called cycles T). Each "T" sub-period duration is 0.25 µsec (the size of 4 pixels in graphic mode 2).

The common point of all Z80A instructions is the need to access the RAM to read the code(s) of the instruction to be executed (called "**opcode**" for operation code). This reading, called "**opcode fetch**", is performed during a first cycle called M1. Each cycle M performs a basic instruction :

- Reading a RAM opcode ("opcode fetch").
- Reading or writing in a byte by the Z80A internal code that executes the opcode.
- Reading or writing an input / output port ("IO REQ").
- Bus: request or acknowledge.
- Interruption: request or acknowledge.

A cycle M consists of several T cycles, one of which has the particularity of taking into account the signal sent to the /WAIT pin by an external component. This wait cycle is commonly named **Tw**. At this Tw cycle it honors an active low signal on /WAIT and runs wait cycles until this line getting up again. This is particularly the case for the instructions:

- OPCODE FETCH, during the **2nd cycle T** of the first M Cycle (M1).
- Reading memory, during the **2nd cycle T** of second M Cycle (M2).
- Writing memory, during the **2nd cycle T** of 3rd M Cycle (M3).
- IO REQ, during the **3rd cycle T**.

When the Z80A "performs" a cycle Tw, it looks at its wait line (in this case that connected to the gate array) and if it is active, then it will generate another cycle Tw. This allows to block the processor indefinitely if the circuit that drives the line wait decides to.

This blocking is only possible if the processor has allowed it in a **Tw** cycle and the GATE ARRAY/ASIC made the blocking request at that time.

The trick of the **GATE ARRAY/ASIC** designers has been to **continually generate 3 Tw followed by a "no Tw" cycle**. When the Z80A, for a Cycle T that makes a wait, falls on one of these cycles Tw, it will run the "remaining" sequence of Tw, which has the effect of "**linearizing**" the instructions over 4 T Cycles.

In other words, if an instruction started on an "aligned" T-cycle and ends with a number of T-cycles which is not a multiple of 4, then the next instruction will be "stretched" during its first cycle "M".

To illustrate this, diagram A on the following pages describe an **OUT(#nn),A** instruction (1 opcode #D3) that needs 11 cycles to run. This instruction has an odd number of T-cycles (so no multiple of 4) and is immediately followed by a second **OUT (#nn),A**.

When running the first **OUT(#nn),A**, none of the WAIT signals of the Z80A falls at the same time as a WAIT signal from GATE ARRAY. The instruction is executed in 11 T-Cycles (0.25 x 11 = 2.75 µSec).

When the second **OUT(#nn), A** is executed, the WAIT signal from the Z80A during cycle M1 occurs at the same time as the WAIT signal from the GATE ARRAY. This causes the generation of a 2nd wait T-cycle by the Z80a (see "**Wait extent**" on the diagram A). During this 2nd wait cycle, the GATE ARRAY does not send a WAIT signal, which stops the generation of wait cycles by the z80a. Therefore this 2nd OUT (#nn), finds itself "stretched" by 0.25 µsec to match the memory access pattern defined by the gate array. Its execution then takes 3 µsec (12 t-cycles).

And so on. As long as the "rectified" instruction itself contains an "unaligned" number of T-cycles, the duration of the following instruction will also be rectified. Thus in the previous example, if a NOP (4 t-cycles) is added behind the 2nd **OUT(#nn),A**, then the cycle T2 of the M1 cycle will generate a wait at the same time as the gate array, which will lengthen the NOP duration of 0.25 µsec (so 1.25µsec in total). Another NOP after this first NOP will be aligned. Indeed, the T2 cycle of its M1 cycle will be aligned with the "non-wait cycle" of the GATE ARRAY, and the instruction will then run in 4 T-Cycles (1 µsec).

The same goes for all the instructions whose M1 cycle lengthens according to the timing of the gate array in order to maintain priority on the RAM accesses to the address provided by the CRTC every 4 T-Cycles.

As part of an I/O writing operation, this mechanism is used to stall, compared to the beginning of a µsec, the moment when I/O begins. In this case on **the cycle T2 of the I/O M cycle**. The Z80A puts the I/O address on the data bus (a reminder of address &BD00 for writing to a CRTC register).

The GATE ARRAY clocks the CRTC at 1 MHz (but not quite at the same time as for the AY-3-8912). The CRTC periodically checks whether the Z80A IORQ signal is active in order to determine if it is affected by the I/O. If this is the case when writing, it can recover the value to select one of its registers or update the one that is selected.

Depending on the instruction which generates the I/O, the data is not immediately present when the CRTC is in a state to update its registers, which may have the consequence of deferring the writing of registers between 2 different instructions within the same micro-second. This is particularly the case between the instruction OUT(C),reg and OUTI, whose difference can be highlighted with **R2.JIT** or **R3.JIT** techniques, for example. Diagram B on the following pages parallels these two instructions.

In principle, the ASIC's do not clock the CRTC exactly like the GATE ARRAY ("CRTC's" 3 and 4). There is undoubtedly a lag of the type which I represented on diagram C, which has the consequence of modifying the time of register updates relative to what happens with the GATE ARRAY.
It also explains why the update of a CRTC register takes place on the 5th µsec of the OUTI instruction, regardless of the type of CRTC, while there is a difference of 1 µsec when the update takes place with the OUT(C),R8 instruction.

Diagram A : OUT(#NN),a + OUT (#nn),A + NOP + NOP

OUT(#nn),A	1																																			
	1 µSec																1 µsec												1 µSec							
GA	WAIT							WAIT				WAIT					WAIT				WAIT				WAIT								WAIT			
Z80 Cycle M	M1 - FETCH 0xD3																M2 -READ CYCLE								M3 - OUT										
	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	8	9	10	11	12	13	14	15
No Tcycle	T1				T2				T3				T4				T5				T6				T7				T8				T9			
Z80A Oper	--				MREQ RD				RFSH				RFSH MREQ				--				MREQ RD				MREQ RD				--				WR IORQ			
Z80A Tw					WAIT																WAIT															

OUT(#nn),A	2																																			
					1 µsec												1 µSec												1 µsec							
GA	WAIT				**WAIT**				WAIT				WAIT				WAIT				WAIT				WAIT				WAIT				WAIT			
Z80 Cycle M	M1 - FETCH 0xD3																M2 -READ CYCLE								M3 - OUT										
	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	8	9	10	11	12	13	14	15
No Tcycle	T1				T2				Tw				T3				T4				T5				T6				T7				T8			
Z80A Oper	--				MREQ RD								RFSH				RFSH MREQ				--				MREQ RD				MREQ RD				--			
Z80A Tw					**WAIT**				**WAIT extent**																WAIT											

NOP	3																																			
					1 µsec																															
GA	WAIT				**WAIT**				WAIT				WAIT				WAIT				WAIT				WAIT											
Z80 Cycle M	M1 - FETCH 0x00																																			
	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
No Tcycle	T1				T2				Tw				T3				T4				T1				T2				T3				T4			
Z80A Oper	--				MREQ RD								RFSH				RFSH MREQ				--				MREQ RD				RFSH				RFSH MREQ			
Z80A Tw					**WAIT**				**WAIT extent**																**WAIT**											

NOP	4																																			
																	1 µSec																			
GA																	WAIT				WAIT				WAIT											
Z80 Cycle M																	M1 - FETCH 0x00																			
																	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
No Tcycle																	T1				T2				T3				T4							
Z80A Oper																	--				MREQ RD				RFSH				RFSH MREQ							
Z80A Tw																					**WAIT**															

Diagram B : OUT(C),H vs OUTI on CRTC 0, 1, 2

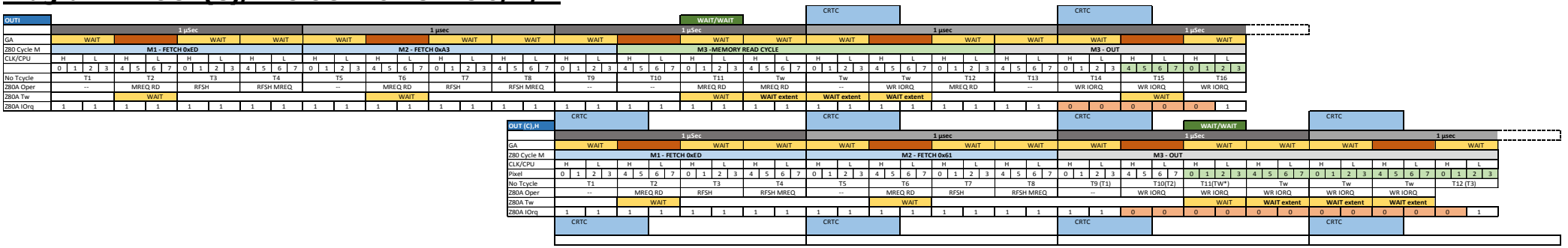
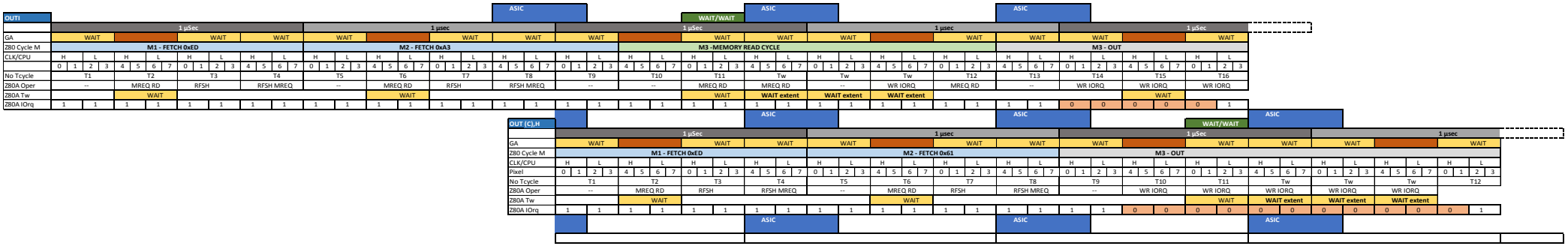


Diagram C : OUT(C),H vs OUTI on CRTC 3, 4



5 OTHER CIRCUITS

5.1 ACCESS

The devices on the CPC are wired in such a way as to partially decode the address used to perform the Input/Output. (Did I ever write "Welcome to CPC"?).

A device is affected by an Input/Output operation as soon as a few precise bits of the address bus are set to 0 and/or 1. This implies that if other bits relative to other devices are also set, then the Input/Output operation will also affect them.

It is therefore possible to send the same value to different devices **simultaneously**.

The advantage may seem relative because the common values useful to several devices are not huge. However, if we measure the consequences of sending unforeseen values to a device, it makes it possible to make "savings" of register(s) by judiciously modifying the value of the Z80A B and/or C registers.

In an environment highly constrained by machine time, it may have the advantage of assigning other uses to these registers.

		Register B (or A)								Register C (or n)								
Circuit	**r/w**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Select Addr**
PAL	w	**0**	**1**	x	x	x	x	x	x	x	x	x	x	x	x	x	x	&7F00 (*)
Gate Array	w	**0**	**1**	x	x	x	x	x	x	x	x	x	x	x	x	x	x	&7F00
CRTC Select	w	x	**0**	x	x	x	x	**0**	**0**	x	x	x	x	x	x	x	x	&BC00
CRTC Write	w	x	**0**	x	x	x	x	**0**	**1**	x	x	x	x	x	x	x	x	&BD00
CRTC Status	r	x	**0**	x	x	x	x	**1**	**0**	x	x	x	x	x	x	x	x	&BE00
CRTC Read	r	x	**0**	x	x	x	x	**1**	**1**	x	x	x	x	x	x	x	x	&BF00
ROM Select	w	x	x	**0**	x	x	x	x	x	x	x	x	x	x	x	x	x	&DF00
Printer	w	x	x	x	**0**	x	x	x	x	x	x	x	x	x	x	x	x	&EF00
PPI Port A	r/w	x	x	x	x	**0**	x	**0**	**0**	x	x	x	x	x	x	x	x	&F400
PPI Port B	r/w	x	x	x	x	**0**	x	**0**	**1**	x	x	x	x	x	x	x	x	&F500
PPI Port C	r/w	x	x	x	x	**0**	x	**1**	**0**	x	x	x	x	x	x	x	x	&F600
PPI Control	w	x	x	x	x	**0**	x	**1**	**1**	x	x	x	x	x	x	x	x	&F700
FDC Status	r	x	x	x	x	x	**0**	x	**1**	**0**	x	x	x	x	x	x	**0**	&FB7E
FDC Data	r/w	x	x	x	x	x	**0**	x	**1**	**0**	x	x	x	x	x	x	**1**	&FB7F
FDC Motor	w	x	x	x	x	x	**0**	x	**0**	**0**	x	x	x	x	x	x	x	&FA7F

(*) Note that bit 14 of the PAL selection address must be at 1 on CPCs equipped with CRTCs 0, 1, 2. It can be at 0 or 1 on CRTCs 3 and 4. For compatibility reasons, it is strongly advised to always set bit 14 to 1 to select PAL.

Finally, still from the perspective of constrained environments, it may be useful to consider that not all bits of a value sent to a device are useful.

For a CRTC, this is true for register update data, such as R9 truncated to 5 bits, as well as the value of the register number.

In other words, you can select R9 with the value 9, but also &31 (00110001), and update it with the value 7, but also &27 (00100111).

5.2 CPC + / GX 4000

On the CPC+ (or the GX 4000), it is possible to communicate directly with the emulated CRTC, via additional registers created specifically to manage the complementary "PLUS" functions. These functions allow for example, the masking of data, performing fine shifts, defining lines and addresses of rupture, or even to stall a raster or "DMA" interrupt (not without bugs however!).

The current objective of this document is not (yet) to analyse the interaction between these registers and the EMULATION of the CRTC. Access to these specific registers does not pass through the Z80A's input/output system described above.

The CPC+ has a page of registers "mapped" on a memory area.
Each register therefore has its own address.
The Z80A accesses these registers with simple writes (or reads) to a given address.

It is therefore possible to access certain functions (e.g. changing colours) in two different ways. The information provided for CRTC 3 in this version of the document relates only to I/O performed via the Z80A OUT Instructions.

It should be noted, however, that the page of these registers, located between &4000 and &7FFF, is accessible via a previously untapped function of the GATE ARRAY.

However, this function itself is conditional on the use of a 17-byte "unlock" sequence which must be sent to the CRTC's selection port (&BC00) :
RQ00, 0, 255, 119, 179, 81, 168, 212, 98, 57, 156, 70, 43, 21, 138, STATE, <ACQ>
RQ00 must be different from the value 0.
STATE=205 for UNLOCK otherwise another value for LOCK.
ACQ represents sending any value if STATE=205 (not needed otherwise).

Remark :
There is no point in sending this unlock sequence to a CRTC 4.
They didn't use the same sequence.
I wouldn't say anything more without the presence of my lawyer.

6 BUILDING A FRAME

6.1 GENERAL LOGIC

The following algorithms describe the general display management logic of an "ideal" CRTC. As we will see later, this logic is sometimes very different depending on the situation.

6.1.1 CHARACTER COUNTING

C0 is incremented by 1
When C0 reaches R0
    C0 goes to 0
    C9 is incremented by 1
    When C9 reaches R9
        C9 goes to à 0
        C4 is incremented by 1
        When C4 reaches R4
            When C5 reaches R5
                C4 goes to 0, C5 goes to 0
                MA is updated from R12/R13
            Otherwise C5 is incremented by 1

6.1.2 SYNCHRONIZATIONS

When C0 reaches R2
    Hsync starts, C3l=0
    C3l is incremented if R3l>0
    When C3l=R3l
        End of Hsync

When C4 reaches R7
    Vsync starts, C3h=0
    C3h is incremented
    When C3h reaches R3h (or 16)
        End of Vsync

6.1.3 CHARACTER DISPLAY

When C0 goes to 0
    Character display is enabled
When C0 reaches R1
    Character display is disabled

When C4 goes to 0
    Character line display is enabled
When C4 reaches R6
    Character line display is disabled

6.1.4 VIDEO POINTER

At each µsec, VMA is incremented by 2 as long as the display is active and C9 is part of the address

```
If C0=R0
Then C0=0                                   true for R0=0
        If C9=R9
        Then C9=0                           true for R9=0
                If C4=R4
                Then
                        If R5=0
                        Then                true for R5=0
                                C4=0        true for R4=0
                                MA=R12/R13
                        Else
                                C4=C4+1 (CRTC's 0, 1 and 2) and C5 management
                Else C4=C4+1
        Else C9=C9+1
Else C0=C0+1
```

6.1.5 DIAGRAMS

The following diagrams describe the overall construction of a frame from the different registers, without modifying the registers during the frame.

Screen definition	Horiz. Reg				Vertical Reg					Vid Ptr		Special
	R0	R1	R2	R3	R4	R5	R6	R7	R9	R12/R13		R8
	63	40	46	14	35	24	25	30	7	&30	0	0

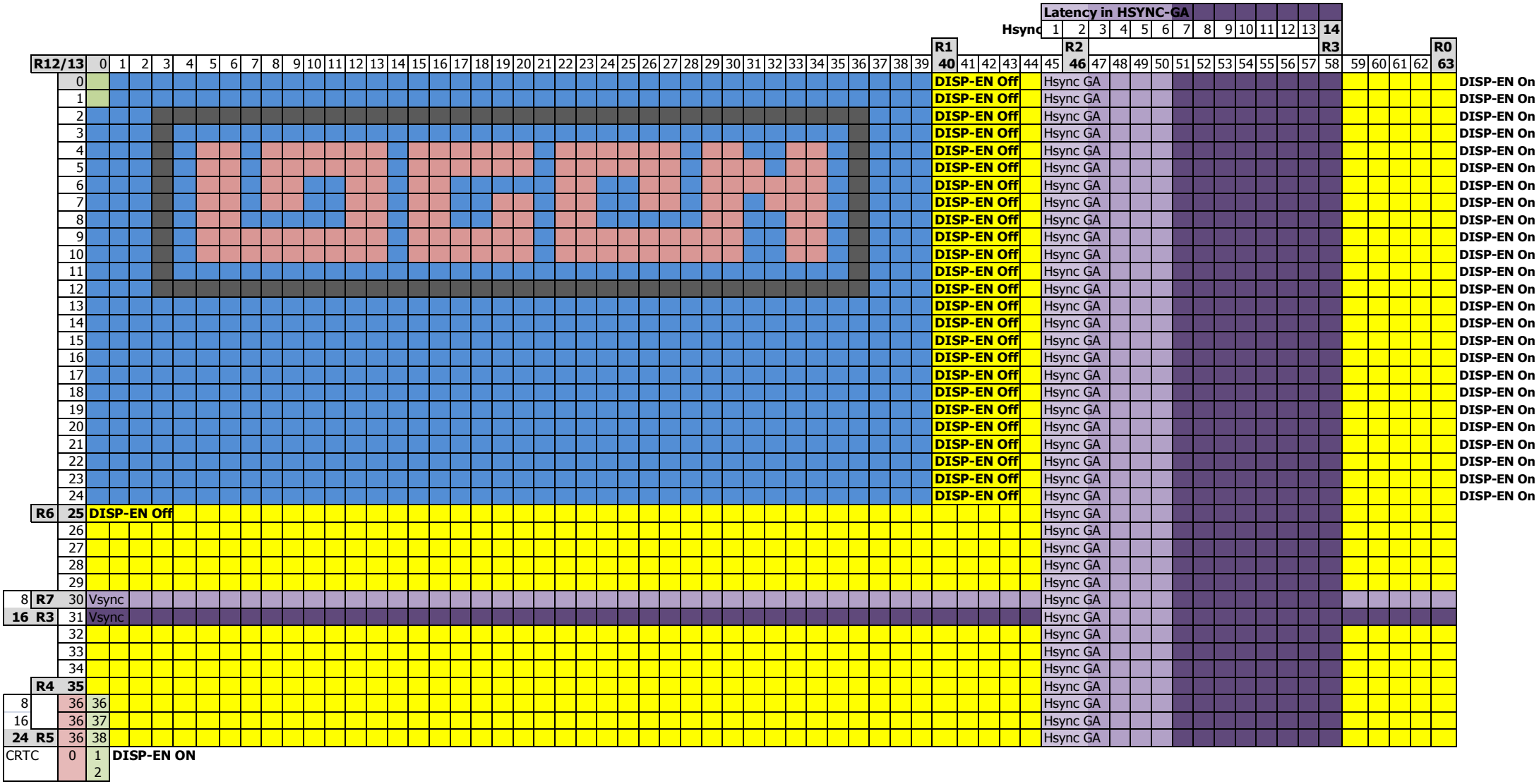
(R0+1) x [((R4+1) x (R9+1)) + R5]
64 x ((36 x 8)+24) = 19968 µs

Description of the first 16 lines shown in green on the diagrams on the following pages:

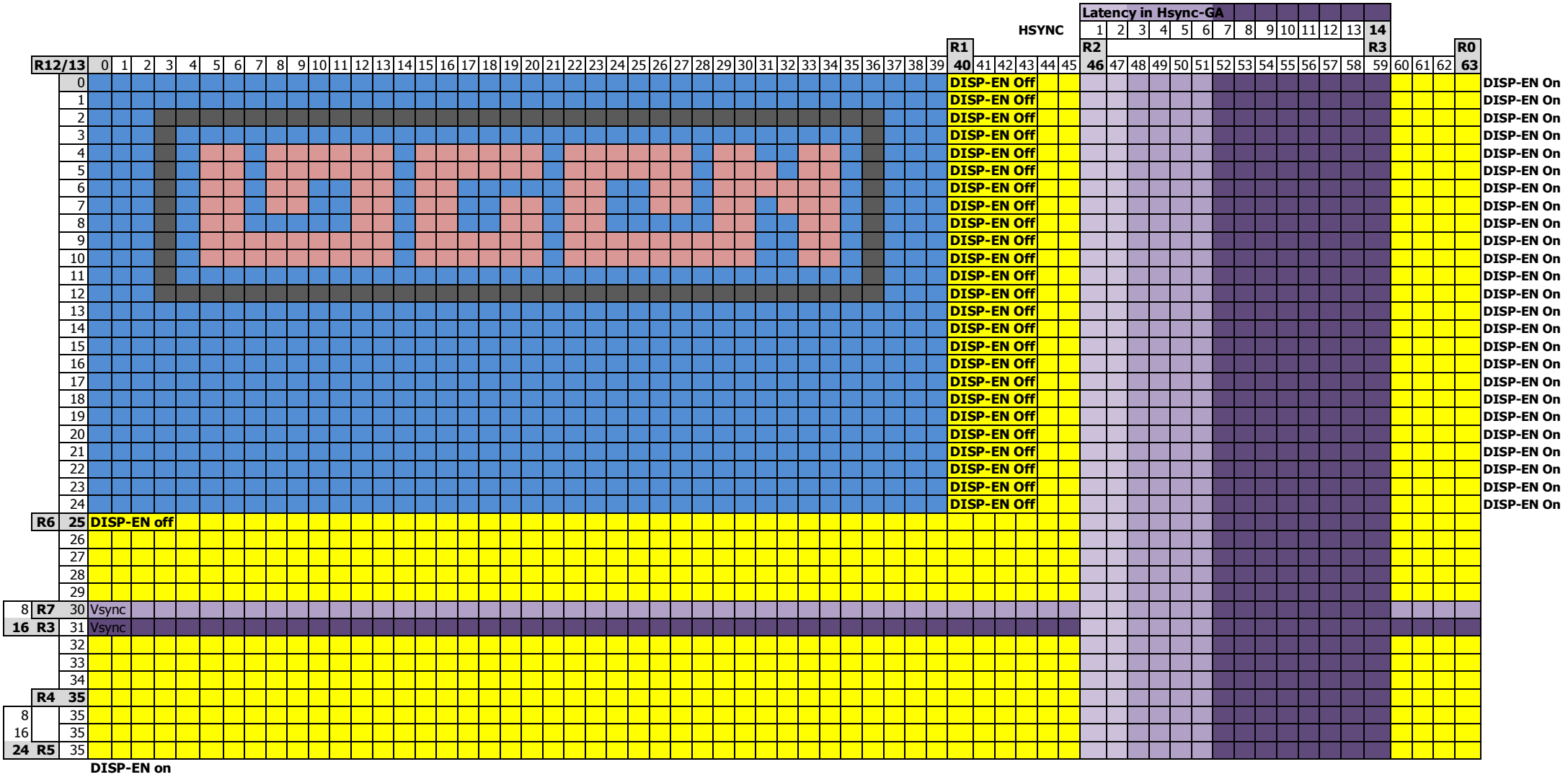
Video Pointer																																
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																	
R12		**C9 (*)**			**R12/13**										0	Hex			**First byte**								**Second byte**							
5	4	2	1	0	9	8	7	6	5	4	3	2	1	0	0	Addr		C9	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	C4
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C000		0																	0
1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	C800		1																	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	D000		2																	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	D800		3																	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	E000		4																	0
1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	E800		5																	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	F000		6																	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	F800	**R9**	**7**																	0
1	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	C050		0																	1
1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	C850		1																	1
1	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	D050		2																	1
1	1	0	1	1	0	0	0	0	1	0	1	0	0	0	0	D850		3																	1
1	1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	E050		4																	1
1	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0	E850		5																	1
1	1	1	1	0	0	0	0	0	1	0	1	0	0	0	0	F050		6																	1
1	1	1	1	1	0	0	0	0	1	0	1	0	0	0	0	F850	**R9**	**7**																	1

(*) C5 on CRTC 3, 4 in vertical adjustment period (R5)

CRTC 0, 1, 2


Latency in HSYNC-GA
Hsync
R12/13
R1
R2
R3
R0
R6
R7
R4
R5
CRTC
DISP-EN Off
DISP-EN On
Hsync GA
Vsync
DISP-EN ON

CRTC 3, 4


HSYNC
Latency in Hsync-GA
R12/13
R1
R2
R3
R0
R6
R7
R3
R4
R5
DISP-EN Off
DISP-EN On
DISP-EN off
DISP-EN on
Vsync

7 SYNCHRONIZATION

7.1 PRINCIPLES

The CRTC is responsible for determining the memory addresses to be displayed, which it communicates to the GATE ARRAY, which reads the data and manages it according to its parameters, to generate colourful and various width pixel sizes.

The CRTC also sends VSYNC and HSYNC signals to GATE ARRAY, which uses them to trigger and send a composite signal to the monitor.

- When the GATE ARRAY receives a HSYNC signal, there is almost 2 µs of latency before the horizontal synchronisation of the monitor begins.
- When the GATE ARRAY receives a VSYNC signal, there are 2 "HSYNC lines" before the vertical synchronisation of the monitor begins.

On a topic that deals with the differences between microsecond scale circuit models, it is important to spatialize the instructions in relation to what is displayed.

Several difficulties are associated with this approach:

- Situate a Z80A instruction in relation to the internal workings of the CRTC, as delays may exist between the update of internal records and the time when characters are displayed.
- Determine when, during an Input-Output instruction, the modification of a CRTC register is effective and considered.

The same type of issue exists for Z80A instructions accessing the GATE ARRAY directly, or when updating the RAM read by the GATE ARRAY.

To "synchronize" the instructions, this document uses a single point of reference to which the instructions of the Z80A affecting the video are aligned.

This reference point is the moment when C0 is considered equal to 0 from the perspective of the CRTC.

There is a delay in display between when the CRTC provides a video pointer, and when the GATE ARRAY displays the corresponding 16-bit character.
This delay is 1 µsec.

This display time lag of the GATE ARRAY with respect to the CRTC would not be a problem if the entirety of what is sent by the CRTC to the GATE ARRAY were **always** delayed by 1 µsec.

But this is not always the case, especially for HSYNC signal management for machines equipped with CRTC's 0, 1 and 2.

When this document details Z80A instructions aligned with the CRTC, C0 counter is presented on 2 "time-line". The one with respect to C0 from the CRTC reference point and the one with respect to the display (pixels, ...) by the GATE ARRAY.
It will be a question of "**C0 from Vsync**" (or **C0vs**) and/or "C0 from GA" when the assignment of the CRTC's registers is decisive in relation to the display of characters by the GATE ARRAY.
The CRTC communicates with Z80A in two different ways (outside of the CRTC 1 read or status registers):

- Via the CRTC VSYNC pin that is connected directly to the 0 bit of port B of PPI 8255.
- Via the Z80A interrupts produced by the GATE ARRAY from the CRTC's HSYNC signals.

7.2 VSYNC SYNCHRONIZATION

The CRTC generates a signal on its VSYNC pin when C4 reaches R7.
This is a reliable indicator to then base any other synchronization method.

The bit 0 of port B of the PPI ("standard" &F5 input-output address) changes to 1 as soon as the VSYNC signal is produced by the CRTC.

In general, programs make "loops" to wait for this bit to go to 1, which leads to a margin of error related to the duration of the instructions of the waiting code.

This margin exists even if the "loops" code is "aligned" through an interrupt generated from a HALT statement located before the waiting loop (the margin is then just "stable").

However, it is perfectly possible to write code capable of setting Z80A code to the microsecond that corresponds to C4=R7 and C0=C9=0.

All data shown in this document will use this **C0vs** reference point. It is of course possible to use the interrupt system as a new reference point.

Chapter 26.6.6 summarizes interruptions according to the different CRTC's.
It should not be forgotten that interruptions are dependent on the differences between the CRTC/GATE ARRAY couples of the different machines.

The principle of the **C0vs** synchronization code is to place a VSYNC wait in a period where the indicator at the PPI level is not yet set to 1.

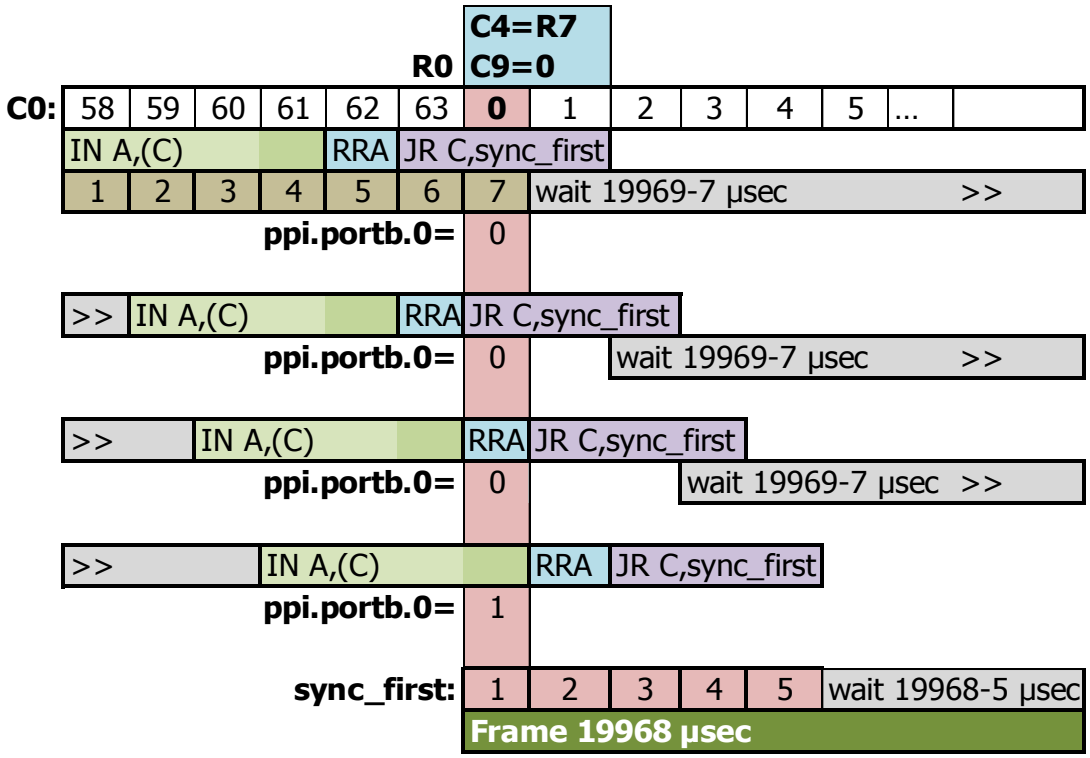
It is then a question of deriving this microsecond code by microsecond at each frame, thanks to a period of **19969** microseconds. This scan is 1 microsecond longer than that produced by the CRTC (with adequate and standard programming of 312 lines of 64 µsec).

This can be seen as a "drift" of the code microsecond-by-microsecond.

The Input-Output instruction, which reads port B of the PPI, is the IN A,(C) instruction which lasts 4 µsec. It is on the 4th µsec of the IN A,(C) instruction that A is updated when the bit 0 of port B changes to 1.

The **C0vs** reference point is calculated in this way and all the information given in this document is correlated with this measurement, which is a single reference point.

Schematic description of this principle :



Synchronisation code :

```
;=============================================================================================
;
;       Vsync synchronization
;       At the end of the synchronization call, the 1st usec is the one on which the CRTC has positioned the Vsync signal
;       Principle
;               - wait for the vsync signal with the margin of error
;               - wait for the vsync signal to finish
;               - derive the vsync test with a loop during 19968+1 us
;               - as soon as the vsync is detected by the test, it is necessary to subtract the duration of the test
;
;=============================================================================================
sync_vbl
                di
                ld b,#f5            ; Wait VBL flag =1
                ld hl,19968-23      ; Nop counter  (minus margins and wait management)
                ld de,-11
sync_wvblon1                        ; Here we are waiting for the beginning of the VBL period (or we are not waiting if we were there already)
                in a,(c)            ;
                rra                 ;
                jr nc,sync_wvblon1
sync_wvbloff1                       ; Flag Vsync CRT goes to 1 (or was already to 1)
                in a,(c)            ; Wait for the flag to go back to 0 (End of Vsync)
                rra
                jr c,sync_wvbloff1
sync_wvblon2                        ; We are now certain that the Vsync signal was not already in progress
                in a,(c)            ;
                rra                 ; margin1 of 7us
                jr nc,sync_wvblon2
sync_wvbloff2                       ; Wait for the Vsync signal to return to 0 by counting the time elapsed
                add hl,de           ; 3       On nop2             On nop3
                in a,(c)            ; 4       2                   1
                rra                 ; 1       1                   1
                jr c,sync_wvbloff2  ; 3/2     2                   3         (bcl)+3+4+1+2=15 / margin 15-5=10
                ex de,hl            ; 1
                call wait_usec      ; 5 >> 6 + 10(margin2)
                ;
                ; Drift zone to wait again for the first manifestation of the flag
                ; the in a,(c) will "go down" nop by nop (frame by frame) until the in recovers the active flag
sync_derive_bcl
                ld b,#f5                    ; 2
```

```
                in a,(c)            ; 4 usec. 0.1.2.[3] (+1)
                rra                 ; 1 usec (+1)
                jr c,sync_first     ; 2/3 (+3)
                ld de,19969-20      ; 3
                call wait_usec      ; 5+(19969-20)
                jr sync_derive_bcl  ; 3 >> 20
sync_first                          ; 6 The flag has been detected at the earliest, and this since 5 usec (2+1+3) C0=R0
                                    ; -1 µs to position on C0=0
                ld de,19968-11      ; 3
                jp wait_usec        ; 3 >> 11 >> de=19968-11

;==============================================================================
; wait "de" usec
; 40+(((de/8)-5) x 8)+(de and 7) nop
; nb - the call of the function is not included
;==============================================================================
wait_usec:
                ld hl,sync_adjust   ; 3
                ld b,0              ; 2
                ld a,e              ; 1
                and %111                    ; 2>8
                ld c,a              ; 1
                sbc hl,bc           ; 4
                srl d               ; 2
                rr e                ; 2>17
                srl d               ; 2
                rr e                ; 2
                srl d               ; 2
                rr e                ; 2>25
                dec de              ; 2>27 8
                dec de              ; 2>29 16
                dec de              ; 2>31 24
                dec de              ; 2>33 32
                dec de              ; 2>35 40 *
                nop                 ; 1>36
wait_usec_01
                dec de              ; 2 -
                ld a,d              ; 1 -
                or e                ; 1 -
                nop                 ; 1 -
                jp nz,wait_usec_01  ; 3 - v=(8 x DE)
                jp (hl)             ; 1>37
                nop                 ; 1 * v=0--7
                nop                 ; 1
                nop                 ; 1
                nop                 ; 1
                nop                 ; 1
                nop                 ; 1
                nop                 ; 1
sync_adjust
                ret                 ; 3>40 *
```

7.3 FAKE VSYNC

The VSYNC signal is part of a tripartite relationship between the CRTC, GATE ARRAY and PPI. CRTC reports status of VSYNC pin to PPI and GATE ARRAY.

The PPI 8255 is a programmable generic device interfacing circuit.
AMSTRAD used 3 different versions of this component:

- NEC D8255AC-5
- NEC D8255AC-2
- TOSHIBA TMP8255AP-5

The last digit indicated is the maximum frequency of the circuit. To my knowledge there is no relevant test to differentiate these models.

It is possible to operate the communication pins of this circuit as input or output according to appropriate programming.

Port B of the PPI is designed on the CPC to work as input, and therefore receive information from the devices to which it is connected. It is possible to reprogram this port as output, by setting to 0 the bit 1 of the control register of the PPI located at the address of I/O #F700. Writing to port B (#F500) a value with the bit 0=1 will therefore put pin 25 of the circuit in the high state. This means that the PPI will send its signal to the CRTC, which in fact will send it back to the GATE ARRAY, since it is also connected to it.

However, I have found, like Kevin Thacker (ArnoldEmu) before me, that this FAKE VSYNC does not work on some CPC's. On other CPC's, the result is incorrect.

On a CRTC 2, programmed with R2=50 and R3=14, VSYNC is in principle no longer generated.

We will see later that a GHOST VSYNC is generated on this CRTC if the VSYNC condition takes place during HSYNC. It should therefore theoretically be possible to do this work instead of the CRTC by placing bit 0 from port B to 1 in the right place and for the right duration.

However, if a PPI-VSYNC is activated for 1024 µsec instead of the one that would have been produced by the CRTC, the signal produced by the PPI is not strong enough to counter the low signal generated by a GHOST VSYNC. But when the CRTC moves its pin back to the low state (when it is already there), then the VSYNC of the PPI is then considered, and the image is stabilized 16 lines higher than expected. An electronics technician would certainly help me to see more clearly.

However, it is still possible to bypass CRTC 2's GHOST VSYNC more easily, by preventing the VSYNC condition to occur during HSYNC and by updating R7 with C4 right after the end of HSYNC (See Chapter 16.4.3, page 163).

WARNING : I don't really know the potential "technical" consequences of sending a signal in the opposite direction to the one intended. Having said that, the risk seems quite low since, for several years, many CPC's have contained several CRTC's welded on top of one another in an orgy of tin. The passive CRTC(s) then taking signals from the active CRTC. In the worst case, a CRTC is worth (still) less than €10.

A bit of history

This is not the first time that an input port has been used as output on the CPC. The older ones will probably remember the download kit which allowed for the retrieval of (paid) programs via a telephone connection connected to the Minitel (I do not know if this system was used with Prestel in U.K.).

The cable of one of these kits was connected from the Minitel socket on the CPC with its JOYSTICK socket, and in order to be able to manage a bidirectional exchange, the keyboard port was then placed in output. Keyboard lines are read through port A of the PPI, via port A of the AY-3-8912 (sound generator).

Even if this had no consequences for the vast majority of users, I witnessed several CPC's whose AY-3-8912 had "permanently lost" part of the bits of some registers of the AY. From memory, it seems to me that these CPCs had particularly high-pitched tones, if it comes to determining which bits were "ravaged".

8 DISPLAY, Z80A & GATE ARRAY

The following diagrams describe in relation to some video ram update instructions by the Z80A and when this update is considered by the GATE ARRAY/ASIC.

Instructions are localized relative to the CRTC's **C0vs** reference point.

The RAM reading by the GATE ARRAY/ASIC for data modified by the Z80A is the same for all CPC's.

8.1 INSTRUCTION LD(HL),reg8 (2 µsec)

HL=0000	reg8=#FF																			
C0vs	00		01		02		03		04		05		06		07		08		09	
Video Ptr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Z80A Inst	LD (HL),reg8				NOP		NOP		NOP		NOP		NOP		NOP		**LD (HL),#00**			
Displayed :	**00**																			

19968-10 µsec

HL=0001	reg8=#FF																			
C0vs	00		01		02		03		04		05		06		07		08		09	
Video Ptr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Z80A Inst	LD (HL),reg8				NOP		NOP		NOP		NOP		NOP		NOP		**LD (HL),#00**			
Displayed :			**00**																	

19968-10 µsec

HL=0002	**reg8=#FF**																			
C0vs	00		01		02		03		04		05		06		07		08		09	
Video Ptr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Z80A Inst	**LD (HL),reg8**				NOP		NOP		NOP		NOP		NOP		NOP		LD (HL),#00			
Displayed :					**FF**															

19968-10 µsec

HL=0003	**reg8=#FF**																			
C0vs	00		01		02		03		04		05		06		07		08		09	
Video Ptr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Z80A Inst	**LD (HL),reg8**				NOP		NOP		NOP		NOP		NOP		NOP		LD (HL),#00			
Displayed :							**FF**													

19968-10 µsec

8.2 INSTRUCTION LD (aaaa),HL (5 µsec)

aaaa=0004	H=#FF, L=#55																**H'=0, L'=0**										
C0vs	00		01		02		03		04		05		06		07		08		09		10		11		12		13	
Video Ptr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Z80A Inst	LD (aaaa),HL										NOP		NOP		NOP		NOP		NOP		NOP		EXX		**Ld (aaaa),HL**			
Displayed :					**00**	**00**																						

19968-17 µsec

aaaa=0005	H=#FF, L=#55																**H'=0, L'=0**										
C0vs	00		01		02		03		04		05		06		07		08		09		10		11		12		13	
Video Ptr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Z80A Inst	LD (aaaa),HL										NOP		NOP		NOP		NOP		NOP		NOP		EXX		**Ld (aaaa),HL**			
Displayed :							**00**	**00**																				

19968-17 µsec

aaaa=0006	H=#FF, **L=#55**																**H'=0**, L'=0										
C0vs	00		01		02		03		04		05		06		07		08		09		10		11		12		13	
Video Ptr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Z80A Inst	**LD (aaaa),**HL										NOP		NOP		NOP		NOP		NOP		NOP		EXX		**Ld (aaaa),H**L			
Displayed :							**55**	**00**																				
							L																					

19968-17 µsec

aaaa=0007	H=#FF, L=#55																H'=0, L'=0										
C0vs	00		01		02		03		04		05		06		07		08		09		10		11		12		13	
Video Ptr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Z80A Inst	**LD (aaaa),HL**										NOP		NOP		NOP		NOP		NOP		NOP		EXX		Ld (aaaa),HL			
Displayed :									**55**	**FF**																		
									L	**H**																		

19968-17 µsec

8.3INSTRUCTION PUSH reg16 (4 µsec)

| **SP=0004** | D=#FF, E=#55 | | |

9 GATE ARRAY

On the CPC, there are 5 models of identified GATE ARRAY. One of the functions of the circuit is to display the pixels depending on the graphical mode and the colour defined for each ink.

- 40007 models (version 1: Ferranti Matrix ULA20RA023, version 2: Ferranti Matrix ULA16RA023) and 40008 (version 2) are pin compatible. It can be noted that there are different markings of the 40007, also available under 40007-4 or 40007-4x. These models are often mounted with a heatsink spread with thermal paste on motherboards.
- The 40010 model exists under 2 versions (1st version 37aa (Matrix LSI HSG3170) and 2nd version 36aa (matrix LSI HSG3130, 23% smaller)). Thanks to Gérald VINCENT for this information.
- ASIC 40226 is used on the low-cost CPC (CRTC 4).
- ASIC 40489 (CRTC 3) is used on CPC + and GX 4000.

AMSTRAD has created an impressive array of different motherboards. This implies that these 3 components can be found on some 464 and 6128 models. I did not see a motherboard of 664 capable of supporting a 40007/40008. Finally, the component is always mounted on a bracket.

Model	Gate Array Nb capacity	40007	40008	40010
464	1	☑		
464	2	☑	☑	☑
664	1			☑
6128	1	☑		
6128	1		☑	☑
6128	2	☑	☑	☑

Note 1: Chapter 3.2.2 of WEKA binders contain an error. The pins of the 40007/40008 are not compatible with those of the 40010.

Note 2 : The models of 6128 with a single location equipped with a 40007 are quite rare (MC0057A motherboards from mid-1988). The gate array is covered with a heat sink.

In general, there is a large amount of 464's equipped with various models.
The 664's are mainly equipped with 40008 and 40010.
The 6128's are mostly equipped with 40010 version 36AA.

Contrary to popular belief, there are many identifiable differences between these components. The 40007/40008 seems to be in advance of 1/16 MHz on the 40010 when it processes the bits of the byte fetched from VRAM.

9.1PIXELS

The GATE ARRAY/ASIC is responsible for converting the memory read by the CRTC into pixels.

> It performs this operation by considering both parameters: the **graphics mode** and the **colour associated** with the pixel to be displayed.

This document is not devoted to the GATE ARRAY in detail, but it should be remembered that the CPC has 4 graphic modes, of which 3 are "official".

The graphic mode determines the horizontal size of a pixel (based on the number of colours that pixel can take).

A pixel can be encoded on 1, 2 or 4 bits.
A pixel occupying 1 bit is the finest resolution that GATE ARRAY can produce.

Mode	VRAM BYTE							
0	A0	B0	A2	B2	A1	B1	A3	B3
1	A0	B0	C0	D0	A1	B1	C1	D1
2	A0	B0	C0	D0	E0	F0	G0	H0
3	A0	B0	x	x	A1	B1	x	x
Bit Nb:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Displayed Pixels							
A3	A2	A1	A0	B3	B2	B1	B0
A1	A0	B1	B0	C1	C0	D1	D0
A0	B0	C0	D0	E0	F0	G0	H0
0	0	A1	A0	0	0	B1	B0
1	2	3	4	5	6	7	8

Definition
2 pixels (16 colors)
4 pixels (4 colors)
8 pixels (2 colors)
2 pixels (4 colors)

A 17th colour, stored in the GATE ARRAY, is displayed when BORDER is activated

This pixel coding represents an index in a table containing 5-bit colour-coded colours.

Colour Index				INKR			COLOUR				
				7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	C1	C1	C1	C1	C1
0	0	0	1	0	1	0	C2	C2	C2	C2	C2
0	0	1	0	0	1	0	C3	C3	C3	C3	C3
0	0	1	1	0	1	0	C4	C4	C4	C4	C4
0	1	0	0	0	1	0	C5	C5	C5	C5	C5
0	1	0	1	0	1	0	C6	C6	C6	C6	C6
0	1	1	0	0	1	0	C7	C7	C7	C7	C7
0	1	1	1	0	1	0	C8	C8	C8	C8	C8
1	0	0	0	0	1	0	C9	C9	C9	C9	C9
1	0	0	1	0	1	0	C10	C10	C10	C10	C10
1	0	1	0	0	1	0	C11	C11	C11	C11	C11
1	0	1	1	0	1	0	C12	C12	C12	C12	C12
1	1	0	0	0	1	0	C13	C13	C13	C13	C13
1	1	0	1	0	1	0	C14	C14	C14	C14	C14
1	1	1	0	0	1	0	C15	C15	C15	C15	C15
1	1	1	1	0	1	0	C16	C16	C16	C16	C16

When the GATE ARRAY displays its pixels, **it is too "fast" for those of MODE 2**.

Indeed, on gate arrays 40007, 40008, 40010 and ASIC 40226 (CRTC 4), pixels in mode 2 are displayed one pixel earlier than for other graphics modes.

They are displayed in "advance" of 1/16 µsec (0.0625 µsec).

In other words, the BORDER "stops" 1 pixel earlier on a line in mode 2 and starts 1 pixel earlier when C0=R1 (if you have not changed graphics mode during the line).

Since it is possible to change the graphic mode between each line and/or during a line, this must be considered if it is necessary to align pixels displayed in different graphic modes.

ASIC 40489 of the CPC+ is not affected by this discrepancy.
Whatever the graphic mode on this machine, the pixels are aligned.

9.2COLOURS

When the colour table (ink colour and border colour) is updated, the consideration of this colour change is different dependent on the CRTC version.

However, the time when the update takes place is the same regardless of the mode.

9.2.1 BORDER AND GRAPHIC MODE 2

On some CPC's, the colour update of the border when the CPC is **in graphical mode 2** can have a visual incidence. Indeed, two pixels from another colour may appear at the beginning and at the end of the µsecond where the colour changes.

This problem is linked to a power fault that affects the gate array or the resistors and capacitors that surround it to generate colours, but nevertheless I mention it here as this is an issue on a significant number of CPC's. It does not depend on the CRTC, nor the GA model, the phenomenon which has been observed on one GA 40010 (28818 / 36AA) but not on another of the same reference. Finally, fiddling with the power connector to establish a more direct power contact mechanically causes the parasitic pixels to disappear.

Some "redundant" gate array colours (beyond 27) may have a different effect from their "counterpart" colours in the palette, as the parasitic pixels appear (or not). It is useful to recall here that these colours are not strictly identical to their counterpart in the palette, because of the method of generating the colours of the CPC, based on resistance sets and high impedance.

9.2.2 SPEED OF PROCESSING

The following two pages of diagrams describe how different instructions which update the colour of an ink are processed and how the GATE ARRAY/ASIC takes this into account when displaying its pixels.

This colour on the diagrams indicates which pixels according to the mode are affected by the colour change.

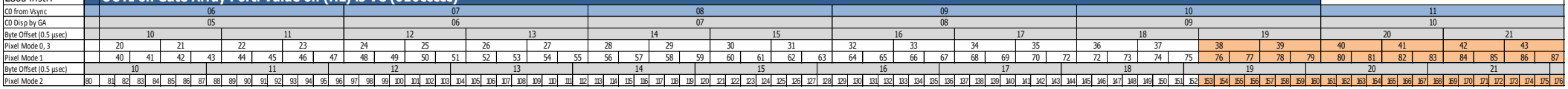
Instructions are calibrated according to **C0vs** calculated from the VSYNC reference point.

WARNING :
It should not be forgotten that it is the display of pixels in mode 2 that starts earlier than for other modes. Updating the colour of an ink, on the other hand, is considered in the same way regardless of the graphic mode.

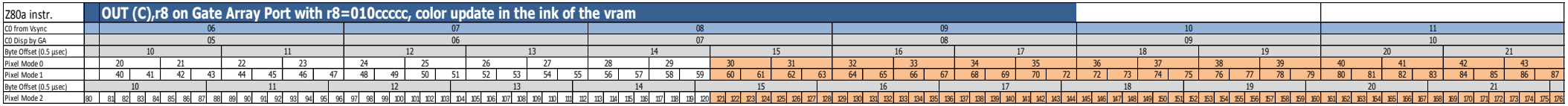
The colour therefore changes on the 2nd pixel of a byte in mode 2, but on the pixels 0 of the bytes processed for the other graphic modes.

CRTC 0, 1, 2 / GATE ARRAY 40010

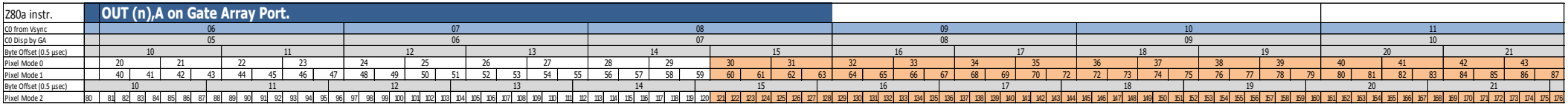
Z80a instr.	OUTI on Gate Array Port. Value on (HL) is VC (010ccccc)



Z80a instr.	OUT (C),r8 on Gate Array Port with r8=010ccccc, color update in the ink of the vram

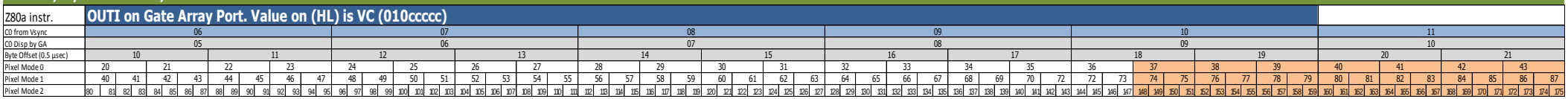


Z80a instr.	OUT (n),A on Gate Array Port.

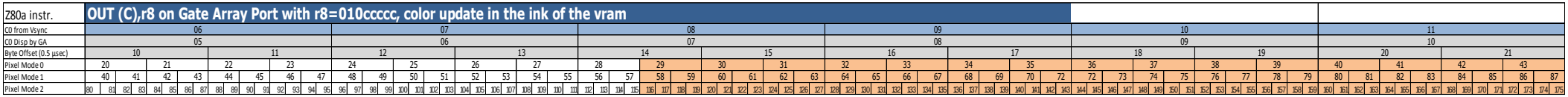


CRTC 3, 4 / ASIC 40489, 40226

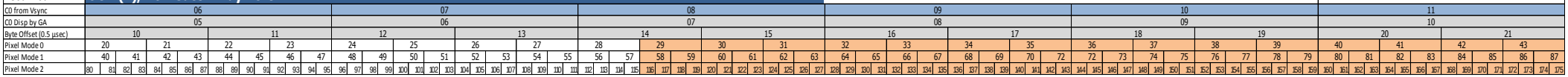
Z80a instr.	OUTI on Gate Array Port. Value on (HL) is VC (010ccccc)



Z80a instr.	OUT (C),r8 on Gate Array Port with r8=010ccccc, color update in the ink of the vram



Z80a instr.	OUT (n),A on Gate Array Port.



9.3 GRAPHIC MODE

9.3.1 GENERAL

A HSYNC signal provided by the CRTC is composed of several periods within the GATE ARRAY.

A period, which can be called HSYNC-GA, has a maximum duration capped at 6 µsec and cannot exceed the value defined in R3.

For all CPCs without exception (CRTC's 0 to 4) **a HSYNC of at least 2 µsec is required for the graphic mode update**.

The update of the internal register containing the mode is effective on the 3rd µsecond of the Z80A instruction of I/O **OUT (C),R8** to the GATE ARRAY

The minimum "latency" period for the GATE ARRAY to generate a C-HSYNC signal is very slightly less than 2 µsec (between 1.8750 and 1.9375c. µs)..

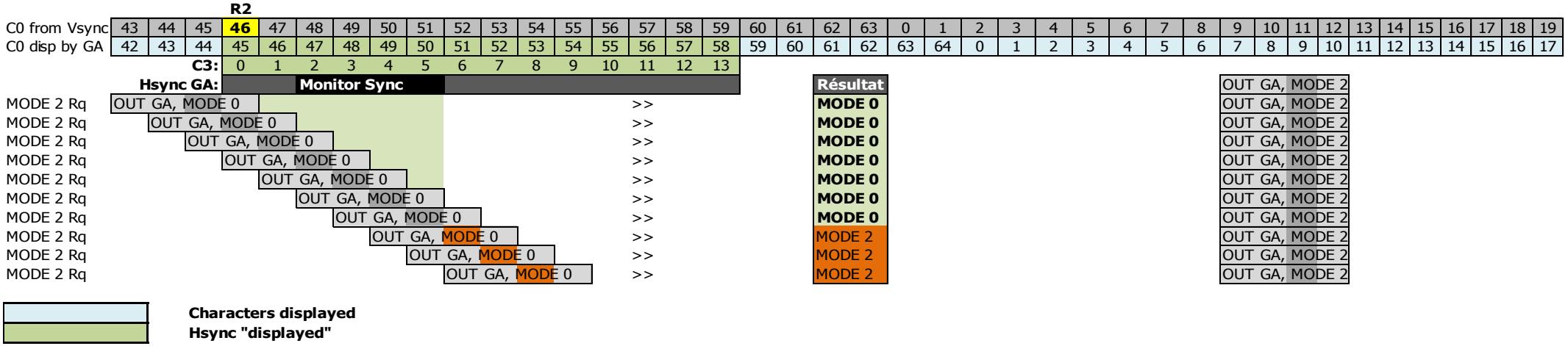
If the HSYNC-GA is programmed to 2 µsec (via the CRTC's R3l programming), then **a very short C-HSYNC signal is produced**. **On CRTC 0 and 1, it lasts 2 or 3 pixel-M2 (1 pixel-M2=0.0625 µsec). On CRTC 2 it lasts 1 or 2 pixel-M2. However, this signal is much too short to be detected by the monitor.** This setting does not prompt the monitor to give up its guts and allows graphics mode to be changed mid-line without affecting horizontal synchronization.

HSYNC is considered more quickly by the GATE ARRAY on CRTC's 0, 1 and 2, than on the ASIC's of CRTC's 3 and 4. ASIC's **delay HSYNC by 1 µsec, as they do for character display.**

On the VSYNC reference point principle, the following diagrams describe **how a mode change is accounted** for by a Z80A OUT(C),r8 instruction before and during HSYNC.

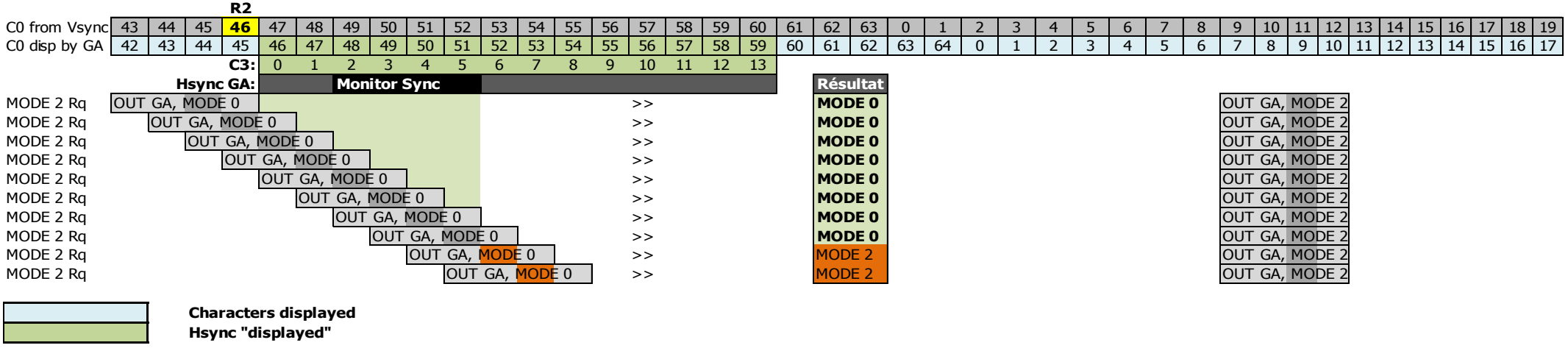
9.3.2 CRTC 0, 1, 2

CRTC-R2=46 / CRTC-R3=14 (HSYNC size)



9.3.3 CRTC 3, 4

CRTC-R2=46 / CRTC-R3=14 (HSYNC size)



9.3.4 MODE SPLITTING

9.3.4.1 GENERAL

A **Pixel-M2** designates a pixel whose horizontal size is that of one pixel in graphic mode 2.
A pixel in graphic mode 0 or 3 is composed of **4 Pixel-M2**.
A pixel in graphic mode 1 is composed of **2 Pixel-M2**.

It is possible to change the graphics mode during a line as long as this change of mode precedes a HSYNC programmed with a length R3>1.

When R3 is programmed with the value 2, the change of mode carried out by the GATE ARRAY occurs after the display has been restored on the CRTC's 0, 1, 2 and 4. (to be verified for CRTC 3).

On CRTC's 0, 2 and 4, when the display is reactivated by the GATE ARRAY (R3=2), **1 Pixel-M2 is however displayed in the previous graphics mode**. On CRTC 1, the reactivation of the display corresponds exactly to the start of the mode change by the GA, which does not allow you to take advantage of a **nice mode 2 pixel** of the previous graphics mode.

On CRTC's 0, 1 and 2, the GATE ARRAY switches the mode update to the 6th pixel mode 2 (i.e. the 3rd pixel mode 1, the 2nd pixel mode 0).

On the CRTC 4, the GATE ARRAY switches the mode change to the 4th pixel mode 2 (i.e. the 2nd pixel mode 1, the middle of the 1st pixel mode 0).

Given **that mode update occurs during the processing of the byte read in ram**, the algorithm for determining the PEN by the GA switches during processing. Consequently, the calculated PENs are distorted for the rest of the pixels to be displayed in the byte. They are partly composed of the rotations performed for the previous mode.

It is also possible to modify, on CRTC's 0, 1 and 2, the "HSYNC non-display" zone via techniques called **R2.JIT** and **R3.JIT**. (JIT = "Just In Time").

R2.JIT reduces the non-display area of the HSYNC from the left and **R3.JIT** stops the HSYNC 0.25 µsec after its scheduled end. (See Chapters 15.3 and 14.5.4 for more information).

In the context of a graphics mode change, the **R3.JIT** technique does not shorten the period required for the graphics mode update. (You didn't think I wasn't going to try...). Indeed, if this technique is used on the 2nd µsec of the **HSYNC**, the change of graphics mode does not take place. The change of MODE is considered only on an **R3.JIT** during the 3rd µsec. This implies a non-display time of 2.25 µsec for a mode update. This nevertheless has the advantage of moving the moment when a "mixed" mode 2 pixel is visualized in another graphic mode.

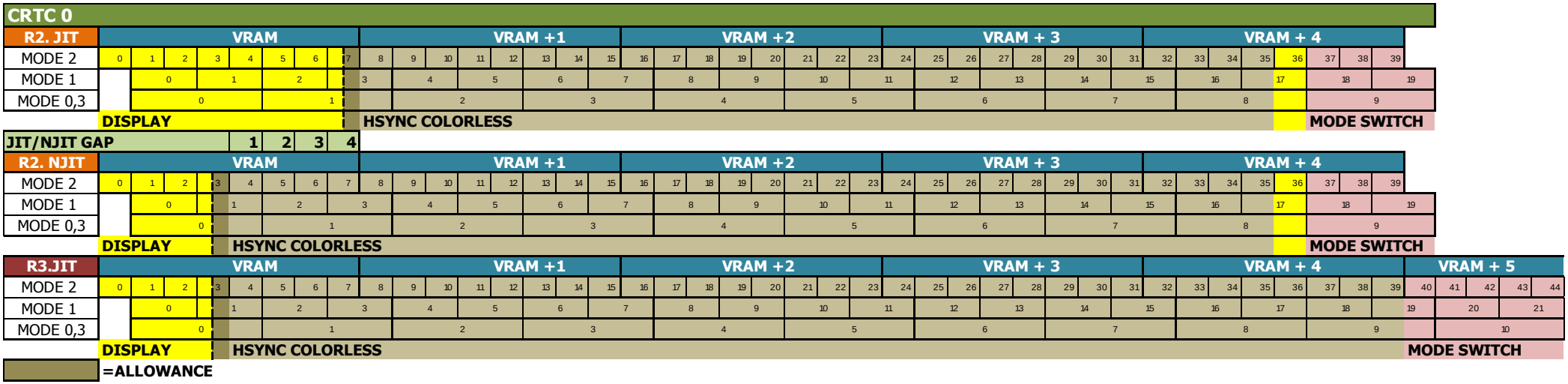
The **R3.JIT** is also of great interest insofar as **it allows the modification of the duration of the "official" 4 µsec of the C-HSYNC monitor with a gap of 0.25 µsec instead of the 0.5 µsec** used "classically" to perform horizontal scrolling at byte level (These 2 values are not exact because of the GA CSYNC algorithm. See Chapter 16.2.3 page 159).

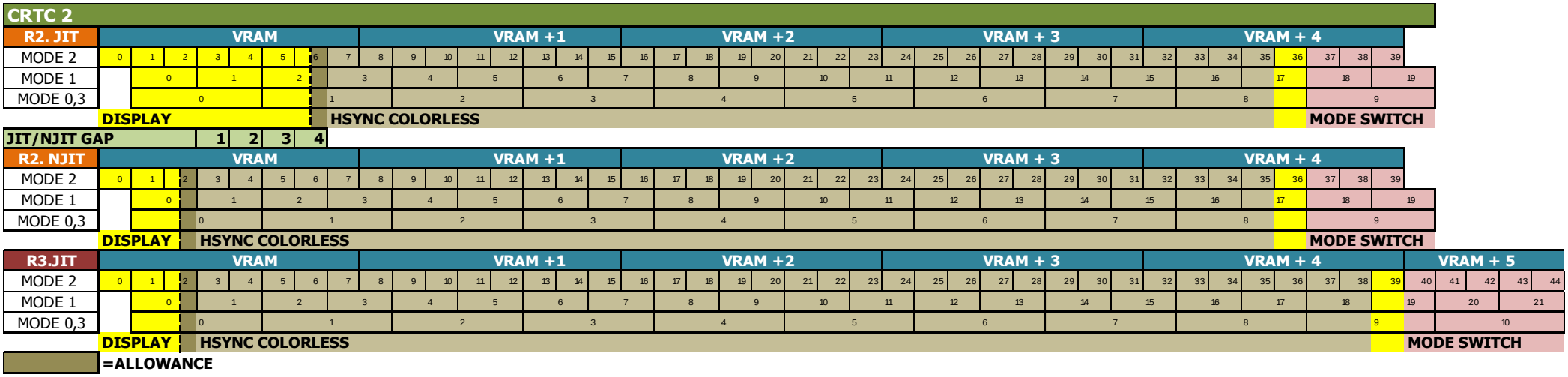
The display shutdown delay achieved with **R2.JIT** does not change anything when the display is reactivated, unlike the **R3.JIT**. The GAP between **R2.JIT** and **R2.NJIT** is **3 Pixel-M2 (0.1875 µsec) on CRTC 1** and **4 Pixel-M2 on CRTC's 0 and 2 (0.25 µsec)**.

9.3.4.2 HSYNC UNDER THE MICROSCOPE

The diagrams on the following pages indicate, according to the CRTC's:

- The moment when the display stops when a HSYNC has been requested (with however a tolerance of ½ Pixel-M2 (0.03125 µsec)) using **R2.JIT** or **R2.NJIT**.
- The gap in Pixel-M2 between an **R2.JIT** and **R2.NJIT** technique.
- The moment when the display resumes for **1 Pixel-M2** (area indicated in yellow).
- When the graphics mode changes (shown in pink).
- When the display stops when a **HSYNC** is stopped using **R3.JIT**.


CRTC 0
R2. JIT
VRAM
VRAM +1
VRAM +2
VRAM + 3
VRAM + 4
MODE 2
MODE 1
MODE 0,3
DISPLAY
HSYNC COLORLESS
MODE SWITCH
JIT/NJIT GAP
1
2
3
4
R2. NJIT
VRAM
VRAM +1
VRAM +2
VRAM + 3
VRAM + 4
MODE 2
MODE 1
MODE 0,3
DISPLAY
HSYNC COLORLESS
MODE SWITCH
R3.JIT
VRAM
VRAM +1
VRAM +2
VRAM + 3
VRAM + 4
VRAM + 5
MODE 2
MODE 1
MODE 0,3
DISPLAY
HSYNC COLORLESS
MODE SWITCH
=ALLOWANCE


CRTC 2
R2. JIT
VRAM
VRAM +1
VRAM +2
VRAM + 3
VRAM + 4
MODE 2
MODE 1
MODE 0,3
DISPLAY
HSYNC COLORLESS
MODE SWITCH
JIT/NJIT GAP
1
2
3
4
R2. NJIT
VRAM
VRAM +1
VRAM +2
VRAM + 3
VRAM + 4
MODE 2
MODE 1
MODE 0,3
DISPLAY
HSYNC COLORLESS
MODE SWITCH
R3.JIT
VRAM
VRAM +1
VRAM +2
VRAM + 3
VRAM + 4
VRAM + 5
MODE 2
MODE 1
MODE 0,3
DISPLAY
HSYNC COLORLESS
MODE SWITCH
=ALLOWANCE

CRTC 1

R2. JIT	VRAM								VRAM +1								VRAM +2								VRAM + 3								VRAM + 4								
MODE 2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	
MODE 1		0		1		2			3	4		5		6		7	8		9		10		11		12		13		14		15		16		17		18		19		
MODE 0,3		0				1				2				3			4				5				6				7			8					9			
	DISPLAY								HSYNC COLORLESS																												MODE SWITCH						

JIT/NJIT GAP	1	2	3

R2. NJIT	VRAM								VRAM +1								VRAM +2								VRAM + 3								VRAM + 4								
MODE 2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	
MODE 1		0		1		2		3		4		5		6		7	8		9		10		11		12		13		14		15		16		17		18		19		
MODE 0,3		0				1				2				3			4				5				6				7			8					9			
	DISPLAY					HSYNC COLORLESS																																	MODE SWITCH					

R3.JIT	VRAM								VRAM +1								VRAM +2								VRAM + 3								VRAM + 4								VRAM + 5				
MODE 2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
MODE 1		0		1		2		3		4		5		6		7	8		9		10		11		12		13		14		15		16		17		18		19		20		21	
MODE 0,3		0				1				2				3			4				5				6				7			8					9				10			
	DISPLAY					HSYNC COLORLESS																																	MODE SWITCH								

	=ALLOWANCE

CRTC 4

R2. NJIT	VRAM +1								VRAM +2								VRAM + 3								VRAM + 4								VRAM + 5								VRAM + 6						
MODE 2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
MODE 1		0		1		2		3		4		5		6		7	8		9		10		11		12		13		14		15		16		17		18		19		20		21		22		23
MODE 0,3		0				1				2				3			4				5				6				7			8					9				10				11	
	DISPLAY								HSYNC COLORLESS																																		MODE SWITCH							

	=ALLOWANCE

9.3.4.3 CRTC 0, 1, 2 : COOKING OF PIXELS WITH R3.NJIT

When the GATE ARRAY switches from one graphics mode to another mode, it is already processing one-byte data according to the previous mode, and result obtained reflects its internal logic.

With **R2.NJIT**, the display stops:
- For 2 µsec on CRTC's 0 and 1 (32 Pixel-M2).
- Lasting 2.0625 µsec on CRTC 2 (33 Pixel-M2).

With **R2.JIT**, the display stops:
- Lasting 1.75 µsec on CRTC 0 (28 Pixel-M2).
- Lasting 1.8125 µSec on CRTC's 1 and 2 (29 Pixel-M2).

On CRTC's 0 and 2, the display is restored 1 Pixel-M2 before the GATE ARRAY changes the graphics mode. **The 30th Pixel-M2 displayed therefore has the characteristics of the previous mode, for its fraction corresponding to 1 Pixel-M2**.

On CRTC 1, it should be considered that the "**Hsync No Disp**" zone indicated on the diagrams is longer by 1 Pixel-M2. The first Pixel-M2 obtained when the display is activated is only displayed for CRTC's 0, 2 and 4. CRTC 1 does not display this pixel.
This additional pixel is symbolized in purple on the following diagrams.

There is a notable difference between the GATE ARRAY 40007/40008 and the 40010. Indeed, in certain situations, when bits of the data in VRAM have already been used in the calculation of a colour number of the pixels of the starting mode, these bits are considered to be 0 (GA 40010) or 1 (GA 40008) for the calculation of the new graphic mode colour numbers.

9.3.4.3.1 MODE 2 TO MODE 0.1.2.3

On a line displayed in MODE 2, the data display resumes:
- On CRTC's 0 and 2, from the **5th Pixel-M2 of the 5th VRAM byte**.
- On CRTC 1, from the **6th Pixel-M2 of the 5th VRAM byte**.

On CRTC's 0 and 2, the first pixel displayed is in MODE 2 (PEN 1 or 0).

Since the MODE 2 pixels are "ahead" of the pixels of the other modes, the GATE ARRAY will display **4 new Pixel-M2 after the 5th**, unless the mode has not changed.
This represents the display of 9 M2-pixels from a single byte.

The GATE ARRAY considers that its bit counter is not at 0 but at 5.

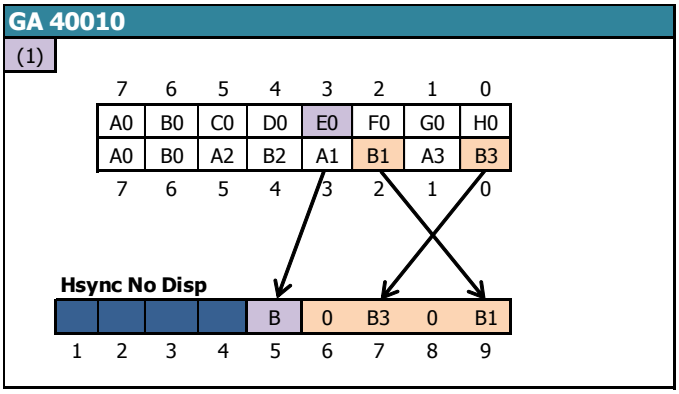
From the 6th bit of the byte read in VRAM, it "starts" processing the calculation of the colour number as if they were the 1st bits of a byte. Thus it considers bits b2,b1 and b0 of the byte in VRAM as those it would have encountered in bits b7,b6 and b5 of this same byte.

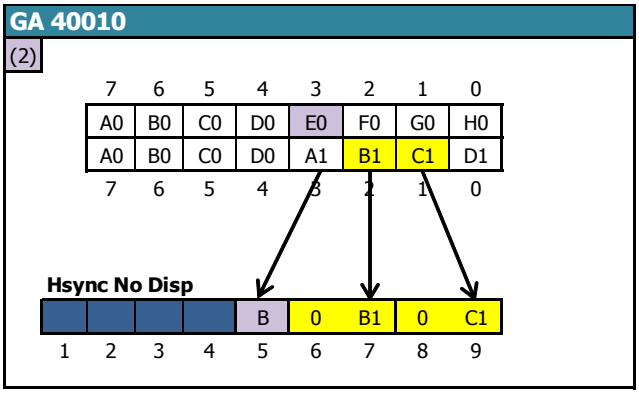
Bits missed by the GATE ARRAY 40010 are considered 0, which limits colour number combinations:
- **MODE 2 to MODE 0**: Colour numbers **0, 1, 4, 5**
- **MODE 2 to MODE 1**: Colour numbers **0, 1**
- **MODE 2 to MODE 3**: Colour numbers **0, 1**

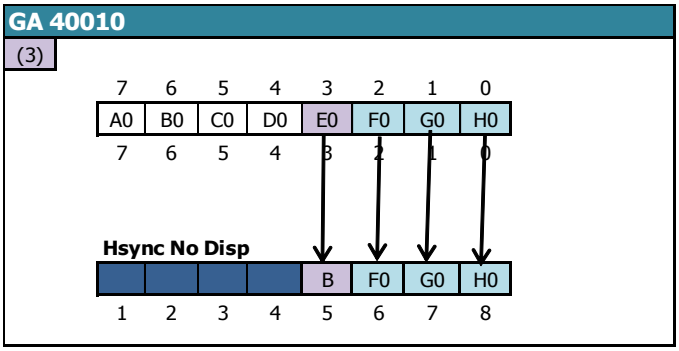
The table below describes the interpretation by the GATE ARRAY 40010 of the VRAM byte processed when the redisplay is activated, according to the new graphics mode requested.

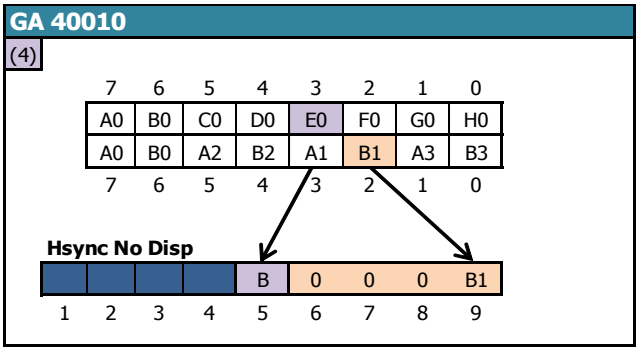
Mode to Mode		VRAM Byte							
2	0	b7	b6	b5	b4	b3	b2	b1	b0
2	1	b7	b6	b5	b4	b3	b2	b1	b0
2	2	b7	b6	b5	b4	b3	b2	b1	b0
2	3	b7	b6	b5	b4	b3	b2	b1	b0

GA 40010 : Displayed Pixels								
Hsync No Disp	(1)	0	b0	0	b2			
Hsync No Disp	(2)	0	b2	0	b1			
Hsync No Disp	(3)	b2	b1	b0				
Hsync No Disp	(4)	0	0	0	b2			
1	2	3	4	5	6	7	8	9









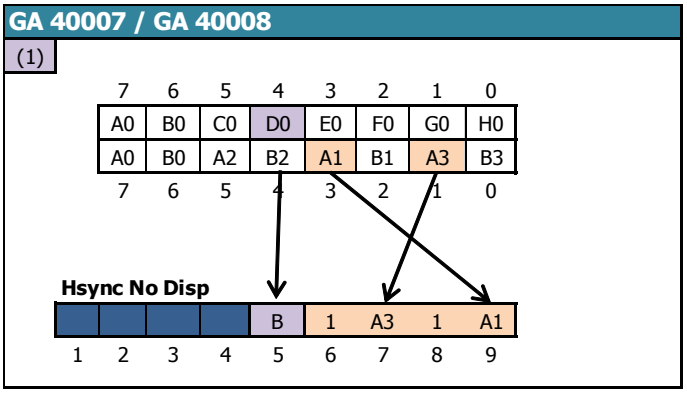
The GATE ARRAY 40007 and 40008 are ahead of 0.0625 µsec compared to the 40010 when they recover the bits allowing for the constitution of the colour number. Missed bits are considered to be 1, which limits colour number combinations:

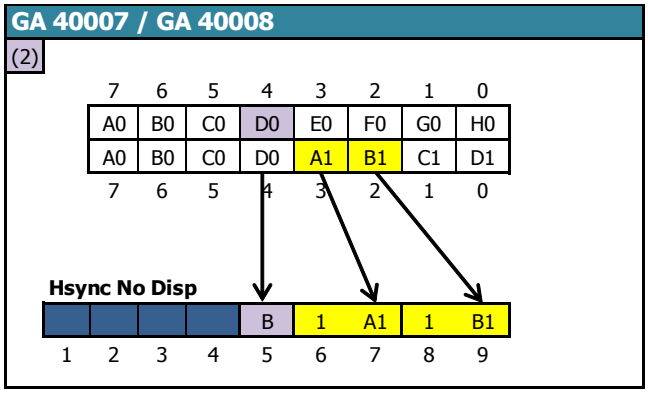
- **MODE 2 to MODE 0**: Colour numbers **10, 11, 14, 15**
- **MODE 2 to MODE 1**: Colour numbers **2, 3**
- **MODE 2 to MODE 3**: Colour numbers **2, 3**

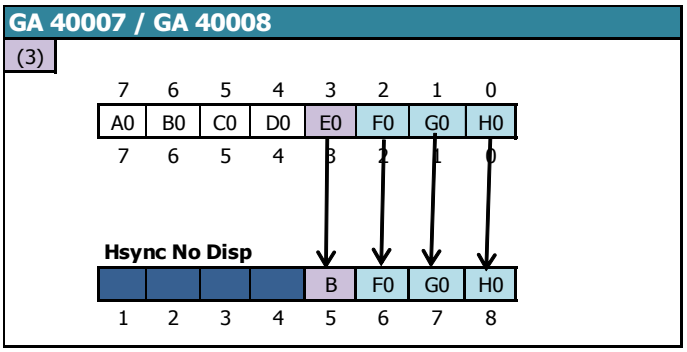
The table below describes the interpretation by the **GATE ARRAY 40007 and 40008** of the VRAM byte processed when the redisplay is activated, according to the new graphics mode requested.

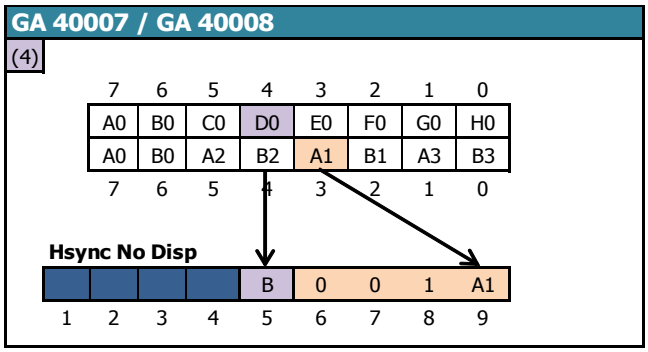
Mode to Mode		VRAM Byte							
2	0	b7	b6	b5	b4	b3	b2	b1	b0
2	1	b7	b6	b5	b4	b3	b2	b1	b0
2	2	b7	b6	b5	b4	b3	b2	b1	b0
2	3	b7	b6	b5	b4	b3	b2	b1	b0

GA 40007/8 : Displayed Pixels								
Hsync No Disp	(1)	1	b1	1	b3			
Hsync No Disp	(2)	1	b3	1	b2			
Hsync No Disp	(3)	b2	b1	b0				
Hsync No Disp	(4)	0	0	1	b3			
1	2	3	4	5	6	7	8	9









9.3.4.3.2 MODE 0 TO MODE 0.1.2.3

On a line displayed in MODE 0, the data display resumes:

- On CRTC's 0 and 2, from the **4th Pixel-M2 of the 5th VRAM byte.**
- On CRTC 1, from the **5th Pixel-M2 of the 5th VRAM byte**.

On CRTC's 0 and 2, the first Pixel-M2 displayed is the last of the **4 pixel-M2 of the MODE 0 pixel** displayed in its original colour (i.e. a colour number between 0 and 15). In other words, only ¼ of the MODE 0 pixel is displayed before the GATE ARRAY displays the following pixels.

If MODE 2 is the new mode required, then the GATE ARRAY will display 3 new Pixel-M2's after the 4th which represents in total the display of 7 pixels-M2 from the same byte.

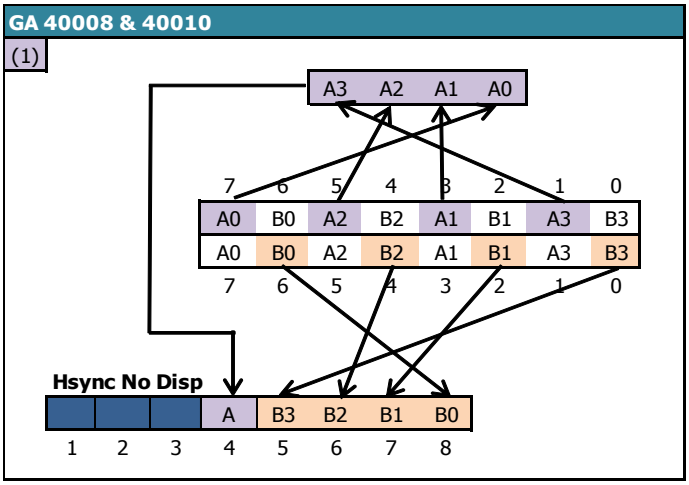
If the new required mode is not MODE 2 then the GATE ARRAY will display 4 new Pixel-M2's after the 4th.

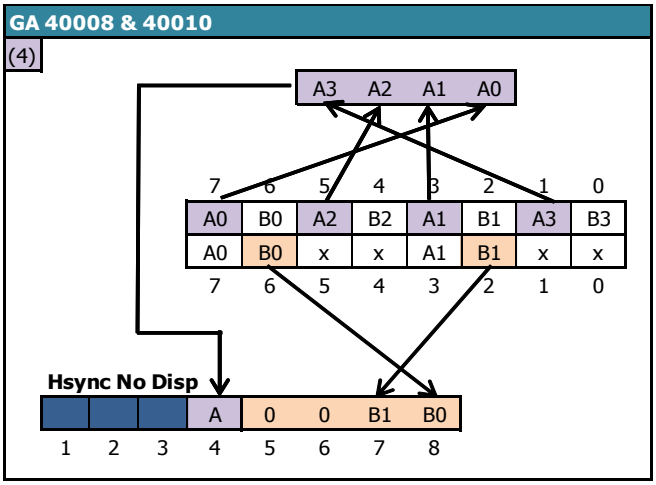
Finally, if the new mode required is 1 or 2, then the GATE ARRAY, to calculate the new colour numbers, "reuses" bits already used to calculate the colour number of the previous pixel.

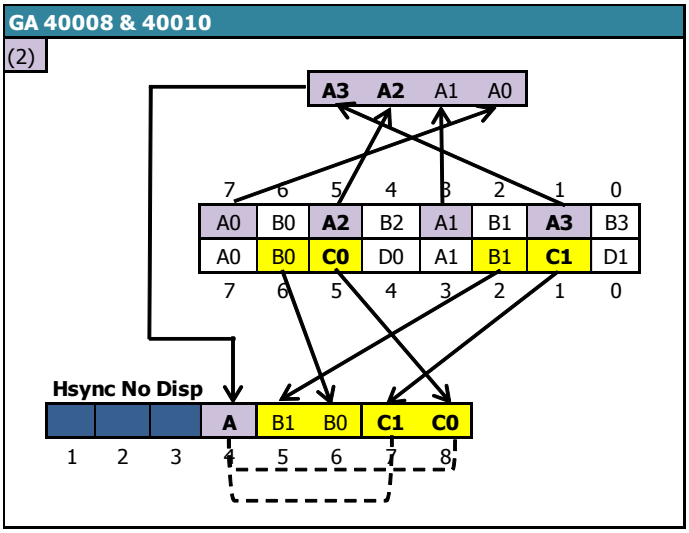
The table below describes the interpretation by the GATE ARRAY 40007, 40008 and 40010 of the VRAM byte processed when the redisplay is activated, according to the new graphic mode requested.

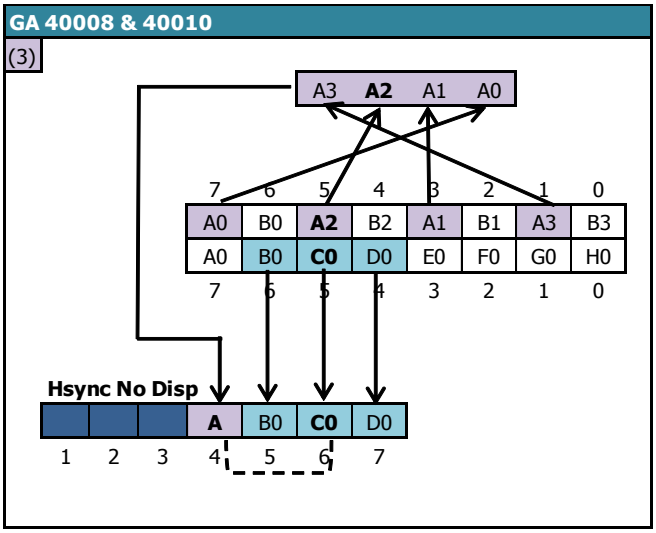
Mode to Mode		VRAM Byte							
0	0	b7	b6	b5	b4	b3	b2	b1	b0
0	1	b7	b6	b5	b4	b3	b2	b1	b0
0	2	b7	b6	b5	b4	b3	b2	b1	b0
0	3	b7	b6	b5	b4	b3	b2	b1	b0

Displayed Pixels							
Hsync No Disp	(1)	b0	b4	b2	b6		
Hsync No Disp	(2)	b2	b6	b1	b5		
Hsync No Disp	(3)	b6	b5	b4			
Hsync No Disp	(4)	0	0	b2	b6		
1	2	3	4	5	6	7	8









When switching from MODE 0 to mode 1 or 2, the common use of the data bits of the byte read in VRAM creates a constraint by linking the colour numbers of the Pixel-M2 displayed in position 4 with that of the pixels located after.

- **MODE 0 to MODE 1**: The colour number of the last pixel in MODE 1 corresponds to bits 3 and 4 of the colour number of the 1st pixel in MODE 0.
  - 1st pixel-M2 N° colours 0,1,2,3 = 2nd pixel MODE 1 with colour number 0.
  - 1st pixel-M2 N° colours 4,5,6,7 = 2nd pixel MODE 1 with colour number 1.

- 1st pixel-M2 N° colours 8,9,10,11 = 2nd pixel MODE 1 with colour number 2.
- 1st pixel-M2 N° colours 12,13,14,15 = 2nd pixel MODE 1 with colour number 2.

- **MODE 0 to MODE 2**: The colour number of the 2nd pixel in MODE 2 corresponds to bit 2 of the colour number of the 1st pixel in MODE 0.
  - 1st pixel-M2 No. of colours 0..3, 8..11 = 2nd pixel MODE 2 of colour 0
  - 1st pixel-M2 No. of colours 4..7, 12..15 = 2nd pixel MODE 2 of colour 1

9.3.4.3.3 MODE 1 TO MODE 0.1.2.3

On a line displayed in MODE 1, the data display resumes:

- On CRTC's 0 and 2, from the **4th Pixel-M2 of the 5th VRAM byte**.
- On CRTC 1, from the **5th Pixel-M2 of the 5th VRAM byte**.

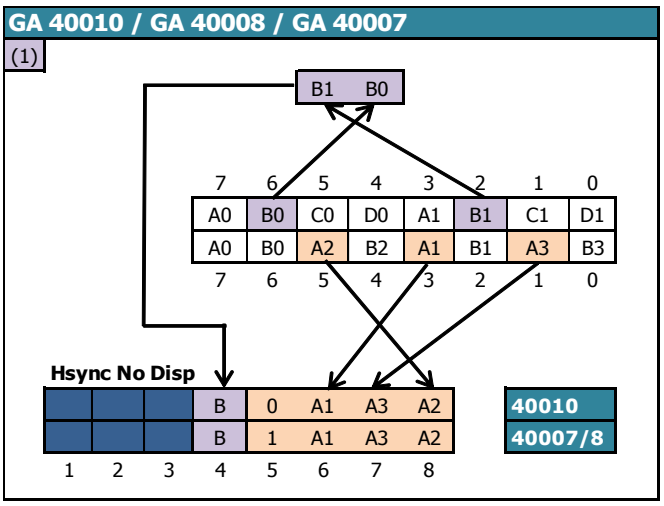
On CRTC's 0 and 2, the first Pixel-M2 displayed is the last of the **2 pixel-M2 of the MODE 1 pixel** displayed in its original colour (i.e. a colour number between 0 and 3). In other words, only half of the MODE 1 pixel is displayed before the GATE ARRAY displays the following pixels.

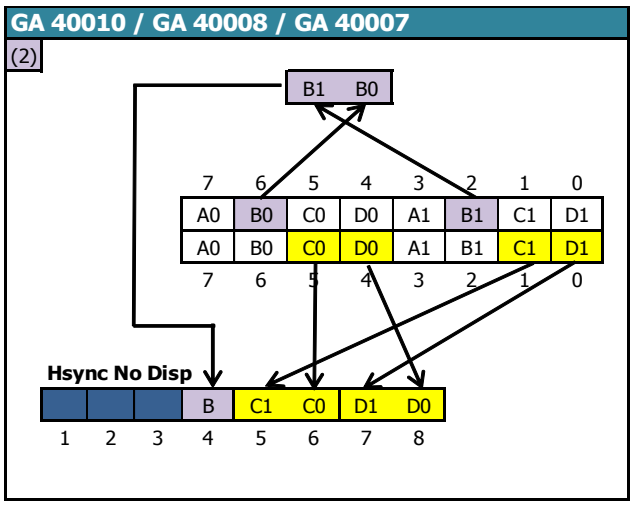
If MODE 2 is the new requested mode, then the GATE ARRAY will display 3 new Pixel-M2's after the 4th, which represents in total the display of 7 pixel-M2's from the same byte.
If the new required mode is different from MODE 2 then the GATE ARRAY will display 4 new Pixel-M2's after the 4th.

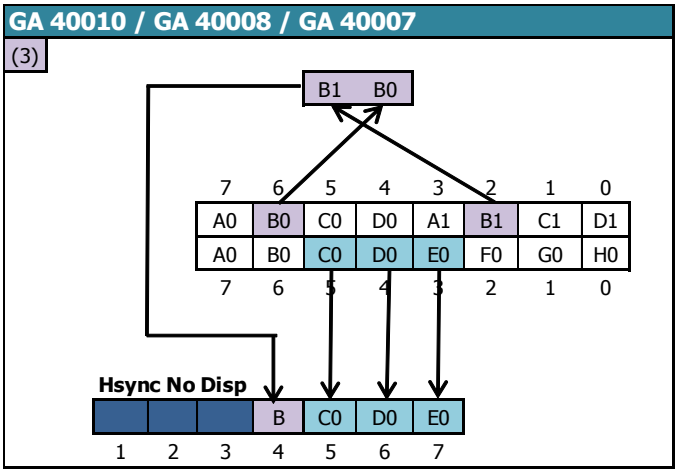
The following table describes the interpretation by the GATE ARRAY 40007, 40008 and 40010 of the VRAM byte processed when the display is reactivated, according to the new graphics mode requested.

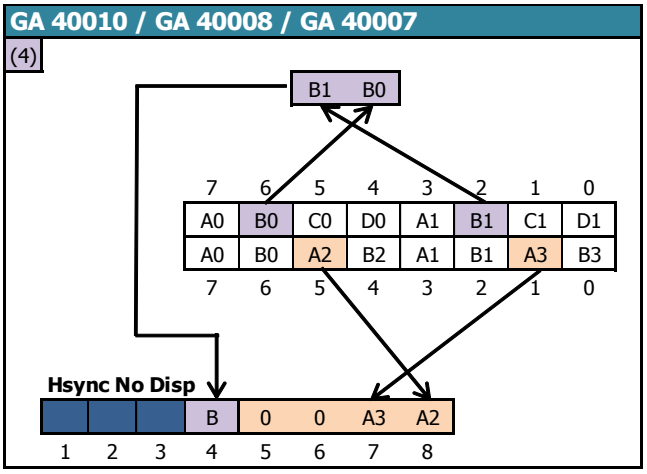
Mode to Mode		VRAM Byte							
1	0	b7	b6	b5	b4	b3	b2	b1	b0
1	1	b7	b6	b5	b4	b3	b2	b1	b0
1	2	b7	b6	b5	b4	b3	b2	b1	b0
1	3	b7	b6	b5	b4	b3	b2	b1	b0

Displayed Pixels							
Hsync No Disp	(1)	0/1	b3	b1	b5		
Hsync No Disp	(2)	b1	b5	b0	b4		
Hsync No Disp	(3)	b5	b4	b3			
Hsync No Disp	(4)	0	0	b1	b5		
1	2	3	4	5	6	7	8









9.3.4.3.4 MODE 3 TO MODE 0.1.2.3

On a line displayed in MODE 3, the data display resumes:

- On CRTC's 0 and 2, from the **4th Pixel-M2 of the 5th VRAM byte**.
- On CRTC 1, from the **5th Pixel-M2 of the 5th VRAM byte**.

On CRTC's 0 and 2, the first Pixel-M2 displayed is the last of the **4 pixel-M2 of the MODE 3 pixel** displayed in its original colour (i.e. a colour number between 0 and 3). In other words, only ¼ of the MODE 3 pixel is displayed before the GATE ARRAY displays the following pixels.

If MODE 2 is the new required then the GATE ARRAY will display 3 new Pixel-M2's after the 4th which represents in total the display of 7 pixel-M2's from the same byte.

If the new required mode is different from MODE 2 then the GATE ARRAY will display 4 new Pixel-M2's after the 4th.

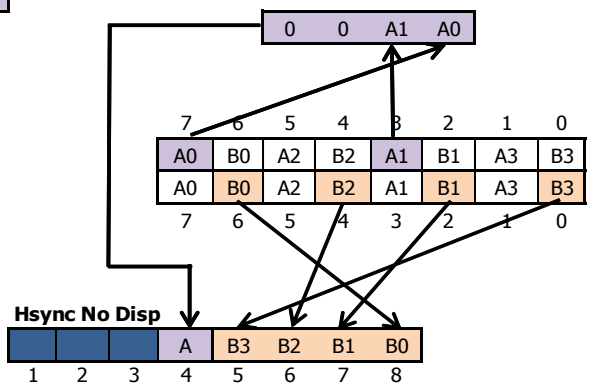
The following table describes the GATE ARRAY's interpretation of the VRAM byte processed when redisplay is enabled, depending on the new graphics mode requested.

Mode to Mode		VRAM Byte							
3	0	b7	b6	b5	b4	b3	b2	b1	b0
3	1	b7	b6	b5	b4	b3	b2	b1	b0
3	2	b7	b6	b5	b4	b3	b2	b1	b0
3	3	b7	b6	b5	b4	b3	b2	b1	b0

Displayed Pixels							
Hsync No Disp	(1)	b0	b4	b2	b6		
Hsync No Disp	(2)	b2	b6	b1	b5		
Hsync No Disp	(3)	b6	b5	b4			
Hsync No Disp	(4)	0	0	b2	b6		
1	2	3	4	5	6	7	8

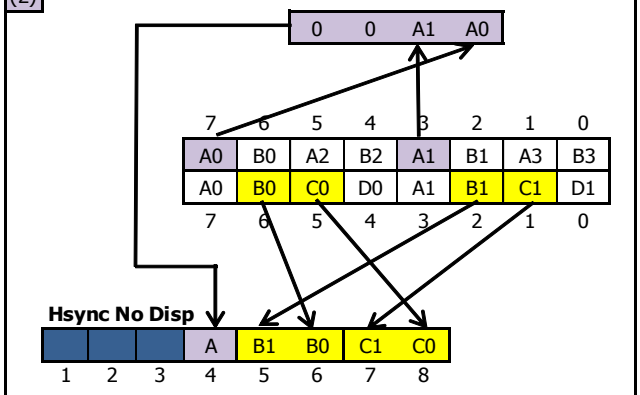
GA 40010 / GA 40008 / GA 40007

(1)



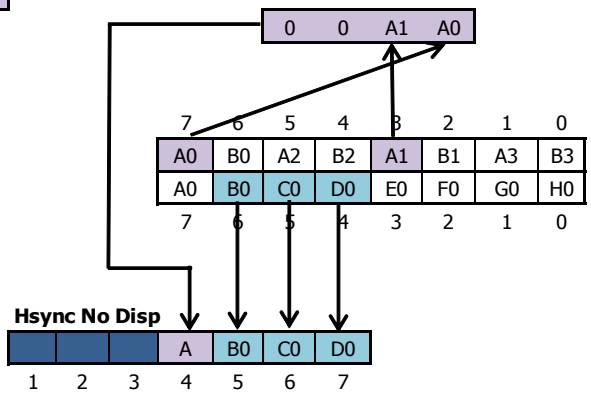
GA 40010 / GA 40008 / GA 40007

(2)



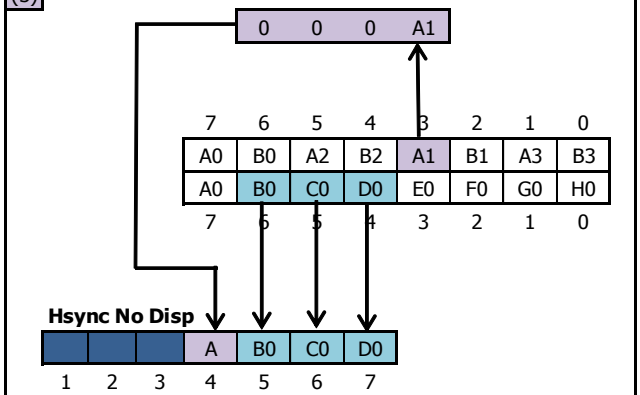
GA 40010

(3)



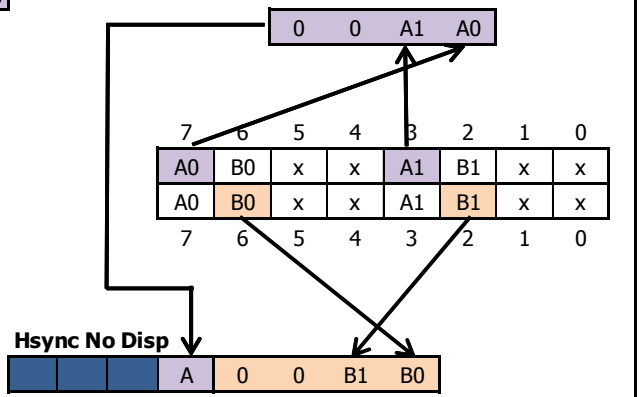
GA 40008 / GA 40007

(3)



GA 40010 / GA 40008 / GA 40007

(4)



9.3.4.4 CRTC 0, 1, 2 : PIXEL COOKING WITH R3.JIT

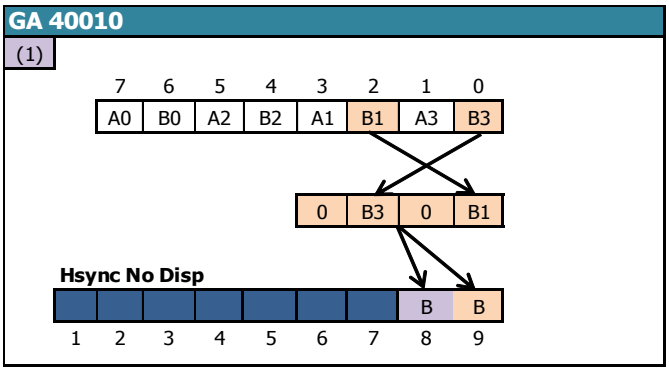
The **R3.JIT** technique can delay the end of a HSYNC by 0.25 µsec.

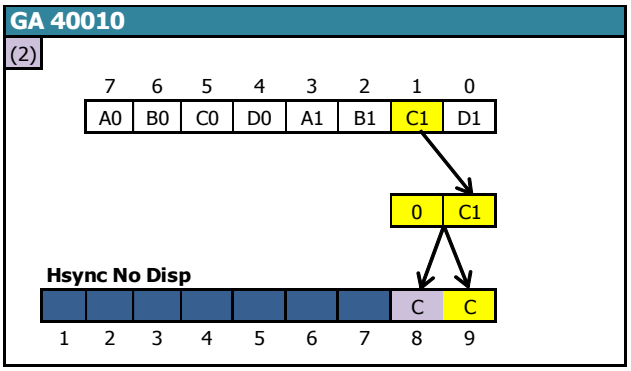
On CRTC's 0 and 1, the **HSYNC** ceases after the last black Pixel-M2 of the HSYNC.
On CRTC 2, the **HSYNC** stops on the last Pixel-M2 of the HSYNC and allows the visualization of an additional pixel.
This additional pixel is symbolized in purple on the following diagrams

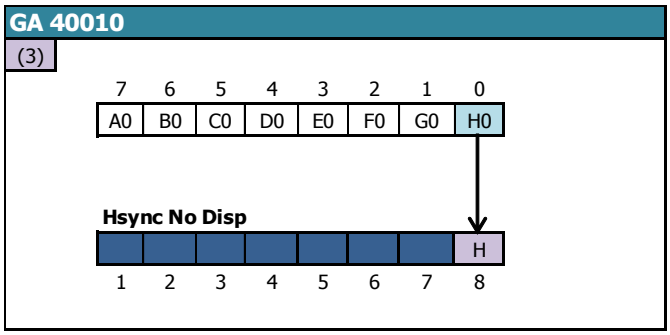
9.3.4.4.1 MODE 2 TO MODE 0.1.2.3

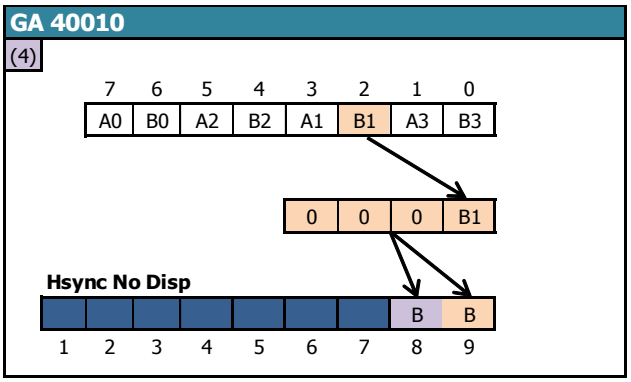
Mode to Mode		VRAM Byte							
2	0	b7	b6	b5	b4	b3	b2	b1	b0
2	1	b7	b6	b5	b4	b3	b2	b1	b0
2	2	b7	b6	b5	b4	b3	b2	b1	b0
2	3	b7	b6	b5	b4	b3	b2	b1	b0

GA 40010 : Displayed Pixels									
Hsync No Disp								(1)	(1)
Hsync No Disp								(2)	b1
Hsync No Disp								(3)	
Hsync No Disp								(4)	(4)
1	2	3	4	5	6	7	8	9	



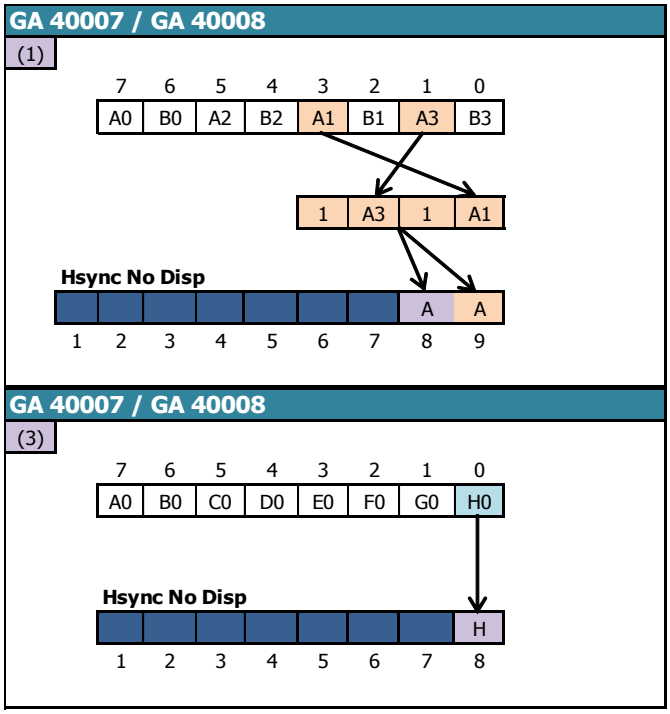


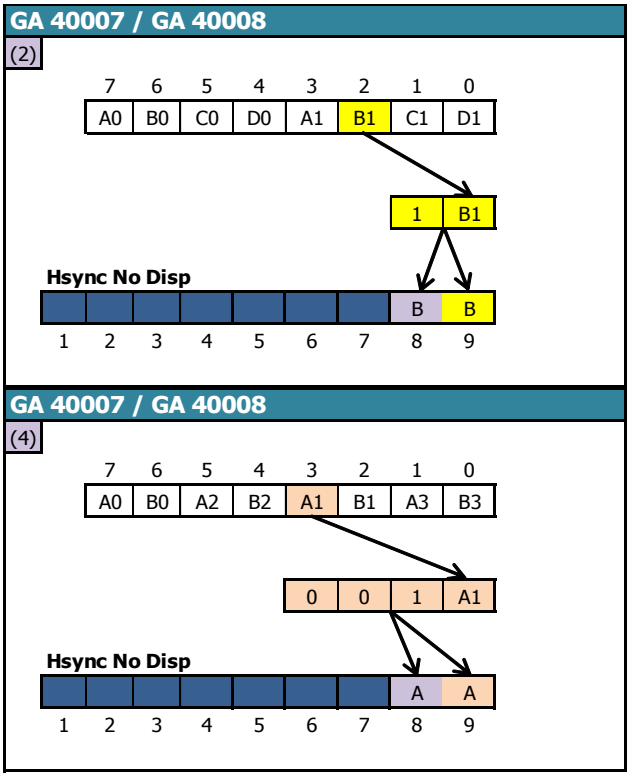




Mode to Mode		VRAM Byte							
2	0	b7	b6	b5	b4	b3	b2	b1	b0
2	1	b7	b6	b5	b4	b3	b2	b1	b0
2	2	b7	b6	b5	b4	b3	b2	b1	b0
2	3	b7	b6	b5	b4	b3	b2	b1	b0

GA 40007/8 : Displayed Pixels									
Hsync No Disp								(1)	(1)
Hsync No Disp								(2)	b2
Hsync No Disp								(3)	
Hsync No Disp								(4)	(4)
1	2	3	4	5	6	7	8	9	

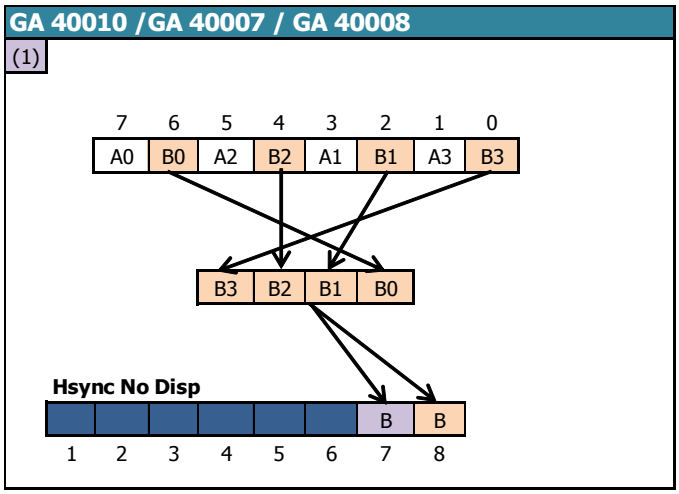


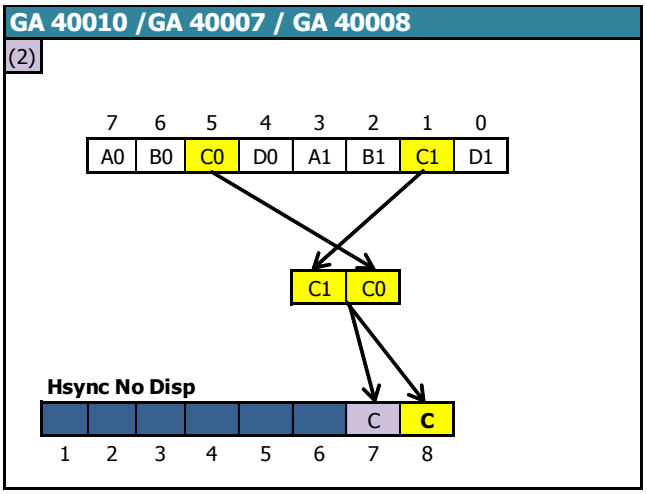


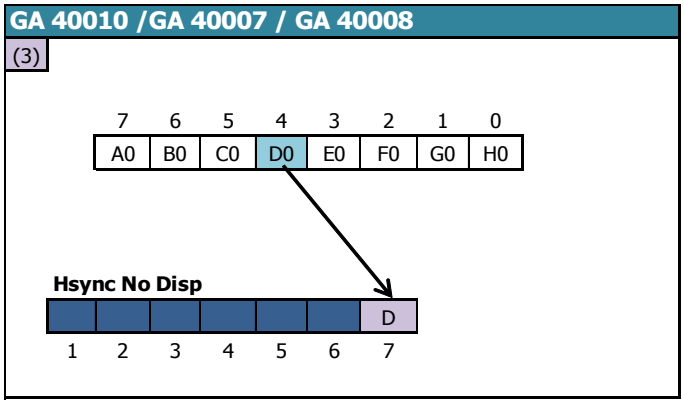
9.3.4.4.2 MODE 0 TO MODE 0.1.2.3

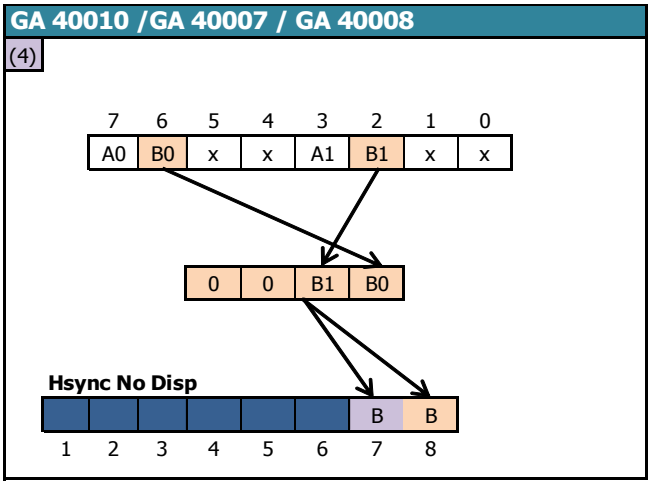
Mode to Mode		VRAM Byte							
0	0	b7	b6	b5	b4	b3	b2	b1	b0
0	1	b7	b6	b5	b4	b3	b2	b1	b0
0	2	b7	b6	b5	b4	b3	b2	b1	b0
0	3	b7	b6	b5	b4	b3	b2	b1	b0

Displayed Pixels		
Hsync No Disp	(1)	(1)
Hsync No Disp	(2)	(2)
Hsync No Disp	(3)	
Hsync No Disp	(4)	(4)
1 2 3 4 5 6 7 8		



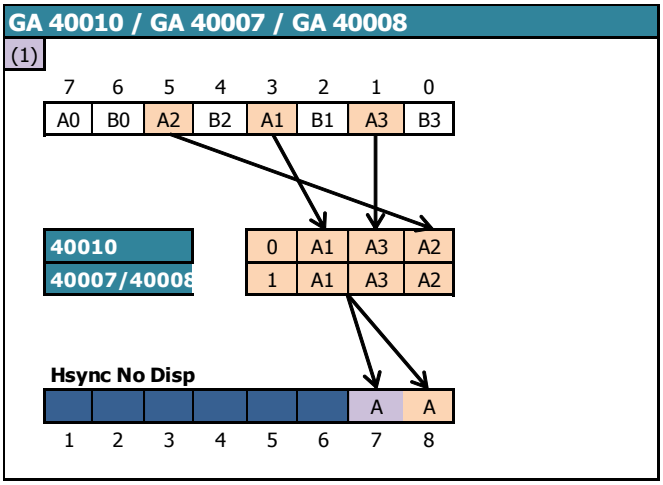


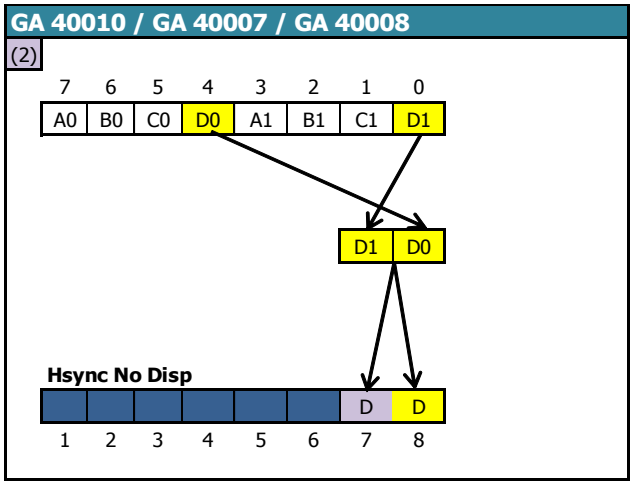


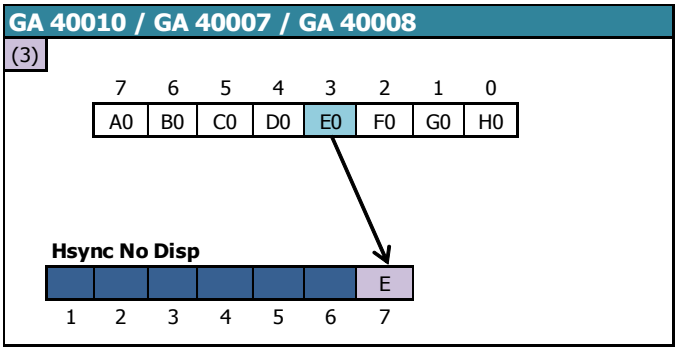


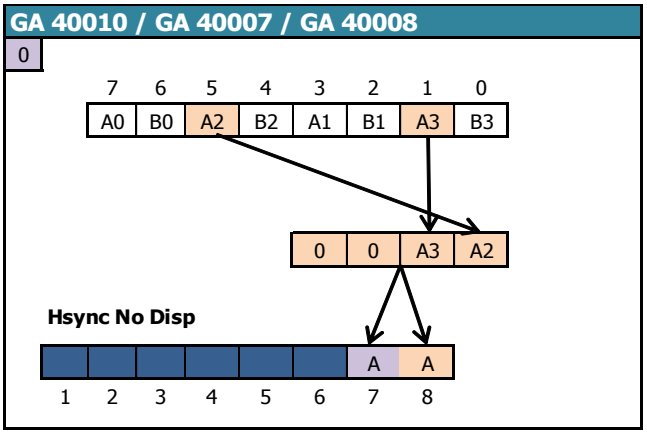
9.3.4.4.3 MODE 1 TO MODE 0.1.2.3

Mode to Mode		VRAM Byte								Displayed Pixels		
1	0	b7	b6	b5	b4	b3	b2	b1	b0	Hsync No Disp	(1)	(1)
1	1	b7	b6	b5	b4	b3	b2	b1	b0	Hsync No Disp	(2)	(2)
1	2	b7	b6	b5	b4	b3	b2	b1	b0	Hsync No Disp	(3)	
1	3	b7	b6	b5	b4	b3	b2	b1	b0	Hsync No Disp	(4)	(4)
										1 2 3 4 5 6 7 8		





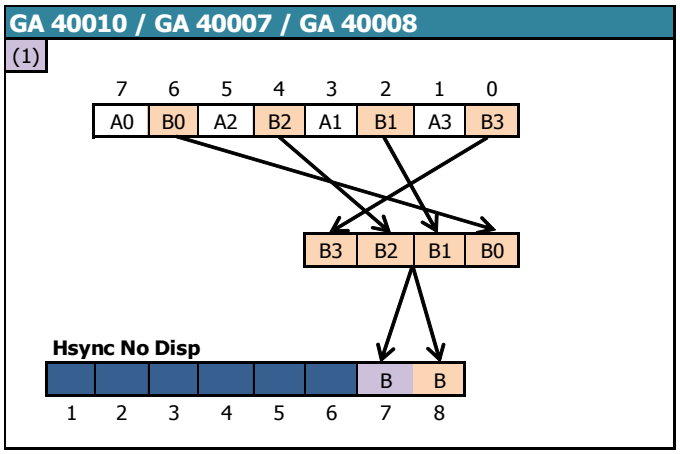


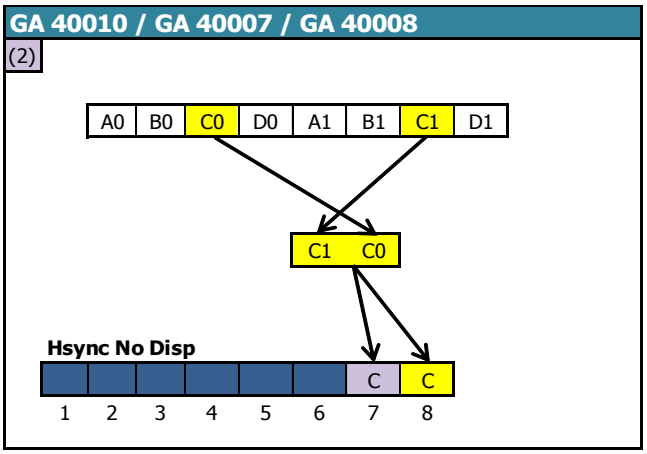


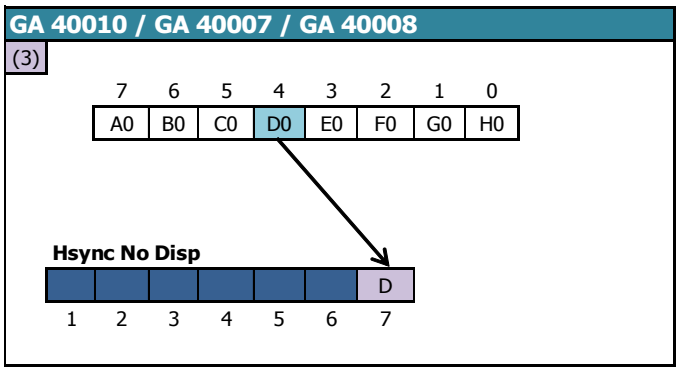
9.3.4.4.4 MODE 3 TO MODE 0.1.2.3

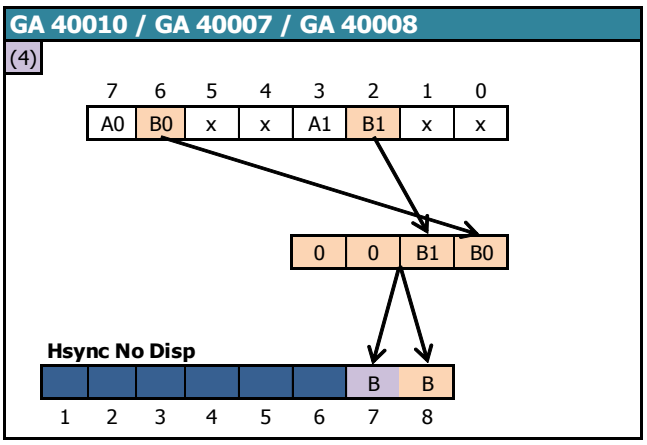
Mode to Mode		VRAM Byte							
3	0	b7	b6	b5	b4	b3	b2	b1	b0
3	1	b7	b6	b5	b4	b3	b2	b1	b0
3	2	b7	b6	b5	b4	b3	b2	b1	b0
3	3	b7	b6	b5	b4	b3	b2	b1	b0

Displayed Pixels								
Hsync No Disp						(1)	(1)	
Hsync No Disp						(2)	(2)	
Hsync No Disp						(3)		
Hsync No Disp						(4)	(4)	
1	2	3	4	5	6	7	8	









9.3.4.5 CRTC 4 : PIXEL COOKING

When the PRE-ASIC switches from one graphics mode to another mode, it is already processing one-byte data according to the previous mode, and the final result obtained reveals its internal logic.

The display stops for 2 µsec (32 Pixel-M2).
The display is restored 1 Pixel-M2 before the PRE-ASIC changes the graphics mode.

The 36th Pixel-M2 displayed therefore has the characteristics of the old mode, for its fraction corresponding to 1 Pixel-M2.

9.3.4.5.1 MODE 2 TO MODE 0.1.2.3

On a line displayed in MODE 2, data display resumes from the **3rd Pixel-M2 of the 7th VRAM byte**.

The first pixel displayed is in MODE 2 (PEN 1 or 0).

Since the MODE 2 pixels are "ahead" of the pixels of the other modes, the PRE-ASIC will display **6 new Pixel-M2's after the 3rd**, unless the mode has not changed.
This represents the display of 9 M2-pixels from a single byte.

The table below describes the interpretation by the PRE-ASIC of the VRAM byte processed when the display is reactivated, according to the new graphics mode requested.

Mode to Mode		VRAM Byte							
2	0	b7	b6	b5	b4	b3	b2	b1	b0
2	1	b7	b6	b5	b4	b3	b2	b1	b0
2	2	b7	b6	b5	b4	b3	b2	b1	b0
2	3	b7	b6	b5	b4	b3	b2	b1	b0

ASIC 40226: Displayed Pixels								
No Disp	b5	0.b2.b0.b4		0	b1	0	b3	
No Disp	b5	b0	b4	0	b3	0	b2	
No Disp	b5	b4	b3	b2	b1	b0		
No Disp	b5	0.0.b0.b4		0	0	0	b3	
1	2	3	4	5	6	7	8	9

9.3.4.5.2 MODE 0 TO MODE 0.1.2.3

On a line displayed in MODE 0, the data display resumes from **2nd Pixel-M2 of the 7th byte of the VRAM**.

The first Pixel-M2 displayed is the last of the **4 pixel-M2 of the MODE 0 pixel** displayed in its original colour (i.e. a colour number between 0 and 15). In other words, only ¼ of the MODE 0 pixel is displayed before the PRE-ASIC displays the following pixels.

If the new graphics mode required is MODE 2, then the PRE-ASIC displays 5 new Pixel-M2's after the 2nd, which in total represents the display of 7 pixel-M2's from the same byte.

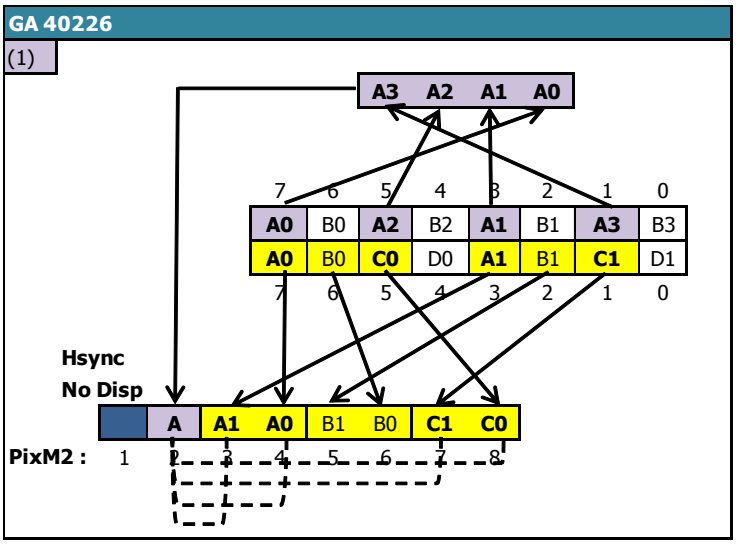
If the new requested mode is different from MODE 2, then the PRE-ASIC displays 6 new Pixel-M2's after the 2nd one.

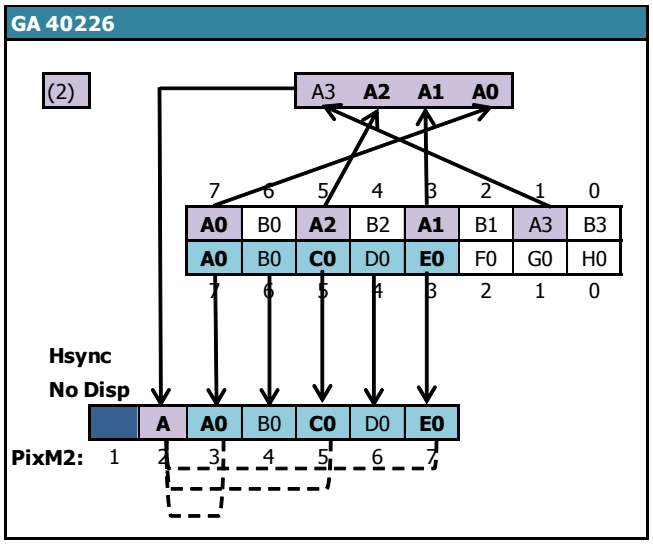
Finally, if the new required mode is 1 or 2, then the PRE-ASIC, to calculate the new colour numbers, "reuses" bits already used to calculate the colour number of the previous pixel.

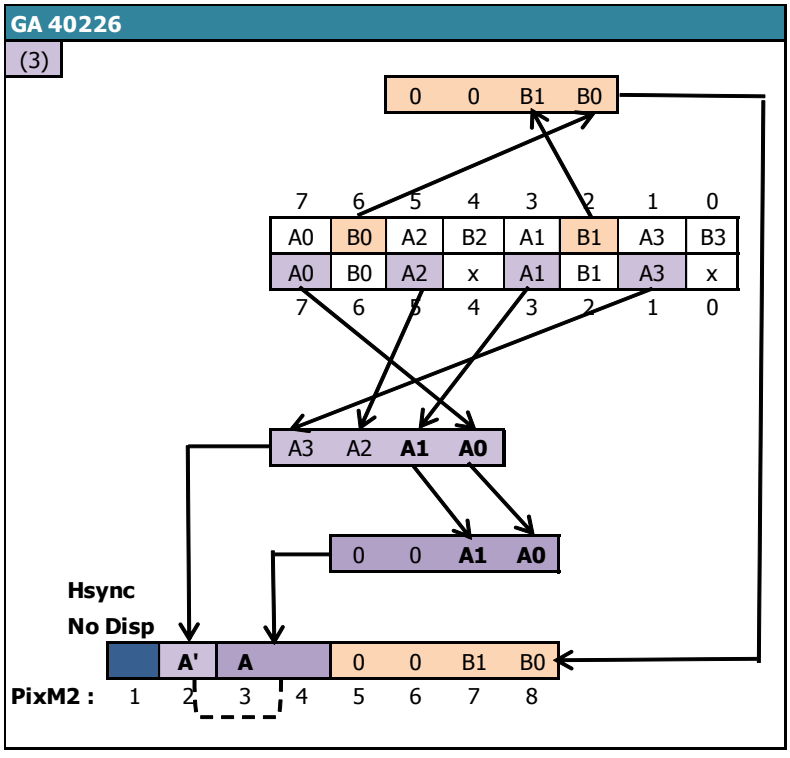
The table below describes the interpretation by the PRE-ASIC of the VRAM byte processed when the display is reactivated, according to the new graphics mode requested

Mode to Mode		VRAM Byte							
0	0	b7	b6	b5	b4	b3	b2	b1	b0
0	1	b7	b6	b5	b4	b3	b2	b1	b0
0	2	b7	b6	b5	b4	b3	b2	b1	b0
0	3	b7	b6	b5	b4	b3	b2	b1	b0

ASIC 40226: Displayed Pixels							
b1.b5.b3.b7				b0	b4	b2	b6
(1)	b3	b7		b2	b6	b1	b5
(2)	b7	b6		b5	b4	b3	
(3)	(3)			0	0	b2	b6
1	2	3	4	5	6	7	8







9.3.4.5.3 MODE 1 TO MODE 0.1.2.3

On a line displayed in MODE 1, the data display resumes from **2nd Pixel-M2 of the 7th byte of the VRAM**.

The first Pixel-M2 displayed is the last of the **2 pixel-M2 of the MODE 1 pixel** displayed in its original colour (i.e. a colour number between 0 and 3). In other words, only half of the MODE 1 pixel is displayed before the PRE-ASIC displays the following pixels.

If the new graphics mode required is MODE 2, then the PRE-ASIC displays 5 new Pixel-M2's after the 2nd, which in total represents the display of 7 pixel-M2's from the same byte.

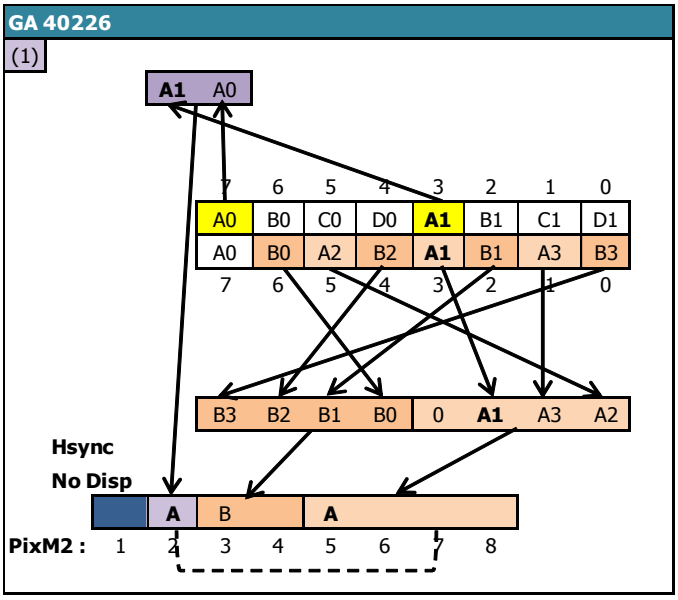
If the new requested mode is different from MODE 2, then the PRE-ASIC displays 6 new Pixel-M2's after the 2nd one.

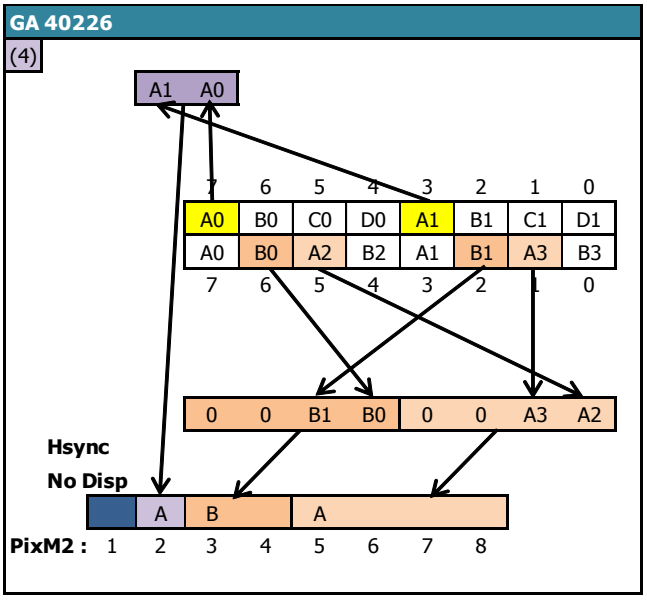
Finally, if the new mode required is 1 or 2, then the PRE-ASIC, to calculate the new colour numbers, "reuses" bits already used to calculate the colour number of the previous pixel.

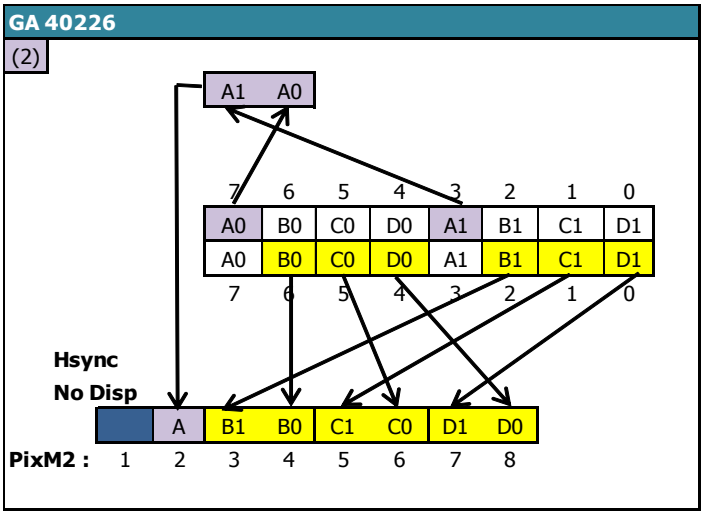
The table below describes the interpretation by the PRE-ASIC of the VRAM byte processed when the display is reactivated, according to the new graphics mode requested.

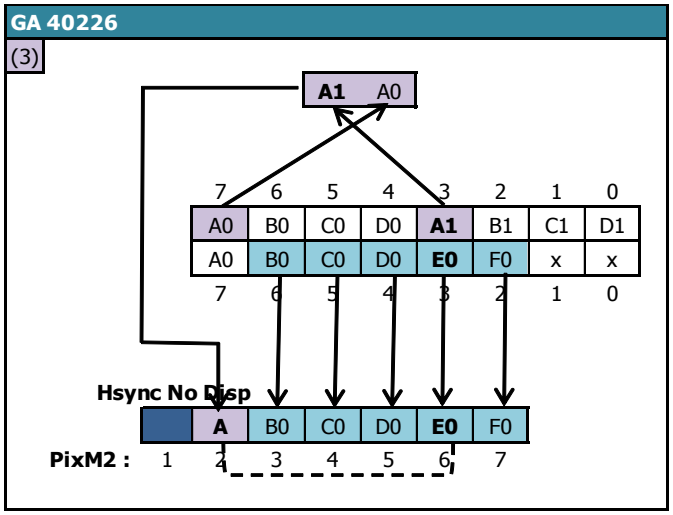
Mode to Mode		VRAM Byte							
1	0	b7	b6	b5	b4	b3	b2	b1	b0
1	1	b7	b6	b5	b4	b3	b2	b1	b0
1	2	b7	b6	b5	b4	b3	b2	b1	b0
1	3	b7	b6	b5	b4	b3	b2	b1	b0

Displayed Pixels							
(1)	b0.b4.b2.b6	0	b3	b1	b5		
(2)	b2	b6	b1	b5	b0	b4	
(3)	b6	b5	b4	b3	b2		
(4)	0.0.b2.b6	0	0	b1	b5		
1	2	3	4	5	6	7	8









9.3.4.5.4 MODE 3 TO MODE 0.1.2.3

On a line displayed in MODE 3, the data display resumes from **2nd Pixel-M2 of the 7th byte of the VRAM**.

The first Pixel-M2 displayed is the last of the **4 pixel-M2 of the MODE 3 pixel** displayed in its original colour (i.e. a colour number between 0 and 3). In other words, only the quarter of the MODE 3 pixel is displayed before the PRE-ASIC displays the following pixels.

If MODE 2 is the new graphics mode required, then the PRE-ASIC displays 5 new Pixel-M2's after the 2nd which in total represents the display of 7 pixel-M2's from the same byte.

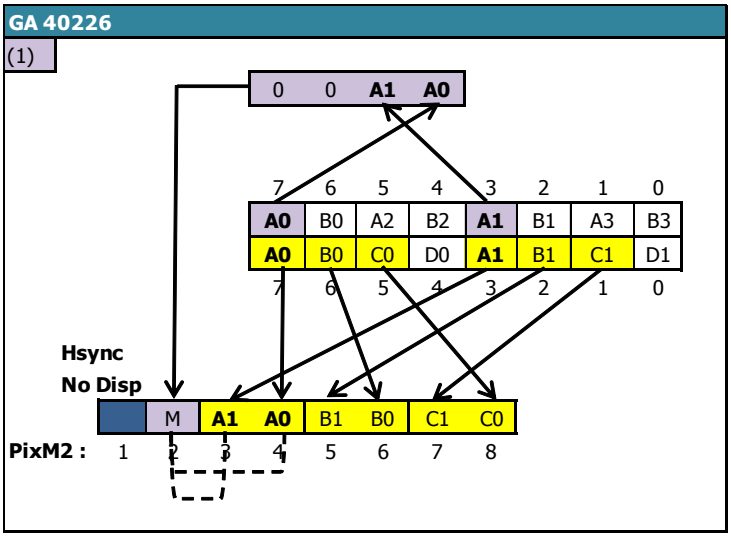
If the new requested mode is different from MODE 2 then the PRE-ASIC displays 6 new Pixel-M2's after the 2nd one.

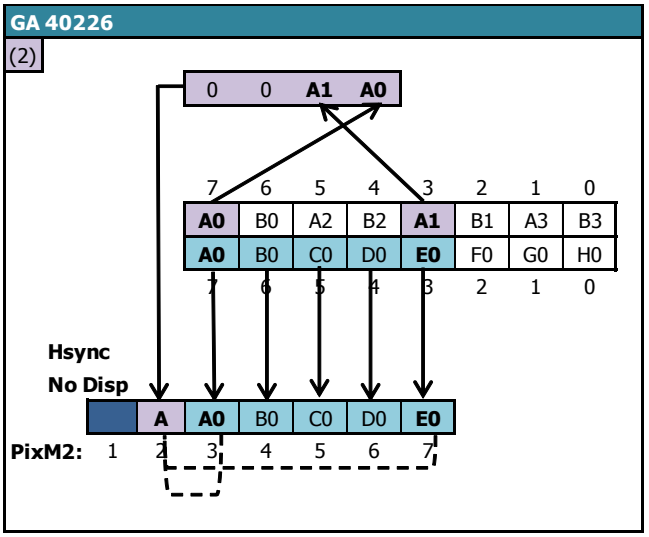
Finally, if the new mode required is 1 or 2, then the PRE-ASIC, to calculate the new colour numbers, "reuses" bits already used to calculate the colour number of the previous pixel.

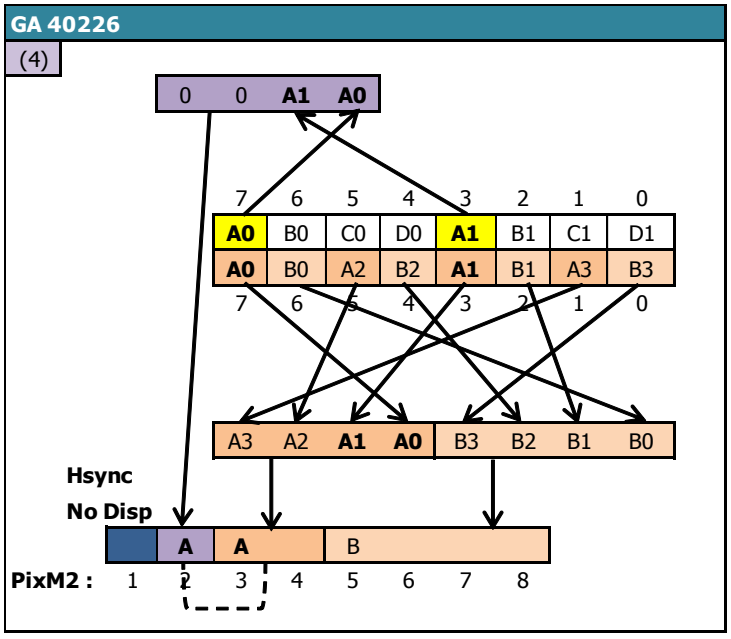
The table below describes the interpretation by the PRE-ASIC of the VRAM byte processed when the redisplay is activated, according to the new graphics mode requested.

Mode to Mode		VRAM Byte							
3	0	b7	b6	b5	b4	b3	b2	b1	b0
3	1	b7	b6	b5	b4	b3	b2	b1	b0
3	2	b7	b6	b5	b4	b3	b2	b1	b0
3	3	b7	b6	b5	b4	b3	b2	b1	b0

Displayed Pixels							
(3)	b1.b5.b3.b7		b0	b4	b2	b6	
(1)	b3	b7	b2	b6	b1	b5	
(2)	b7	b6	b5	b4	b3		
0.0.b3.b7			0	0	b2	b6	
1	2	3	4	5	6	7	8







10 COUNTER : REGISTER R9

10.1 GENERAL

Register 9 allows you to set the number of vertical lines of each character line.

In principle, the C9 counter, which determines the raster line of the character line, increments to the value of R9. When IVM interlace mode is enabled, the counting algorithm of C9 is different depending on the CRTC type (See Chapter 19.3).

When it returns to 0, C4 is incremented (and in "principle" should return to 0 when C4=R4).

The internal "row" counter C9 is connected directly to the bits 11,12 and 13 of the VRAM pointer. C9 delimits, in a space of 16 KB, 8 slices of 2 KB for 8 different possible "lines".

Extended to 64k of addressable space, this represents 8 slices of 8 KB for the 8 "lines". This principle contributes to the vertical non-linearity of the display ("curtain" effect when the memory is transferred without cutting) well known to any CPC regular.

The definition of R9 is 5 bits which results in "vertical characters" of up to 32 lines. Bits 3 and 4 of the counter are not considered in the calculation of the video pointer on C9 values that exceed 7.

However, this is information that must be considered if C9 is to count beyond 7.

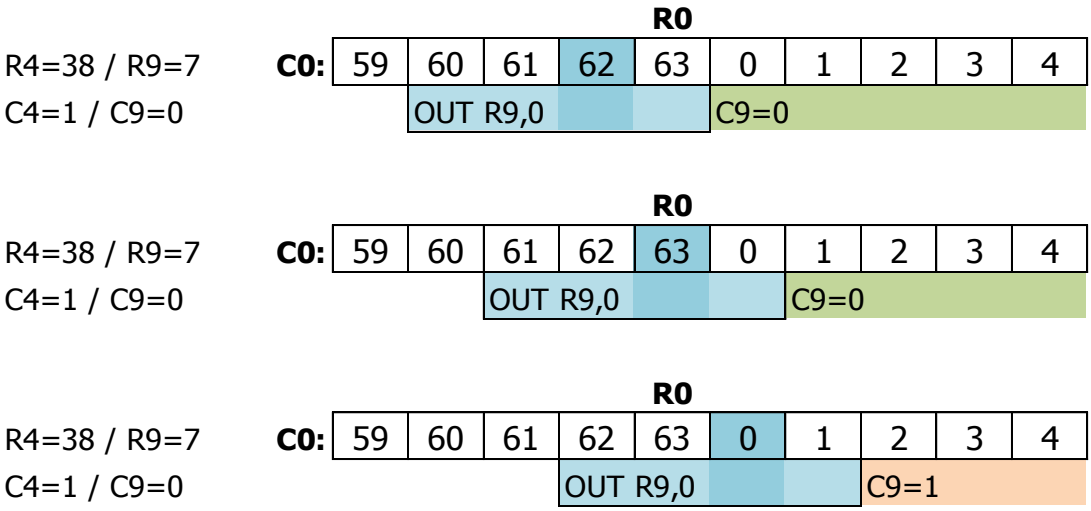
Indeed, even if C9=8 gives the display of a line 0 (bit 3 being "ignored"), this value does not always allow for the consideration of the operations that take place during the change of a line-character (C4).

10.2 TIME LIMITS

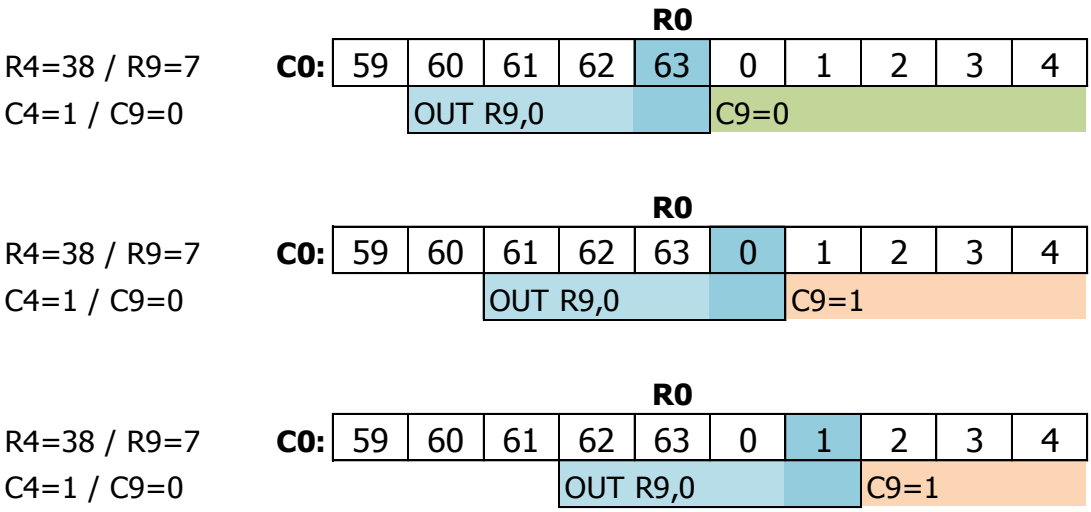
R9 update is considered while C0<=R0.

In the diagrams below, R9 is set to 0 when it was greater than 0 before.

CRTC 0 , 1, 2



CRTC 3, 4



10.3 COUNTING RULES

If R9 is updated with 0 while C9 is greater than 0, for example, C9 will count to its maximum value (31) before going back to 0.

There is an exception to this rule for CRTC's 0 and 2, when R9 is updated with 0 on the last line of the last character of the frame.

10.3.1 CRTC 0

It can be read sometimes that the modification of R9 on CRTC 0 (and 2) is not considered for the line that follows the one where the update took place because the counter for this register would be "buffered". However, the CRTC does not "buffer" its counters. **It simply tests the equivalence of the counters with its registers at very precise moments**.

The only value that is saved in a buffer in the CRTC is the video pointer because it is reloaded at each line start.(See Chapters 17 and 20.2).

The value of C9 is compared with R9 when C0<2 (just as C4 is compared with R4) t**o determine a state that indicates whether the line was the last on the frame.**

This state allows the CRTC to know if C4 will be reset to 0 or if C4 will be incremented on the next line (if C9=R9). C4 is a counter which must "officially" be able to exceed R4, especially in vertical adjustment.

Note : CRTC's 3 and 4, for which C4 does not exceed R4 in adjustment, "solidifies" the additional lines with C4=R4.

If R9 or R4 is modified **when C0>1**, it does not change anything. Case closed.

Indeed, the last line state is no longer updated when C0>1. If the last line state is set, C4 will change to 0 and C9 too (except when R0=0, which is a special case).

If the last line state is not set when C0>1, then C4 will be incremented if C9=R9 when C0 goes to 0, otherwise (if C9<>R9), then C9 is incremented and C4 keeps its current value.

We can break down this management of incrementing C9 and C4 when we are not on a last line (or in additional management) into 2 steps which takes place when C0==R0:
(a) If C9==R9 then C4++
(b) If C9==R9 then C9=0 otherwise C9++

With this principle:
(a) C4 will increment each time C9==R9, whatever the value of R4.
(b) C9 will return to 0 if C9 was worth R9, or increment until reaching the new limit of R9 if this register was modified during the line.

Note: In additional management from the last line of the frame, the test is more complete in a) because it integrates an additional equivalence (C4==R4) so that C4 is only incremented once.

There is a **special case** of this counting logic, when R9 is modified exactly at the position C0==R0, and C9==R9 before arriving at C0==R0. When R9 is modified exactly to C0==R0, the register value is actually affected in the circuit **between a) and b) of the processing indicated previously**. This has an impact on the result of C4 and C9, since we can end up with a case where C4 and C9 are both incremented.

10.3.1.1 GENERAL CASE

If R9 is changed with the value of C9, it will change to 0 on the next line.
Example : C9=0 and R9 is updated with 0, which was 7 before.
Example : C9=3 and R9 is updated with 3, which was 7 before.

If R9 is modified with a value less than C9, then C9=C9+1 (overflow of C9)
Example : C9=3 and R9 is updated with 1. Next C9 will be 4.
If the C9 counter overflows, it will count to its maximum value (31) before looping back to 0.

If R9 is modified with a value greater than C9, then C9=C9+1. C4 is unchanged.
Example : C9=0 and R9 is updated with 7. C9 will be 1 on the next line.

10.3.1.2 EXCEPTION TO THE GENERAL CASE : LAST LINE OF FRAME

Updating R9 on this last line when C0>1 has no impact on the next C9, because C4 will be forced to 0 and therefore C9 will also change to 0.

If C4=R4 and C9=R9 when C0<2, then the "**last frame line**" state is triggered.

If C4<>R4 or C9<>R9 when C0<2 then the "**last frame line**" state is disarmed.

The "last frame line" state is no longer evaluated when C0>1.

If the "**last frame line**" state is armed when C0>1, C4 will be reset to 0 on the next line (after the additional line(s) if R5>0) and C9 will change to 0.

Programming R9 (or R4) when C0>1 will not prevent C4 and C9 from returning to 0.

10.3.2 CRTC 1

10.3.2.1 GENERAL CASE

If R9 is changed to the current value of C9, then on the following line:
C9 goes to 0.
C4 is incremented by 1, and goes to 0 if it was R4, and in this case the offset is considered.

If R9 is modified with a value less than C9
C9=C9+1 and C4 is unchanged (until the end of the overflow of C9).
The offset will be changed only if C4=C9=C0=0

If R9 is modified with a value greater than C9, then C9=C9+1. C4 is unchanged.
Example : C9=0 and R9 is updated with 7. C9 will be 1 on the next line.

10.3.2.2 NO EXCEPTION

Everything is pure logic, in an unspeakable and tasteless simplicity. :-)

10.3.3 CRTC 2

The management of C9/R9 is linked to that of C4/R4.

Chapter 12.4.2, page 94 describes counting and updating management for the two registers.

Apart from the cases where C4 and C9 are zeroed because of a "**Last Line**" state, similar to that existing on CRTC 0, C9 counts up to R9, and returns to 0 once C9=R9. C4 is then incremented (or reset to 0) depending on the situation.

If register C9 overflows because R9 has been updated with a value less than C9 (and apart from a programmed reset of C9), the counter will count up to its maximum value (31) before looping back to 0, until the value of R9 is reached again.

10.3.4 CRTC 3, 4

10.3.4.1 GENERAL CASE

When R9 is changed, its value is considered immediately in the following way.

If R9 is changed with a value less than or equal to C9, then C9 changes to 0 on the next line, and C4 goes to 0 (if C4=R4) otherwise C4=C4+1.

Simply put, it is impossible to "overflow" C9 on these CRTC's.

It is not a simple equality test which takes place, but a "more complex" comparison performed by the ASIC:
If current-C9 > R9 then next-C9=0

Example : If C9 was 4, and R9 is changed with 1 (whereas it was 7 before), then C9 will go to 0 (and C4=C4+1 or 0 depending on the value of C4 and R4)

After C9=0, C9 will be incremented to the new value of R9. 1 in the example.

This management sometimes allows for dual compatibility with CRTC's 0, 1 and 2.

Indeed, if a program modifies R9 as part of a line-to-line rupture, a classic way to proceed is to:

- Update R9=0 on the last line of the frame on CRTC 0 or 2, so that the next line C9=0 is also considered the last frame (if R4=0).
- Update R9=0 on the first line of the fraùe on CRTC 1, so that the next line is considered the last frame (if R4=0).

In both situations, CRTC's 3 and 4 will put C9=0 back on the next line.

Previous R9=7				CRTC 0 (HD6845SP)			CRTC 1 (UM6845R)			CRTC 2 (MC6845P)			CRTC 3, 4		
C4=R4 (>0)			Event	Result on next line			Result on next line			Result on next line			Result on next line		
Case	**R9**	**C9 cur**	**Upd R9**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**
1	7	0	0	0	**C4++/0 (**)**	**N/Y (**)**	0	0	Yes	0	**0 (*)**	**If C0<=R1**	0	0	Yes
2	7	1	0	C9=C9+1 (2)	unmodified	No	C9=C9+1 (2)	unmodified	Yes if C4=0	C9=C9+1 (2)	unmodified	No	0	0	Yes
3	7	2	0	C9=C9+1 (3)	unmodified	No	C9=C9+1 (3)	unmodified	Yes if C4=0	C9=C9+1 (3)	unmodified	No	0	0	Yes
4	7	3	0	C9=C9+1 (4)	unmodified	No	C9=C9+1 (4)	unmodified	Yes if C4=0	C9=C9+1 (4)	unmodified	No	0	0	Yes
5	7	4	0	C9=C9+1 (5)	unmodified	No	C9=C9+1 (5)	unmodified	Yes if C4=0	C9=C9+1 (5)	unmodified	No	0	0	Yes
6	7	5	0	C9=C9+1 (6)	unmodified	No	C9=C9+1 (6)	unmodified	Yes if C4=0	C9=C9+1 (6)	unmodified	No	0	0	Yes
7	7	6	0	C9=C9+1 (7)	unmodified	No	C9=C9+1 (7)	unmodified	Yes if C4=0	C9=C9+1 (7)	unmodified	No	0	0	Yes
8	7	7	0	0	**0/C4++ (***)**	**Y/N (***)**	C9=C9+1 (8)	unmodified	Yes if C4=0	0	**0**	**If C0<=R1**	0	0	Yes

(*) if R9 is updated outside of hsync period **(**) C4=0 if R9 is updated on C0<2** **(***) C4=0 if R9 is updated on C0>1**

Previous R9=7				CRTC 0 (HD)			CRTC 1			CRTC 2			CRTC 3, 4		
C4<>R4			Event	Result on next line			Result on next line			Result on next line			Result on next line		
Case	**R9**	**C9 cur**	**Upd R9**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**
1	7	0	0	0	**C4=C4+1**	No	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No
2	7	1	0	C9=C9+1 (2)	unmodified	No	C9=C9+1 (2)	unmodified	Yes if C4=0	C9=C9+1 (2)	unmodified	No	0	C4=C4+1	No
3	7	2	0	C9=C9+1 (3)	unmodified	No	C9=C9+1 (3)	unmodified	Yes if C4=0	C9=C9+1 (3)	unmodified	No	0	C4=C4+1	No
4	7	3	0	C9=C9+1 (4)	unmodified	No	C9=C9+1 (4)	unmodified	Yes if C4=0	C9=C9+1 (4)	unmodified	No	0	C4=C4+1	No
5	7	4	0	C9=C9+1 (5)	unmodified	No	C9=C9+1 (5)	unmodified	Yes if C4=0	C9=C9+1 (5)	unmodified	No	0	C4=C4+1	No
6	7	5	0	C9=C9+1 (6)	unmodified	No	C9=C9+1 (6)	unmodified	Yes if C4=0	C9=C9+1 (6)	unmodified	No	0	C4=C4+1	No
7	7	6	0	C9=C9+1 (7)	unmodified	No	C9=C9+1 (7)	unmodified	Yes if C4=0	C9=C9+1 (7)	unmodified	No	0	C4=C4+1	No
8	7	7	0	C9=C9+1 (8)	unmodified	No	C9=C9+1 (8)	unmodified	Yes if C4=0	C9=C9+1 (8)	unmodified	No	0	C4=C4+1	No

Previous R9=7				CRTC 0 (HD)			CRTC 1			CRTC 2			CRTC 3, 4		
C4=R4 (>0)			Event	Result on next line			Result on next line			Result on next line			Result on next line		
Case	**R9**	**C9 cur**	**Upd R9**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**
1	7	0	0	0	**C4++/0 (**)**	**N/Y (**)**	0	0	Yes	0	**0 (*)**	**If C0<=R1**	0	0	Yes
2	7	1	1	0	**C4++/0 (**)**	**N/Y (**)**	0	0	Yes	0	**0 (*)**	**If C0<=R1**	0	0	Yes
3	7	2	2	0	**C4++/0 (**)**	**N/Y (**)**	0	0	Yes	0	**0 (*)**	**If C0<=R1**	0	0	Yes
4	7	3	3	0	**C4++/0 (**)**	**N/Y (**)**	0	0	Yes	0	**0 (*)**	**If C0<=R1**	0	0	Yes
5	7	4	4	0	**C4++/0 (**)**	**N/Y (**)**	0	0	Yes	0	**0 (*)**	**If C0<=R1**	0	0	Yes
6	7	5	5	0	**C4++/0 (**)**	**N/Y (**)**	0	0	Yes	0	**0 (*)**	**If C0<=R1**	0	0	Yes
7	7	6	6	0	**C4++/0 (**)**	**N/Y (**)**	0	0	Yes	0	**0 (*)**	**If C0<=R1**	0	0	Yes
8	7	7	7	0	0	Yes	0	0	Yes	0	**0 (*)**	**If C0<=R1**	0	0	Yes

(*) if R9 is updated outside of hsync period **(**) C4=0 if R9 is updated on C0<2**

Previous R9=7 C4<>R4			Event	CRTC 0 (HD) Result on next line			CRTC 1 Result on next line			CRTC 2 Result on next line			CRTC 3, 4 Result on next line		
Case	**R9**	**C9 cur**	**Upd R9**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**
1	7	0	0	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No
2	7	1	1	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No
3	7	2	2	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No
4	7	3	3	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No
5	7	4	4	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No
6	7	5	5	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No
7	7	6	6	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No
8	7	7	7	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No	0	C4=C4+1	No

Previous R9=7 C4=R4			Event	CRTC 0 (HD) Result on next line			CRTC 1 Result on next line			CRTC 2 Result on next line			CRTC 3, 4 Result on next line		
Case	**R9**	**C9 cur**	**Upd R9**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**
1	7	2	1	C9=C9+1 (3)	unmodified	No	C9=C9+1 (3)	unmodified	Yes if C4=0	C9=C9+1 (3)	unmodified	No	0	0	Yes

Previous R9=0 C4=R4=0			Event	CRTC 0 (HD) Result on next line			CRTC 1 Result on next line			CRTC 2 Result on next line			CRTC 3, 4 Result on next line		
Case	**R9**	**C9 cur**	**Upd R9**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**	**C9**	**C4**	**Offs Upd**
1	0	0	0	0	0	Yes	0	0	Yes	**0 (*)**	C4=0 / C4+1	if C0<=R1/No	0	0	Yes
2	0	0	1	**0/1 (**)**	0	**Y/N(**)**	C9=C9+1 (1)	0	Yes (C4=0)	**0 / 1 (*)**	C4=0 / C4+1	if C0<=R1/No	C9=C9+1 (1)	0	No
3	0	0	2	**0/1 (**)**	0	**Y/N(**)**	C9=C9+1 (1)	0	Yes (C4=0)	**0 / 1 (*)**	C4=0 / C4+1	if C0<=R1/No	C9=C9+1 (1)	0	No
4	0	0	3	**0/1 (**)**	0	**Y/N(**)**	C9=C9+1 (1)	0	Yes (C4=0)	**0 / 1 (*)**	C4=0 / C4+1	if C0<=R1/No	C9=C9+1 (1)	0	No
5	0	0	4	**0/1 (**)**	0	**Y/N(**)**	C9=C9+1 (1)	0	Yes (C4=0)	**0 / 1 (*)**	C4=0 / C4+1	if C0<=R1/No	C9=C9+1 (1)	0	No
6	0	0	5	**0/1 (**)**	0	**Y/N(**)**	C9=C9+1 (1)	0	Yes (C4=0)	**0 / 1 (*)**	C4=0 / C4+1	if C0<=R1/No	C9=C9+1 (1)	0	No
7	0	0	6	**0/1 (**)**	0	**Y/N(**)**	C9=C9+1 (1)	0	Yes (C4=0)	**0 / 1 (*)**	C4=0 / C4+1	if C0<=R1/No	C9=C9+1 (1)	0	No
8	0	0	7	**0/1 (**)**	0	**Y/N(**)**	C9=C9+1 (1)	0	Yes (C4=0)	**0 / 1 (*)**	C4=0 / C4+1	if C0<=R1/No	C9=C9+1 (1)	0	No

(*) According authorization of last line in Hsync **(**) C9=0 if R9 updated on C0>1 / C9=1 if R9 updated on C0<2**

11 COUNTER : REGISTER R5

11.1 GENERAL

The R5 register allows you to add so-called vertical adjustment lines at the end of the frame.

The objective of this register is to complete the total number of vertical rows displayed when the (R4+1)x(R9+1) rows of a frame does not correspond to 312 rows (for the European standard)

This register contains a number of lines on 5 bits (0 to 31) which corresponds to the maximum possible adjustment compared to the maximum number of possible lines of a character, fixed by R9. Indeed, CRTC's 1, 2, 3 and 4 do not respect this functional principle because the adjustment is not supposed to contain several characters. In other words, if R5/R9+1>1, R4 should in principle be adjusted by the developer.

If R5 = 0 then there is no specific adjustment.
If R5 > 0 then an adjustment takes place with the creation of R5 additional lines.

If one of the two interlace modes is programmed (R8=1 or R8=3) then an additional adjustment line is added on the even frames, after any lines generated via R5.

The implementation of the vertical adjustment function is the source of several differences between the CRTCs. On CRTCs 0, 3 and 4, **there is no specific C5 counter** and C9 is used for comparison with R5. On CRTCs 1 and 2, there is a specific counter C5 used in conjunction with C9 to allow management of "characters" within the adjustment lines.

The C4 incrementation follows different logic when R5>0:

- On CRTC 0, C4 is incremented only on the "last line" when R5>0 (settled before C0==3 of the last line). C9 is compared with R5 and R9.
- On CRTC's 1 and 2, C4 increments regardless of the value of R4 each time C9=R9, as long as C5 has not reached R5. For the first additional line, C4=R4+1 if C9==R9.
- On CRTC's 3 and 4, C4 does not increment and is equal to R4. C9 is only compared with R5.

11.2 COUNTING IN VERTICAL ADJUSTEMENT

11.2.1 GENERAL

The following diagrams describe the diversity of the methods of counting (C4, C9, C5) and updating the video pointer during this management, considering that R5 and R9 are not modified during the adjustment. The registers are initialized as follows (R4=10, R5=16**, R9=3,** R1=40, R0=63)

CRTC 0			
C4	C9	PTR-VRAM	LINE
11	0	0	&0
11	1	0	&800
11	2	0	&1000
11	3	0	&1800
11	4	40	&2000
11	5	40	&2800
11	6	40	&3000
11	7	40	&3800
11	8	40	&0
11	9	40	&800
11	10	40	&1000
11	11	40	&1800
11	12	40	&2000
11	13	40	&2800
11	14	40	&3000
11	15	40	&3800

CRTC 1, 2				
C4	C9	C5	PTR-VRAM	LINE
11	0	0	0	&0
11	1	1	0	&800
11	2	2	0	&1000
11	3	3	0	&1800
12	0	4	40	&0
12	1	5	40	&800
12	2	6	40	&1000
12	3	7	40	&1800
13	0	8	80	&0
13	1	9	80	&800
13	2	10	80	&1000
13	3	11	80	&1800
14	0	12	120	&0
14	1	13	120	&800
14	2	14	120	&1000
14	3	15	120	&1800

CRTC 3, 4			
C4	C9	PTR-VRAM	LINE
10	0	0	&0
10	1	0	&800
10	2	0	&1000
10	3	0	&1800
10	4	0	&2000
10	5	0	&2800
10	6	0	&3000
10	7	0	&3800
10	8	0	&0
10	9	0	&800
10	10	0	&1000
10	11	0	&1800
10	12	0	&2000
10	13	0	&2800
10	14	0	&3000
10	15	0	&3800

11.2.2 CRTC 0

On CRTC 0, HITACHI engineers saved a C5 counter to use C9 instead. In additional management, C9 is compared with R9 and R5.

The new limit of C9 is no longer R9 at the end of the line, but R5 at the beginning of the line (R5 must be greater than 0 for there to be at least one line generated, except in the context of interlace modes).

The additional management becomes true if R5 becomes greater than 0 before C0==3 of the last line of the frame (when C4==R4 and C9==R9). When this additional management is activated, it will end when C9 calculated for the new line==R5.

In order to prevent resetting C9 to 0 from leading to a loop if R5>R9+1, the incrementation of C4 and C9 follows a specific logic. As long as C4<>R4 in additional management, C9 can no longer be zeroed. This allows C9 to exceed R9 and reach R5, to end the additional management.

In additional management, the processing of **C4 and C9 takes place in 2 times when C0==R0**:
(a) If C9==R9 & C4==R4 (last line true) then C4++
(b) If C9==R9 then C9=0 otherwise C9++

With this principle:

- C4 will only increment once on a last line true at the end of the line (when C0 reaches R0-1 and C9==R9/C4==R4).
- C9 will return to 0 if C9 was worth R9, or increment until reaching the new limit of R9 if this register was modified during the line.

This logic however has its limits, because if R4 is modified during the last line, C4 will not increment, except in the particular case of updating R4 or R9 to C0==R0.

When R4 or R9 is modified exactly to **C0==R0**, the register value is actually affected in the circuit **between a) and b) of the processing indicated previously**. This has an impact on the result of C4 and C9, since we can end up with a case where C4 and C9 are both incremented.

Example 1: On a last line C9==R9=7/C4==R4=38 with R5>0, the value of R9 is modified with 12 on the position C0==R0. On test a) R9 is still worth the old R9 (7) and C4 is incremented (C4=39). On test b) R9 takes its new value (12) and C9 increments (C9=8). We therefore went from C4=38/C9=7 to C4=39/C9=8.

Example 2: On a last line C9==R9=7/C4==R4=38 with R5>0, the value of R9 is modified with 12 before C0==R0, and the value of R4 is modified with 50 on the position C0==R0. On test a) R9 is no longer equal to R9 (7<>12) and C4 does not increment (38). On test b) R4 takes its new value. R9 is still not equal to R9, and C9 increments (C4=38/C9=8).

C9 continues to be compared with R9 **to consider the video pointer (VMA'=VMA) when C0=R1 and C9=R9.**

This is not "practical" for managing several characters automatically during adjustment, but it allows you to move to another address without C9 being equal to 0.
Thus, the diagram on the next page, which uses the data from the first diagram shows the impact of an R9 update being adjusted to change the VMA' pointer.

CRTC 0				
C4	C9	PTR-VRAM	LINE	
11	0	0	&0	
11	1	0	&800	
11	2	0	&1000	
11	**3**	0	&1800	**R9**
11	4	40	&2000	**OUT R9,10**
11	5	40	&2800	
11	6	40	&3000	
11	7	40	&3800	
11	8	40	&0	
11	9	40	&800	
11	**10**	40	&1000	**R9**
11	11	80	&1800	
11	12	80	&2000	
11	13	80	&2800	
11	14	80	&3000	
11	15	80	&3800	

In the example:

- When C9 reaches R9 (=3), then the video pointer is updated with the one that has been memorized when C0=R1 and C9=R9. (R1=40)
- R9 is modified with 10 while C9=4
- When C9 reaches R9 (=10), then the pointer is once again updated, and therefore goes to 80.
- Finally, when C9 reaches R5, vertical adjustment ends.

11.2.3 CRTC 1, 2

On these circuits, the C5 and C9 counters are dissociated.

The C9=R9 management considers the video pointer (VMA'=VMA) when C0=R1.
C9 is zeroed when C9=R9 and C4 is incremented.
R9 update is considered for current C9 counting.

CRTC 1, 2					
C4	C9	C5	PTR-VRAM	LINE	
11	0	0	0	&0	
11	1	1	0	&800	
11	2	2	0	&1000	
11	3	3	0	&1800	**R9**
12	0	4	40	&0	**OUT R9,10**
12	1	5	40	&800	
12	2	6	40	&1000	
12	3	7	40	&1800	
12	4	8	40	&2000	
12	5	9	40	&2800	
12	6	10	40	&3000	
12	7	11	40	&3800	
12	8	12	40	&0	
12	9	13	40	&800	
12	10	14	40	&1000	**R9**
13	0	15	80	&0	

If an additional line is added via the activation of the interlace function, this logic is not modified. The counting is done as if this line had been added to R5.

Example: if you program R4=37, R9=7 and R5=7, then the value of C4 will be 38 during the first 8 additional lines. The additional interlace line will then be part of the character C4=38.
If R5 is programmed with 8 in this example, then the value of C4 will be 38 on the R5 Additional lines, and the value of C4 will be 39 on the additional interlace line. (once in 2 if parity is not frozen, since the additional line is added only at the end of a even frame).
See chapters 19.3 and 19.5 concerning the notions of interlace and parity.

11.2.4 CRTC 1

When the additional management is true, C9 counts up to R9 and when it goes back to 0, C4 is incremented without taking R4 into account.

If R9 and/or R4 are modified on the last line of the frame while additional lines are programmed, their value is taken into account immediately to postpone the end of the frame (in other words the

additional management is deactivated). The additional management only becomes definitively irreversible if C4==R4 and C9==R9 when C0 reaches R0.

However, if C4==R4 and C9==R9 in position C0==R0, and R4 or R9 is modified in this position, then the **additional management remains true**. The values of R4 and R9 are used for counting and the lines thus calculated are **part of** the additional lines.

If C4=0 before the additional management, then VMA is updated with R12/R13 and not VMA', and this as long as C4=1 (new value of C4 in additional management). In other words, the management of R1 for the update of the video pointer no longer takes place. It is then possible to modify the offset on each line C9 of C4=1 as one would do when C4=0, or on any value of C4 in RFD (see chapter 11.6, page 85)

However, if R4 was modified to C0==R0 with R4>0, then VMA is not updated with R12/R13 when C4=1.

Note: If R9 is modified to C0==R0 when C4==R4=0, then VMA continues to be updated with R12/R13 when C4=1 in additional management.

11.2.5 CRTC 2

In additional management, C4 is incremented each time C9 reaches R9 (C9 then goes to 0), whatever the value of R4.

If R9 and/or R4 are modified on the last line of the frame while the additional lines are programmed, their value is taken into account immediately for the counting of C9 and C4, and the lines will be **part of** the additional lines.

11.2.6 CRTC 3, 4

On CRTC's 3 and 4, the C4 counter is not incremented when additional management begins.
C4 is equal to R4. However, the video pointer is updated before the start of the additional lines (VMA'=VMA) when C0=R1.

Additional management just set's C9 to 0 and compare's C9 with R5 to deactivate this management, without updating the video pointer.

This "solidifies" the last character with the character generated in vertical adjustment, and the interlace line if it has been scheduled.

In the previous example, the character C4=10 contains 16 more rows.

11.3 UPDATING R5 DURING AN ADJUSTMENT

Whatever the CRTC, if R5 is modified with C5+1 (or C9+1 on CRTC's 0, 3, 4) on line C5 (or C9), then the vertical adjustment is "stopped".

In this situation, C4=C9=0 for the next line (whatever the current value of C9), unless the conditions are present so that an interlace line is generated.

The "latest line" state is then processed correctly and led to reset C4.

11.3.1 CRTC's 0, 2

When the number of the next additional line (C5+1 on CRTC 2, C9+1 on CRTC 0) reaches R5, additional management R5 ends.

The additional interlace line (R8) is managed if the conditions for its generation are respected.

If R5 is modified with a value less than C5+1/C9+1, then the counter overflows and continues to count up to 0 to reach the new R5 value.

Be careful however because on CRTC 0, the update of R5 during the last line of frame is considered only when C0<3. In other words, if R5 is modified when C0>2 on the last line (C9+1=R5), then the value of R5 is not considered (C4 and C9 will then go back to 0).

If no interlace line was scheduled, then C4 and C9 return to 0.

11.3.2 CRTC 1

When the number of the next additional line (C5+1) reaches R5, additional management R5 ends.

The additional interlace line (R8) is managed if the conditions for its generation are respected.

If R5 is modified with a value less than C5+1, then the C5 counter is overflowing and continues to go back to 0 to reach the new R5 value.

However, **if the new value of R5 is set to 0**, this causes a bug that deactivates (slyly) the reset of C4 for the new frame. Indeed, CRTC 1 activates an internal additional management state if R5>0 when C4 should return to 0 at the end of the frame (C4=R4, C9=R9). In principle, additional management waits until C5+1=R5 before setting C4 to 0 and deactivating this state.

But if R5 becomes zero during additional management, the state is not deactivated, C4 does not return to 0 and C5 loops. C4, however, continues to be compared to R4 to process the change from C4 to 0. **The additional management, however, remains activated**. Thus, if C5+1 reaches an R5>0, then the additional management changes C4 to 0 before deactivating its state. In other words, **it is possible to change C4 and C9 to 0 on any line with this method by modifying R5 with a value greater than 0 according to the value reached by C5+1.**

11.3.3 CRTC's 3, 4

When the number of the next additional line (C9+1) reaches R5, additional management R5 ends.

The additional interlace line (R8) is managed if the conditions for its generation are respected.

If R5 is modified with a value below C9+1, then the line is considered the last and additional management ends.

Whether with R5 or R9, it is impossible to overflow C9.

11.4 R5 UPDATE BEFORE AN ADJUSTMENT

11.4.1 CRTC's 1, 2, 3, 4

R5 management is considered on each C0 position.

On CRTC 1, the R5 update on C0==R0 with a value greater than 0 when it was 0 triggers a bug described in chapter 11.6.

11.4.2 CRTC 0

The R5>0 update after position C0>2 on the last line of the frame is not considered because its evaluation is completed. See Chapter 13.2.

Unless an interlace line has been defined, the next line displayed will be C4=C9=0 (respect for the "**Last Line**" condition).

11.5 VSYNC DURING VERTICAL ADJUSTMENT

11.5.1 CRTC's 0, 1, 2

R7 can be positioned on one of the values reached by C4 in vertical adjustment to trigger a VSYNC.

11.5.2 CRTC 0

On CRTC 0 the VSYNC is blocked when R7 is updated with C4 on a position C0<2.

11.5.3 CRTC 2

If R7 is positioned with C4 during a HSYNC, a GHOST VSYNC begins.

11.5.4 CRTC's 3, 4

The VSYNC only starts when C4=R7 on position C0=C9=0.

11.6 RUPTURE FOR DUMMIES (R.F.D.) ON CRTC 1

On the CRTC 1, there is a very interesting bug when R5 is updated with a value different from 0 on the position C0=R0 of some C9s when R5 is equal to 0. Note that this bug can be triggered by other less practical methods (see chapter 13.7.1.2, page 118). Going from R5>0 to R5=0, or from R5>0 to another value of R5>0 does not trigger this bug.

I name **RFD** the rupture technique that stems from this bug.

Activating an **RFD** sets several statuses.

- The first status defines the VMA update source (VMA' or R12/R13) when C0=0.
- The second one activates parity management in the C9/R9 test carried out in IVM when C0 reaches R1 (and which allows the positioning of the VMA update state when the C9=R9 condition on C0=R1 is verified)

The value of R5 is of little importance, **except on certain CRTC 1's** for the value **#10**.
This CRTC will be identified as 1-B while waiting to find out if the difference noticed really comes from the CRTC. The effect obtained by this value will be identified as **RFD#10**.
CRTC 1-A is defined as CRTC 1 which **does not support RFD#10.**
The additional management continues to be managed normally if R5 is not reset to 0 after the RFD has been triggered. If additional management is not desired for the frame, the reset of R5 can be done at any position of C0 after the activation of the RFD.
It is therefore possible to trigger an RFD via an "OUT R5,1", immediately followed by an "OUT R5,0" if it is only the RFD effect that is desired. If additional management and an RFD effect are desired, it is possible to do so by updating R5 once on C0=R0 with the number of additional rows desired.

Common processing of C4 versus C9 is not affected.

The RFD demonstrates that there is a state that allows CRTC 1 to accept consideration of R12/R13 at the start of the line regardless of the value of C4.

As a reminder, on the first character of a frame, VMA is assigned with R12/R13 and is loaded with VMA' on the other characters. VMA' is itself loaded with VMA when C9 reaches R9 and C0 reaches R1 (to account for the advance of the offset on the next row relative to the BORDER). The update of VMA with R12/R13 or VMA' takes place from C0=0.

VMA's update status via R12/R13 is usually true when C4=0, and then becomes false (VMA is then reloaded with VMA'). However, if for some reason the condition to test that the BORDER has been reached on the last line of the character **is not met**, then the VMA update status via R12/R13 remains true regardless of the value of C4.

It is important to note however that if the condition C0=R1 is not met (because R1>R0), then the condition C9=R9 is enough to deactivate the update of VMA with R12/R13. It is therefore not enough for R1>R0 to be able to modify the offset on each line.

RFD sets another status of the IVM interlace which involves frame parity in the C9=R9 equivalence. Thus, VMA update via R12/R13 is activated by the RFD and this state persists as long as the only true equivalence is C0=R1. However, the RFD does not modify the counting mode associated with the IVM mode.

In other words, this state remains active as long as the parity of the frame combined with the calculation mode of C9 comes into play on the comparative processing of C9. It is thus possible to end up with a situation equivalent to R1>R0 (See Chapter 17.4.2) while R1<R0.

The C9=R9 "out of parity" condition continues to be processed normally for the calculation of C4 and the reset of C9. The **RFD** does not affect the counting mode of C9 but determines the comparative values of C9 when C0=R1 activate VMA assignment with VMA' (instead of VMA=R12/R13).

11.6.1 RFD AND FRAME PARITY

The activation of an **RFD** implies a test of R9 carried out with the current parity.

On the **first frame (case 1)**, VMA' is no longer updated when C0 reaches R1 (because the test C9=R9 is faulty), and the characters are repeated. However, C4 continues to be managed whenever C9 reaches R9. VMA's update status is stuck on R12/R13, so it's possible to change the address on every line of every character, as if C4 was 0.

On the **second frame (case 2)**, VMA' is updated when C0 reaches R1 and C9=R9. Parity no longer corrupts the test and therefore characters do not repeat because VMA is updated with VMA'. On this frame, a R12/R13 update is no longer considered as soon as VMA has been updated with VMA' (so when C9=R9). A **RFD** triggered on the last line C9=R9 disables the state allowing VMA to be updated with R12/R13. However, the **RFD#10 of the CRTC 1-B** always allows, when C9=R9, to deactivate the parity management in the test C9=R9 (See next Chapter).

C4 increments in both situations, because the frame remains perfectly synchronized on C4=R7.

The switching of parity, which takes place once each time the frame starts (C4=C9=C0=0 and R9 is odd), causes a stroboscopic effect between the two frames, except probably to position judiciously, at the right moment, R1>R0 so that all rows repeat equally on both frames. But we will see that such a subterfuge is useless because there is something simpler.

11.6.2 IVM ON/OFF.

As we have just seen, the repetition of the first character on the frame defined as case 1 is the consequence of a fault in the equivalence test C9=R9 allowing the assignment of VMA' (and which locks the VMA update status). This C9 test involves frame parity. According to this parity, the test allowing VMA' assignment is assigned between each frame.

It is however possible to **fix the parity of a frame**, knowing that this **parity switches with each new frame** (or some other specific cases. See Chapter 19.5.3).
For example, it is enough to activate and deactivate the IVM mode with an **odd R9** to set an **even parity** (OUT R8,3 followed by OUT R8,0). I will name this action "**IVM ON/OFF**". Be careful, however, to perform these updates on an even C9 because the C9 bit 0 can go to 0 on the line and therefore corrupt the counting if C9 was odd.

If an "**IVM ON/OFF**" takes place **before a RFD**, then all the frames will contain lines for which VMA' is no longer updated with VMA (case 1 mentioned in the previous Chapter).

In other words, **it is then possible to modify the offset on each line of the entire frame without any additional formality required**.

If an "**IVM ON/OFF**" takes place **after the RFD**, then all the frames will contain lines for which VMA' is updated each time C9=R9 (case 2 mentioned in the previous Chapter). In other words, it is possible to modify the offset until C9<>R9 from the line where the **RFD** is generated. The "**IVM ON/OFF**" having taken place after the R5 update, **it is on the new frame that the even parity will switch to become odd and cause this management from the second frame and the following**.

On **CRTC 1-B**, value #10 in R5 **deactivates parity management in test C9=R9**, while other values **activate this parity management (including value #10 on CRTC 1-A)** .

The brand and model of **CRTC 1-B** does not differ from other CRTC 1's (UM6845R).
This does not seem related to the batch number: the RFD#10 operates for example on a 6128 UM6845R-8804T, but not on a 6128 with UM6845R-8802T.
Out of 7 machines tested, 3 had this additional capacity.
If you want to identify this capacity on a CRTC 1, you can use SHAKER 2.1 by running the SHAKE21B.BIN module, test "O".

We can define that RFD:

- Activates a status authorizing the update of VMA with R12/R13 when C0 = 0 (this status is disabled when the test C9 = R9 on C0 = R1 is true).
- Active (or deactivates on the **CRTC 1-B** with RFD#10) a status authorizing the consideration of parity in the test C9 = R9 carried out when C0 = R1.

When the IVM state which authorizes the consideration of parity in the test C9 = R9 becomes active with a RFD, it is no longer possible to deactivate it until the frame is finished.
Therefore :

- If a RFD activates this status, an RFD#10 can no longer deactivate it.
- If a RFD #10 (**CRTC 1-B**) deactivates this status, a RFD "not #10" allows it to be reactivated.

Note: The "Sync Interlace" mode (IS ON/OFF) does not allow (to my knowledge) for fixing of the parity of the frame. The alteration of processing between each frame is not impacted if R8 changes between value 1 and 0 (or 2).

11.6.3 R.F.D. IN SUMMARY.

Method to change the video pointer on a character without having to worry about R4 or R7.

- On C0=R0, change R5 from 0 to 1 on any line different from C9=R9 for a C4 where the offset must be modified.
  - OUT R5,1+OUT R5,0
- Modify R12/R13 on the line before the R5 update.
- Make R8 go from 3 to 0 once on an even C9 line.
  - OUT R8,3+OUT R8,0

Method to change the address on each line.

- Wait for a new frame (C4=C9=0).
- Make R8 go from 3 to 0 once on an even C9 line.
  - OUT R8,3+OUT R8,0
- On C0=R0, change R5 from 0 to 1 on any line 1 time for the frame.
  - OUT R5,1+OUT R5,0
- Modify R12/R13 on the line preceding the one where the address must be modified.

This method is not a conventional RLAL, insofar as C9 participates in the construction of the video pointer. C9 is always 0 in RLAL. It is always possible to duplicate the video ram of the different C9s, but this implies a serious consumption of the ram according to the value of R9 programmed.

11.6.4 R.F.D. AND OTHERS CRTC

Page 120 of the HITACHI technical guide (CRTC 0) contains a table showing the anomalies that can occur when certain registers are modified during the display. Concerning R5, it is particularly indicated that if an update of R5 takes place on C0=R0, then there are "certain cases" where R5 is not really considered...Really ?

The contradictions in this table are a real playground!

11.7 R6 AND VERTICAL ADJUSTMENT

When C4 reaches R6, the display of the data is stopped.
R6 must be positioned according to the values reached by C4 to display the corresponding characters generated during the adjustment.

On CRTC 1 there is special management of R6 when it is 0, which allows for the activation of the BORDER in a non-definitive manner.

This particular treatment is also managed by CRTC 1 in vertical adjustment.

The technique which uses this management is called "split-border".
This technique is possible on CRTC's 0, 3 and 4 using an R8 function (See Chapter 19.2, page 187).

11.8 ADJUSTMENT DURING INTERLACE

In interlace mode, R5 lines are added to the frame.

On CRTC 2, it is quite logical, because this CRTC does not require, as for the other CRTC, an R4 update to double the number of characters displayed (since each character contains 2 times less lines in interlace mode) and an adaptation of the registers associated with C4, such as R6 and R7.

On the CRTC's 0,1,3 and 4, R5 lines are added on each frame.

If, for example, we have 38 characters of 8 lines and 8 additional lines, you must program R4=37 and R5=8 (38 x 8 + 8 = 312). If we switch to an IVM interlace mode (R8=3), it is necessary to keep the same number of lines, double the number of characters displayed. R4 must therefore be equal to 75. (38x2 – 1)

However R5 must remain at 8 : 76 x 4 + 8 = 312.

In all cases, the C9/C4 counters continue to evolve according to the interlace treatment logic associated with the lines according to the CRTC (See Chapter 19.8).

For example, on CRTC 1 in the previous example, C4 will increment 2 times during the R5 adjustment period because for each C4, there will be 4 C9.

11.9 INTERLACE ADJUSTMENT LINE

In INTERLACE mode, a specific vertical adjustment management is carried out which results in an additional line on each even frame. This adjustment occurs in very specific conditions, which depend on the state of R8 but also on the current parity of the frame.

See Chapter 19.3, page 192.

This adjustment is independent of that made via R5. When the adjustment condition is filled, the "Interlace" line is added **after** the lines possibly scheduled in R5.

The adjustment condition (interlace mode (IVM/non-IVM) activated and even frame) is evaluated on the last line of a frame, when C0=R0, and only if R8 contains the right value on the last line. This latest line can be one of the adjustment lines displayed via R5.

It is therefore possible to update R8 on one of the lines displayed via R5 to activate or deactivate the treatment of the interlace line.

On CRTC 2, if the interlace mode is disabled (R8=0) while the "Interlace" line is displayed, then the "**Last Line**" condition is cancelled. The current line is no longer considered as an interlace line. C9 then continues to count to R9, and C4 is increasing if it is different from R4. It is perfectly possible to reprogram R4 with C4 (equal to the old R4+1) in order to reactivate the "**Last Line**" state when C9 reaches R9.

12 COUNTING : REGISTER R4

12.1 GENERAL

The CRTC's R4 register defines the number of "character lines" to be displayed. A character-line is composed of several raster-lines, the number of which is fixed by the register R9.
See Chapter 10, page 71.

The line and character are numbered starting from 0.
The number programmed represents a value to reached.

When all lines of a character are displayed (C9=R9), then C4 is incremented (the counter returns to 0 if C4 was R4 before being incremented).

The value of C4 is compared with R7 to trigger VSYNC and compared with R6 to trigger BORDER.

Note however that the Interlace mode of the CRTC delays the triggering of the VSYNC.

When C4=R4 and all the rows of the last character are displayed, additional row management can optionally begin if R5 > 0 or Interlace mode is enabled.

Note: Except for CRTC's 3 and 4, additional lines via R5 continue to increment C4 (which therefore exceeds R4 in this situation).

Note: This management of additional line(s) is managed when C0<2 on CRTC 0.

When C4 changes to 0 (and while it is 0 on CRTC 1), VMA is updated with R12/R13 content.

On CRTC 2 however, VMA is updated with VMA', which is itself updated with R12/R13 when C0=R1 from the last line (when C9=R9).

12.2 CRTC 0

As long as C0<2, the CRTC assesses whether C9=R9 and C4=R4 **to determine if it is on the last line of the frame**. It no longer repeats this test on the other values of C0>1.

It is therefore not necessary to anticipate the programming of R4 (or R9) on the current line for the last line condition to be true on the following line. It is possible to modify R4 or R9 on the current line as long as C0<2 to validate or invalidate the "**Last Line**" state (and thus validate the reset of C4 on the following line).

If the desired objective is to maintain C4 and C9 at 0 so that each new line is a "last line" (see next chapter on the line-to-line rupture), it is possible to position R9 or R4 with the value 0 on C0<2 of the first line of the frame (when C4=C9=0 when R4>0 or R9>0).

Thus, by positioning R4 or R9 at 0 (with an OUT(C),reg8 starts on C0vs=#3E (if R0=#3F)) allows to satisfy the "**last line**" condition when C0=0. (or on C0=1 if the OUT(C),reg8 starts on C0vs=#3F). This feature simplifies compatibility with CRTCs 1, 3 and 4 which require positioning R4 and R9 at 0 when C9=C4=0.

When the CRTC has determined that it is on the last line of the frame, C4 will go to 0 no matter what happens (as well as C9). If R4 and/or R9 are modified during the line when C0>1 while the "**Last Line" state is set, this does not change anything for C4 which will therefore return 0** (on the next line).

Conversely, the "**Last Line**" state can be canceled if R4 or R9 is modified before C0>1 (and the equality C4=R4 and C9=R9 is not (or no longer) satisfied). If this state is not set, C4 will be incremented if R9 is still equal C9 when C0 goes back to 0.

C4 is incremented beyond the value set in R4 :

- If the "**Last Line**" state is false and C9<>R9 when line change. C4 will "overflow" to its maximum value (127) before looping back (if a new R4 update does not occur before).
- When additional lines are added. This can occur when R5>0 (or R5=0 with R0<2), or if the "Interlace" mode is activated on even frames (See Chapter 19.3, page 192). Note that if C4 exceeds R4 on at least one of these events, **it will return to 0 once the additional management is completed.**

12.2.1 CASE STUDY : LINE-TO-LINE RUPTURE (R.L.A.L.)

The objective of this technique is to obtain consecutive lines for which C9=C4=0, in order to be able to modify the address via R12 and/or R13.
For the following two examples, the registers R4 and R9 are considered to be **greater than 0**.

From the first line of a frame:
If R9 and/or R4 are positioned at 0 when C9=C4=0, this line will not be considered the last on the frame (the test took place when C0=0 and C0=1).
On the second line, C4 will then be incremented (C4 = 1). And C9 will go to 0.
At the beginning of this second line, C4(=1)<>R4(=0). C9=R9=0.
The CRTC will therefore increment C4 regardless of the value that is programmed in R4,…
This is not the desired goal.

If, on the fisrt line, we program R4=1 and R9=0 (when C0>1)(instead of R4=0 and R9=0), things will go better. If this line has not been considered the last, C4 will increment on the second line (C4 = 1) (and C9 will be equal to 0).

The test that took place on the second line (on C0=0) indicates that it was the last one (C4=R4=1 and C9=R9=0). C4 will therefore increase to 0 on the third line.

But if we want this third line and all the others to be considered as the last, we must ….

…. change R4 with 0 on the second line when C0>1. And that's it.

From the last line of a frame (C9=R9 and C4=R4):
This line N is considered as the last one of the frame and then, on the next line N+1, C9=C4=0. If R9 and R4 are updated to 0 on this line or on the line N+1 when C0<2, this line will be considered as the last one. C9 and C4 will both change to 0 on the line N+2.

This line and the next ones will be considered as the last ones.

Note : When C9 becomes equal to R9 (last line), it is necessary to wait until C0=2 to modify R9, because CRTC 0 requires special treatment with C0=0 and 1 to manage C9 and disable in particular the additional line management (enabled by default).

C4:	0																		
C9:	**7**																		
C0:	**0**	**1**	2	3	4	5	6	7	8	9	10	11	12	…	58	59	60	61	62	63
R9:	**7**	**7**	**0**	0	0	0	0	0	0	0	0	0	0	…	0	0	0	0	0	0
R4:	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0	0	0	0	0	0
	OUT R9,0														Update R12/R13 before C0=0					

How do we get out?
To stop getting lines C4=0 and C9=0, it is necessary to modify R4 and/or R9. Since each line is also a "**Last Line**", C4 and R9 will be reset to 0 on the next line, if the update of R4 and/or R9 takes place when C0>1.

With a view to common code for CRTC 2, it is advisable to:

- Modify R12/R13 before C0=R1.
- Manage the update of R9 during HSYNC as described in Chapter 12.4.2.
- Do not set R2=0 (so that the BORDER does not remain activated).

12.3 CRTC 1

If R4 is updated with the value of C4:

- If we were on line C9 between 0 and R9-1, then C9=C9+1 (C4 will go to 0 when C9 goes back to 0 and the offset (R12/R13) will be considered).
- If we were on the last line (C9=R9), then C9 goes to 0, C4=0 and R12. R13 is considered.

If R4 is updated with a value less than C4, then C4 will increment to its maximum value (127) before looping back.

Unlike CRTC 0, putting R4=0 on the last line of a screen will cause an "overflow" of C4, the value 0 being handled as a general case.

If we want to put R4 to 0 so that C4 loops to 0, we must do it only when C4 = 0.

Creating a line-to-line rupture on this CRTC is trivial.
Simply set R4 and R9 to 0 when C4 and C9 are both at 0.

12.4 CRTC 2

12.4.1 LAST LINE CONCEPT

As on the CRTC 0, there is a concept of "**Last Line**" which irremediably arms the reset to 0 of C4 and C9 on the following line.

When this "Last Line" state is set, it can no longer be modified.

This means that if this "**Last Line**" state is true, then C4 and C9 will change to 0 on the next line, regardless of the values subsequently programmed in R4 and R9 during the rest of the line.

The "**Last Line**" status is evaluated at the start of the line (at position C0=0) or **during an update of R4 and/or R9 if a "Last Line Management" status is true**.

If C4<>R4 or C9<>R9 on position C0=0, then the "Last Line" state is false.
The "Last Line Management" state is true unless C4=0 and C9=0 (in which case it is false). In other words, if we are not on the last line at the start of the line, it will be possible to update the "**Last Line**" state during the line (by modifying R9 and/or R4 outside of a HSYNC), except if we are on a first line (C9=C4=0) (in this case the updates of R9 and/or R4 are not considered when evaluating the "**Last Line**" state).

The "**Last Line Management**" state allows, if it is true, to evaluate the conditions of the "**Last Line**" state on positions C0>0 from an update of R9 and/or R4. There is an **exception to this rule**, if the R9 and/or R4 update that satisfies the "**Last Line**" condition occurs during a HSYNC (the "**Last Line**" state remains false).

If C4=R4 and C9=R9 on position C0=0, then the "Last Line" state is true (the "**Last Line Management**" state is false). However, there are 2 exceptions for which the "**Last Line**" state is false:

- If the previous line was a last line. A state (yes, another one) "**Last Previous Line**" is managed on the last position C0 of the **HSYNC**.
- If a **HSYNC** takes place on position C0=0.

During a **HSYNC**, a test is performed on position C0=R2+R3-1, in order to determine if line N is a last line for line N+1 (at C0=0). Thus, if C4=R4 and C9=R9, then "**Last Previous Line**" is true, otherwise it is false.

Furthermore, if C4<>R4 or C9<>R9 on this last position of the HSYNC, this updates the "**Last Line Management**" state by setting it to true. Thus, if "**Last Line**" was false and its management also because of an "already last" previous line (evaluated during the HSYNC), this makes it possible to force the re-evaluation of the last line in order to circumvent the exception on the C0 positions>=(R2+R3)

When the "**Last Line**" state is true, nothing can prevent C4=0 (and C9=0) from passing to the next line, unless an additional line is programmed via R5. If R5 is programmed with a value greater than 0 at any position of C0, then the additional lines programmed in R5 will precede the reset of C4 and C9. When displaying these additional lines, C4 will be incremented (overtaking R4)(See Chapter 11.2.3).

If an objective is to maintain the reset to 0 of C4 and C9 on each line, it is possible to force the "**Last Line**" state on a first line, by creating the condition at the end of **HSYNC** which allows it to be considered.

In other words, it is enough to authorize the management of the "**Last Line**" on the last character of the **HSYNC** to allow this CRTC to manage the resetting to 0 of C4 as expected. **Modifying R9 wisely before and after HSYNC achieves this goal.**

12.4.2 CASE STUDY: LINE-TO-LINE RUPTURE (R.L.A.L.)

The objective of this technique is that all the lines displayed start with C4=0 and C9=0, so that the registers R12 and R13 are modified before C0=R1 is considered.

Suppose we are on the "**Last Line"** of a frame with C4=R4 and C9=R9 (R4 and/or R9 are greater than 0). We know that on the next line, C9=0 and C4=0, but we want that for the next lines, C9 and C4 are also at 0 (and be able to change the address via R12 / R13).

It will be necessary to reprogram R9 and R4 with 0 so that the **last line** condition is active on each line. Each new line will be **a first and last line** of the frame.

For this example, the **HSYNC** will be positioned where C0=1 (via R2=1).

Consideration should be given to constraints related to other management flaws during **HSYNC**, such as not enabling the **VSYNC** pin and disabling the BORDER on C0=0.

We will consider that this **HSYNC** will have the minimum time necessary for a horizontal synchronization, namely 6 µsec (via R3 = 6) because it will be necessary to modify R9 twice.
This may as well be done as soon as possible when the index on the CRTC register is already selected.

If C9=C4=0 and the "**Last Line**" condition is true on the last character of the HSYNC, then the "**Last Line**" state can no longer be managed. By modifying R9 during **HSYNC** so that the last line condition is false, it is possible to authorize this management of evaluation of the "**Last Line**" state. By modifying R9 with a value other than that of C9, it is possible to activate the "**Last Line Management**" so that it can be considered.

Once the HSYNC is complete, returning the value of R9 to the same as that of C9 (so 0), allows to indicate what the correct limit of C9 is, so that the condition "**Last Line**" is treated correctly.

Example with 3 lines:

1st line :
It is assumed here that C4=R4=0 and C9=R9=7, and the HSYNC is shown in orange :

		R2																		
C4:	0																			
C9:	**7**																			
C0:	0	1	2	3	4	5	6	7	8	9	10	11	12	…	58	59	60	61	62	63
C3:		1	2	3	4	5	6													**R1**
R9:	7	7	7	7	7	7	7	0	0	0	0	0	0	…	0	0	0	0	0	0
R4:	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0	0	0	0	0	0
						OUT R9,0					Update R12/R13 before C0=R1									

This line is considered the last line of the frame because C9=R9 and C4=R4 when C0=0. This "**Last Line**" condition is true because there is no **HSYNC** on C0=0 and the previous line (C9=6) was not a last line (evaluation made at the end of HSYNC of C9=6).
R12 and/or R13 are modified before C0=R1 so that VMA' is updated with R12/R13 because we are... on the last line.

Note 1 : This assignment of R12/R13 depends on the "last line" state when C0=R1 (and not on the equality C4=R4 and C9=R9 which positions this state). This implies that changing R9 before equality C0=R1 does not prevent this assignment.

Note 2 : Changing R9=0 on this line is no longer considered for the "**Last Line**" state because this state has already changed to "true". It will however possible to create the "**Last Line**" condition on C0=0 of the following line.

2nd line :

		R2																		
C4:	0																			
C9:	**0**																			
C0:	0	1	2	3	4	5	6	7	8	9	10	11	12	…	58	59	60	61	62	63
C3:		1	2	3	4	5	6													**R1**
R9:	**0**	0	0	**1**	1	1	1	**0**	0	0	0	0	0	…	0	0	0	0	0	0
R4:	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0	0	0	0	0	0
		OUT R9,1				OUT R9,0					Update R12/R13 before C0=R1									

On this first line 0 (C4=R4=C9=R9=0), the "**Last Line**" state is true.

However, since the "**Last Line**" condition has not changed during the previous **HSYNC**, the previous line is considered a last line. As such, the current line is no longer considered as a last line unless the "**Last Line Management**" is reactivated during the **HSYNC** (via the condition C4<>R4 or C9<>R9 on the last position of the HSYNC).

Thus, on C0=6 (last character of the **HSYNC**) the test C9(0)=R9(1) is false.
This unlocks the "**Last Line**" evaluation.
The OUT R9,0 has the effect of modifying R9 when C0=7 and the **HSYNC** is finished.

On C0=7, the CRTC triggers the reset of C4 and C9 to 0 ("**Last Line**" condition satisfied). This reset to 0 of C9 (and of C4) can then no longer be "cancelled".

If, in the example above an OUT R9,10 was added behind the OUT R9,0 (R9 would be updated with 10 on C0=11), it would not change anything for C9 and C4, which would return to 0 on the next line.

Thus, on the 3rd line we will have C9=R9=0 and C4=R4=0.

Note 1:
If the HSYNC had been positioned after the R9 update with 0 on the 1st line, then it would not have been considered as a last line when evaluating C9=R9/C4=R4 at the end of HSYNC on line C9=7. It would then not have been necessary to modify R9 during the HSYNC of line 2 for the "**Last Line**" state to remain armed for the 3rd line.

3rd line :

		R2																		
C4:	0																			
C9:	**0**																			
C0:	0	1	2	3	4	5	6	7	8	9	10	11	12	…	58	59	60	61	62	63
C3:		1	2	3	4	5	6													**R1**
R9:	**0**	0	0	**1**	1	1	1	**0**	0	0	0	0	0	…	0	0	0	0	0	0
R4:	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0	0	0	0	0	0
		OUT R9,1				OUT R9,0					Update R12/R13 before C0=R1									

This line does not pose any problem either, because the "**Last Line**" condition is again false on C0=6 (HSYNC) and becomes true on C0=7 when R9 goes to 0 (to satisfy C9=R9/C4=R4 (Excluding HSYNC)). And so on...

Note 1 : For practical reasons it is perfectly possible to move the OUT R9,1 instruction by 4 µsec (on C0=4) (R9 is different from C9 on C0=R2+R3-1).

Finally, just as on CRTC 0 to "get out" of "**Last Line**" treatment, it must be remembered that when R9 is put back at a value greater than 0, this value is no longer considered if the CRTC considers that it is on a last line.

It will put C9 and C4 back to 0 on the next line, as CRTC 0 would.

If the "**Last Line**" condition was left active during the last HSYNC (there were two last lines with the same C9/R9), C9 will be treated in relation to the value of R9. C9 will be incremented if C9<>R9 as would be the case for a CRTC 1, but C4 will be unconditionally incremented if C9=R9 (and C9 will return to 0).

With a view to common code for CRTC 0, it is advisable to:

- Treat the update of R9 on the last line as on the previous lines so that the next line is considered a last line, in order to ensure compatibility with the "**Last Line**" operation of the CRTC 0.
- Do not modify R9 before C0=2 for the same reasons.

12.5 CRTC 3, 4

If R4 is updated with the value of C4:

- If we were on the line C9 between 0 and R9-1, then C9=C9+1, C4 will go to 0 when C9 will return to 0 and the offset (R12/R13) is considered.
- If we were on the last line (C9=R9), then C9 goes to 0, C4=0 and R12/13 are considered.

The modification of register 4 is considered immediately at the end of the line.

If we want to put R4 to 0 so that C4 loops to 0, we must do it when C4 = 0.

If R4 is updated with a value less than C4, then there is overflow of the C4 counter. (unlike what happens with C9/R9).

It should also be noted that additional management of vertical lines does not increment C4 beyond R4 as occurs with CRTC's 0, 1 and 2.

Creating a line-to-line rupture on this CRTC is even more trivial than on CRTC 1, because you simply set R4 and R9 to 0 when C4=0.

Indeed, C9 increases to 0 if C9>R9 which improves the level of compatibility between CRTC's 0 and 1.

13 COUNTING : REGISTER R0

13.1 GENERAL

The CRTC's R0 register is used to define the number of characters which the circuit will generate on a line. A C0 counter (also called HCC by some eccentrics :-)) counts from 0 up to and including the value of R0. This register contains the desired number of CRTC characters per "line" minus 1.

When C0 is reset to 0, different counters are updated (C4, "C5", C9, ...).

The C0 counter is also comparable to R1 (border/video pointer management) and R2 (HSYNC management).

One might expect the condition Cx=Rx to define the incrementing or resetting of other counters.

However, if C0 does not reach certain values defined internally, this poses some problems, especially on CRTC 0 because processing take place on specific values of C0.

The purpose of this processing is to allow the reset of counters or simply to manage the increment of other counters.

It is worth recalling that there is a discrepancy between the CRTC's internal registers and the actual display of the corresponding characters by the GATE ARRAY and/or ASIC. The CRTC counters are ahead of the GATE ARRAY display.

The GATE ARRAY does not respect this "shift" to generate the HSYNC's. Thus, the HSYNC occurs on the C0 of the "CRTC" and not that displayed by the GATE ARRAY.

ASIC's, however, respect this delay to generate the HSYNC when the character corresponding to C0=R2 is displayed, which corresponds to C0+1 of the CRTC.

13.2 CRTC 0

13.2.1 THE FIRST 3 MICROSECONDS

This CRTC performs operations on precise positions of C0 (arming resets and increments of counters, inhibition of processing).

The detail of this operation is necessary to better understand the specific processing of resetting C4 to 0, which only occurs when the CRTC assesses that it is on **the last line of the frame**.

The designers of the circuit probably "preferred" to manage under conditions a reset of C4, because this counter was planned to **exceed R4** on all CRTC's "not-ASIC'd".

This is particularly the case when R5 is greater than 0 (it then becomes the new limit of C9+1) or when the Interlace mode is active during an "even" frame.

These different "tests" were spread over several values of C0:

- **When C0=0**, then counters C4 and C9 are updated according to states decided on the previous line. Then the counters are evaluated to schedule their update on the next line. If C9 has reached R9 then C4 will be programmed to be incremented on the next line. The incrementing of C4 will eventually be cancelled on C0=2 if this line was not the last on the frame (C4<>R4 or C9<>R9) or if there are additional lines to add (R5>0 and/or Interlace Line "on")

- **When C0=1**, then the management of C9 is again authorized for the next C0=R0. If this "authorization" is not given (context R0=0) then counter C9 is no longer managed on the next C0. If R0 is set to 0 on position C0=0, C9 is previously updated with respect to the last value of R9 if R0 of the previous line was >1. As for C0=0, the last line state is set to true (if C4=R4, C9=R9) or false otherwise.

- **When C0=2**, on the last line of frame, the CRTC determines if additional line management should take place. If this additional management is activated, then C4/C9 counting will follow a specific logic (see chapter 11.2.2).

Therefore :

- If **R0=0**, then C0 never reaches 1 (and therefore remains at 0) and **C9 can no longer count**. It then remains **frozen with the value it had before R0=0.**
    - If C9<>R9 on the first C0=0 for which R0=0, then **all of the CRTC counters are frozen** as long as R0=0. For example, freezing R0=0 for 64x8 µsec amounts to "forgetting" 8 lines (C4-1 if R9=7)

    - If C9=R9 on the first C0=0 for which R0=0, **then C4 is incremented once on the second C0=0**. C4 is only incremented once because the additional management is "deactivated" only on C0=2 (an additional management therefore only increments C4 once if C4==R4 and C9==R9.After this second C0=0, **all of the CRTC counters are frozen as long as R0=0**.

  Note : Since C9 management takes place only once if C0 does not exceed 1, updates to registers R4, R5 and R9 are no longer considered as long as R0=0. On the other hand, R8 continues to be considered each time C0=0.

- If **R0=1**, then C0 never reaches 2, and the additional management of line(s) set by default in C0=0 remains engaged if C4=R4 and C9=R9 (C0=0 or C0=1). C9 can increment according to R9. If R5=0, the additional management then lasts 1 line of 2 µsec before ceasing (C4+1, C9=0). On the next line, the end of additional management reset C4 and C9 to 0.

The conditions for **resetting C4 to 0** are the followings ones :

- When C0<2, C4=R4 and C9=R9 conditions the "**Last Line**" state, which then activates an "**additional management**" state for which C4 is incremented unconditionally. Note that during this additional management, C9 continues to be tested with R9 for the update of VMA/VMA' (but C9 is not zeroed when C9==R9).

- When C0=2,the additional management state is deactivated if there was no line programmed (R5=0 or no "Interlace Line".

If the CRTC was not on the last line (or on the last line of the frame with an additional line engaged), the fate of C4 is sealed: **If C9==R9 then C4=C4+1** (and in additional management one only once if C4 was worth R4)

If R5 is programmed on the last frame line with a value greater than 0 **before C0=3** (C0=0, 1 or 2), then R5 additional lines will be considered (with C4=R4+1 on the following line if C9==R9 at the end of line).

On the other hand, if R5 becomes greater than 0 when C0>2, then no additional line will be added to the frame. The next line will correspond to a new frame with C4=C9=0.

13.2.2 FREEZE OF VSYNC

Other operations take place on the CRTC 0 between the values 0 and 2 of C0.
Particularly to determine if a **VSYNC** will be authorized for the next occurrence of C0=0.

Each time C0=2, a state validates the update of C4=R7 on the next C0=0. This state is cancelled when C0=0.

Thus, if R0 is modified with a value lower than 2 on the line preceding the equivalence C4=R7 (On the line where C9=R9 and C4=R7-1), the VSYNC will not be authorized on C0=0. This will also cause the VSYNC to block for the value C4=R7.

A modification of R7 when C0<2 having the purpose of "triggering" a VSYNC after C0=0 (when C4<>R7) will also cause the blocking of the VSYNC for this value C4=R7.

Unblocking the **VSYNC** is then no longer possible for this value of C4, except for two specific conditions. See Chapter 16.3 for more description of these conditions.

13.2.3 FREEZE OF ADDITIONAL ADJUSTMENT LINE

If an additional line is programmed (R5=1 for example), we will have on this line C4=R4+1 and C9=0. If R0 goes to 0 on C0=0 of this line, then C9 remains fixed at 0 and C4 can only go to 0 when C9 is managed again (as soon as C0=1).

C4 and C9 remain fixed as for a "non-additional" line for which C9<>R9. Modifying R5 for a period during which R0=0 has no effect on the value of the counters while C0 remains at 0, because C9 is no longer managed in relation to R5 as long as C0 has not returned to 1.

Note that when C9 management is inhibited, the update of both R9 and R4 are also ignored while R0=0.

13.2.4 FREEZE OF C9

The fate of C9 is specific when R0=0 since it is no longer incremented.
On the first step of C0=0, when R0 becomes equal to 0, the value of C9 is calculated for the first time with respect to R9 (for example if C9 was worth 4 and R9=7, C9 goes to 5).

<u>On the first C0=0 :</u>

- The CRTC manages different updated states on previous C0=0,1,2 to update C4 and C9 on C0=0. Each state is cancelled once the counter is updated. In principle, 4 states are possible :
  - Counting C4
  - Reset C4
  - Counting C9
  - Reset C9

  Note that the processing of the status of C9 and its management of counting is subordinated to another status of 'management C9'.
- Tests are performed to determine the next states of C4/C9, specifically with respect to counting management C9/R9 or with respect to that of C9/R5 if additional management

is decided (in this case, it is the value of R5 with respect to C0 which is used to set the end of the additional management)
- Additional management is decided if C4=R4 and C9=R9. It would in principle be confirmed on C0=2 if C0 managed to reach this value.
- C9 processing management is disabled. It would in principle be activated on C0=1 if C0 succeeded in reaching this value.

On the second C0=0
- The management of C9 has not been reactivated (because C0 has not reached 1). The value of C9 therefore remains the same as that which it had on the first "C0=0"
- Two situations then arise :
  - If C9 had reached R9 on the first C0=0, then the reset of C9 had been armed as well as the increment to C4. With C9 being frozen, only C4 will increment.
  - If C9 was different from R9 on the first C0=0, then the increment to C9 will have been armed (but not the increment to C4). With C9 being frozen, no counter will have been moved.

On the third C0=0 (and the following ones) :
- The management of C9 has not been reactivated (because C0 has not reached 1). The value of C9 therefore remains the same as that which it had on the first "C0=0".
- In the situation where C4 has been incremented on the second "C0=0", this increment is deactivated because it has taken place. All counters are frozen.

Reminder : R5 is the quantity of line(s). C9 is compared with R9 before its increment and will become the current line if C9 thus calculated has not reached R5. In this case, C9 and C4 return to 0 before the line is displayed.
R5=0 indicates that there is "in principle" no line.

If R0 is modified to become greater than 1 again, **increment management of C0 resumes normally** without the other counters being affected.

However, **if the conditions of a last line are satisfied on C0=0 or C0=1, then C9 will no longer be managed with R9 but with R5**. Going through C0=2 does not cancel it.
It will stop only when C9 computed=R5 on C0=0.

The vertical adjustment management has 2 particularities on the CRTC 0:
- **C4 is incremented only once, whatever the value of R5.**
  C4 returns to 0 once the adjustment is complete, whatever the value of R4.
  This means that C4 is no longer managed when C9 reaches R9 during an additional management. Note however that the CRTC continues to manage VMA/VMA' transfers specific to C0=R1 when C9=R9. (See Chapter 11, page 78)

Note : On CRTC's 1, 2, 3 and 4, the "**Last Line**" assessment takes place during all C0 values, **whereas CRTC 0 only manages it on C0=0 and C0=1 (and completes the status on C0=2 for additional lines).**

However, CRTC 2 arms a last-line reset to 0 that can no longer be disarmed afterwards, regardless of the values of R9 and R4.

Respecting the strongest constraint for the "**Last Line**", if R4 or R9 is changed so that its value is updated to C0=0 (OUT triggered on C0vs=#3E on CRTC's 0, 1, 2 and on #3D on CRTC's 3, 4 for

R0=#3F) or C0=1 then only one common instruction is required without updating specifically the code. If, however, it is necessary to modify R9 and R4 to satisfy "**Last Line**" conditions, it is necessary to anticipate R9 or R4 update.

See Chapter 12, page 90, and Chapter 10, page 71.

13.2.5 CASE STUDY: R0=1

When R0 is 1 (and C0<2) we obtain "lines" of 2 characters (2 µsec).

If C4=R4 and C9=R9 when C0=0, then we are potentially on the last line of the frame. But C0 not having reached 2, the additional line management is not disarmed and will take place.

If C4=R4 and C9=R9, then the "frame" ends when C0 reaches R0. This is particularly the case when R4 and R9 are 0, to create "frames" of 2 µsec. (The first line is then also the last)

Therefore :

- When C0=0 (or C0=1), the CRTC assesses whether it is on the last line, and if so, arms an internal flag by default to trigger vertical adjustment.
- When C0=2, the CRTC assesses the conditions for disarming this vertical adjustment mechanism (in particular by testing the value of R5).

Consequently :

- If C0 never exceeds 1, and R4=R9=0, the CRTC generates a "1 line" vertical adjustment every "frame" when C4=R4 and C9=R9.
- The CRTC can no longer disarm the additional management (test of R5 and/or "Interlace on even frame programmed") and a 2 µsec "frame" is generated. This results in a one-time increment of C4 beyond the value programmed in R4. **However, the adjustment stops after 1 line**. For some reason I haven't determined yet, C9 (not C9+1) is compared to R5 to stop the adjustment.
  Note: C4 and C9 return to 0 when the adjustment is complete

When R4=R9=0, each "line" of 2 µsec is therefore immediately followed by a "line" of 2 µsec for which C9=0 and C4=1.

Once this "additional" line of 2 µsec is completed, C4 and C9 reach 0 and the offset is updated. In this situation, the R12/R13 registers can therefore be considered "only" after 4 µsec, whereas this is possible every 2 µsec on a CRTC's 1, 2, 3 or 4 (and even 1 µsec when R0=0 on these CRTC's). If R9>0, this "line" of 2µsec (for which C4=1) occurs after the last value of C9 (when C9=R9).

13.2.6 CASE STUDY: R0=0

When R0 is 0 and C0=0, then C0 remains at 0 (frame or line of 1 µsec).

Things get a bit more complicated:

- **C9 is frozen**
  Since C0 never reaches 1, C9 is no longer managed.
  If, for example, it was equal to 3 when R0 went to 0, it will remain at 3 regardless of the value of R9, as long as R0=0
- **C4's last hiccup**
  If C9=R9 on C0=0 (when R0 goes to 0) then C9 is no longer managed, but C4 will however be incremented regardless of the value of R4. C4 is incremented without C9 returning to 0. **Magical !**
  After that, no more counter will be managed except C0.
- **Additional management**
  If C9=R9 and C4=R4 (when R0 goes to 0), then we are in the same situation as in the previous paragraph, except that an additional management is activated, and which will remain so when C0 can once again exceed 1. It is then R5 which controls the end of the additional management. To stop this management, program R5 with C9+1

To summarize : **C9 remains "frozen" with the value it had before R0=0 and therefore no longer increments, nor returns to 0.**

If C9=R9 and C4<>R4 then C4=R4+1. When R0>0, C4 continues to be managed via C9/R9.
If C9=R9 and C4=R4 then C4=R4+1. When R0>0, C4 is managed by C9/R5.
In the other cases, C4 remains fixed

Example **Context : C0=R0=C4=R4=C9=R9=R5=0**			
Counters	C0	C9	C4
1st character C0=0	0	0	0
The current video pointer is updated. CRTC-VMA'=CRTC-VMA=R12/R13 When C0 loops first time on R0, C0 goes from 0 to … 0. C4=0. The CRTC is at the end of the "frame" (it has just finished the R4/R9 lines of a 1 µsec frame (C4=R4/C9=R9))			
2nd character C0=0	0	0	1
C9=R9 and C4=R4. C4 is incremented (i.e. 1). We are on additional management. Note that C9 is not "really" reset to 0 because it is no longer managed.			
3rd character C0=0 (and next ones…)	0	0	1
The CRTC is in additional management, but C9 is frozen.			
When R0 becomes > 2 again	0	0	1
The additional management being in progress, it can no longer be cancelled on C0=2 C9+1 will be compared to R5 on the next C0=0			
Next C0=0 after C0=R0>2	0	1	1
C9+1 being different from R5, then C9 is incremented.			
Example 2 (movie sequel) **We will put R0=0 on the first C0 after C0=R0>2**			
Counters	C0	C9	C4
1st character C0=0	0	1	1
Take values from the end of the first example			
2nd character C0=0	0	1	1
C9 is frozen. Additional management is still ongoing.			

13.2.7 CASE STUDY: VERTICAL RUPTURE LAST LINE (R.V.L.L.)

A BIT OF TECHNICAL STORY… THE R.V.I.

Initially developed on CRTC 0 using a knife and string, an ancient ancestral technique called "RVI" (Invisible Vertical Rupture © Overflow) is intended to allow the choice of the C9 number of the visible line by creating small lines in the non-visible part of the screen (during HSYNC), while changing the address of the visible line. To achieve this, there are at least ***3 different methods****, which can be combined: Either by modifying R9, or by changing the size of the lines (R0), or by limiting their number (synchronous action Z80A / CRTC). This allows you to individually control the address of many more rows than the 2k (among 16k of a page) allowed by the value C9=0*

One of the major constraints of this technique is ***the interdependence between the lines****. Also, you will really enjoy the next Chapter.*

Indeed, the value of C9 calculated for the new row displayed depends on the value of C9 of the previous row. Creating an algorithm that automatically determines the desired evolutions of C9 according to the current C9 further alters the CPU available on a line of 64 µsec.

It is in principle the value of R9 that sets the maximum value of C9. It should be heed that only the 3 bits of C9 are "preserved" for the composition of the address.

One of the corollaries of this technique is usually the update of CRTC-VMA via an update of R12 and / or R13. For this, it is necessary that C9 and C4 return to 0.
Note : For CRTC 2, it is also necessary for R12/R13 to be modified before C0=R1 and C0 reaches R1 when C9=R9, but you are not on the right Chapter.

For example, if C9=8 corresponds to C9=0 for address constitution, C4 does not return to 0 and therefore R12/R13 cannot be considered. For a change of address to be considered, it is necessary that ***C9 "loops" and returns to 0, as well as C4****. In addition, if the C9 displayed is a "****Last Line****", it will cause C9 to be reset to 0.*
Note : For CRTC 1, there is no need for C4 to go back to 0 for the address to be taken into account. This is true whenever C0=0 while C4=0.

When 2 µsec frame-lines are created in the edge of the screen (or not if we want to expose the underside of the case ...) while R9>0, then C9 will increment until it reaches R9..

But in this situation, when C9=R9 (so here C0=1, since R0=1), an additional "line" will be generated with C4=1 and C9=0. This "additional" management results in the creation of an additional "line" when C9 reaches R9.

The change of address does not take place because C4=1. It is on the next line frame that C4 will return to 0 (with C9). One consequence is that the number of C9 increments for these "2µ lines" is reduced by 1 compared to CRTC's 1, 2, 3, 4.

This does not allow access, with R0 = 1, to C9 = 7 or 5 by changing the offset, because it is necessary on the one hand that ***R9 is equal to 7 or 5, but also that it could have looped back to 0 before****.*

The final values of odd C9 do not help matters, unless you are a pro of the Interlace "controlled" (blah, blah, lawyer, blah, blah), but I digress...

As this technique was developed on CRTC 0, the use of R0=0 was quickly abandoned because of the difficulty in understanding the behaviour of counters at the time..

*On CRTC's 1, 2, 3 and 4, this value (R0=0) is not a problem. Neither blocking C9 nor additional line C4=1, but it is more subtle for CRTC 2. It is therefore much simpler to create this type of rupture in the edge of the screen allowing access to all the values of C9 on the main line. The reset to 0 "triggered" on the "**Last Line**" is particularly interesting here compared to CRTC 1.*

At first glance, the notion of "**Last Line**" on CRTC 0 (and 2) seems restrictive, because it does not allow "immediate" consideration of updates to the R4 and R9 registers.

In addition, CRTC 0 is limited to do rupture at least 2 µsec for "correct" operation of the C9 counter. Finally, if a "last line" occurs when this line is 2 µsec, then a new line of 2 µsec is generated (C4=1).

However, the principle of "**Last Line**" represents a great tool to **program in advance a reset of C4 and C9**, especially in the context of a vertical rupture, which I will call « Rupture Verticale Last Line » (in perfect Frenglish).

Indeed, if we consider that each visible line is a "**Last Line**", then **each first rupture of 2 µsec will be the first line of a frame with C4=C9=0**.
Thus, it is possible to anticipate the modification of R9 knowing that the next C9 will be equal to 0.

In this situation, the selection of the next C9 is greatly simplified since it is enough to set the desired number in R9 and create the appropriate number of ruptures (during HSYNC if the objective is to hide them) in order for C9 on the next line to be equal to that programmed in R9.

The update of the offset (R12/R13) then takes place during the first rupture of 2 µsec. Since the last line (C9=R9) is always the visible line larger than 2 µsec, the CRTC does not generate an additional rupture of 2 µsec.

This means that it takes 7 ruptures of 2 µsec to reach C9=7 (14 µsec in total).

This also means, except to cheat in a barbaric way with the value of the offset, that the rupture of 2 µsec begins essentially early in the 14 µsec in order to bring C9 to the desired value.

For this purpose, the value of R0 must therefore correspond to the moment when the ruptures of 2 µsec begin. Otherwise R0 must be equal to 63 – (R9 x 2) (for R9>1).

In the diagram below the pink areas correspond to the OUT instructions on R0. R0-1 corresponds to the update of R0 at the beginning of the line (and therefore started at the end of the previous line), and R0-2 corresponds to the value of R0 for "hidden" ruptures.

R0-1	R0-2	R9				R2		OUT R0-2											OUT R0-1		New C9
63	x	0	C0:	...	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	**0**
50	12	1	C0:	...	47	48	49	50	**0**	1	2	3	4	5	6	7	8	9	10	11	12	**1**
59	1	2	C0:	...	47	48	49	50	51	52	53	54	55	56	57	58	59	**0**	1	0	1	**2**
57	1	3	C0:	...	47	48	49	50	51	52	53	54	55	56	57	**0**	1	0	1	0	1	**3**
55	1	4	C0:	...	47	48	49	50	51	52	53	54	55	**0**	1	0	1	0	1	0	1	**4**
53	1	5	C0:	...	47	48	49	50	51	52	53	**0**	1	0	1	0	1	0	1	0	1	**5**
51	1	6	C0:	...	47	48	49	50	51	**0**	1	0	1	0	1	0	1	0	1	0	1	**6**
49	1	7	C0:	...	47	48	49	**0**	1	0	1	0	1	0	1	0	1	0	1	0	1	**7**
						C3:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
																						R3=15

Note : The 2nd OUT statement that begins on the 63rd character of the line is intended to resize the line for the next C9. This statement cannot start on the 64th character without causing C4 to overflow on the resized line (See Chapter 13.7).

On a CTM monitor, the visible part of a line is 48 µsec, and 16 µsec is not visible. However, for the frame to be centered on screen, the HSYNC must start on the 51st character, when C0 reaches 50 (if R0=63). And so that R2 is programmed with this value.

To have the 14 µsec necessary to reach C9=7, R0 must be programmed with 49 and it would in principle be necessary that R2 is at 0 on this precise line (while inhibiting HSYNC by protecting the area with a value of R3 covering all other occurrences of C0=0).

A HSYNC is not necessary on each line, but it is possible to position R2 with 50, knowing that the lines C9=7 benefit from the synchronization of the other lines, but this implies that there are not several consecutive C9=7 lines. This is what is done in the R.V. demonstration provided with SHAKER, or lines 7 are out of sync. This is not a problem on native monitors.

To be able to use all the ram without this constraint, simply set R2=49.
If C9=7 is not used, it is possible to set R2=50.

13.3 CRTC 1

R0 accepts all values without causing any problem for other counters.

If R0 is 0, then C9 and R4 continue to be managed normally.

The offset can be modified according to the timings indicated in the diagrams of the following pages.

Note 1: It is possible to create 14 "hidden" ruptures, allowing to simply access all the values of C9, especially since the offset is considered while C4 = 0 (and not when it goes back to 0).

Note 2: The update of one of the offset registers taking at least 4 µsec, the only Z80A update instruction covers 4 "C0 line frames" when R0=0.

Note 3: Using the OUTI instruction to modify R0 has unintended consequences compared to the same change made with an OUT(C),R8. The modification of the R0 register is effective on the 5th µsecond of the OUTI statement, and on the 3rd µsecond of the OUT(C),R8 instruction.

In principle, the OUT(C),R8 instruction must start 2 µseconds later than the OUTI instruction to obtain the same result. But when the OUTI instruction is used on CRTC 1 to modify R0, **the comparison of C0 with R0 takes place after the assignment of R0 with the new value in some cases**. Consequently, on the position where C0 should have gone to 0, if R0 is modified on the last µsecond of the OUTI instruction, then C0 is compared with the new value of R0, which can lead to an overflow of C0.

Example: if R0 was 49 and the 5th µsecond of the OUTI is at the position following C0=49, and R0 is modified with 20, then C0 will not be equal to 0 but to 50. In other words, the value of R0 is considered during the 6th µsecond after the start of the OUTI instruction in some case :-).

13.3.1 CASE STUDY : VERTICAL INVISIBLE RUPTURE (R.V.I.)

The R.V.I., on a CRTC 0, is very restrictive because placing R0 < 2 poses some constraints and limits the number of ruptures that can be made in the "invisible" part of the line. Indeed, when R0 is 1, an additional line is generated, and when R0 is 0, C9 is affected. This technique implies that C9 goes back to 0 so that R12 and / or R13 are considered, and this imperatively requires a high number of ruptures. It is impossible on CRTC 2, in particular because of the constraints of considering R12/R13.

When it comes to writing a program running on the CRTC's other than CRTC 0, the R0=0 value is not a particular problem. If the objective remains to access all C9, we will nevertheless prefer the R.V.L.L. on CRTC 0 and 2 which avoid a transitional management algorithm between the C9. CRTC's 1, 3 and 4 do not have this option.

CRTC 1, compared to CRTC 3 and 4 (and although CRTC 3 has many other ways to change the offset of each line) has an additional advantage to manage a R.V.I (if I have not already written it). Indeed, the offset is considered as long as C4=0, whatever the value of C9, unlike all other CRTC's. (See Chapter 20.3).

This allows the modification of the offset without C9 having "looped" to 0 and allows to limit the number of ruptures necessary to reach the desired C9 with a "new" offset, which allows for the placement of R2 at 50 for perfect screen centering.

The following tables describe the 64 transitional states between two C9. The method used involves updating R0 one or two times per line, as well as updating R9, R12 and/or R13.
A joy for all SudokuZ80A fans!

The values of the new C9s (shown in red) correspond to a CRTC 1 "C4=0" possibility. For CRTC 3 and 4, simply replace the values with those listed below each table. It should be noted that the passage C9=0 to 7 implies that R0 of the main line is equal to 49, limiting de facto the possibility of setting R2 to 50. This is not at all inconvenient for a CRTC 4, since it is the value that allows centering on this circuit. (See Chapter 15)

C9=0

Old C9	R0.1	R0.2	R9
0	63		0
0	63		1
0	59	0	2
0	57	0	3
0	55	0	4
0	59	0	5
0	58	0	6
0	57	0	7

	…	47	48	49	**50**	51	52	53	54	55	56	57	58	59	60	61	62	63	New C9
					R2								OUT R0.2				OUT R0.1		
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	0
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	1
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	**0**	0	0	0	2
C0:	…	47	48	49	50	51	52	53	54	55	56	57	**0**	0	0	0	0	0	3
C0:	…	47	48	49	50	51	52	53	54	55	**0**	0	0	0	0	0	0	0	4
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	**0**	0	0	0	5
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	**0**	0	0	0	0	6
C0:	…	47	48	49	50	51	52	53	54	55	56	57	**0**	0	0	0	0	0	7
C3:					0	1	2	3	4	5	6								
											R3								

Compatibility CRTC 3 & 4

Old C9	R0.1	R0.2	R9
0	59	0	1
0	53	0	5
0	51	0	6
0	49	0	7

	…	47	48	**49**	50	51	52	53	54	55	56	57	58	59	60	61	62	63	New C9
				R2															
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	**0**	0	0	0	1
C0:	…	47	48	49	50	51	52	53	**0**	0	0	0	0	0	0	0	0	0	5
C0:	…	47	48	49	50	51	**0**	0	0	0	0	0	0	0	0	0	0	0	6
C0:	…	47	48	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7

C9=1

Old C9	R0.1	R0.2	R9
1	63		1
1	59	0	4
1	63		2
1	58	0	3
1	56	0	4
1	54	0	5
1	59	0	6
1	58	0	7

	…	47	48	49	**50**	51	52	53	54	55	56	57	58	59	60	61	62	63	New C9
					R2								OUT R0.2				OUT R0.1		
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	0
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	**0**	0	0	0	1
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	2
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	**0**	0	0	0	0	3
C0:	…	47	48	49	50	51	52	53	54	55	56	**0**	0	0	0	0	0	0	4
C0:	…	47	48	49	50	51	52	53	54	**0**	0	0	0	0	0	0	0	0	5
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	**0**	0	0	0	6
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	**0**	0	0	0	0	7
C3:					0	1	2	3	4	5	6								
											R3								

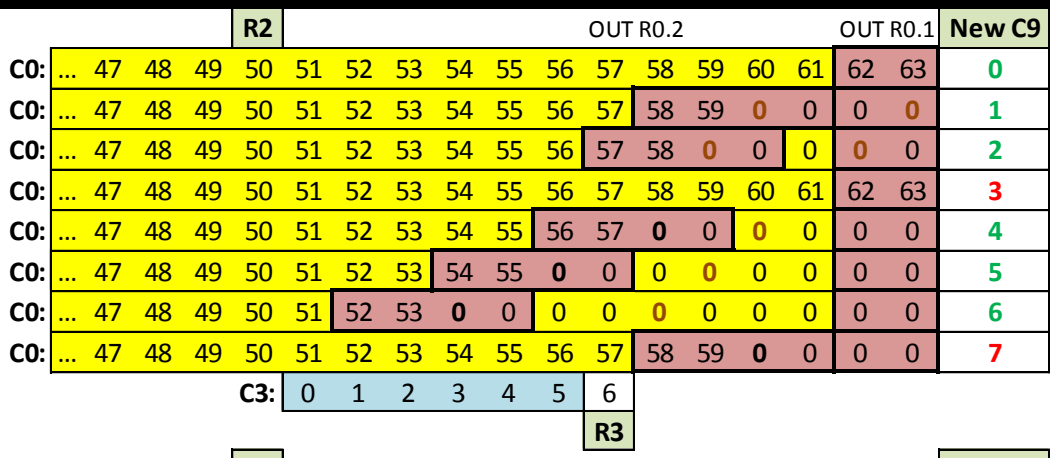
Compatibility CRTC 3 & 4

Old C9	R0.1	R0.2	R9
1	59	0	3
1	52	0	6
1	49	0	7

	…	47	48	49	**50**	51	52	53	54	55	56	57	58	59	60	61	62	63	New C9
					R2														
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	**0**	0	0	0	2
C0:	…	47	48	49	50	51	52	**0**	0	0	0	0	0	0	0	0	0	0	6
C0:	…	47	48	49	50	**0**	0	0	0	0	0	0	0	0	0	0	0	0	7

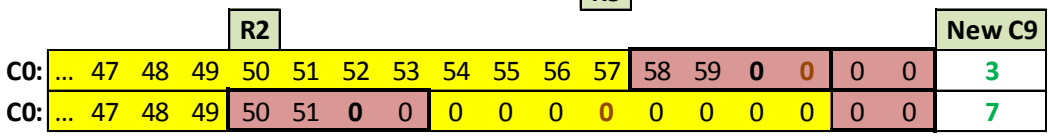
C9=2

Old C9	R0.1	R0.2	R9	New C9
2	63		2	0
2	59	0	2	1
2	58	0	2	2
2	63		3	3
2	57	0	4	4
2	55	0	5	5
2	53	0	6	6
2	59	0	7	7



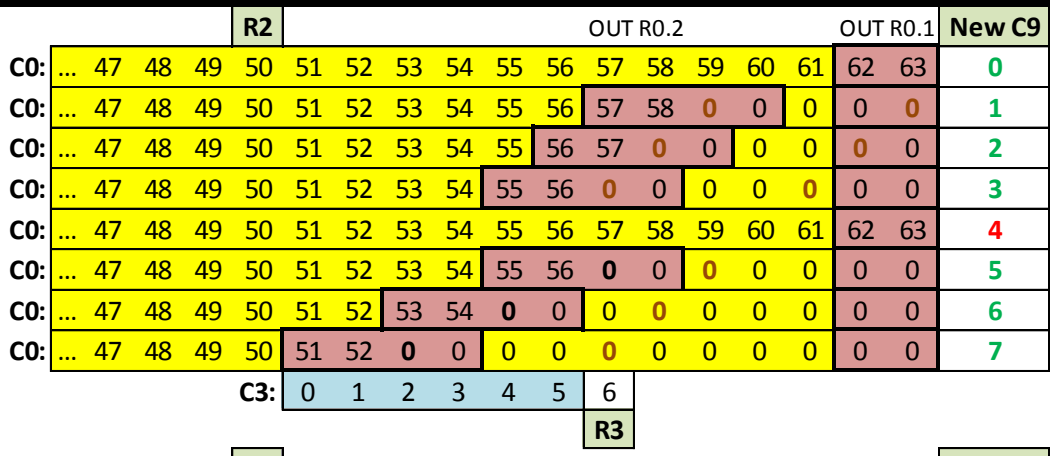
Compatibility CRTC 3 & 4

Old C9	R0.1	R0.2	R9	New C9
2	59	0	3	3
2	51	0	7	7



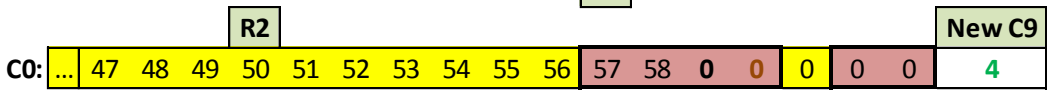
C9=3

Old C9	R0.1	R0.2	R9	New C9
3	63		3	0
3	58	0	3	1
3	57	0	3	2
3	56	0	3	3
3	63		4	4
3	56	0	5	5
3	54	0	6	6
3	52	0	7	7



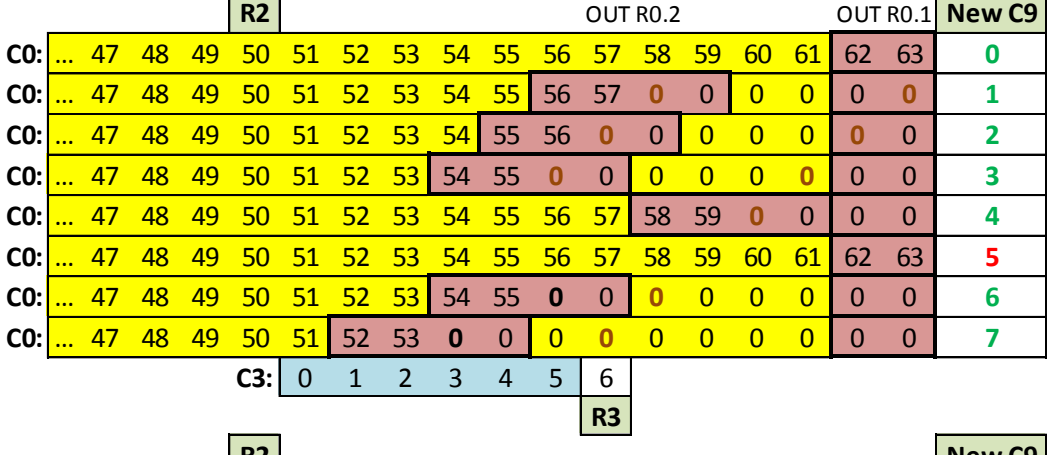
Compatibility CRTC 3 & 4

Old C9	R0.1	R0.2	R9	New C9
3	58	0	4	4



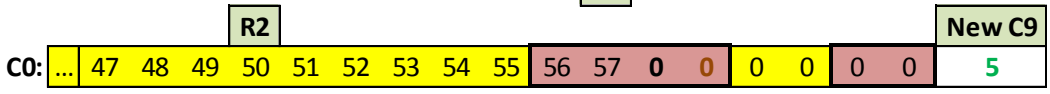
C9=4

Old C9	R0.1	R0.2	R9	New C9
4	63		4	0
4	57	0	4	1
4	56	0	4	2
4	55	0	4	3
4	59	0	4	4
4	63		5	5
4	55	0	6	6
4	53	0	7	7



Compatibility CRTC 3 & 4

Old C9	R0.1	R0.2	R9	New C9
4	57	0	5	5



C9=5

Old C9	R0.1	R0.2	R9
5	63		5
5	59	0	8
5	59	0	7
5	59	0	6
5	59	0	5
5	58	0	5
5	63		6
5	54	0	7

				R2								OUT R0.2					OUT R0.1		New C9
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	0
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	1
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	2
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	3
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	4
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	0	0	0	0	0	5
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	6
C0:	…	47	48	49	50	51	52	53	54	0	0	0	0	0	0	0	0	0	7
C3:					0	1	2	3	4	5	6								
R3																			

Compatibility CRTC 3 & 4

Old C9	R0.1	R0.2	R9
5	56	0	6

				R2															New C9
C0:	…	47	48	49	50	51	52	53	54	55	56	0	0	0	0	0	0	0	6

C9=6

Old C9	R0.1	R0.2	R9
6	63		6
6	59	0	9
6	59	0	8
6	59	0	7
6	59	0	6
6	58	0	6
6	57	0	6
6	63		7

				R2								OUT R0.2					OUT R0.1		New C9
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	0
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	1
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	2
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	3
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	4
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	0	0	0	0	0	5
C0:	…	47	48	49	50	51	52	53	54	55	56	57	0	0	0	0	0	0	6
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	7
C3:					0	1	2	3	4	5	6								
R3																			

Compatibility CRTC 3 & 4

Old C9	R0.1	R0.2	R9
6	55	0	7

				R2															New C9
C0:	…	47	48	49	50	51	52	53	54	55	0	0	0	0	0	0	0	0	7

C9=7

Old C9	R0.1	R0.2	R9
7	63		7
7	59	0	10
7	59	0	9
7	59	0	8
7	59	0	7
7	58	0	7
7	57	0	7
7	56	0	7

				R2								OUT R0.2					OUT R0.1		New C9
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	0
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	1
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	2
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	3
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	0	0	0	0	4
C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	0	0	0	0	0	5
C0:	…	47	48	49	50	51	52	53	54	55	56	57	0	0	0	0	0	0	6
C0:	…	47	48	49	50	51	52	53	54	55	56	0	0	0	0	0	0	0	7
C3:					0	1	2	3	4	5	6								
R3																			

13.4 CRTC 2

R0 accepts all values without causing any problem for other counters.

However, one of the peculiarities of this CRTC is to inhibit many treatments during HSYNC. See Chapters 15.6 (page 149) and 14 (page 124).

In particular :

- A GHOST VSYNC is generated when C4=R7 if this happens during the HSYNC.
  See Chapter 16.4.3 (page 163).
- C4's reset management (when on the last line of a frame) is disabled under certain conditions. See Chapters 12 (page 90) and 10 (page 71).
- The border deactivation condition (when C0=0) is also no longer supported, and the BORDER from the previous row remains active. See Chapter 15.5.2, page 149.

This CRTC displays 1 byte (0.5µsec) of BORDER before C0=0, but unlike CRTC 0, which does the same thing, the R8 SKEW DISP function does not exist to work around this problem. So, we cannot (to my knowledge) prevent the appearance of 1 byte of BORDER.
See Chapter 19, page 186.

R12/R13 content is "transferred" to CRTC-VMA' when C0 reaches R1.
However, there is a special case when R1 is 0 on the last line of the frame.
See chapter 17.4.3 , page 177 and chapter 20.3.3, page 236.

CRTC-VMA' is transferred to CRTC-VMA at the beginning of the frame.
This implies that it is impossible to change offset if C0 does not reach R1.
See Chapter 17.4.3, page 177.

Changing the offset therefore implies that at least one microsecond of BORDER exists before C0=R0, so that C0 can be equal to R1.

The treatment of C4 and C9 on this CRTC partly follows the logic of CRTC 0, insofar as there is a notion of arming the reset to 0 or the increment of C4 on the condition of "**Last Line**".

The CRTC determines if it manage a "**Last Line**" on position C0=0 or via an update of R4 and/or R9 outside HSYNC and under conditions (See chapter 12.4.1).

Depending on the "**Last Line**" state, it arms the reset of C9 (and C4) to 0 for the next line, regardless of the values that are programmed afterwards in R9 and R4 during the line.

This concept is of paramount importance to enable the management of an RVLL on this CRTC.

When it is necessary to create the conditions allowing an offset change at each "line", C4, R4, C9, R9 must all be at 0 when C0=0, and C0 must have met R1 after R12 and/or R13 are updated.

Without a clever trick, C4 will overflow on the second line.
See Chapter 0, page 75.

If R0 is positioned at 3, for example, CRTC 2 will generate 16 "lines" of 4 µsec.

In this situation, care must be taken to ensure that R2 is greater than 0:

- So that HSYNC must start when C0>0 in for VSYNC to be considered when C4 reaches R7. However, this can be "bypassed".
- So that the BORDER is disabled (or at least does not remain enabled) when C0=0, which is a more annoying constraint.

13.4.1 CASE STUDY: VERTICAL RUPTURE LAST LINE (R.V.L.L.)

CRTC 2 collects an anthology of various constraints that must be considered if we want to achieve a vertical rupture...

Let's say it right away, performing a "traditional" RVI on this CRTC is impossible for a simple reason: For the offset update, C0 must reach R1 **on the last line. Modifying R12/R13 after C0>R1 does not allow consideration for the next line**. However, the RVI is based on the principle of reaching the last line in the invisible part of the screen, and it would therefore be necessary that a micro-line "in the edge" has a C0 that reaches R1 (when C9 = R9).

Setting R1=1 when R0=1 will activate the BORDER from C0=1.
Unless you can modify R0 and R1 at the same time, not much more will be displayed....

It is therefore necessary that C0 can reach R1 on the visible line, which must also be the last of the frame. **This is good because RVLL works on this principle!**

CRTC 2 doesn't have any problem for the counting of C9 and C4 when R0=0. As such, it has the same potential for "hidden" ruptures as a CRTC 1, i.e. 14 ruptures of 1 µsec, which should have made it possible to solve the problem of centering encountered on CRTC 0, whose potential for hidden rupture is 7.
But, unfortunately, this is only possible if you can activate the last line state on a first line

R0-1	R0-2	R9		R2										OUT R0-2					OUT R0-1		New C9
63		0	C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	**0**
58	4	1	C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	**0**	1	2	3	4	**1**
57	2	2	C0:	…	47	48	49	50	51	52	53	54	55	56	57	**0**	1	2	0	1	2	**2**
57	1	3	C0:	…	47	48	49	50	51	52	53	54	55	56	57	**0**	1	0	1	0	1	**3**
55	1	4	C0:	…	47	48	49	50	51	52	53	54	55	**0**	1	0	1	0	1	0	1	**4**
58	0	5	C0:	…	47	48	49	50	51	52	53	54	55	56	57	58	**0**	0	0	0	0	**5**
57	0	6	C0:	…	47	48	49	50	51	52	53	54	55	56	57	**0**	0	0	0	0	0	**6**
56	0	7	C0:	…	47	48	49	50	51	52	53	54	55	56	**0**	0	0	0	0	0	0	**7**
			C3:					0	1	2	3	4	5	6								
														R3								

Like CRTC 0, there is a notion of "**Last Line**" to schedule an automatic reset to 0 of C9 and C4. However, activating this state on a first line is possible only if this same state is false on the last position C0 of the **HSYNC**.

In other words, two first lines C9=C4=0 are impossible without modifying R9 and/or R4 during the HSYNC to allow the "**Last Line**" state to be considered, otherwise C4 overflows dirty.

It is impossible to modify R9 during HSYNC to allow this state to be considered and to allow this reset of C9 to 0 (as it can be done in a line-to-line rupture) because if you want to modify R9 during HSYNC, there is no magic trick to modify R0 in parallel to create hidden ruptures.

However, it is very useful to remember here that R9 is a 5-bit register of which only 3 bits participate to the video pointer. When line C9=8 is displayed, line C9=0 is displayed.

It is therefore possible to manage the contiguity constraints of the C9 using this principle, as can be seen in the following table:

R0-1	R0-2	R9
55	0	8
54	0	9
53	0	10
52	0	11
51	0	12
50	0	13
49	0	14
48	0	15

		R2							OUT R0-2								OUT R0-1		New C9
C0:	…	47	48	49	50	51	52	53	54	55	**0**	0	0	0	0	0	0	0	**8 (0)**
C0:	…	47	48	49	50	51	52	53	54	0	0	0	0	0	0	0	0	0	**9 (1)**
C0:	…	47	48	49	50	51	52	53	0	0	0	0	0	0	0	0	0	0	**10 (2)**
C0:	…	47	48	49	50	51	52	0	0	0	0	0	0	0	0	0	0	0	**11 (3)**
C0:	…	47	48	49	50	51	0	0	0	0	0	0	0	0	0	0	0	0	**12 (4)**
C0:	…	47	48	49	50	0	0	0	0	0	0	0	0	0	0	0	0	0	**13 (5)**
C0:	…	47	48	49	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**14 (6)**
C0:	…	47	48	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**15 (7)**
C3:		0	1	2	3	4	5	6											
		R3																	

The demonstration provided with SHAKER uses a frame without identical contiguous C9 using all 8 blocks. However, the first and last lines use the "C9 and 7" principle in order to avoid this contiguity (by creating a line 0 via C9=8 and a line 1 via C9=9).

It should be noted, however, that to reach C9=6 and C9=7 (via C9=14 and 15), it is necessary for R0 to reach 49 and 48 respectively. This is not an ideal situation if the visible lines need to be centered.

The earlier C0=R2 occurs, the more "hidden ruptures" appear on the left side of the screen.

If you're curious you can watch the intro of "**AMAZING DEMO 2021**". A simple method to bypass the constraint of contiguous rows is to alternate 1 row out of 2 the table management of C9 to reach. Thus, a line 1 is then necessarily followed by a line C9 + 8 and the problem is solved.

Notes : Since CRTC 0 cannot create hidden ruptures of 1 µsec, this means that the first data to appear on the left is the 3rd byte of the line, followed by a 1/2 µsec of BORDER. On a CRTC 2, it would be the 1st byte of the line (when R0=0) followed by a 1/2 µsec of BORDER. Finally, on CRTC's 1, 3 and 4 in the same R2 context, it would be 2 bytes which would appear on the left.

13.5 CRTC 3, 4

R0 accepts all values without causing any problems for other counters.

If R0 is 0, then C9 and R4 continue to be managed normally.

The offset can be modified according to the timings indicated in the diagrams of the following pages, Chapter 13.8.

The techniques of R.V.L.L. are inapplicable on these CRTC's.

The main "subtlety" of these CRTC's is the change from C9 to 0 if R9 is programmed with a value lower than the current C9 (which allows a "certain" compatibility with CRTC 0).

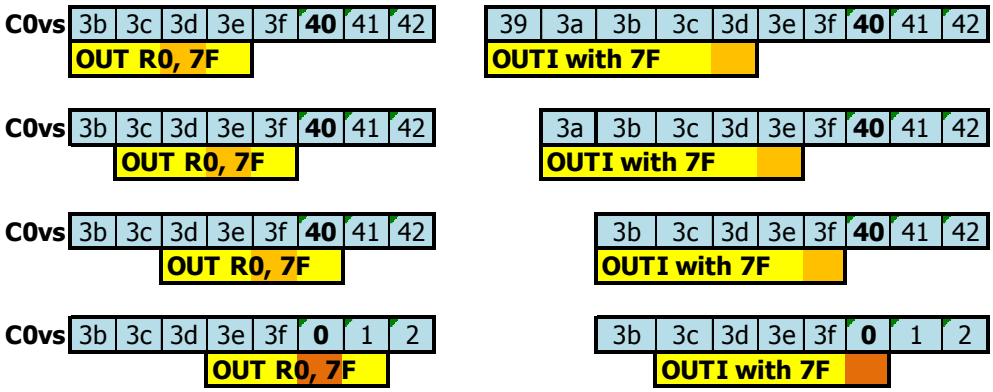
It is possible to carry out an R.V.I. with the constraint of a loop from C9 to 0 so that the offset is considered. (See Chapter 13.3.1).

Be careful, however, not to forget that the I/O takes place with 1 µs delay, which implies placing the OUT 1 µsec before those that would be done on a CRTC 1.

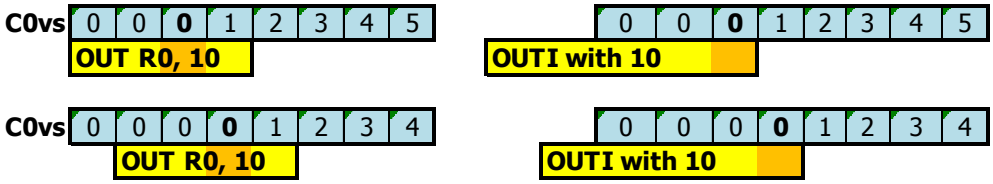
13.6 R0 UPDATE

13.6.1 CRTC 0, 2 : CHRONOGRAM

Previous R0=#3f Update of R0 not considered (too late) Update of R0 ok (just in time)

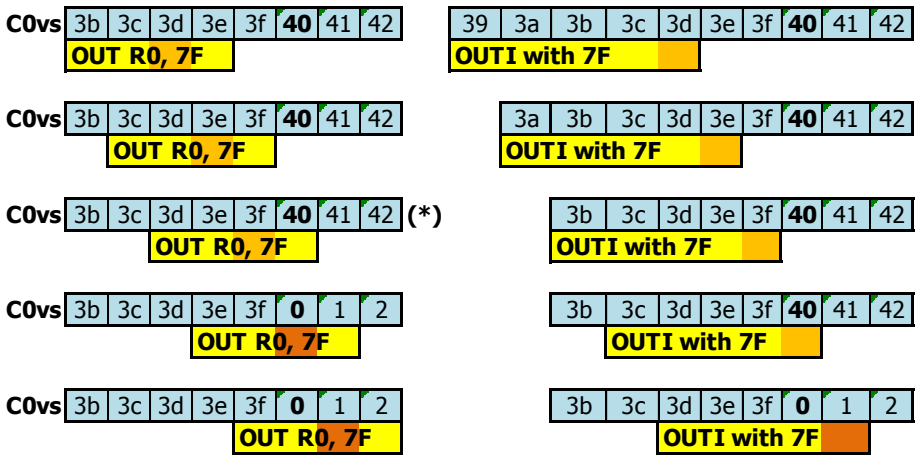


Previous R0=0



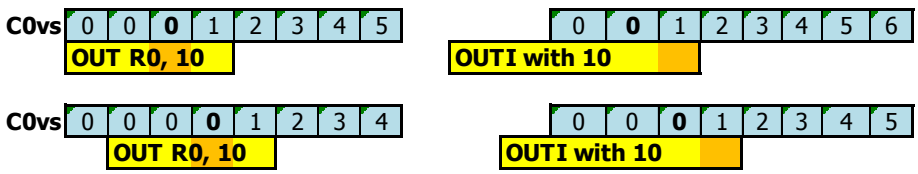
13.6.2 CRTC 1 : CHRONOGRAM

Previous R0=#3f Update of R0 not considered (too late) Update of R0 ok (just in time)



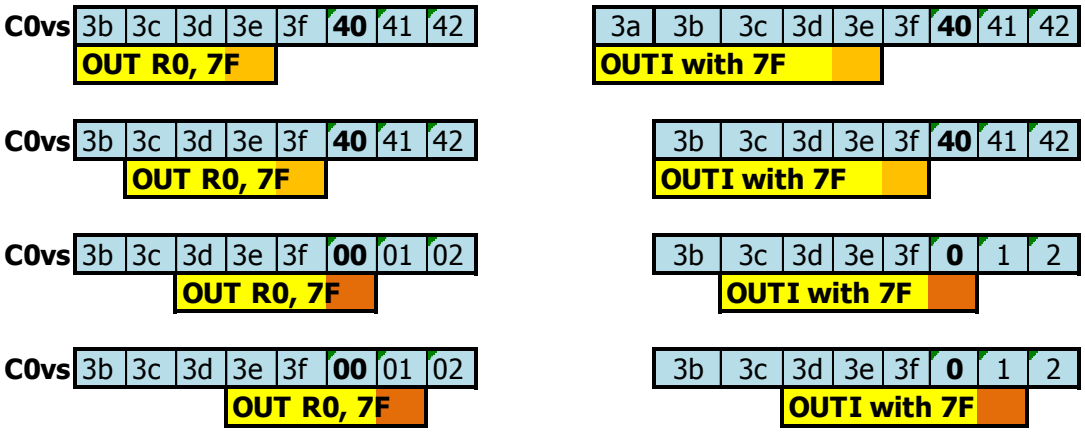
(*) RFD activated on CRTC 1
if R4 and/or R9 modified until C0=7F (new R0) on last line of frame

Previous R0=0

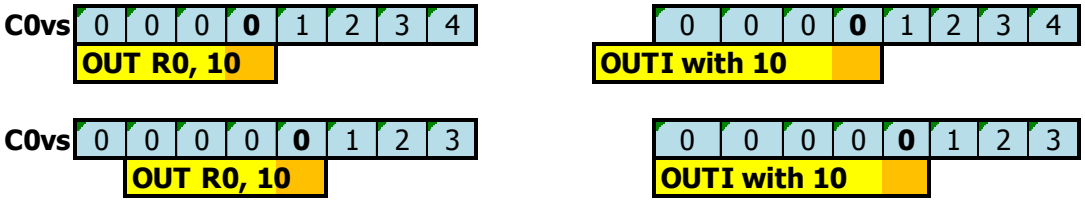


13.6.3 CRTC 3, 4 : CHRONOGRAM

Previous R0=#3f Update of R0 not considered (too late) Update of R0 ok (just in time)



Previous R0=0



The C0 counter never exceeds the value of R0 when R0 is updated according to the timing described in the previous diagrams.

If R0=0 and C0 is 0, it will not overflow, and C0 will always be 0.

In general, the value programmed in R0 is 63 so that 64 µsec flows before the counter goes back to 0. This loopback allows the C0 counter to return to the value of R2 periodically every 64 µsec.

When R0 is reprogrammed during the line, it is necessary to plan to reprogram R2 so that the HSYNC takes place in the same place every 64 µsec and that there is only one of more than 2 µsec.

In principle, it is rather advisable to generate 1 HSYNC per line, but a CTM monitor will easily support not having 1 per line (blah, blah, heretics, blah, blah..., purists, blah, blah,...).

As previously stated, on CRTC 0, updating R0 with a value of less than 2 prevents the CRTC from carrying out certain specific treatments that have consequences on counting.

The minimum possible time between R13 and R12 (in this order) is 8 µsec.
See Chapter 23.1, page 245.

13.7 SPECIAL CASES

There are certainly other exceptions that still remain to be established in a nice table but here are nevertheless some cases identified, according to the CRTC.

13.7.1 CRTC 1

There is a difference in CRTC 1's consideration of the update of R0 on a specific position of C0 according to Z80A instruction used for this update. The two instructions concerned are OUTI (I/O on the 5th µsec) and OUT(C),reg8 (I/O on the 3rd µsec). This reflects a difference in internal processing time at the level of the Z80A instructions, as well as an internal processing phase shift between this CRTC and CRTCs 0 and 2.

13.7.1.1 R0 UPDATE: OUTI

The comparison of C0 with R0, to determine whether C0 should be incremented or reset to 0, takes place after R0 is updated at the 5th µsecond of the instruction of the OUTI instruction. To consider the updated value of R0 correctly, it must be considered that R0 is actually modified 6 µsecond after the start of the OUTI instruction.
For more details, see chapter 13.3, page 107

13.7.1.2 R0 UPDATE: OUT(C),R8

When C0 reaches R0 on the last line of a frame on CRTC 1, some internal end-of-frame states are updated. If R0 is modified on position C0=R0 of this last line of the frame (C9=R9/C4=R4/R5=0) in order to enlarge it, and the last line condition is canceled (by modifying R9 and/or R4) during this enlargement, this paradox will generate an RFD state, which will be triggered at the end of the line.

In this case, the offset change via R12/R13 is considered while considering the IVM parity state. Just like an RFD triggered via R5, the management of an IVM ON/OFF (Interlace Video Mode) allows to freeze the parity. For more details, see chapter 11.6, page 85.

However, there is an additional subtlety depending on the register(s) modified during the extended line.

If R9 is modified so that C9<>R9 at the end of the last line, then R12/R13 are considered, and 1 frame out of 2, there is a default when updating VMA' with VMA (e.g. line repetition).

If R4 is modified (but C9==R9) so that C4<>R4 is at the end of the last line, then R12/R13 are considered 1 frame out of 2. The second frame, C4 is not reset to 0.

In both situations, setting the parity (IVM ON/OFF) freezes the frame (see RFD chapters).

13.7.2 CRTC 0

When C0=R0, the CRTC will increment C4 if a reset to 0 has not been scheduled.
This reset of C4 is fully effective from C0=2 when the CRTC has assessed that it is on a last frame line (C4=C4 and C9=R9) and that there is no additional management (C9=R5, R8=1 or 3 and interlace line condition is ok, on the first frame where the register is updated on even C9).

Note that if an additional management is in progress, the reset to 0 is programmed and will normally occur at the end of this management.

To fully understand what I am going to describe, we must keep in mind that when the condition C0=R0 occurs after C0=2 **as part of additional management**, then C4 is incremented regardless of the value of R4, and C9 goes to 0 when C0 returns to 0. CRTC 0 saves C5 and uses C9 during additional management.

If R0 is programmed with a value of 1, then the additional management test is not performed, and the increment of C4 remains programmed when C9=R9. As part of the R0 update times, this management can be **partially accomplished** if the update takes place on C0 =1 while R0 = 1 and we want to modify it with a value greater than 1 (for example to pass a line to 64 µsec ...).

If the update of R0 takes place when **C0=1 and C9=R9**, this causes an overflow of C4 without C0 and C9 being reset to 0, **even if C4 is not equal to R4.**
We are in additional line management when:

- C4 is unconditionally incremented.
- C9 can no longer go back to 0, even when C9=R9.

The C0=R0 test (with a value of 1) takes place before the register is updated, causing "additional management to begin". But the R0 register was however updated early enough to be taken into account for the increment of C0 as indicated in the previous Chapter (See Chapter 13.6.1) :

Screen Grid:		56	57	58	59	60	61	62	63	0	1	2	3	4	5	6	7
	C0:	...	56	57	58	59	**0**	1	0	1	0	**1**	2	3	4	5	6	7
R9=7	**C9:**	...	4	4	4	4	5	5	6	6	7	7	7	7	7	7	7	7
R4>=6	**C4:**	...	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7

OUT R0,1 OUT R0,63

In this example, two situations are possible on the line C9=7 when R9=7 and when C0 reaches R0 (which is then 63 thanks to the second OUT).

13.7.2.1 We are not on a "last line of frame" (C4<>R4, C9=R9)

Additional management is automatically disabled on the C0=2 position (C4<>R4) because R5 is 0. The consequence is then just an overflow of C4 from C0 = 2, without C4 going back to 0 then as it would have done in additional management.
Note that this **overflow would not have occurred if C9 had been different from R9.**
In the example, C4 would have remained equal to 6.
C9/C4 counting control remains classic.
If, for example, it was necessary to compensate the two "lines" of 2 µsec created after C0=59 so that 8 complete lines would be displayed, it would be necessary to program R9 with 9 to add 2 new lines of 64 µsec.

13.7.2.2 We are on a "last line of frame" (C4=R4, C9=R9)

The last line status has been enabled with C4=R4, C9=R9, regardless of the value of R4.
This is also true if R4=0.
As in the previous case, C4 is incremented from C0=2.
Additional management is evaluated on the C0=2 position.
In this position, incrementing C4 without C9 returning to 0 **leaves the additional management activated**.

If, for example, it was necessary to compensate the two "lines" of 2 µsec created after C0=59 so that 8 complete lines were displayed, **it would be necessary to program R5 with 10** to add 2 new lines of 64 µsec. If R5 is not reprogrammed and was 0, C9 will increment to display lines 8 to 31, until it reaches R5. When the additional management is complete, then C4 returns to 0.

If we want to avoid that C4 is incremented during an "enlargement" of R0 which was 1, **it is therefore imperative to do it on the position C0 = 0**.

13.8 OFFSET ACCORDING TO C0

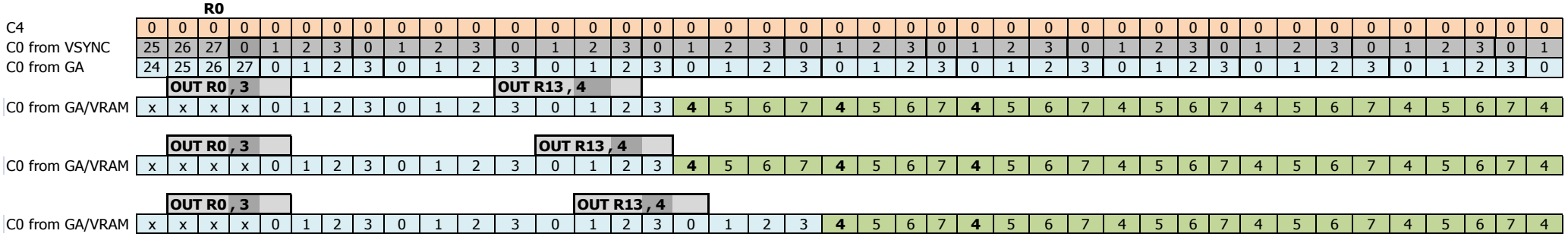
The following diagrams describe how to consider a change in offset (R12 and/or R13) from the current value of **C0vs**. CRTC 2 is not represented here as these tests were initially performed with an R1 value greater than R0.

Offset updates are represented with the colour green..

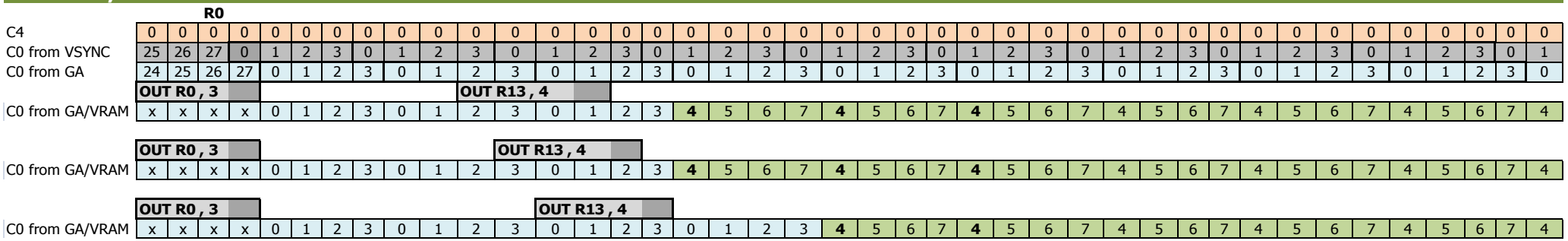
Initial data :
CRTC-R4=0
CRTC-R9=0
CRTC-R1=4
CRTC-R13=0

13.8.1 4 µsec FRAMES (R0=3)

CRTC 0, 1

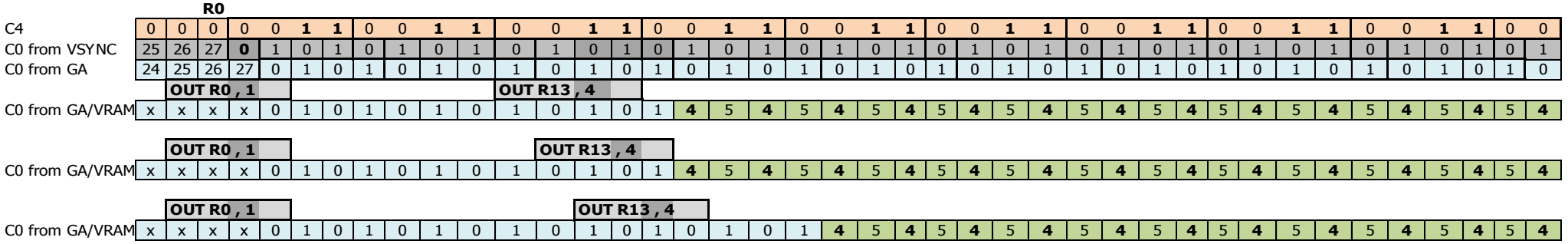


CRTC 3,4



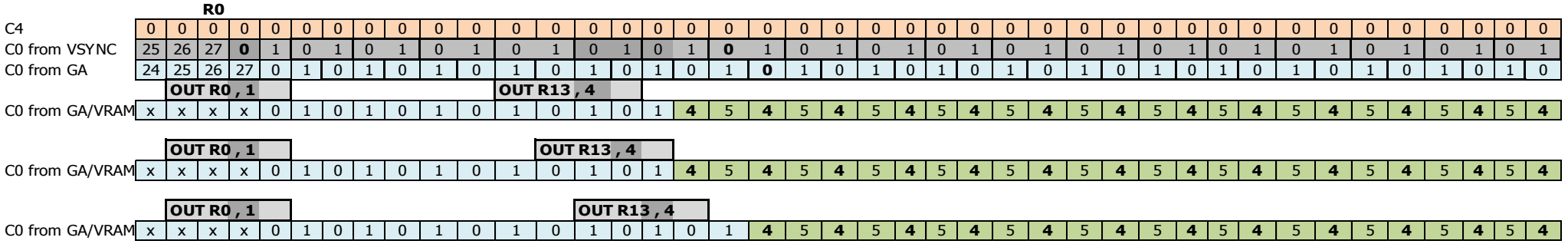
13.8.2 2 µsec FRAMES (R0=1)

CRTC 0

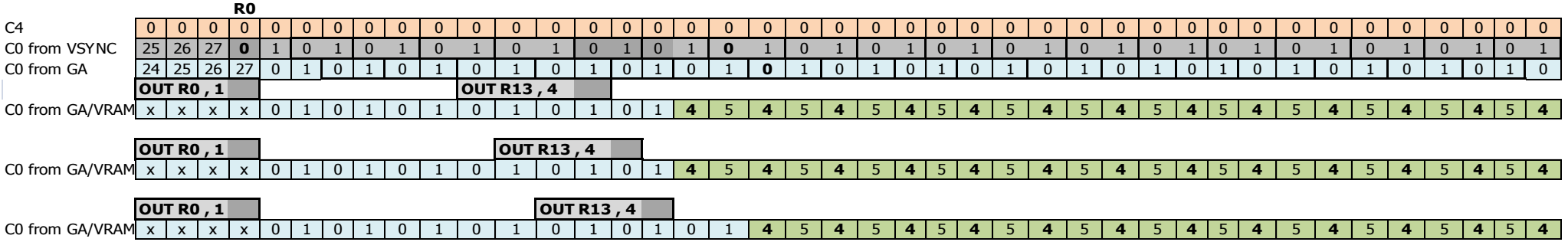


The event C0 = R0 after 2 µsec leaves C4 = 1 for the 2nd period of 2 µsec, which represents a vertical adjustment "not canceled" (because C0 is never equal to 2)

CRTC 1

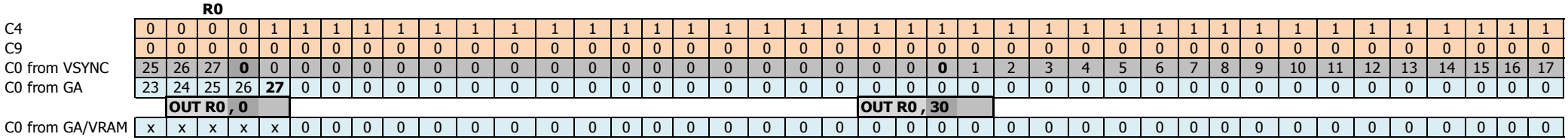


CRTC 3, 4



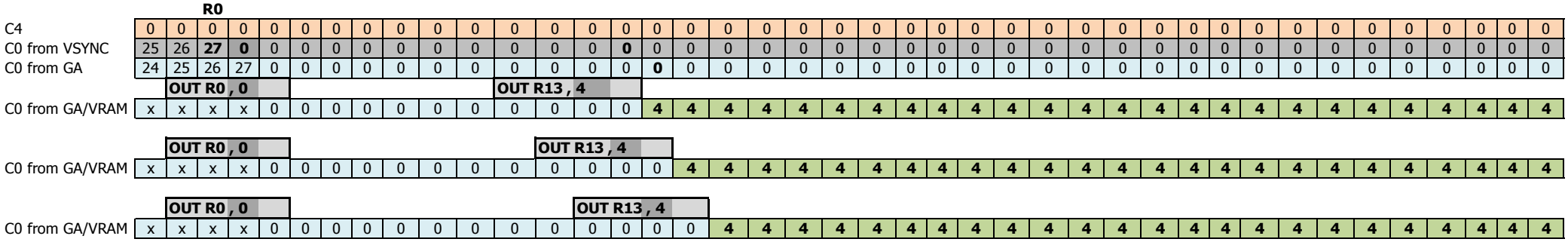
13.8.3 1 µsec FRAMES (R0=0)

CRTC 0

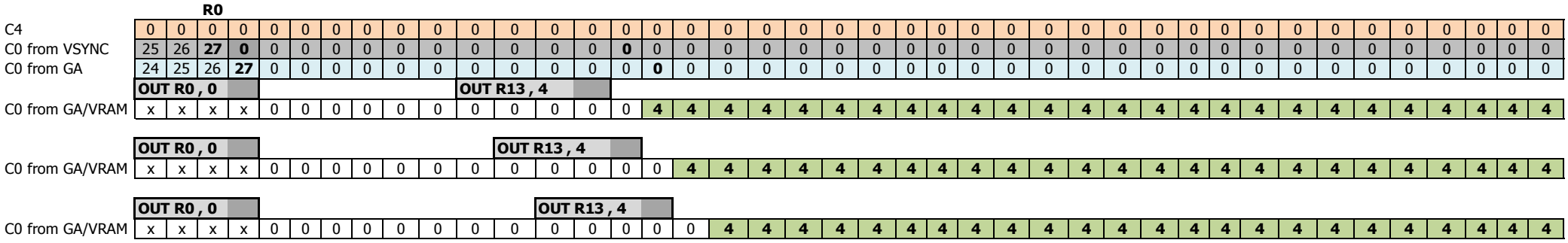


R12 / R13 cannot be considered until C4 and C9 both go back to 0

CRTC 1



CRTC 3,4



14 SYNCHRONISATION : REGISTER R3

14.1 GENERAL

Register 3 may contain 2 different pieces of information, depending on the CRTC 6845 model.

In general, it allows to fix:

- The duration of HSYNC (in **R3l**).
- The duration of VSYNC for some CRTC's (in **R3h**).

In the various diagrams, the HSYNC period is represented using a counter **C3l** which starts at 0 on C0=R2 and which counts until it reaches the value of **R3l**. R3L contains a number of µSeconds. The HSYNC starts as soon as the C3L counter reaches the value of R3L, at the start of the character.

CRTC	7	6	5	4	3	2	1	0
0	Vsync	Vsync	Vsync	Vsync	Hsync	Hsync	Hsync	Hsync
1	x	x	x	x	Hsync	Hsync	Hsync	Hsync
2	x	x	x	x	Hsync	Hsync	Hsync	Hsync
3	Vsync	Vsync	Vsync	Vsync	Hsync	Hsync	Hsync	Hsync
4	Vsync	Vsync	Vsync	Vsync	Hsync	Hsync	Hsync	Hsync

Other CRTC	7	6	5	4	3	2	1	0
MC6845*1	Vsync	Vsync	Vsync	Vsync	Hsync	Hsync	Hsync	Hsync
UM6845E	Vsync	Vsync	Vsync	Vsync	Hsync	Hsync	Hsync	Hsync

Hsync CRTC				
0	0	0	0	**No Hsync**
0	0	0	1	1 nop
0	0	1	0	2 nop
0	0	1	1	3 nop
0	1	0	0	4 nop
0	1	0	1	5 nop
0	1	1	0	6 nop
0	1	1	1	7 nop
1	0	0	0	8 nop
1	0	0	1	9 nop
1	0	1	0	10 nop
1	0	1	1	11 nop
1	1	0	0	12 nop
1	1	0	1	13 nop
1	1	1	0	14 nop
1	1	1	1	15 nop

Hsync CRTC				
0	0	0	0	**16 nop**
0	0	0	1	1 nop
0	0	1	0	2 nop
0	0	1	1	3 nop
0	1	0	0	4 nop
0	1	0	1	5 nop
0	1	1	0	6 nop
0	1	1	1	7 nop
1	0	0	0	8 nop
1	0	0	1	9 nop
1	0	1	0	10 nop
1	0	1	1	11 nop
1	1	0	0	12 nop
1	1	0	1	13 nop
1	1	1	0	14 nop
1	1	1	1	15 nop

Vsync CRTC				
0	0	0	0	**16 lines**
0	0	0	1	1 line
0	0	1	0	2 lines
0	0	1	1	3 lines
0	1	0	0	4 lines
0	1	0	1	5 lines
0	1	1	0	6 lines
0	1	1	1	7 lines
1	0	0	0	8 lines
1	0	0	1	9 lines
1	0	1	0	10 lines
1	0	1	1	11 lines
1	1	0	0	12 lines
1	1	0	1	13 lines
1	1	1	0	14 lines
1	1	1	1	15 lines

14.2 VSYNC LENGTH

First-generation CRTC's generate a 16-line VSYNC signal.

The manufacturers have improved the circuit by adding a function to parameterize the number of lines, using the high-4 bits of R3 (**R3h**).

In order to ensure compatibility of programs created for the first generation of circuits, the value 0 for the CRTC's integrating the new function corresponds to 16 lines of VSYNC.

The high-4 bits of R3h is used to indicate an exact number of lines, except for 0 therefore, which means 16 lines.

The BASIC ROM of CPC initializes R3h with 8 (R3=1000xxxx).

Thus, CRTC's 1 and 2, which do not manage these bits, generate 16 lines of VSYNC while CRTC's 0, 3 and 4 generate 8. (When R7 is already programmed before C4=R7).

When a VSYNC starts on CRTC 0, 3 and 4, it is possible to modify the number of lines along the way.

Thus, if R3H was worth 9 and R3h is modified with the value 8 on the 8th line of the VSYNC, then the VSYNC stops at the end of the HSYNC of this line.

On the other hand, if R3h is modified with the value 8 on the 9th line, then the VSYNC line counter will count until its definition limit (4 bits) by creating 16 lines, then will generate 8 additional lines again.

Since CRTC's 1 and 2 do not know how to manage a VSYNC with a configurable number of lines, **it is advisable to avoid using this function** on CRTC's 0, 3 and 4. This can lead to incompatibilities for a program that would have the good idea to synchronize on the end of the VSYNC for example, or to test the VSYNC in a period greater than that defined in R3h and less than 16.

This involves systematically reprogramming this register because initialization by the ROM creates a difference which could be avoided.

The game "***3D STARSTRIKE***", published by "Realtime Games Software" in 1985, was developed on CRTC 0. The developers did not modify the programmed value in ROM for the length of the VSYNC. This causes a bug during the game on CRTC 1 and 2, because the firing cursor is not in phase with the display and flashes, while this is not the case on CRTC 0, 3 and 4.

Note :
When the VSYNC signal is generated, the Gate Array will take care of the management of the VSYNC (26 black lines including 4 monitor signal lines). The programmed length in R3h has no real use for synchronization. Thus program a VSYNC CRTC of 1 line (R3h=1) on CRTC 0 has no impact on image synchronization. This is not the case, however, for CRTC 3 and 4, which need the CRTC VSYNC signal to be active on the 2nd HSYNC encountered since C4==R7 so that the C-VSYNC Monitor is generated by the ASIC or the Pre-Asic (see chapter 16.2).

14.3 HSYNC GATE ARRAY VERSUS CRTC

When the GATE ARRAY receives a HSYNC signal from the CRTC, it displays black color for 2 µsec (approximately because it is possible to reduce this period) and then generates a C-HSYNC signal for the monitor whose maximum duration is 4 µsec (during this period he stops displaying colors). If the value programmed in R3 is greater than 6, the GATE ARRAY will display black color again for the remaining period.

The GATE ARRAY partly uses the VSYNC and HSYNC signals produced by the CRTC. Even if the role of the HSYNC signal is more important than that of the VSYNC signal, the GATE ARRAY remains (almost) master of the beginning and the end of the C-HSYNC signal. The VSYNC signal is only used to start the VSYNC management of the GATE ARRAY (CRTC 0, 1 and 2), which will then deal with the start and duration of the C-VSYNC signal using the HSYNC signal.

If the HSYNC CRTC time is greater than 6, it does not affect the horizontal synchronization of the monitor. With a value less than 6, the C-HSYNC signal time for the monitor drops below 4 µsec. In this situation, the C-HSYNC signal is no longer stopped at the end of a 4 µsec count (64 x 0.0625µsec) by the GATE ARRAY, but by receiving the HSYNC end signal of the CRTC. The signal is therefore not reduced by 1, 2 or 3 µsec exactly.

However, it can be useful to have a "CRTC" HSYNC time greater than 6 because the CRTC does not consider the update of all its registers in the same way during this period and therefore makes it possible to inhibit processing. In addition, the C-VSYNC signal is produced by the GATE ARRAY only when the CRTC signals an end of HSYNC.

14.4 HSYNC AND FRAME POSITION

It is possible to use the C-HSYNC produced by the GATE ARRAY to shift the lines more finely. By reducing the size of 1 µsec, the line is "shifted" with a offset of 0.5 µsec on the right (1/2 CRTC character). The monitor deflectors and the GA signals are in principle fairly precise (see chapter 16.6).

The start of the HSYNC CRTC signal reported to the GATE ARRAY will determine the start of the C-HSYNC signal. The C-HSYNC signal begins one or two pixels-M2 before the start of the 3rd µsec (according to the CRTC) and stop under 2 conditions. Either the CRTC signals an end of HSYNC, or the GATE ARRAY has totaled very exactly 4 µsec.

When R3l>=6, it is the accounting by the GA of the 4 µs which conditions the end of C-HSYNC signal because the end of HSYNC-CRTC occurs one or two pixels-M2 after these 4 µsec.

On the other hand, if R3l is less than 6, then it is the end of HSYNC produced by the CRTC which interrupts C-HSYNC. The duration of the C-HSYNC then consider the deadline where the CRTC signals the end of HSYNC to the GATE ARRAY. And this situation is less precise according to the CRTC.

Some games and demos used R3l values 5 and 6 to shift the screen by 8 Pixels (mode 2) to manage a horizontal scroll. This is particularly the case on the game "***Skatewars***", edited by "Ubi Soft" in 1989. Jon Menzies uses values &85 and &8E, which goes to values 5 and 6 for the size of the C-HSYNC generated by the GATE ARRAY. In some parts of "***Yao Demo***", created in 1990, Fabien Fessard also uses values 5 and 6 to slow down his scrollings.

When **R3l drops**, excluding R3.JIT, the image is shifted to the right of **half a unit** (0.5 µsec if R3l drops by 1) whatever the value of R2. However, **values 5 and 6 should be avoided** to obtain an exact and independent phase shift from the CRTC.

On CRTC 1, for example, if R3l=5, then the duration of C-HSYNC is approximately 3,1250 µsec, while with R3l=6, the duration of C-HSYNC is exactly 4 µsec. 4-3.1250 = 0.875/2 = 0.4375 or 7 Pixels (graphi mode 2) instead of the expected 8 pixels. If R3L=4, then C-HSYNC lasts 2,1250 µsec approximately. The difference between 3,1250 and 2,1250 is exactly 1 µsec and the screen is in principle well offbeat of 8 pixels.

It is therefore preferable to use the 4/5 values because these two values allows to generate C-HSYNC signals, the difference of which is very exactly 1 µsec, whatever the type and tolerance of the CRTC. A CTM monitor must be adjusted if it loses sync with R3l=4.

Continuously (and CPU-intensive) switching of these values between lines **can divide by 2 the difference between 2 positions** (i.e. a division by 4).

It is also possible to cause oscillation around a given position by varying the C-SYNC positioning by 16 (or more) mode 2 pixels between each line. This is done by modifying R2 between each line. This technique, imagined by Rhino (Batman group) allows to create pretty frames at the origin of acronyms of graphic modes.

R3.JIT technique can shift the end of the HSYNC by **0.25 µsec**, therefore it's possible to obtain more combinations.

The following table describes the durations of the **C-HSYNC's** expressed in µsec according to the values of R3l (in **R3.JIT** and **R3.NJIT** mode) and according to the CRTC's. I indicated a range of 2 values that I could see, while waiting to have more precise measures.

	R3 NJIT 2	R3JIT 2	R3 NJIT 3	R3JIT 3	R3 NJIT 4	R3JIT 4	R3 NJIT 5	R3JIT 5	R3 NJIT 6
CRTC 0	0,0625 0,1250	0,3125 0,3750	1,0525 1,1250	1,3125 1,3750	2,0625 2,1250	2,3125 2,3750	3,0625 3,1250	3,3125 3,3750	4,0000 4,0000
CRTC 1	0,1250 0,1875	0,3125 0,3750	1,1250 1,1875	1,3125 1,3750	2,1250 2,1875	2,3125 2,3750	3,1250 3,1875	3,3125 3,3750	4,0000 4,0000
CRTC 2	0,0625 0,1250	0,3125 0,3750	1,1250 1,1875	1,3125 1,3750	2,0625 2,1250	2,3125 2,3750	3,0625 3,1250	3,3125 3,3750	4,0000 4,0000

When R2 increases by 1, the screen is shifted to the left by 16 pixels M2.
When R2 decreases by 1, the screen is shifted to the right by 16 pixels M2.
When R3l increases by 1 (if <6), the screen is shifted to the left by 8 pixels M2.
When R3l decreases by 1 (if >2), the screen is shifted to the right by 8 pixels M2.
A precise position of C-HSYNC can therefore be determined by updating both R2 and R3l.

If 2 positions are switched from one line to the other, the CTM monitor tries to synchronize the line between these 2 positions.

The "**module D test 6**" of SHAKER 2.4 demonstrates that it is now possible on CPC to perform a **smooth hardware scroll at pixel mode 1** (2 M2 pixels) using these different principles.

Note :

A short C-HSYNC may also have a discolouration effect on some non-CTM displays. This is for example the case if the CPC is connected to an ATARI SC1425 monitor (which was available with the ATARI 520STE).

If the HSYNC is cut to 6 µsec (R3=6), the GATE ARRAY sends black for approximately 2 µsec followed by the HSYNC signal of 4 µsec. If the colour of the BORDER has been defined with something other than black (which a HSYNC longer than 6 would generate) this has a discolouration impact on the line because each colour component is calibrated according to the colour present over the following 3 µsec (position 7, 8 and 9).

If the BORDER is not black and R3<9, the colour will be affected on the line. On this type of monitor, it is therefore possible to obtain more colours.

This can also cause unexpected results (especially on LCD screens).

14.5 UPDATING R3 DURING HSYNC

It is possible to change the value of R3l when C3l counts, which can affect the length of the HSYNC.

If R3l is changed with a value less than C3l, then C3l is overflowing, **except for CRTC 1 with a value of 0, which cancels the current HSYNC**.

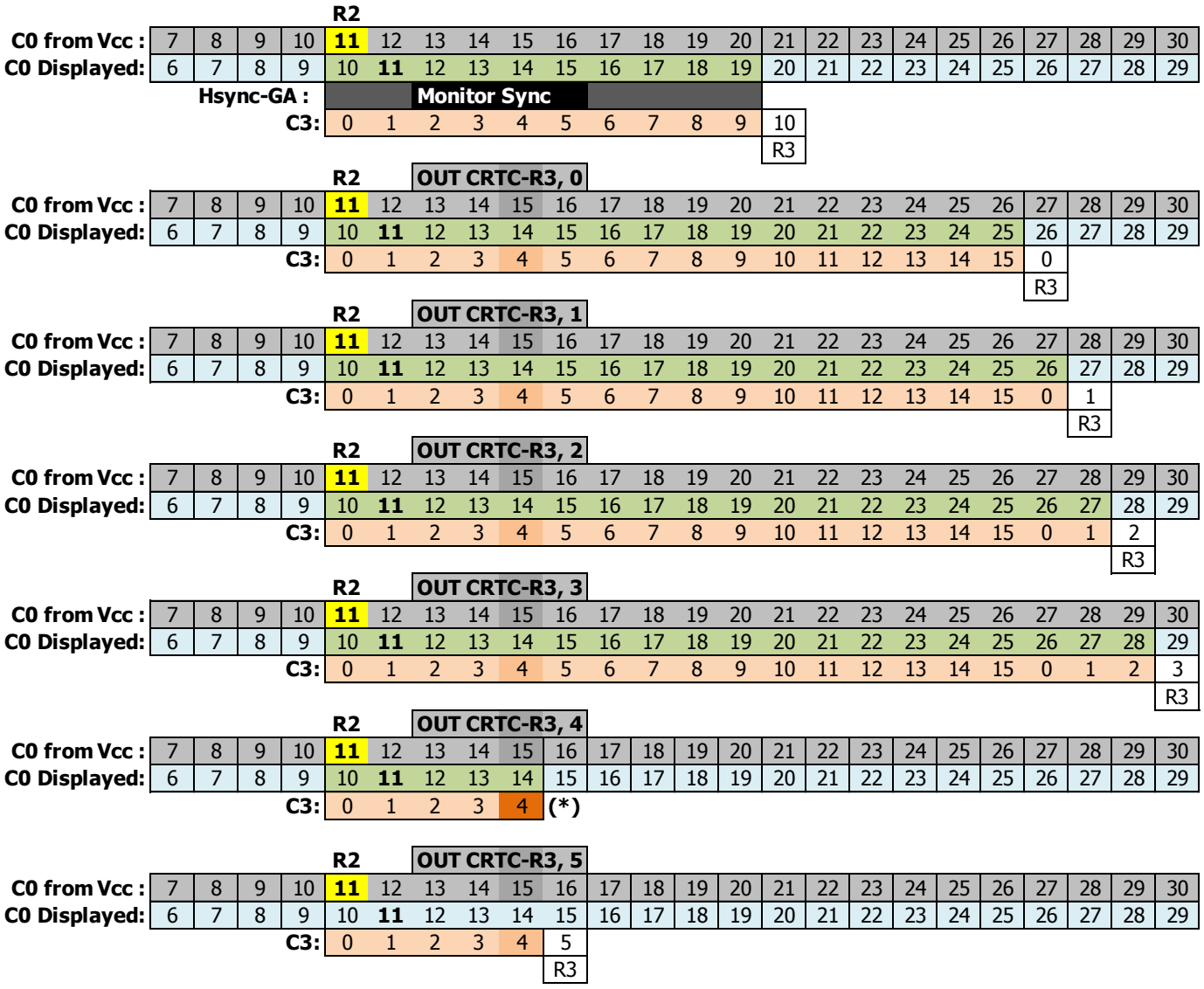
The **R3.JIT** technique consists in modifying R3l with the value of C3l at the time or C0 is on the position corresponding to C3 to interrupt the HSYNC wildly.

Unlike the **R2.JIT** technique, which only affects colorization, **R3.JIT** also has an impact on the duration of the C-HSYNC signal if it is operated during the 4 µsec when produced by the GATE ARRAY.

The update of R3 during this count is considered according to the situations described below.

14.5.1 CRTC 0, 2

CRTC-R2=11 / CRTC-R3=10 (HBL Size = 10 chars)



	Z80A (OUT (C),r8) instruction
	Register update
	HSYNC Zone
	Characters displayed

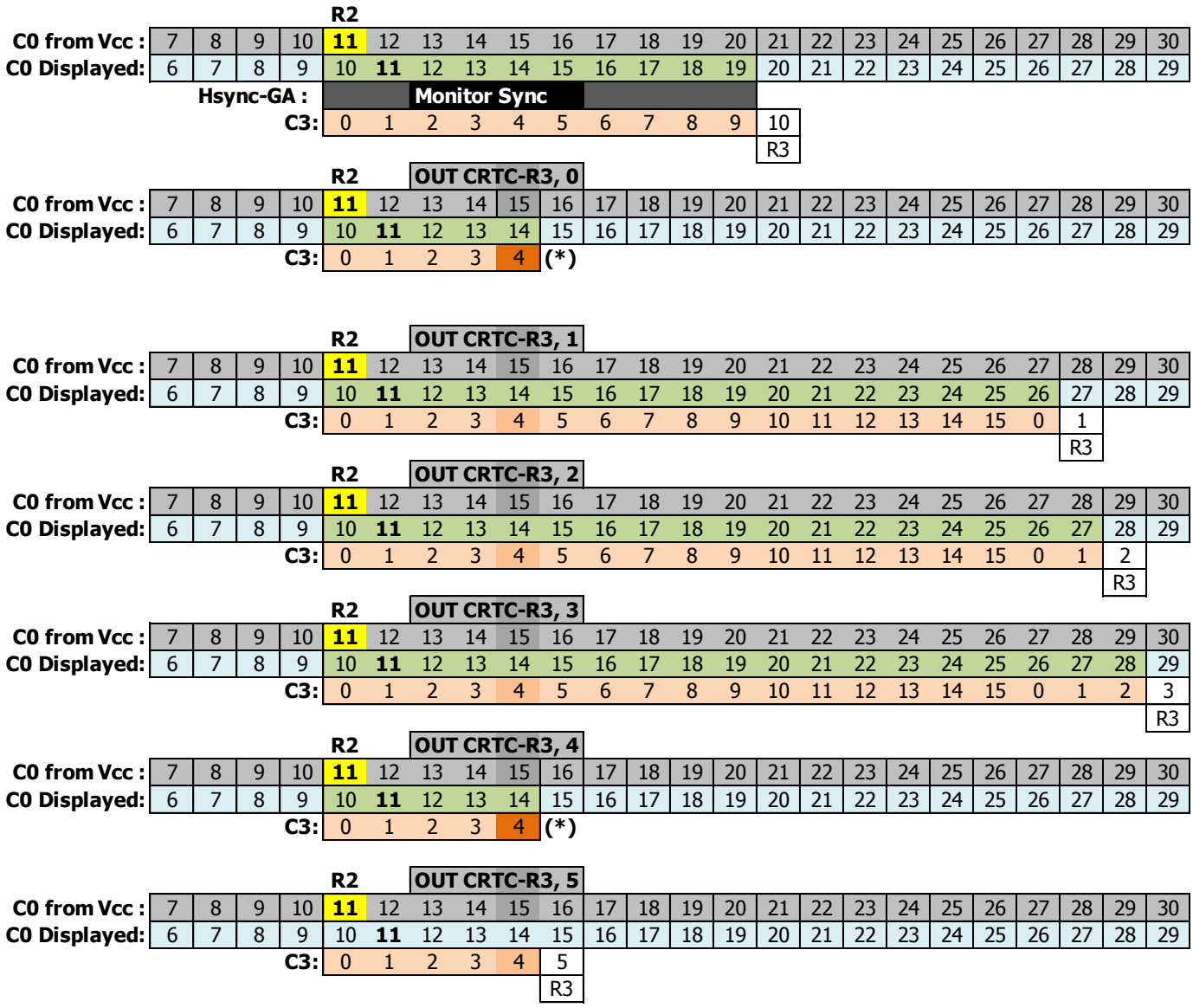
(*) R3.JIT : The precise update of R3l interrupts the HSYNC after 0.25 µsec after its normal end if R3l had been programmed in advance. See next Chapters.

14.5.2 CRTC 1

CRTC-R2=11 / CRTC-R3=10 (HBL Size = 10 chars)

During an R3 update during HSYNC, CRTC 1 behaves like CRTC's 0 and 2, except when R3 is updated to 0. Indeed, the processing of R3=0 (no HSYNC) continues to be managed during HSYNC for this CRTC.

On CRTC's 0 and 2 in this condition, HSYNC continues and 0 is treated as a value to reach.

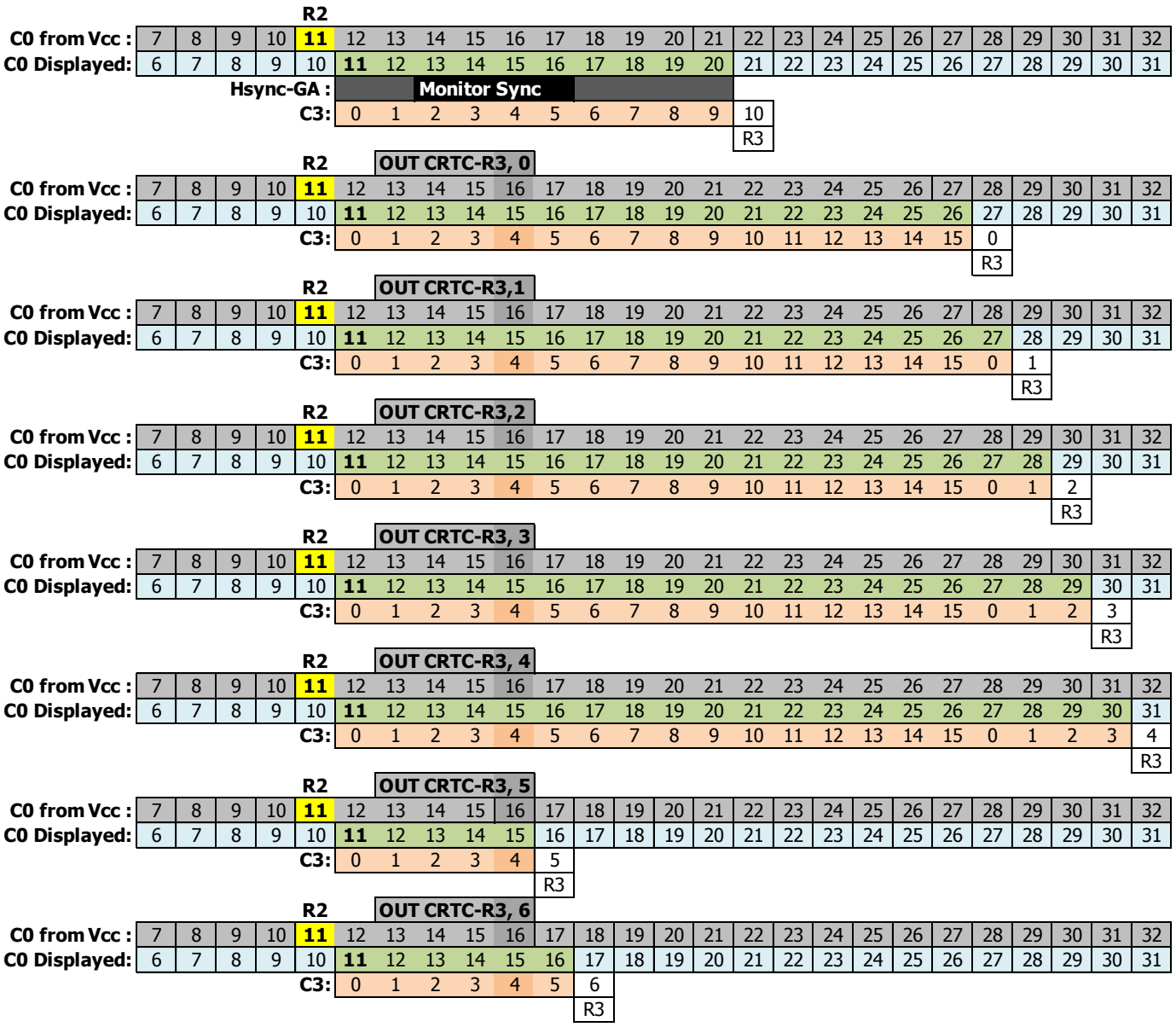


	Z80A (OUT (C),r8) instruction
	Register update
	HSYNC Zone
	Characters displayed

(*) R3.JIT. The precise update of R3l interrupts the HSYNC after 0.25 µsec after its normal end if R3l had been programmed in advance. See next Chapters.

14.5.3 CRTC 3, 4

CRTC-R2=11 / CRTC-R3=10 (HBL Size = 10 chars)



	Z80A (OUT (C),r8) instruction
	Register update
	HSYNC Zone
	Characters displayed

14.5.4 ZOOM ON R3.JIT

If R3l is modified via an **OUT(C),r8** with the value of C3l when C0 is at position which corresponds to C3l while R3l was greater than this value, then the **HSYNC** stops on CRTC's 0, 1 and 2. This technique is called **R3.JIT**

The use of OUTI does not allow this technique to be used on these CRTC's.

Whether with OUT(C),r8 or OUTI, **this technique does not work on CRTC's 3 and 4**, which synchronize the HSYNC with the display.

When a HSYNC starts, it does so at different positions depending on the CRTC's.

See Chapter 9.3.4.2, page 52 for more details on this subject.

The update of R3l can be carried out on all the positions of C0 during which the HSYNC takes place.

On the CRTC's 0 and 1, the first µsecond of HSYNC is special, because interrupting it with the value 0 interrupts the HSYNC prematurely instead of delaying its end by 0.25 µsec.

Note 1: Interrupting R3 with 0 using an OUTI prevents the HSYNC from starting.

The value R3=0 for a CRTC 2 means that the HSYNC will be 16 µsec.

On **R3.JIT** with R3=0, the end of the HSYNC takes place after the last **Pixel-M2** of the current µsec.

On a CRTC 0, the HSYNC starts on the 5th pixel-M2 and lasts **4 pixel-M2's**.

On a CRTC 1, the HSYNC starts on the 6th pixel-M2 and lasts **3 pixel-M2's**.

Note 2: On CRTC 1, in R3.NJIT (or with OUTI), the HSYNC ends 1 Pixel-M2 later than for CRTC 0 and 2.

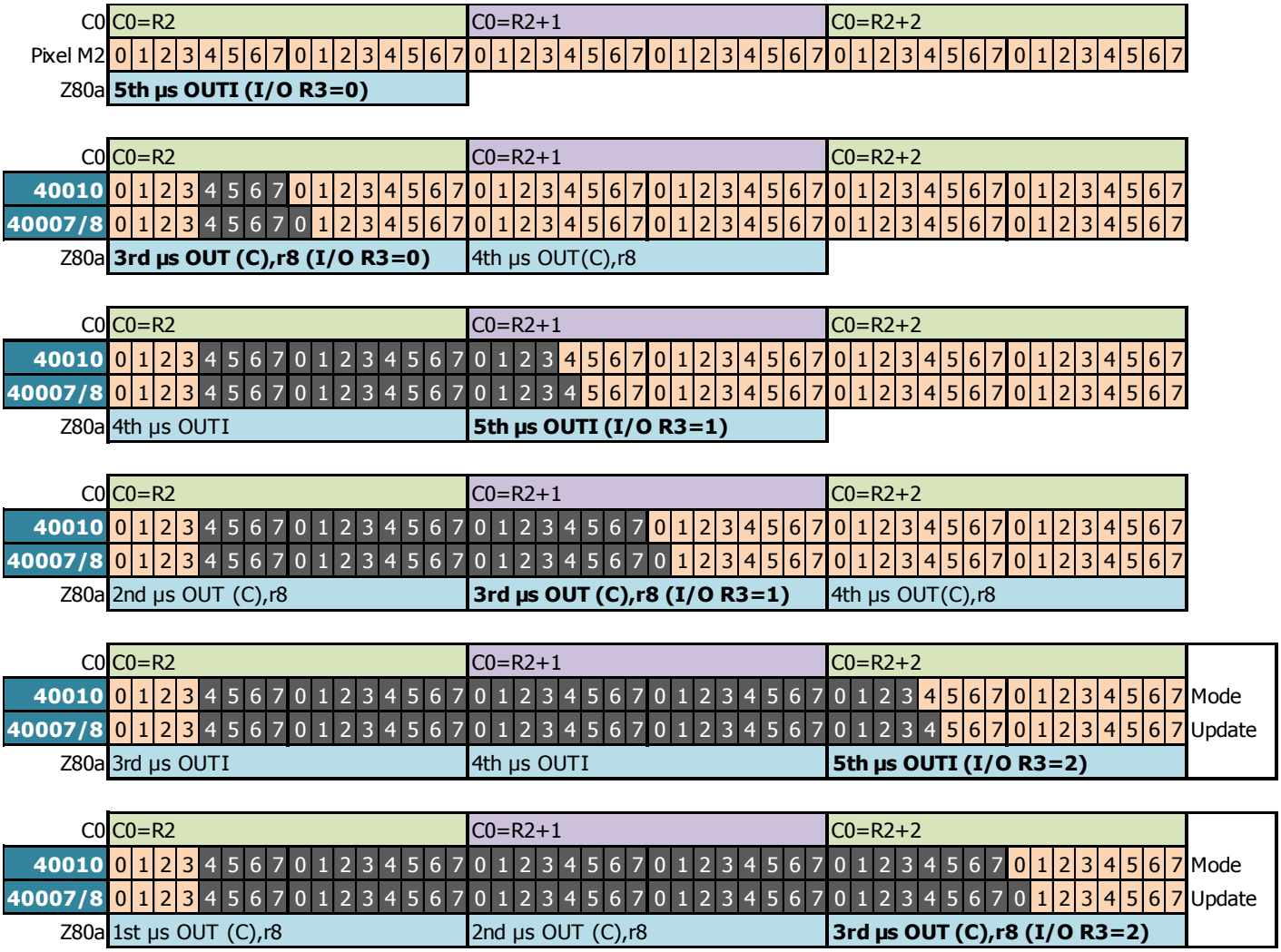
The following diagrams describe the positioning of the HSYNC in **R3.JIT**.

The HSYNC is 1 pixel-M2 longer on the GA 40007 and 40008 compared to the GA 40010.

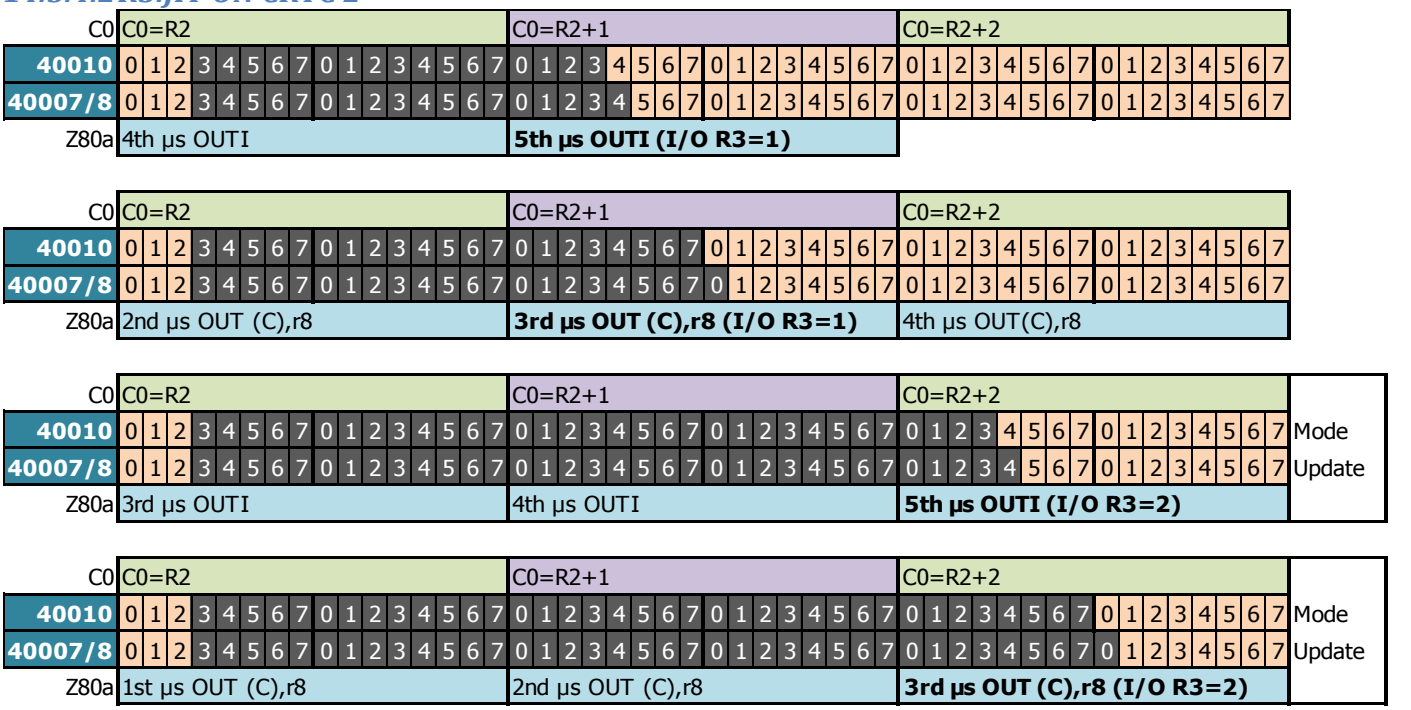
The possibility of changing graphics mode is indicated on the diagrams.

It must also be considered that going from MODE 2 to another MODE "adds" 1 pixel M2 (9 pixels are generated from a byte) and going from MODE 0,1 or 3 to MODE 2 "subtracts" 1 pixel M2 (7 pixels are generated from one byte).

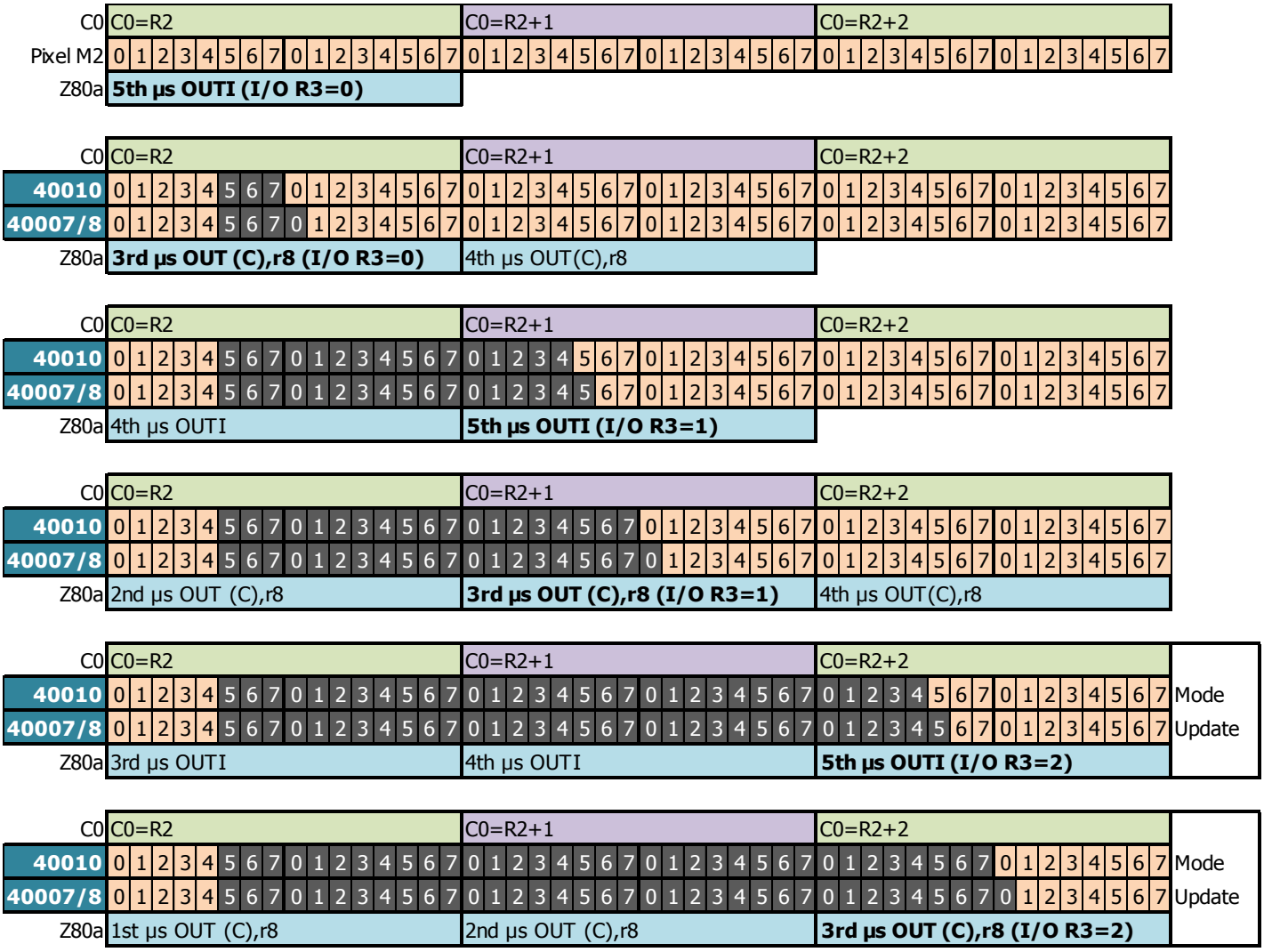
14.5.4.1 R3.JIT ON CRTC 0



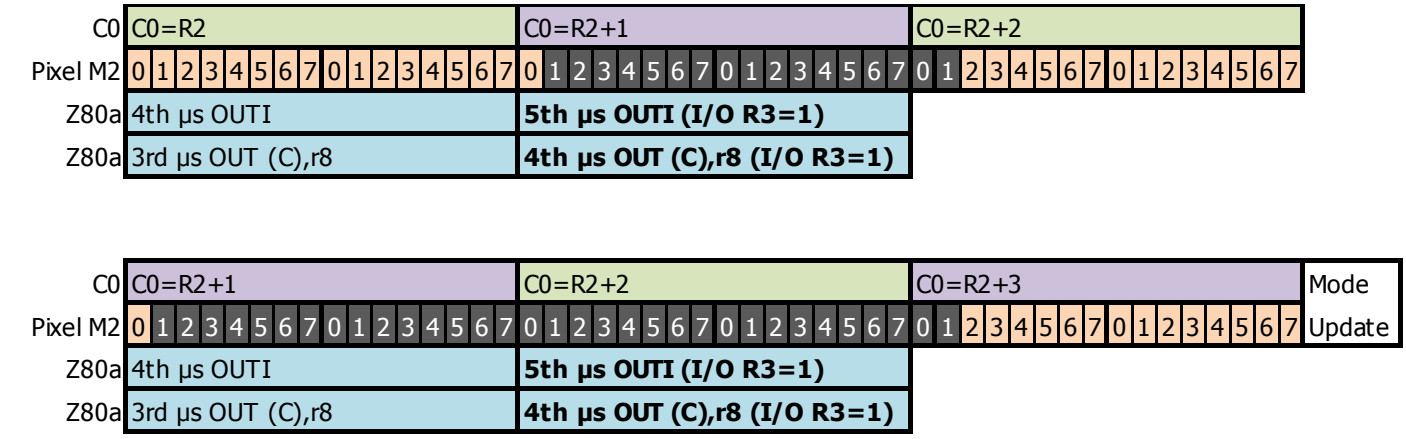
14.5.4.2 R3.JIT ON CRTC 2



14.5.4.3 R3.JIT ON CRTC 1



14.5.4.4 NO R3.JIT ON CRTC 4



14.6 ABSENCE OF HSYNC

When R3=0, CRTC's 0 and 1 do not produce HSYNC (**and therefore no interruption**).

On CRTC's 2, 3 and 4, it is impossible not to generate HSYNC if the C0=R2 condition is satisfied. A value of 0 in R3 will generate a HSYNC of 16 µsec, unless it is interrupted by modifying R3 during HSYNC.

14.7 HSYNC START-UP

14.7.1 CRTC 0, 1, 2

When C0=R2 then a HSYNC is generated over a length of R3 characters.
R2 update occurs during the 3rd µsec of the OUT(C),reg8 instruction.

When the **update of R2 takes place before C0 reaches R2**, the HSYNC is processed by the GATE ARRAY during the display of the previous character, the character C0=R2 having not yet been displayed by the GATE ARRAY.

In this situation, the HSYNC black zone does not start exactly on a character boundary, and not at the same position according to the CRTC's.

- CRTC 0 : The non-display associated to the HSYNC starts from the start of the 5th mode 2 pixel after the start of the displayed CRTC R2-1 character. It was however observed the display of half of the 4th Pixel-M2.
- CRTC 1 : The non-display associated to the HSYNC starts from the start of the 6th mode 2 pixel after the start of the displayed CRTC R2-1 character.
- CRTC 2 : The non-display associated to the HSYNC starts from the start of the 4th mode 2 pixel after the start of the displayed CRTC R2-1 character. It was however observed the display of half of the 3rd Pixel-M2.

If the update of R2 occurs **while C0=R2** (during the 3rd µsec of an OUT(C),r8) then the CRTC sends the HSYNC signal later to the GATE ARRAY. This reflects a delay between the execution of the instruction in Z80A and the consideration by the CRTC.

However, if this update is performed via the OUTI instruction (during the 5th µsec of the instruction) then the HSYNC signal is sent faster to the GATE ARRAY by 0.25 µsec, which then behaves as if R2 had been programmed before that C0=R2.

A **R2.JIT** ("Just In Time") update performed with an OUT(C),r8 causes the HSYNC black zone to appear later than in other situations (R2 programmed before C0=R2 or i/o OUTI on C0=R2)

In this situation, the stop position of the display depends on the type of CRTC:

- CRTC 0, 1: The non-display associated to the HSYNC starts from the start of the 9th mode 2 pixel after the start of the displayed CRTC R2-1 character. It was however observed the display of half of the 8th Pixel-M2.
- CRTC 2: The non-display associated to the HSYNC starts from the start of the 8th mode 2 pixel after the start of the displayed CRTC R2-1 character. It was however observed the display of half of the 7th Pixel-M2.

The HSYNC duration remains counted down with the value programmed in R3 (unless R3 is modified during the HSYNC). The delayed display of the HSYNC black zone does not change the synchronization of the monitor with respect to an anticipated R2 programming.

This special case of update (**R2.JIT**) allows delaying of the start of the displayed HSYNC by 4 mode 2 pixels (i.e. 0.25 µsec) (1 T-State) on CRTC's 0 and 2, and 3 mode 2 pixels (0 .1875 µsec) on CRTC 1.

This technique allows delaying of the end of display by the GATE ARRAY of the black zone from 0.1875 to 0.25 µsec. This is interesting in the event of a change of graphic mode during a line to limit the area of absence of display. It should nevertheless be considered that mode 2 is displayed 1 pixel (0.0625 µsec) earlier by the GATE ARRAY than for the other graphic modes.

On the diagrams on the following pages, R3 is fixed at 2.
R2 before modification is greater than 10.

14.7.2 CRTC 3, 4

When C0vs=R2 then a HSYNC will be generated.

This update is considered in order to correspond to the display of the character corresponding to C0 by the GATE ARRAY, but the test is nevertheless carried out with respect to C0vs.

An **R2.JIT** ("Just In Time") update carried out with an OUT(C),r8 does not cause the HSYNC black zone to appear later than in the other situations since the HSYNC is deferred.

The CRTC 4 ASIC simulates the GATE ARRAY, insofar as the display of pixels in mode 2 begins 1 Pixel-M2 before the display of pixels in the other graphics modes.
On this CRTC, the display stop linked to the HSYNC starts from the 19th mode 2 pixel after the start of the displayed CRTC R2-1 character. However, the display of half of this 19th Pixel-M2 pixel was observed.

As a reminder, an I/O via an OUT(C),r8 takes place during the 4th µsec of the instruction on CRTC's 3 and 4. An I/O on the OUTI instruction takes place during the 5th µsecond of the instruction, as for other CRTC's.

The diagrams below illustrate the previous remarks by showing the positioning of the HSYNC according to the moment when the update of R2 takes place and according to the instruction used.

The CRTC 3 diagram will be added in a later version. Currently assumed to be the same as CRTC 4, but with HSYNC starting on the 17th pixel after the start of the displayed CRTC R2-1 character.

14.8 HSYNC AND INTERRUPTIONS

Updating the HSYNC size changes when an interrupt is generated. The GATE ARRAY triggers an interrupt just after the HSYNC end, under certain conditions.

See Chapter 26, page 274 .

14.9 HSYNC SCHEMATICS

The following page describes the display of the black color at the beginning and at the end of an HSYNC produced by the GATE ARRAY.

When a VSYNC ends (at the end of the 26th HSYNC), the black color stops 1 PixelM2 after the end of an HSYNC for CRTC's 0 and 1.

It stops at the same time for CRTC's 2, 3 and 4.

CRTC 0

| Z80A instruction | R2-NJIT / OUT(C),r8 CRTC-R2=10 I/O >> x | | |

15 SYNCHRONIZATION : REGISTER R2

15.1 GENERAL

The CRTC R2 register allows to define the activation condition of the **CRTC HSYNC** signal.
This signal becomes "high" when **C0vs reaches R2**.

During **HSYNC**, in principle, nothing is displayed anymore.

The length in µsec of the **HSYNC-CRTC** is fixed with the 4 less significant bits of the R3 register.
See Chapter 14.1, page 124.

The timing of **HSYNC** is different for different CRTC's and is not based on the beginning of a CRTC character. See Chapter 14.4, page 126.

The display of pixels does not start (and does not stop) on a word boundary, see byte (depending on conditions) for HSYNC. See Chapter 14.7, page 135.

The GATE ARRAY is faster to manage HSYNC than to display characters read by CRTC's 0, 1 and 2. The HSYNC starts earlier and is "visible" on the character preceding the one pointed by R2, which has not yet been displayed.

THE ASIC's (CRTC's 3 and 4) manage a HSYNC consistent with the C0 value displayed, delaying the display of the HSYNC by 1 µsec.

Examples :

- If R2 is 10, on a CPC with CRTC (0, 1 and 2) then HSYNC starts from C0 displayed=9 (R2-1) over a length of R3 µsec.
- If R2 is 10, on a CPC with "CRTC" (3 and 4) then HSYNC starts from C0 displayed=10 over a length R3 µsec.

Because of this discrepancy, the calibration of a CM14 monitor (464+/6128+) is different from that of a CTM 640/644 (464/664/6128).

The CRTC 4 does not come with a CM14, but it behaves like a CPC+ at the HSYNC signal. AMSTRAD calibrated the CTM delivered with this CPC so that the frame is centered on screen. Connected to the CTM 640/644 of another CPC (with a CRTC 0, 1, 2), the image is shifted to the left because HSYNC occurs 1 µsec later.

In order to ensure visual compatibility with the other CRTC's, this can be overcome by programming R2 with a value 1 lower than that programmed for other CRTC's. But without a calibration menu, it's impossible to know if the CRTC 4 is plugged into its original monitor.

Conversely, plugging a CRTC 0, 1, or 2 CPC into a CM14 monitor or the CTM of a CRTC 4 should cause a shift to the right of the image.

On a CTM monitor, the first character visible on the left is the 15th character from the C0=R2 position if R3>5. (If R2=49 (with R0=63), the 1st visible character is C0=63).

This difference between consideration and display does not change the CRTC's counter management behaviour when R2 and R3 are modified.

As we have seen, the GATE ARRAY is slower to display characters than to take into consideration the HSYNC signal, but there is **also a delay between the instruction of the Z80A that updates the CRTC and the speed of the latter to consider.**

Thus, if R2 is modified very precisely on the character (CRTC) concerned, the CRTC receives the information slightly later, reflecting here the internal delays specific to the Z80A OUT instruction. This slightly reduces the size of the "displayed" HSYNC. See Chapter 14.9 page 137.

The CRTC has a period during which it "agrees" to consider R2.

The GATE ARRAY in this situation, receives the HSYNC signal slightly later, which has an impact on the "**displayed**" length of the HSYNC.

HSYNC treatment begins with HSYNC-GA treatment, which lasts a maximum of 6 µsec. The monitor synchronizes for each line thanks to the C-HSYNC signal generated during the HSYNC-GA

If the C-HSYNC signal changes position or length several times during scanning this leads to an image distortion by the monitor.

If the C-HSYNC signal changes position or length (and if this length exceeds 2 µsec) several times during scanning it causes distortion of the image by the monitor.

The C-HSYNC signal is sent between 1,875 and 2 µsec after the start of the HSYNC. If R3l is 2, the signal is too short to be processed by the monitor deflector circuit. The HSYNC-GA is dependent on the HSYNC-CRTC (start and length):

- It begins when the CRTC HSYNC signal becomes active.
- It stops when the CRTC HSYNC signal becomes inactive and lasts less than 6 µsec.

When R3l is greater than 2, the length of the C-HSYNC signal generated by the GATE ARRAY becomes sufficient for the monitor to try to synchronize the line displayed.

Distortion effects may appear:

- If the HSYNC is too short (>2 µsec and < 6 µsec) (C-HSYNC>0 and <4 µsec).
- If multiple HSYNC of length > 2 µsec are generated on the same line.
- If the HSYNC(s) are not vertically aligned.

During a HSYNC, the different CRTC's no longer manage the C0=R2 condition, which prevents a HSYNC from restarting within a HSYNC.

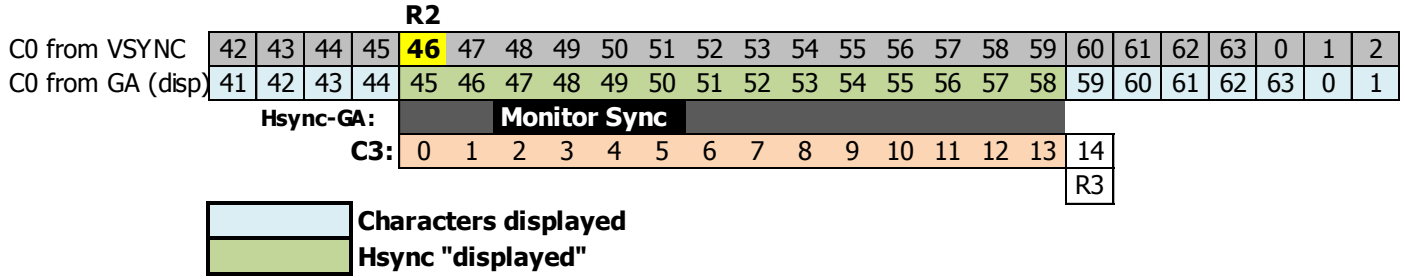
15.2 HSYNC WHEN R2 IS PREDEFINED

The following diagrams show the generation of HSYNC, as it occurs when R2 was programmed before C0vs=R2, which is in principle the general case.

The characters are indicated according to the two timelines, since the HSYNC is managed without "delay" by the GATE ARRAY.

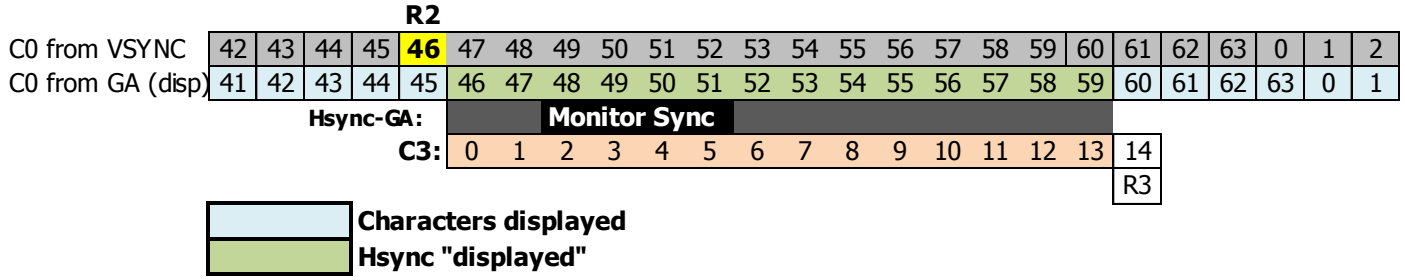
15.2.1 CRTC's 0, 1, 2

CRTC-R2=46
CRTC-R3=14



15.2.2 CRTC's 3, 4

CRTC-R2=46
CRTC-R3=14



15.3 UPDATING R2 DURING HSYNC

15.3.1 GENERAL

During the processing of HSYNC CRTC, an update to R2 is no longer considered if the purpose of this change is to start a new HSYNC during HSYNC.

This lack of consideration avoids a crash of the circuit, which would only generate HSYNC in the situation or R0 would be less than R3. This is particularly the case when R0 is less than R3, which implies that C0 can pass several times on the same value (equal to R2).

On CRTC 0, two HSYNC's cannot be contiguous if position C0=R2 is encountered when C3l reaches R3l, and R3l has not been modified on this position.

On the CRTC's 1, 2, 3 and 4, there is a bug if C0=R2 on C0=R2+R3.

15.3.2 INFINITE HSYNC

The R2 management bug on CRTC's 1, 2, 3 and 4 allow for the creation of an infinite HSYNC (and beyond).

If, for example, we place R0=0, R2=0 and R3=1, we will ask the CRTC to generate a **HSYNC** of 1 µSec at the position C0=R2=0 and ask it to make a HSYNC for all the µsec.

On the 2nd character, C0 is still equal to R2 (=0).

On a CRTC 0, the **HSYNC** will not take place. It will occur on the 3rd C0=0.

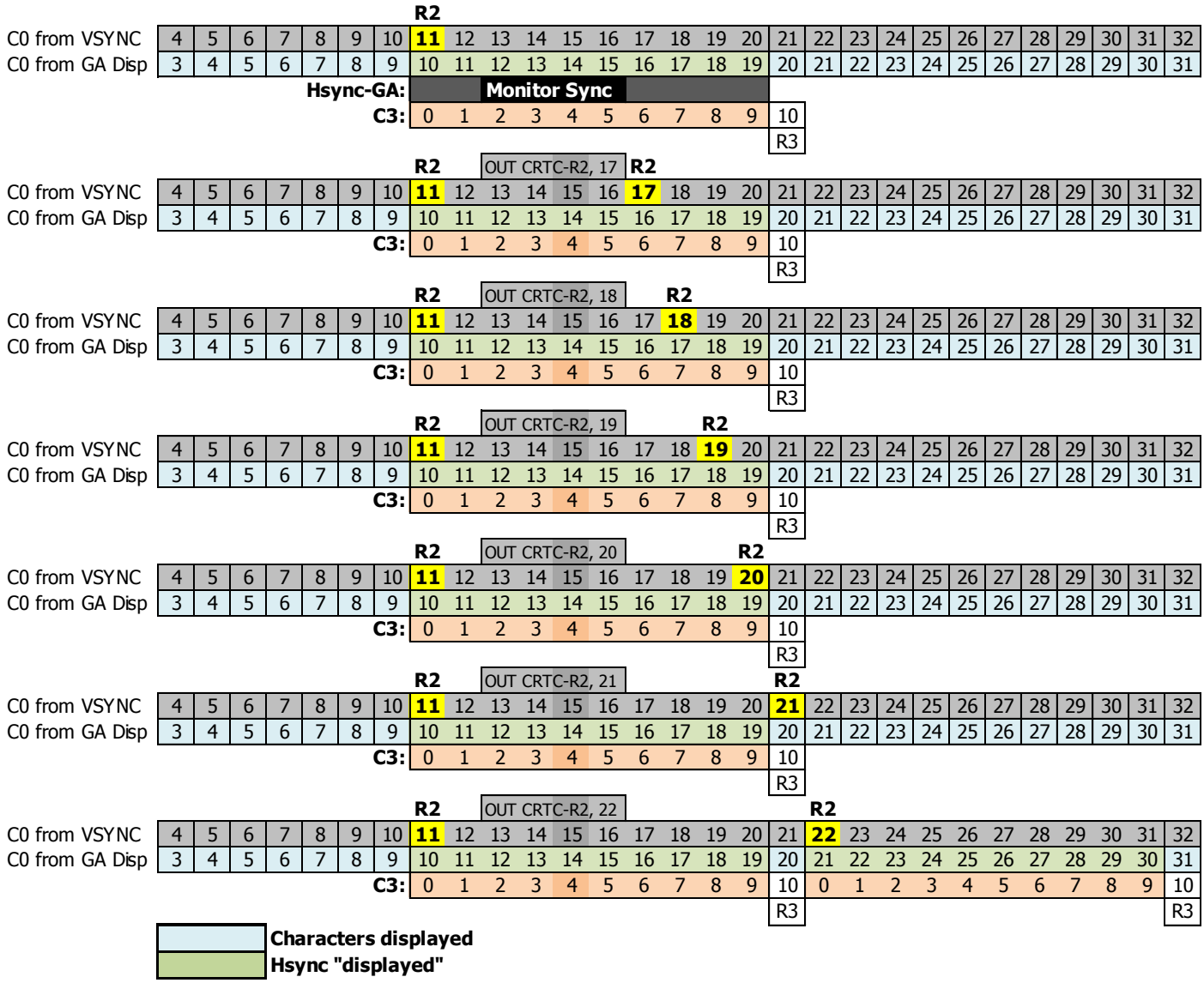
On the other CRTC's, the **HSYNC** does not end and C3 will overflow.
C3 will increment up to 15, return to 0 and then back to 1.

At the end of the overflow of C3, if C0 is still equal to R2 (= 0), then the HSYNC will again continue its route, and C3 overflow again, and so on.

The differences in the consideration of R2 according to the CRTC's are detailed on the diagrams below and describe the update of R2 on different values of C0VS during the **HSYNC**, and its impact.

15.3.3 CRTC 0

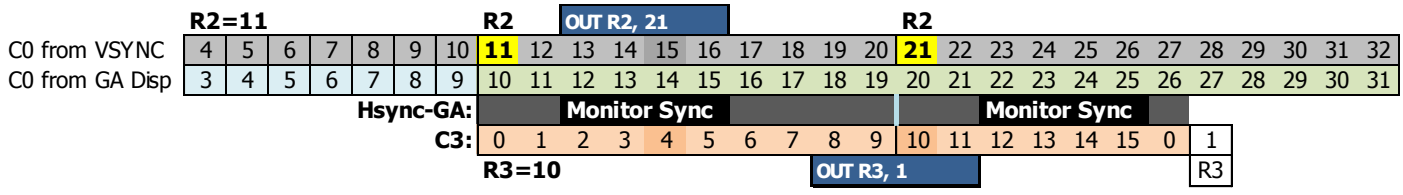
CRTC-R2=11 / CRTC-R3=10 (HSYNC Size=10 chars)



When C3L reaches R3l on position C0=R2+R3l, the HSYNC-CRTC stops (more precisely the black color stops from bit 3 of the displayed byte (e.g. the bits 7.6.5.4 are black)).

On this position, if C0 is again equal to R2 but R3l is modified, then a new HSYNC-CRTC begins without C3l being zeroed (in the same way as for the other CRTC's on this same position without R3l has been modified before).

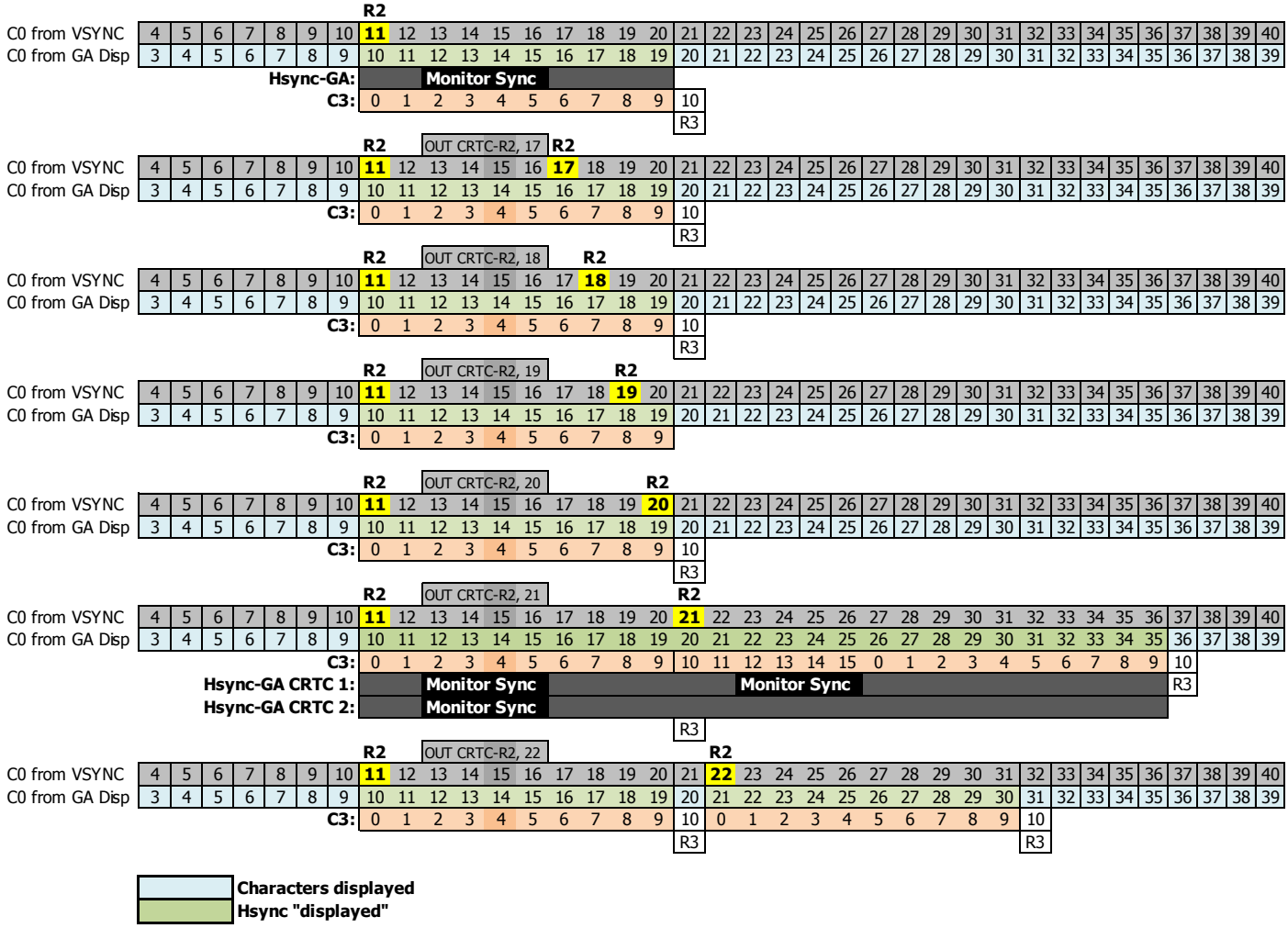
This new HSYNC-CRTC "unexpected" begins around 3.5 Pixel-M2 after the one that has just ended. This corresponds to the start of a **R2.JIT** HSYNC (Just In Time). The diagram in chapter 9.3.4.2 shows how an HSYNC with **R2.JIT** technique begins on this CRTC.



The GATE ARRAY begins a HSYNC again and generates a signal for the monitor if the new limit scheduled in R3l leads the HSYNC to last more than 1,1875 µsec additional (in other words if R3l is >= 2).

15.3.4 CRTC's 1,2

CRTC-R2=11 / CRTC-R3=10 (HSYNC Size=10 chars)



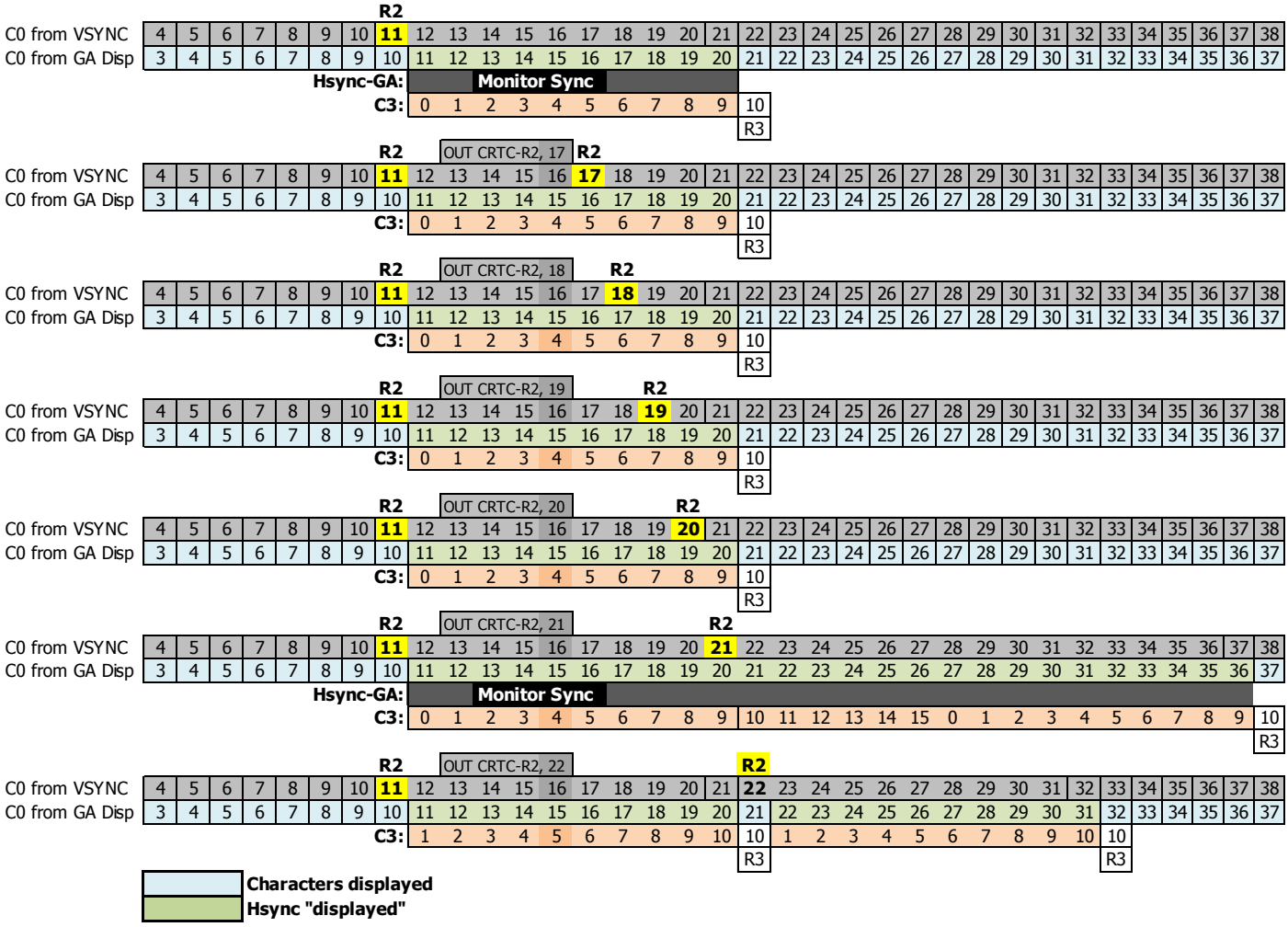
If C0 is equal to R2 on position C0=R2+R3l, the C3l counter overflows (see before the last diagram above). The HSYNC CRTC continues until C3l buckles to reach R3l. An R3l update is considered immediately to bypass the absence of a C3l zeroing.

The CRTC 1, however, has time to generate an "invisible" end of HSYNC, then immediately reactivate the signal for the GATE ARRAY. The latter then reset to 0 its internal character counter and sends a second HSYNC monitor from the 2nd position (position C0vs=23 in the example above). The transition from HSYNC OFF/ON is fast enough to not be visible.

The CRTC 2 does not have time to generate an end of HSYNC and the GATE ARRAY continues to display the black color until the HSYNC of the CRTC occurs (when C3l is again equal to R3l).

15.3.5 CRTC's 3,4

CRTC-R2=11 / CRTC-R3=10 (HSYNC Size=10 chars)



15.4 VSYNC CONSIDERATION DURING HSYNC

15.4.1 GENERAL

On CRTC's 0, 1 and 2, the evaluation of the VSYNC condition is carried out on either the C0 value. C4 must be equal to R7, either because C4 arrives there, or because R7 has been programmed with the value of C4. This is also true when C4 reaches R7 from additional line management.

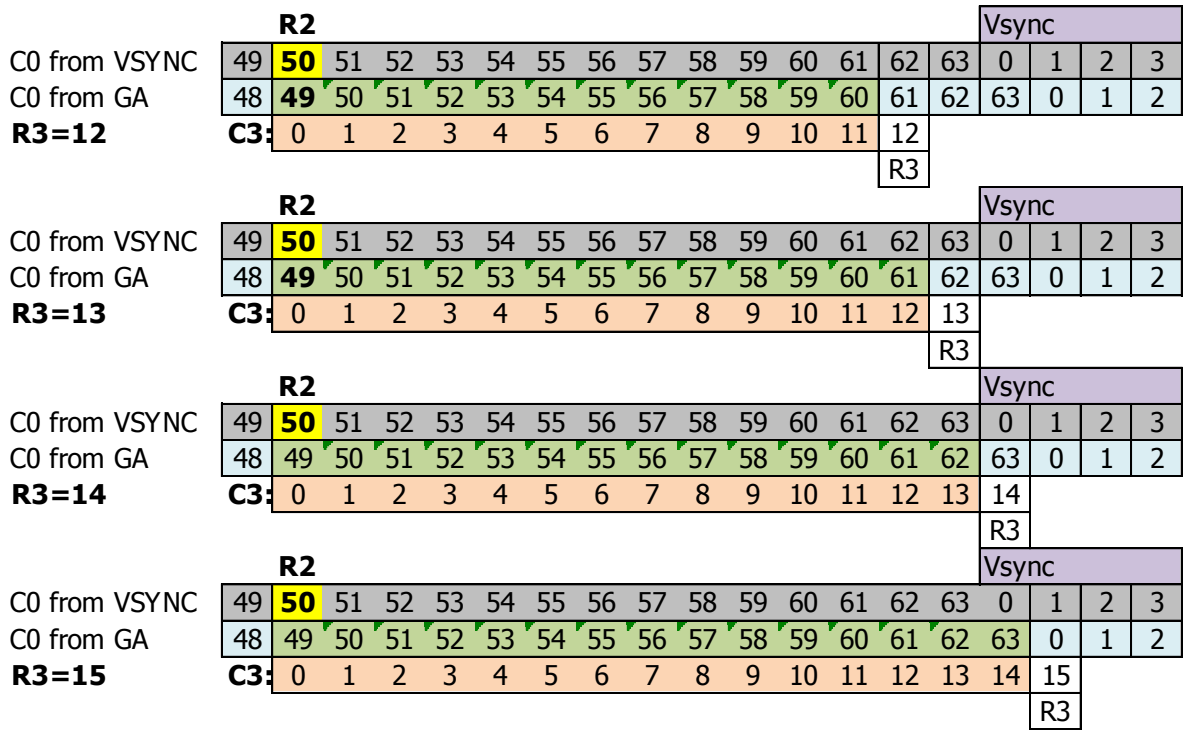
On CRTC's 3 and 4, the assessment of the VSYNC condition only takes place when C4 goes to R7. If R7 is modified with C4, the VSYNC does not occurs.

For all CRTC's except the CRTC 2, there is no problem with a **VSYNC** condition occurring during HSYNC. See the following paragraph for details on CRTC 2.

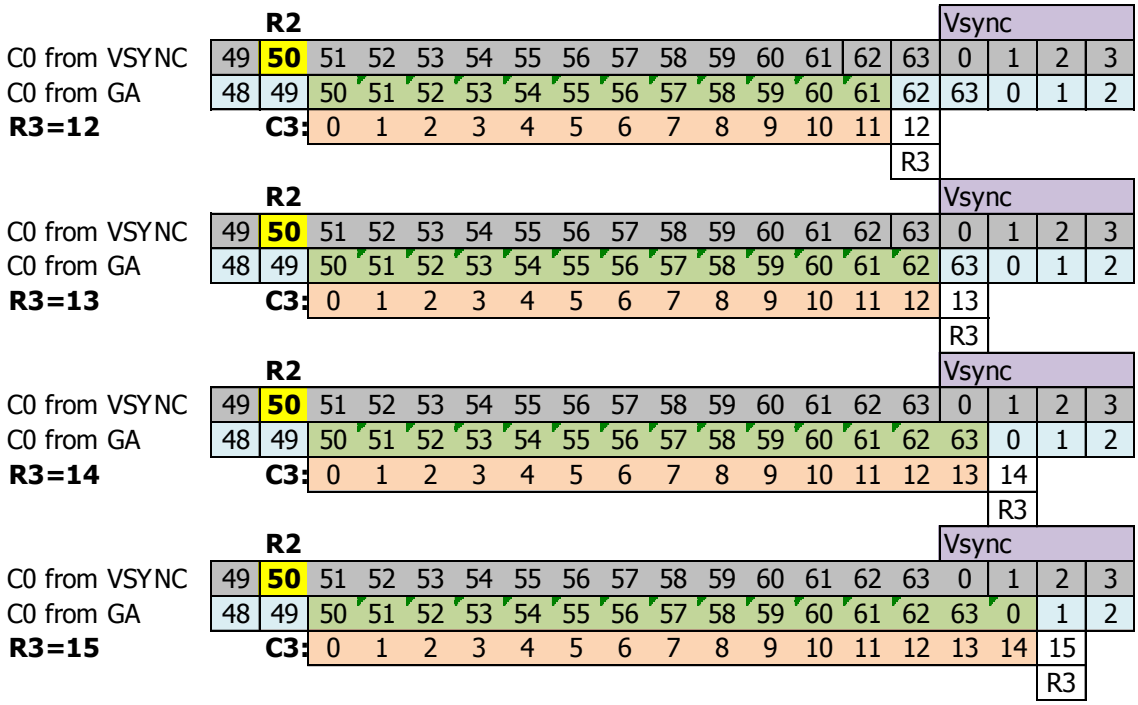
The value of R2 has been applied to a full frame in the following schemas and describes HSYNC's of different lengths that encroach on the VSYNC test area.

The part of the line on the diagrams is the one that corresponds to C4=R7-1 (or previous if R7 is 0), C9=7 (when R9=7).

15.4.2 CRTC 0, 1



15.4.3 CRTC 3, 4



15.4.4 CRTC 2

The **VSYNC** condition is evaluated for all values of C0 during which C4=R7.

If this evaluation occurs while the **HSYNC** is in progress, then a **GHOST VSYN**C starts.

As we have seen in the previous chapter, 2 **HSYNC's** cannot be glued without the first HSYNC overflowing, which implies that the processing of the HSYNC is not completely completed on the position C0=R2+R3l-1 because the counter has not yet been reset.

A VSYNC condition taking place on a position between **C0=R2 and the position of C0 after R3l+1 characters** will therefore trigger a **GHOST VSYNC, unless R2=0.**

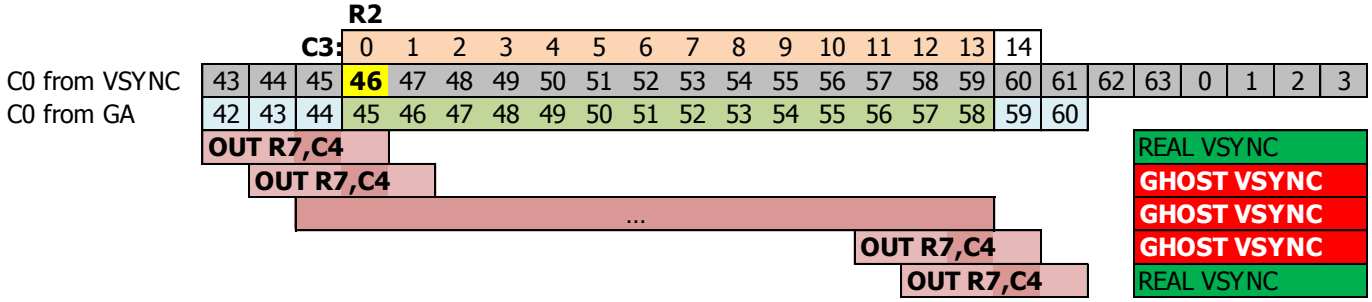
When **R2=0**, the HSYNC starts on C0=0, but the VSYNC has had time to be processed and it occurs normally.

If the VSYNC condition were only "ignored" during the HSYNC, it would occur immediately when leaving HSYNC, since the C4=R7 condition is still true, and the evaluation takes place on all values of C9 and C0.

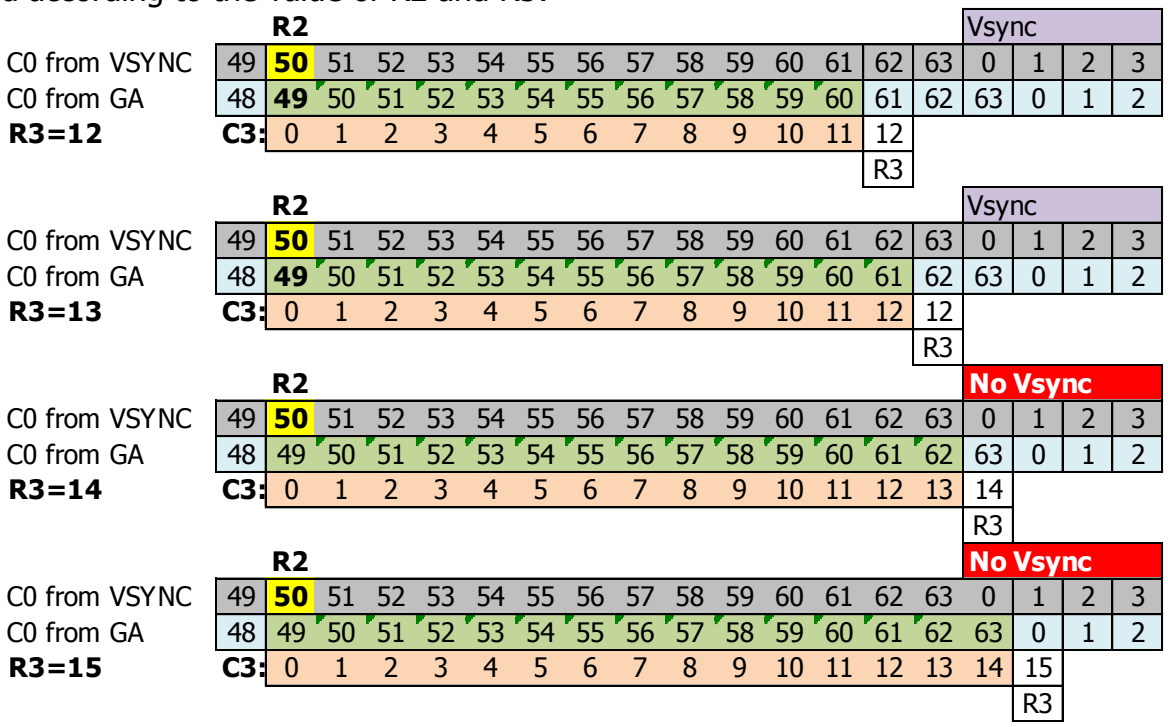
There is certainly a nice conflict on the activation of the VSYNC pin (pin 40) when the HSYNC pin (pin 39) is yet in the high state.

VSYNC can no longer occur because the CRTC has enabled its HSYNC counter and will therefore only be able to accept a new VSYNC condition when **GHOST VSYNC** is completed.

The following diagram precisely describes the inhibition of VSYNC when R7 is modified with the current C4:



The following diagrams describe the different values of C0 on which the VSYNC condition is evaluated according to the value of R2 and R3:



It is possible to circumvent CRTC 2's limitation to treat its VSYNC in several ways if you want to position R2 and R3 freely.

One solution is to manage R7 yourself, so that the VSYNC cannot be processed during HSYNC, and position R7 with C4 once the HSYNC is finished.

The FAKE VSYNC mentioned in chapter 7.3 does not work properly on all the CPC's I have been able to test. It is therefore a solution to avoid while we don't know what this difference is related to and whether it can be overcome. From my observations, it is not the update of bit 0 with 1 of the PPI's port B that generates a VSYNC, but rather the precise moment when the CRTC cancels the GHOST VSYNC.

Another solution is to change the size of the HSYNC at the right time.
The value of R3 is then reduced at the time the VSYNC condition is to be evaluated.

By doing so, VSYNC takes place normally.

		R2							OUT R3, 12							Vsync			
C0 from VSYNC	49	**50**	51	52	53	54	55	56	57	58	59	60	61	62	63	0	1	2	3
C0 from GA	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	0	1	2
R3 init=14	**C3:**	0	1	2	3	4	5	6	7	8	9	10	11	*12*	*13*	*14*			

		R2							OUT R3, 12							Vsync			
C0 from VSYNC	49	**50**	51	52	53	54	55	56	57	58	59	60	61	62	63	0	1	2	3
C0 from GA	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	0	1	2
R3 init=15	**C3:**	0	1	2	3	4	5	6	7	8	9	10	11	*12*	*13*	*14*	*15*		

Note 1:
If R2 and R3 are programmed so that the HSYNC ends on the next line, it means that there are 2 HSYNC's per line. But we also have an active HSYNC on C0=0 and the "**Last Line**" condition is then not evaluated. Given that it is false at the start of the line, this causes the overflow of C4 at the end of the frame (See Chapter 12.4.1 and 15.6).

15.5 BORDER AND HSYNC

15.5.1 CRTC 0, 1, 3, 4

The management of background or BORDER display, with conditions C0=0 and C0=R1, is considered for all CRTC's except 2.

15.5.2 CRTC 2

The condition for restoring the background display occurs when C0=0 (and C4 has never reached R6).
BORDER is enabled when C0=R1.
However, during HSYNC, this C0=0 test is not performed.
(to see if it is the same with C0=R1 to activate the BORDER)

In this condition, **BORDER is not disabled**.
That is BORDERLINE, we can say!

15.6 CRTC 2 AND HSYNC

The CRTC 2 is the champion in the differences when handling certain conditions occurring on specific values of C0 during a HSYNC:

- When the programming of R2 and R3 causes C0 to reach the position preceding C0=0. If a HSYNC takes place on this position, then the VSYNC on C4=R7 is considered activated but without the VSYNC signal being transmitted to the GATE ARRAY (GHOST VSYNC). If R7 is programmed with the value of C4 during HSYNC, this triggers a GHOST VSYNC.
- If R2 is programmed with 0, **the VSYNC condition is detected early enough to occur normally**.
- A HSYNC placed on position C0=0 does not allow the BORDER to be deactivated before a next C0=0.
- A **"Last Line"** condition effective on position C0=0 is cancelled if a HSYNC is placed on position C0=0. Also, C4 increments instead of going to 0 at the end of the frame.
- An update of R4 or R9 is ignored if it occurs during a HSYNC to set the **"Last Line"** state to true.

Some examples :

- If R0=63, R2=50 and R3=14, then the frame scrolls without a black band. The display is on but there is no more VSYNC.
- If R0=63, R2=50 and R3=15, then the frame is no longer displayed, it scrolls without a black band. The display is off (Border), there is no more VSYNC and C4 continuously overflows.
- If R0=63, R2=0 and R3=6, then the frame is no longer displayed but a black band scrolls. The display is deactivated (Border), the VSYNC is present but C4 overflows and therefore the frame scrolls.

15.7 THE RIGHT MOMENT...

If R2 is updated without "precautions", a new HSYNC may occur during or after the end of the current line. In this situation, the monitor must handle multiple HSYNC's or the absence of HSYNC after the update, and this results in horizontal distortion of the image.

The monitor tries to set its image to the new position of C0=R2. Since it takes several lines to achieve this, this results visually in a gradual shift of the lines. This is the same principle as when the length of the HSYNC is changed with a value of less than 6 to shift the image.

Many demos have used these principles (R2 and/or R3) to perform horizontal image distortions or scrolling.

As for R7 at the vertical level, it is possible to avoid this horizontal synchronization stall of the monitor by acting so that C0 returns to the new value of R2 in the same place as the old value of C0=R2. This requires modifying R0 so that it goes back to 0 earlier if the new C0=R2 to be reached is higher than the old one, or conversely to enlarge R0 so that C0 goes back to 0 later if the new C0=R2 to be reached is lower than the old one. You just have to imagine that it is the counters that come to place where they should be.

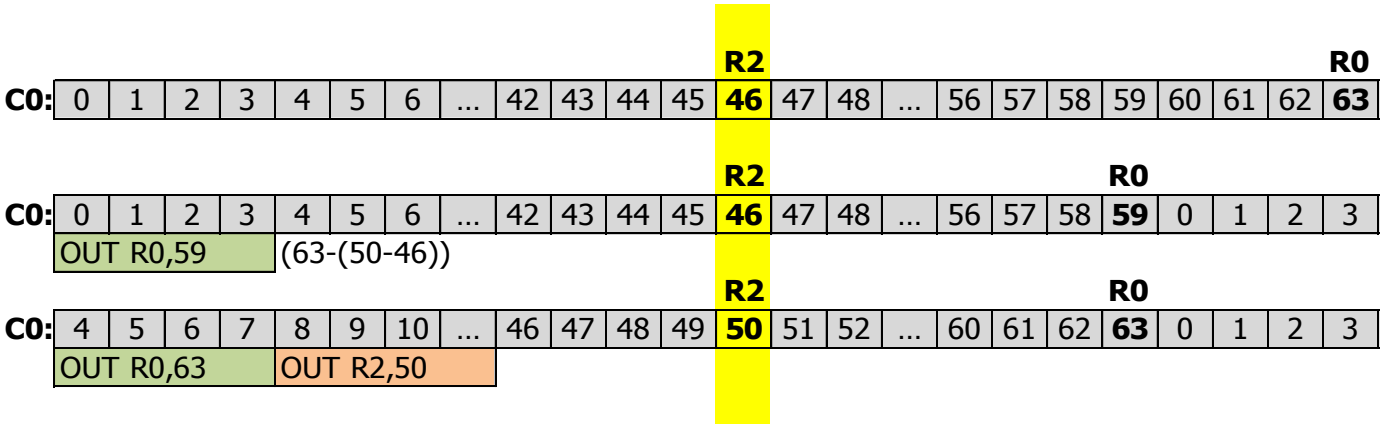
These little gymnastics of repositioning the counters prevent the monitor from losing the HSYNC (or having two in less than 64 µsec), which can be annoying for a program which cannot suffer this type of visual artifact.

Otherwise, if the CPC is not intended to serve the video broadcasting diktat of the organizers of large competition meetings (to which the diktat mentioned in Chapter 16.7 is subject) or adapt to pretty LCD flat screens, it is always possible to hide this stall by modifying R2 in an area not displayed (for example during the VSYNC).

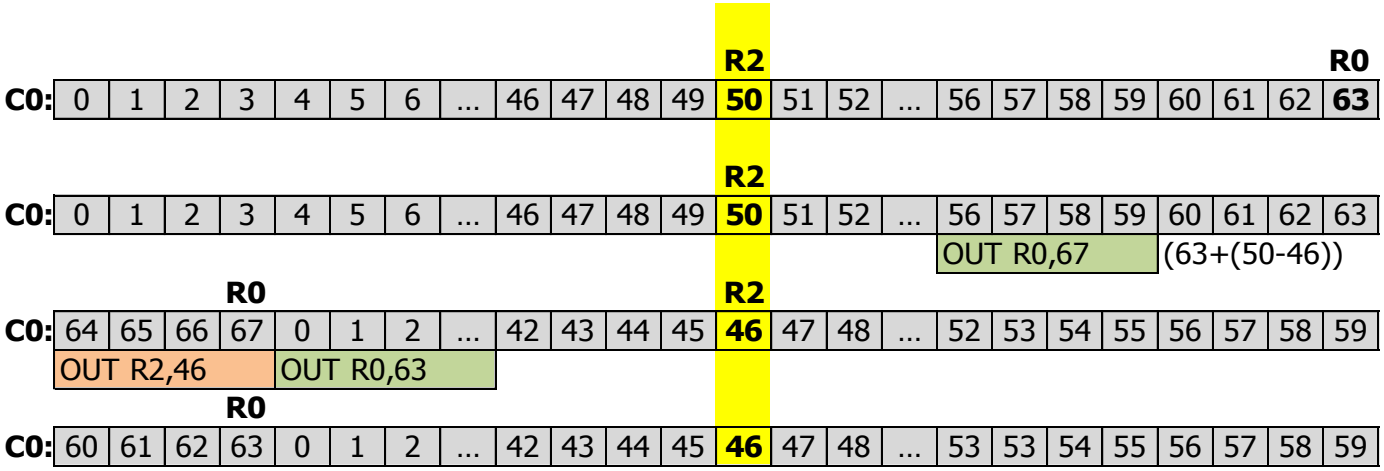
It is also possible, as per vertical synchronisation, to act on R1 and / or R6, or even set all the inks to black at a "specific time".

Setting the H-Hold potentiometer, however, requires a small flat screwdriver to be set to a CTM monitor.
Here are nevertheless 2 diagrams to translate the gymnastics of the counters necessary to go from R2 from 46 to 50 and vice versa, from 50 to 46 (the choice of these values being neither fortuitous nor independent of my will) without traumatizing the screen.

15.7.1 GO FROM R2=46 TO R2=50 ON 64 µSEC LINES.



15.7.2 GO FROM R2=50 TO R2=46 ON LINES OF 64 µsSEC



16 SYNCHRONIZATION : REGISTER R7

16.1 GENERAL

The R7 register is used to fix the moment when the **CRTC activates its VSYNC signal**.
This signal is **received by the GATE ARRAY**, which supports the vertical synchronization signal sent to the monitor.

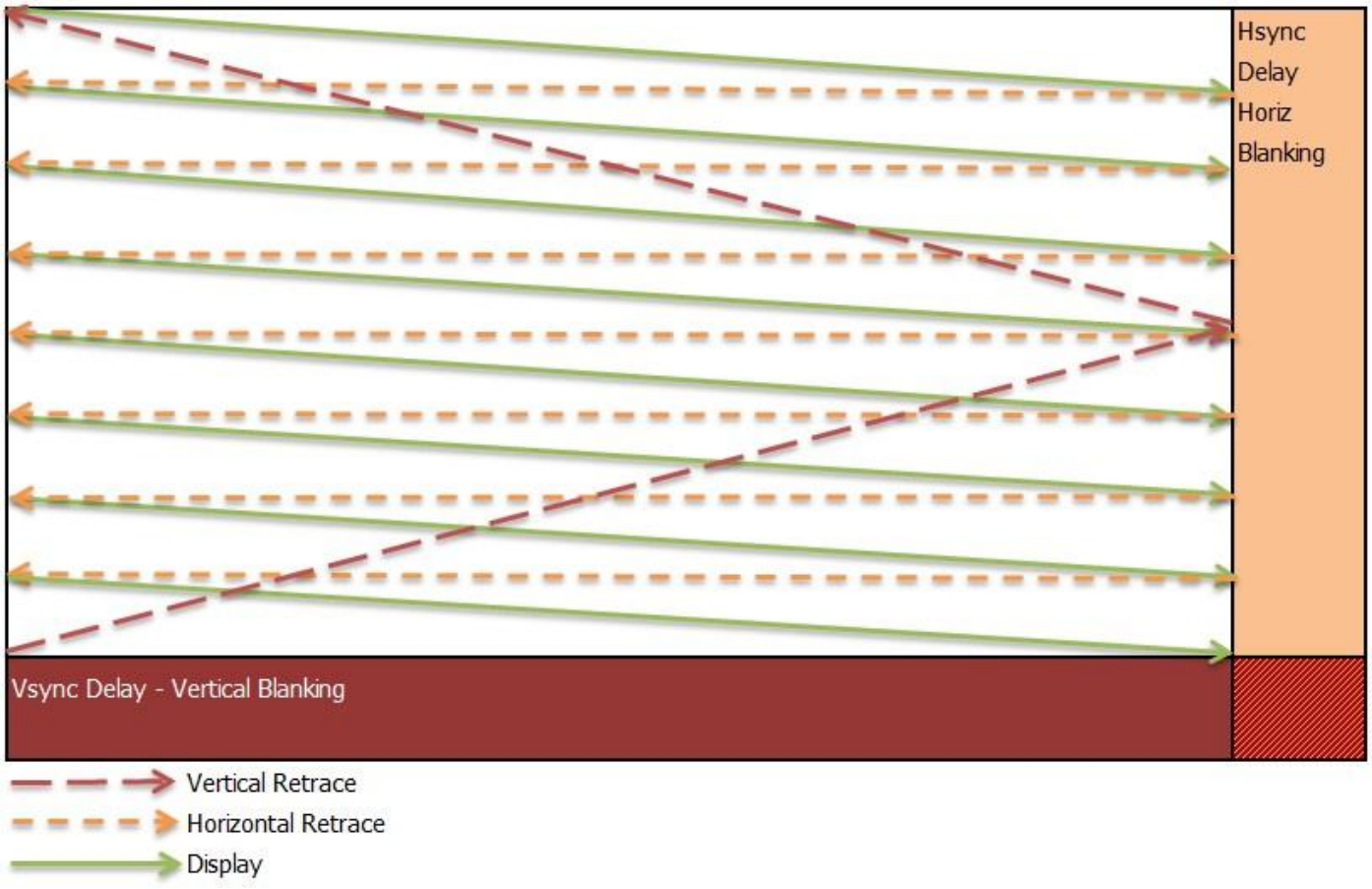
The CRTC VSYNC signal is generated **when C4 reaches R7**.
However, there are some exceptions, specific to each CRTC.

When the VSYNC CRTC begins, the GATE ARRAY supports completely (or partially on CRTC's 3, 4) the monitor synchronization signal. The GATE ARRAY initializes in particular a **V26 counter** with value 0, which will be incremented at the end of each HSYNC.

When V26 reaches 2, the GATE ARRAY activates a "composite" synchronization C-SYNC signal for the monitor, until V26 reaches 6.
This duration represents **4 lines of 64 µseconds in a standard case (R0 = 63)**.

Note : The V26 counter management by the GATE ARRAY takes place whatever the size of the HSYNC encountered. If R3l=1 after the start of the VSYNC, the 1 µsec HSYNC's encountered when C0=R2 will update the V26 HSYNC counter of the GATE ARRAY just before C0=R2+1.

It is from the VSYNC signal sent by the GATE ARRAY that the monitor beam rises diagonally upwards, zigzagging to reposition itself on the character where the signal was received:



The vertical deflector of the monitor constantly descends at a constant speed. Although deliberately exaggerated in the diagram above, the beam descends when displaying the image from left to right. This cannot be seen thanks to internal tilt adjustments. This phenomenon is less noticeable when the beam returns from right to left because the horizontal deflector goes faster during a HBL and the slope is less.

During the VSYNC period, the GATE ARRAY does not display a character, but the CRTC continues to manage its counters and pointers. According to the Motorola documentation ("*MC6845 AD1-465N Fig 14. CRTC Vertical Timing*") this has not always been the case.

This physical return of the electron beam by the monitor is also called Vertical Blanking Line (VBL) because of the real absence of display.

The size of the CRTC VSYNC is expressed in number of lines, each time C0=0.
This number of lines can be programmed on CRTC's 0, 3 and 4 (via register R3h).
It is fixed at 16 for CRTC's 1 and 2 (and for CRTC's 0, 3, 4 when R3h=0).

The size of the VSYNC of the GATE ARRAY is expressed in number of HSYNCs, each time C0=R2. This is the only way for the GATE ARRAY to count the lines it uses to time out the time of the VSYNC. The sending of the C-VSYNC synchronization signal to the monitor by the GATE ARRAY is a step in its processing.

The size of the C-VSYNC signal sent by the GATE ARRAY to the monitor is the delay between the end of the 2nd HSYNC-CRTC and the end of the 6th HSYNC-CRTC. This duration is capped on CRTC's 3 and 4 by the number of VSYNC lines programmed in R3h (see following chapters).

As with HSYNC's, it is possible to generate several VSYNC's during a frame, but the monitor will only be able to lock onto one VSYNC signal.

On a CTM monitor, the image begins to be visible from the 34th line (which represents the second line of the 5th character of 8 lines from the start of the VSYNC).

16.2 VSYNC-CRTC VERSUS VSYNC-GATE ARRAY

16.2.1 VSYNC AREA DISPLAY

When the GATE ARRAY receives the signal emitted by the CRTC (when C4==R7), it sets its counter **V26 to 0**. **The black color will be displayed for 26 lines**. However, the display stops during a period of 4 HSYNC's during which the GATE ARRAY sends the synchronization signal to the monitor. The absence of display during this period is slightly different from the black color displayed, but can still be seen visually on full moon nights (and provided you have unicorn eyes).

The GATE ARRAY reacts as soon as it receives the VSYNC signal from the CRTC. The colorization begins when C4 changes to the value of R7, one microsecond before the display of the character C0=0 (on C9=R7) because the characters are displayed with a delay of one microsecond. This characteristic remains true for CRTC's 3 and 4, whereas it is not the case for the HSYNC signal, which is delayed.

The VSYNC CRTC signal become active under different conditions, which affect when the GATE ARRAY receives the information.

If R7 is programmed with C4 before C4=R7:

- CRTC 0 and 2: The display of the black color starts from the 5th pixel of the VMA word which precedes C4=R7. Note that on these CRTC's, 8 pixels of BORDER are displayed instead of the 2nd byte of VMA.
- CRTC 1: The display of the black color starts from the 6th pixel of the VMA word which precedes C4=R7.
- CRTC 4: The display of the black color starts from the 2nd pixel of the VMA word which precedes C4=R7.

R7.NJIT	C0vs/C4	C0vs=0/C4=R7	C0vs=1	C0vs=2
	C0ga/C4	C0ga=R0/C4=R7-1	C0ga=0	C0ga=1
CRTC 0	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
CRTC 1	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
CRTC 2	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
CRTC 4	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

If R7 is programmed with C4 when C4<>R7:

The **R7.JIT technique consists of updating R7 with the value of C4**.
It does not work on CRTC's 3 and 4, whose VSYNC only starts on C0=0.

On CRTC's 0, 1 and 2 a VSYNC starts sooner or later depending on the instruction used:

- CRTC 0:
  The display of the black color begins on the 5th pixelM2 of the VMA word of C0 for which R7=C4 (bit 3 of the 1st byte of the word) and this regardless of the instruction used (OUT (C), reg8 or OUTI). The display of the VSYNC and the activation of the CSYNC signal are delayed by 1 µsec because R7 is not modified fast enough to be considered immediately.
- CRTC's 1 and 2 :
  - If R7 becomes equal to C4 with OUT (C),reg8: The display of the black color begins on the 9th pixelM2 of the word pointed to by VMA which precedes the position of C0 on which R7=C4 (in other words bit 7 of the 2nd byte of the word pointed to by "C0-1").
  - If R7 becomes equal to C4 via OUTI: The display of the black color begins on the 5th pixel M2 of the word pointed to by VMA which precedes the position of C0 on which R7=C4 (in other words bit 3 of the 1st byte of the word).

R7.JIT	C0vs	C0vs=2	C0vs=3	C0vs=4
	C0ga	C0ga=1	C0ga=2	C0ga=3
CRTC 0	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
CRTC 1	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
CRTC 2	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
Z80a		3rd µs OUT (C),r8 (I/O R7=C4)	4th µs OUT(C),r8	

R7.JIT	C0vs	C0vs=2	C0vs=3	C0vs=4
	C0ga	C0ga=1	C0ga=2	C0ga=3
CRTC 0	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
CRTC 1	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
CRTC 2	Pixel M2	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
Z80a		5th µs OUTI (I/O R7=C4)		

When the 26th HSYNC ends, data display is restored under the following conditions:

- On CRTC's 0 and 1, the background display starts 1 pixelM2 (1/16 Mhz, 0.0625 µsec) after that of the end of HSYNC.
- On CRTC's 2, 3 & 4, the background display starts at the same time as the end of HSYNC.

You can consult chapter 14.9, page 137 to observe the precise detail of the display during an end of HSYNC according to the CRTC and the current graphics mode.

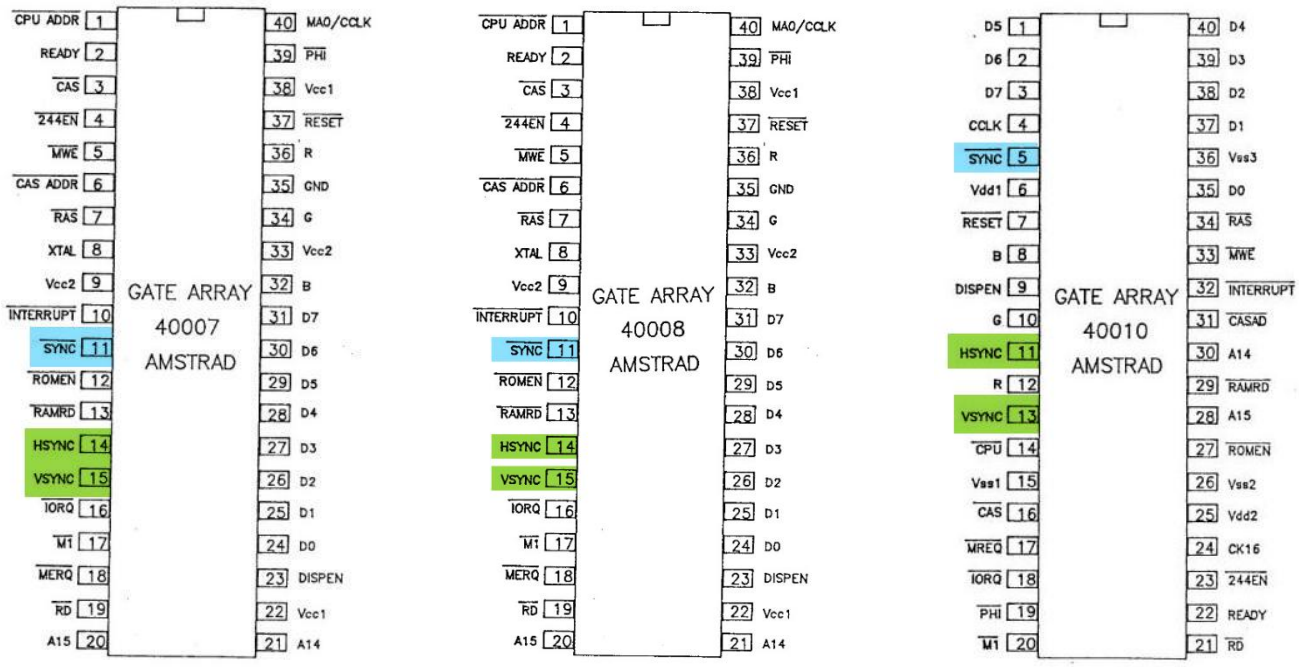
On CRTC's 0, 3 and 4, if the number of lines of the VSYNC is programmed with 1 line in R3h, the GATE ARRAY will support the "display" for 26 lines. This is also true if the duration of the VSYNC produced by the CRTC is reduced to 2 µseconds.

Note :

It may not be a coincidence that the limit of the V26 counter is half the limit of the interrupt line counter of the GATE ARRAY. These counters are necessarily distinct but it is perhaps a saving on the flip-flops which define the limits for each counter within the circuit. The same question may arise for the duration of the monitor signal, which is 4 characters during an HSYNC (H06 between 2 and 6), and 4 HSYNC during a VSYNC (V06 between 2 and 6).

16.2.2 MONITOR C-SYNC SIGNAL

The synchro signal emitted by the GATE ARRAY is called "composite" because the 2 signals HSYNC and VSYNC generated by the CRTC (in green below) are used to generate the signal on a single output (in blue). The C-SYNC signal has a low (active) or high state, the alternation and duration of which allow the components of the monitor to separate them. This synchronization signal is present on pin 4 of the DIN6 video connector.



To my knowledge, the CTM 640/644 (color screens) contain a Sanyo LA7830 (or Nec µPC1378H) component to manage the vertical deflection, and a Sanyo LA7800 to separate the horizontal and vertical synchronization signals. The GT 64/65 (monochrome screens) contain an LA1385 (or nec µPC1031H2) to manage the vertical deflection.

The C-SYNC composite signal of the CPC is of the **XNOR** type. A XNOR B ▶ A XOR NOT (B)
The signal state depends on the HSYNC and VSYNC transitions processed by the GATE ARRAY.

Unlike an AND-type C-SYNC signal, a C-SYNC XNOR allows the HSYNC signal to coexist during the VSYNC period. The signal state for the HSYNC information in this situation is then reversed.

The HSYNC signal from the CRTC is low (0) when inactive and high (1) when active.
The VSYNC signal from the CRTC is low (0) when inactive and high (1) when active.
The C-SYNC signal produced by the GATE ARRAY is active when it is low (0)

GATE ARRAY manages several counters and states in order to correctly process the C-SYNC signal.

- A **H06 counter** counts the number of characters processed during an HSYNC. The GATE ARRAY cannot indeed know C3l from the CRTC.
- A **VSYNC_GA** state becomes true when the VSYNC CRTC signal becomes true (the CRTC VSYNC signal state is no longer used thereafter, except for CRTC's 3 and 4).
- A **V26 counter** counts the number of HSYNCs as long as VSYNC_GA is true.
- The **SIG_GA_HSYNC** and **SIG_GA_VSYNC** states are used to generate CSYNC. Like the CRTC's HSYNCs and VSYNCs, they are low (0) when inactive.

We can define C-SYNC as follows: **C-SYNC=SIG_HSYNC XNOR SIG_VSYNC**

For clarity, I distinguish the horizontal and vertical nature of the C-SYNC signal, respectively with the terminologies C-HSYNC and C-VSYNC.

Counter V26 is incremented by the GATE ARRAY when the CRTC signals an end of HSYNC.
The GATE ARRAY activates the CSYNC signal **when the counter V26 reaches 2. It deactivates this signal when V26 reaches 6**.

On CRTC 0, if the number of VSYNC lines is programmed via R3h=1, the GATE ARRAY autonomously manages the sending of the synchronization signal to the monitor while the VSYNC signal from the CRTC has become inactive. It can be assumed that CRTC's 1 and 2 do the same thing, even if the VSYNC signal of these CRTC's remains active for 16 HSYNCs.

On CRTCs 3 and 4, the VSYNC signal from the CRTC must be active for the synchronization signal to be sent by the ASIC to the monitor. If R3h is programmed with 2, the VSYNC signal from the CRTC is maintained on lines C9=0 and C9=1. The C-VSYNC signal starts at the end of the HSYNC-CRTC of line C9=1 and then stops when C0 returns to 0.

For example, if R3h=2, R3l=13, R2=40 and R0=63, the signal for the monitor is emitted from 53 (40+13) and stops 11 µsec later (63+1-53) when C0 goes back to 0. Depending on the accuracy of the monitor's V-Hold potentiometer setting, this value may be sufficient to synchronize the image vertically. If R3l is enlarged, the signal size decreases. The screen scrolls and can no longer synchronize the image.

Programming R3h>2 on CRTC's 3 and 4 increases the duration of the signal by R0 µsec (except for the 4th line where the signal stops when C0 reaches R2+R3l.

C9/C0 :	0	1	.....	R2	.....	R0
0	VSYNC Black Color		V26 :	1		
1	VSYNC Black Color			2		
2	VSYNC Monitor			3		
3	VSYNC Monitor			4		
4	VSYNC Monitor			5		
5	VSYNC Monitor			6		
6	VSYNC Black Color			7		
7	VSYNC Black Color			8		
0	VSYNC Black Color			9		
1	VSYNC Black Color			10		
2	VSYNC Black Color			11		
3	VSYNC Black Color			12		
4	VSYNC Black Color			13		
5	VSYNC Black Color			14		
6	VSYNC Black Color			15		
7	VSYNC Black Color			16		
0	VSYNC Black Color			17		
1	VSYNC Black Color			18		
2	VSYNC Black Color			19		
3	VSYNC Black Color			20		
4	VSYNC Black Color			21		
5	VSYNC Black Color			22		
6	VSYNC Black Color			23		
7	VSYNC Black Color			24		
0	VSYNC Black Color			25		
1	VSYNC Black Color			26		
2						

The management of the incrementation of V26 at the end of the HSYNC-CRTC causes a noticeable side effect. Indeed, when C4 reaches R7, on line C9=0, C0=0, the CRTC activates its signal VSYNC, and the GATE ARRAY resets counter V26 to 0. But if the size of the HSYNC defined via R3l overflows on the line C4=R7, V26 will change to 1 (end of HSYNC) when it has just changed to 0 (C4=R7).

In this situation, the GATE ARRAY will activate C-VSYNC at the end of the next HSYNC, which will occur R0+1 µsec later, at the start of C9=1. C-VSYNC occurs one line earlier, compared to a frame where HSYNC does not overflow onto the new line. And the screen will therefore be displayed one line lower.

For example, if R2=50 (51 on CRTC 0, 1 and 2, whose HSYNC is not delayed) and R3l=14, then the image goes down one line. If R3l is programmed with the value 13, then V26 will not be incremented when C4=R7, and the VSYNC signal will be sent from the end of the HSYNC which will start on C9=1. The image will then move up one line.

The GATE ARRAY clocks the CRTC via a clock signal CLK whose period is 5 x 0.0625 µsec high and 11 x 0.0625 µsec low. When the CRTC signals a start or an end of HSYNC, there are delays of 1 or 2 pixel-M2 (1 pixel-M2=0.0625 µsec) before and after this clock signal, which depend on the type of CRTC (for the management of the HSYNC signal) and a level of tolerance specific to the circuit.

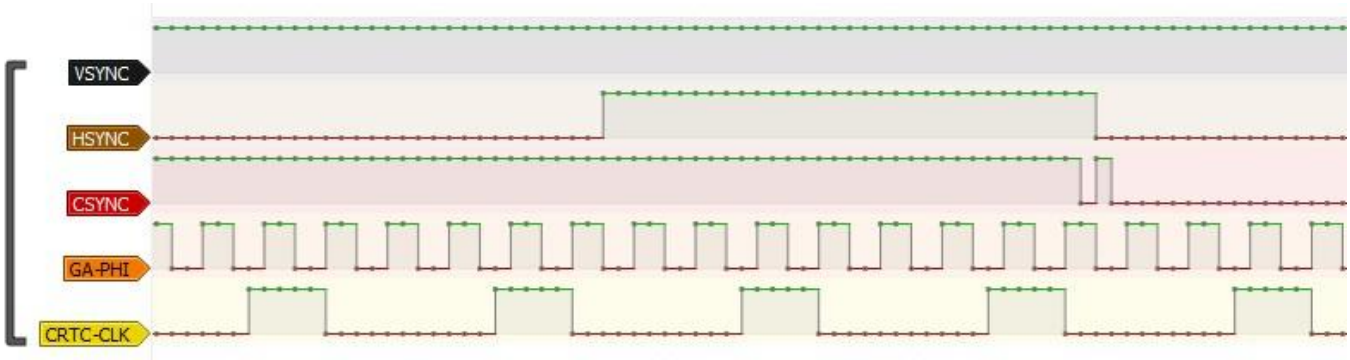
These shifts between the active edges of the signals lead to a phase shift between the start of the HSYNC signal received by the GATE ARRAY and the start of the count to activate the C-HSYNC signal after 2 µsec. A phase shift also exists at the end of the HSYNC signal which triggers the end of the C-HSYNC signal.

The C-HSYNC signal becomes active (low state) 1 or 2 Pixel-M2 before the end of the 2 µs if R3l>=2.
The C-HSYNC signal becomes inactive (high state) in 2 situations:

- 1 or 2 Pixel-M2 after the end of the HSYNC signal
- If the duration of C-HSYNC has reached 4 µsec

The C-VSYNC signal becomes active 1 Pixel-M2 after the end of the 2nd HSYNC.
The C-VSYNC signal becomes inactive 1 Pixel-M2 after the end of the 6th HSYNC.

On the diagram below, the interval between 2 dotted lines is 1/16 Mhz (0.0625 µsec). It shows the 2nd HSYNC signal (programmed with R3l=2) during a VSYNC. We can observe the activation of the C-HSYNC signal for 0.0625 µsec (low state) followed by its deactivation (high state) for 0.0625 µsec before being activated again to generate the C-VSYNC signal, which will last until the end of HSYNC Number 6.

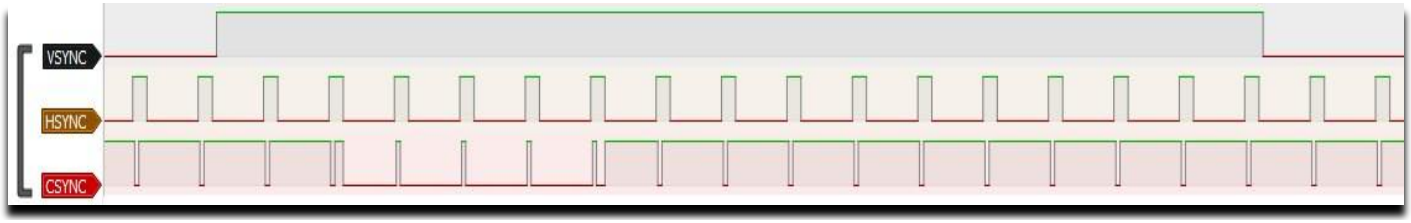


16.2.3 C-SYNC ALGORITHM

The following algorithm represents the simplified logic of C-SYNC signal processing.

```
If VSYNC-CRTC Transition OFF ▶ ON
        V26=0                                   ; HSYNC CRTC counter initialized
        VSYNC_GA=true                           ; GA VSYNC support
        CBLACK_VSYNC=true                       ; Black color enabled for VSYNC

If HSYNC-CRTC Transition OFF ▶ ON
        H06=0                                   ; NOP CRTC counter initialized
        CBLACK_HSYNC=true                       ; Black color enabled for HSYNC

If HSYNC-CRTC Transition ON ▶ OFF
        SIG_GA_HSYNC=LOW                        ; Hsync GA signal inactive
        CBLACK_HSYNC=false                      ; Disable HSYNC black color
        If VYSNC_GA==true                       ; GA VSYNC management if active
                V26++                           ; HSYNC counter incremented
                If V26==2: SIG_GA_VSYNC=HIGH (and CRTC-VSYNC (on CRTC 3/4))
                If V26==6: SIG_GA_VSYNC=LOW
                If V26==26: CBLACK_VSYNC=false      ; Disable black color VSYNC
                VSYNC_GA=false                      ; VSYNC GA completed

If CRTC Transition Character
        H06++                                   ; NOP counter incremented
        If H06==2: SIG_GA_HSYNC=HIGH
        If H06==6: SIG_GA_HSYNC=LOW

BLACKCOLOR=CBLACK_HSYNC or CBLACK_VSYNC
CSYNC=SIG_GA_HSYNC XNOR SIG_GA_VSYNC
```

Below, the measurement on the GATE ARRAY of the CRTC VSYNC and HSYNC signals which arrive there and the CSYNC signal which is generated between HSYNC N°2 and HSYNC N°6.



16.2.4 TOLERANCES

According to measurements made on several CTMs, the duration of the signal emitted for the monitor must be greater than 11-12 µseconds. Below this value, the monitor can no longer "anchor" the image, regardless of the adjustment made with the potentiometer on the back of the monitor.

Beyond 12 µseconds, the image stability depends mainly on the monitor setting.
Depending on the setting, per period of 6 µseconds, the stability is reduced. We can thus have slight jolts of the image over a period of 6 µsec, then a period of clear jumps over the following period, and finally a permanent stall.

16.2.5 CRTC AND GATE ARRAY INTERACTIONS

If the VSYNC signal from the CRTC becomes inactive, the GATE ARRAY continues to manage the C-VSYNC on its side. CRTC's 3, 4 still need the active VSYNC signal from the CRTC to continue sending the C-VSYNC signal to the monitor (the designers of the "CRTC ASIC" reproduced the logic used for the C-HSYNC of CRTC's 0, 1 and 2).

The maximum (programmable on CRTC 0, 3, 4) number of lines of a VSYNC generated by the CRTC is less than the 26 lines of the VSYNC of the GATE ARRAY.

If a VSYNC CRTC is activated again while the GATE ARRAY VSYNC is in progress, then the GATE ARRAY VSYNC line counter is reset to 0. It is incremented each time the GATE ARRAY receives the HSYNC end signal of the CRTC.

In the following example, R3h=0 (16 lines). A second VSYNC CRTC is triggered from C4=12/C9=4, by positioning R7=C4 before the HSYNC. This example works on CRTC's 0, 1 and 2, because they can activate a VSYNC as long as C4=R7, whatever the value of C9.

Update R7	C4	C9	CRTC CNT	GATE ARRAY COUNTER		MONITOR
10	10	0	1	1		Black Color
	10	1	2	**2**		**VSYNC Monitor**
	10	2	3	**3**		**VSYNC Monitor**
	10	3	4	**4**		**VSYNC Monitor**
	10	4	5	**5**		**VSYNC Monitor**
	10	5	6	6		Black Color
	10	6	7	7		Black Color
	10	7	8	8		Black Color
	11	0	9	9		Black Color
	11	1	10	10		Black Color
	11	2	11	11		Black Color
	11	3	12	12		Black Color
	11	4	13	13		Black Color
	11	5	14	14		Black Color
	11	6	15	15		Black Color
	11	7	16	16		Black Color
	12	0		17		Black Color
	12	1		18		Black Color
	12	2		19		Black Color
	12	3		20		Black Color
12	12	4	1	21	1	Black Color
	12	5	2	22	**2**	**VSYNC Monitor**
	12	6	3	23	**3**	**VSYNC Monitor**
	12	7	4	24	**4**	**VSYNC Monitor**
	13	0	5	25	**5**	**VSYNC Monitor**
	13	1	6	26	6	Black Color
	13	2	7		7	Black Color
	13	3	8		8	Black Color
	13	4	9		9	Black Color
	13	5	10		10	Black Color
	13	6	11		11	Black Color
	13	7	12		12	Black Color
	14	0	13		13	Black Color
	14	1	14		14	Black Color
	14	2	15		15	Black Color
	14	3	16		16	Black Color
	14	4			17	Black Color
	14	5			18	Black Color
	14	6			19	Black Color
	14	7			20	Black Color
	15	0			21	Black Color
	15	1			22	Black Color
	15	2			23	Black Color
	15	3			24	Black Color
	15	4			25	Black Color
	15	5			26	Black Color
	15	6				Current Ink

16.3 VSYNC PROTECTION

To prevent infinite VSYNC, CRTC designers have provided two mechanisms.

The first consists in ignoring the comparison of C4 with R7 when R7 is modified during VSYNC. It is not possible to trigger or inhibit a VSYNC during a VSYNC. Thus, modifying the value of R7 with a value of C4 reached during the VSYNC does not cause a new VSYNC. Modifying R7 with a C4 value different from the initial C4 value does not interrupt the current VSYNC.

The second protection mechanism is to check if the equality between C4 and R7 has changed. This condition changes if C4 increments or if R7 is changed. For example, if R7=0, then a VSYNC occurs when C4=0. If R4 is 0, then C4 remains at 0 each time C9 goes back to 0. C4 is therefore 0 during the VSYNC but also after the end of the VSYNC. In this context, there is no more VSYNC. If R7 is modified and C4 is different from R7 when C9 goes back to 0, this allows the VSYNC blocking mechanism to be lifted.

This second mechanism was not renewed by the designers of CRTC's 3 and 4 for AMSTRAD. Even if the equality C4=R7 has not changed (C4=R7=0 for example), a new VSYNC starts immediately as soon as the current VSYNC ends, which causes an infinite VSYNC. In this particular case, the ASIC no longer generates the C-VSYNC for the monitor. (Ouch!).

For all CRTC's, if the comparison between C4 and R7 changes during VSYNC, the second blocking mechanism is overridden. The protection of VSYNC is then ensured only by the first mechanism described above. Thus, if R7=0 and R4=1, a VSYNC occurs when C4=0.

If R3h=0 and R9=7, then C4 will change to 1 on the 9th line, and return to 0 on the 17th line. The CRTC's VSYNC will then start again immediately, creating an infinite VSYNC. In this situation, the GATE ARRAY (CRTC's 0, 1, 2) or ASIC (CRTC's 3, 4) will trigger a new VSYNC if it has finished processing the previous one.

The GATE ARRAY VSYNC is considered complete when the 26th row has been processed.

16.4 CONDITIONS TO CONSIDER

For all CRTCs, the update of R7 with C4 can take place up to the last µsecond preceding C4=R7. In other words, if R7 is modified with the value of C4 on line C4-1, C9=R9 and C0=R0, then the **VSYNC** will be active as soon as C4=R7.

16.4.1 CRTC 0

VSYNC CRTC starts when C4=R7.

If C4 becomes equal to R7 when C0vs reaches 0, then the VSYNC starts on C0vs=0.

16.4.1.1 R7 UPDATE

If R7 is modified with the value of C4, then the VSYNC is triggered immediately if it was not already in progress, **except if this modification occurs when C0vs=0 or C0vs=1**.

If the modification of R7 with the value of C4 took place when C0vs<2, we are in a **BLOCKED VSYNC.** This means that the blocking mechanism described in the previous chapter is active, but without the VSYNC having occurred**.**

The VSYNC can no longer occur on the value of C4=R7 until an unblocking condition is produced.
The **BLOCKED VSYNC of CRTC 0** should not be confused with the **GHOST VSYNC of CRTC 2** (see following chapters). With CRTC 0, only the second blocking mechanism is activated. A VSYNC is possible as soon as the release conditions are true.

When R7=C4 with C0vs>1, the VSYNC is "triggered" during the line.
In this case, the row counter starts with 0.

This VSYNC line counter is initialized at the start of the next line when C0=0.
All the lines for which C4=R7 are affected by this mid-line VSYNC start.
The total duration of the VSYNC is increased by the number of µsec corresponding to the calculation R0 - C0vs (in relation to the exact moment when R7 was updated).
So if a VSYNC is triggered during line number 1, then the VSYNC ends at the end of line 17.

Examples :

- If R7 is modified on C0vs=#36, then the next read of the PPI-PORT B (read 6 µsec later on C0vs=#3C) returns an active status of the VSYNC.
- If R7 is modified on C0vs=#00, then the next read of the PPI-PORT B (read 6 µsec later on C0vs=#06) returns an inactive status of the VSYNC

16.4.1.2 R0 UPDATE

The counter C0 needs to reach the value 2 on the line preceding that where C4=R7 for a VSYNC to be considered. Thus, if the length of the line preceding the VSYNC condition is limited to 0 or 1 µsec (R0=0/1), the latter will not take place and the VSYNC will be blocked for the condition C4=R7, as if it had had place. A new VSYNC is then only possible if an unblocking condition is satisfied (C4 or R7 update).

If R0 (which was greater than 2) becomes equal to 0 on C0=0 of the first line C4=R7, then the VSYNC starts on C0=0, but like all the other counters on this CRTC, the VSYNC line counter C3h is frozen, and the VSYNC is not deactivated if R3h was worth 1 (because C3h can no longer reach R3h).

If R0 (which was greater than 2) becomes equal to 1 on C0=0 of the first line when C4=R7, then the VSYNC starts on C0=0, but the counter can be incremented when C0 returns from 1 to 0. If R3h was 1, then VSYNC stops. It will have lasted 2 µsec in total. This is nevertheless sufficient to trigger the management of the VSYNC by the GATE ARRAY.

16.4.2 CRTC 1

VSYNC starts when C4=R7.

If R7 is modified with the value of C4, then VSYNC is triggered immediately.
If R7 is modified when C0vs=#36, for example, then the next reading of the PPI (at the earliest 5 µsec after, so on C0=#3B) returns an active status of the VSYNC.

The "triggered" activation of the VSYNC counts the line as if the VSYNC had started when C0=0.

As a result, the total VSYNC duration is reduced by the number of µsec corresponding to the value of C0+1 from the moment R7 was updated.

If a VSYNC is triggered during line number 1, then the VSYNC ends at the end of line 16

16.4.3 CRTC 2

VSYNC is considered on all values of C0 and C9 when C4=R7.
If the VSYNC condition occurs during a **HSYNC** from C0=R2 to C0=R2+R3 (1 µsec longer than the visual size of the HSYNC) then the CRTC generates a **GHOST VSYNC.**

If R7 is modified with the value of C4, then the VSYNC is triggered immediately, except during the **HSYNC** period (C0=R2 to C0=R2+R3), which triggers the **GHOST VSYNC**.

A **GHOST VSYNC** means that the CRTC counts the lines as if a VSYNC were taking place by preventing a new VSYNC from occurring, but without the VSYNC pin being enabled.

The "triggered" activation of the VSYNC counts the line as if the VSYNC had started in C0=0.

As a result, the total VSYNC duration is reduced by the number of µsec corresponding to the value of C0+1 from the moment when R7 was updated

If a VSYNC is triggered during line number 1, then the VSYNC ends at the end of line 16.

To get around the problem of no VSYNC on this CRTC, just avoid creating the condition of a **GHOST VSYNC**.

It is possible by positioning R7 far into the cosmos (e.g. 127) and then updating R7 with C4 when C0 is no longer present in the HSYNC period of the line considered.

It is also possible to reduce R3 at the last moment, but it is more tricky. In this case, it should not be forgotten that if the HSYNC overflows on C0=0, then C4 overflows on the last line of the frame and the BORDER remains activated.

16.4.4 CRTC 3, 4

VSYNC starts when C4=R7 and C9=C0=0.

If R7 is modified with the value of C4 while C0>0, it will not trigger VSYNC.

There is no VSYNC reentrancy protection mechanism on these circuits.
If the condition C4=R7 and C9=C0=0 has not changed and is renewed in the absence of an active VSYNC, then a VSYNC starts again.

Finally, it is necessary that the VSYNC CRTC lasts at least 3 lines for the C-VSYNC monitor signal to be generated.

16.5 DELAYED VSYNC

16.5.1 CRTC 0

VSYNC can be delayed when one of the "Interlace" modes is active from C4==R7.

- A delay of half a line when the parity of a frame is even. VSYNC then occurs on C4==R7 and C0=R0/2.
- A delay of one complete line in the particular case where R9 is odd in IVM mode (R8=3) on an odd frame and an odd C4. See chapter 19.5.2

16.5.2 CRTC 1

VSYNC can be delayed when one of the "Interlace" modes is active from C4==R7.

- A delay of half a line when the parity of a frame is even. VSYNC then occurs on C4==R7 and C0=R0/2.

Unlike CRTC's 0, 3 and 4, this CRTC does not correctly synchronize a frame in IVM mode when R7 is odd on an image composed of characters with an odd number of lines (R9 even).

16.5.3 CRTC 2

VSYNC can be delayed when one of the "Interlace" modes is active from C4==R7.

- A delay of half a line when the parity of a frame is even. VSYNC then occurs on C4==R7 and C0=R0/2

16.5.4 CRTC' s 3 & 4

VSYNC can be delayed when one of the "Interlace" modes is active from C4==R7.

- A delay of half a line when the parity of a frame is even. VSYNC then occurs on C4==R7 and C0=R0/2.
- A delay of one complete line in the particular case where R9 is odd in IVM mode (R8=3) on an odd frame and an odd C4. See chapter 19.5.5

16.6 LIMITLESS VSYNC !

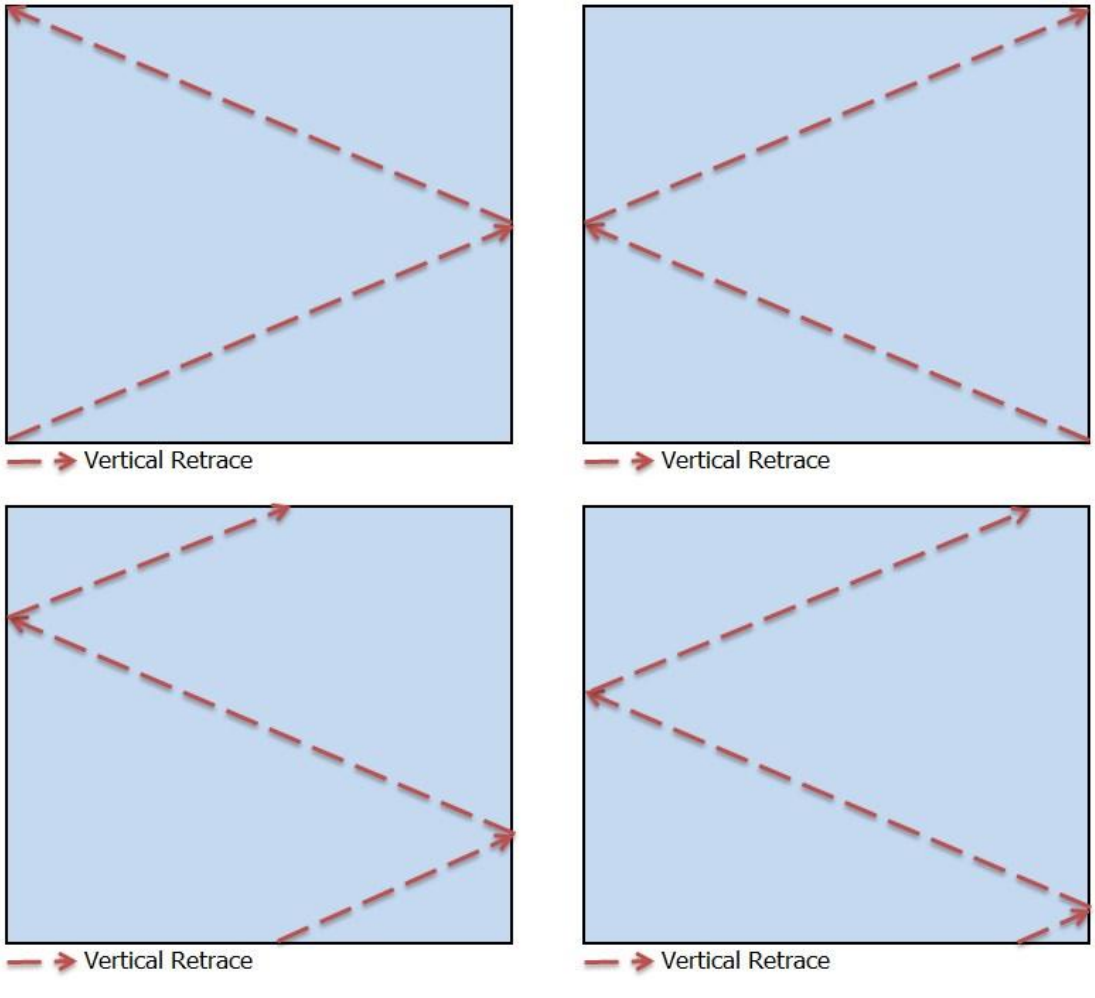
In a system without a GATE ARRAY, if the VSYNC signal from a CRTC was sent directly to the monitor, the monitor would be able to start its return phase to the top of the monitor to satisfy the constraints of the interlaced mode.

As part of an image composed of 2 frames, the CRTC in Interlace mode generates an odd and even frame of 312 lines each, separated by a line on which the VSYNC will be generated in the middle of the line (R0/2). See Chapter 19.3, page 192.

Unfortunately (or fortunately, we are going to see it) it is the GATE ARRAY which filters the HSYNC and VSYNC signals from the CRTC and which generates a mixed signal HSYNC+VSYNC towards the monitor as soon as it arrives on the 2nd HSYNC which follows the VSYNC generated by the CRTC. It thus short-circuits the "native" timing of the CRTC's for "Interlace" operation.

It is still possible to force the GATE ARRAY to send the C-VSYNC signal to the monitor at the right time if you want to obtain an "interlace" image. It suffices for this to place the 2nd HSYNC which occurs after the VSYNC CRTC at half the distance from the line compared to that which usually takes place (i.e. a delay of R0/2 µsec). However, we must not forget to compensate the deficit thus created for the CTM. It is therefore possible to display an "interlace" image quite simply.

The beam rises diagonally to reach the horizontal position left at the bottom of the screen. Depending on the horizontal position of the beam when the request is made, the latter zigzags to reach the horizontal position it has just left.
The following diagrams show the different situations that may arise.

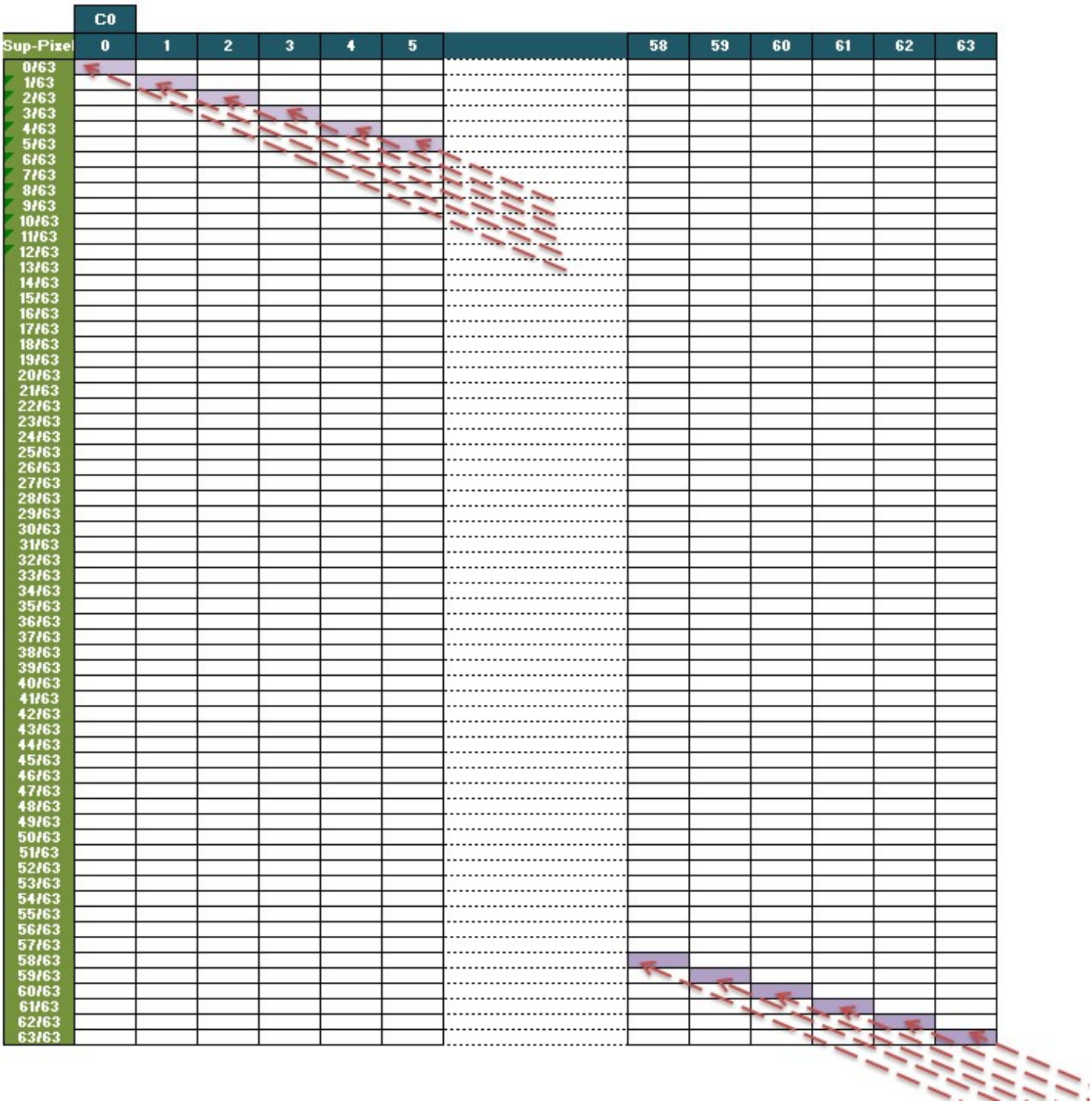


The later the beam starts, the higher the beam rises.

Thus, if the beam starts to rise from the middle of a line, it will rise:

- Higher by ½ pixel than if the beam had gone up from position C0=0.
- Lower by ½ pixel than if the beam had risen from position C0=63.

One idea (not that far-fetched, Cheshirecat would say) is to trigger the 2nd HSYNC-VSYNC on each of the 64 frame positions to achieve 1/64th pixel positioning accuracy.



This diagram shows the positioning of the beam at the vertical level of a pixel when it has arrived at the top of the screen, starting from a VSYNC activated on the different positions of C0 of a line.

The CTM is accurate enough to handle that. CPC magic! Boo to the C64 and Amiga!

The human eye, however, is not able to see movement at this level of precision.
Thus, the CPC on a CTM monitor can manage fluid vertical scrolling at 1/64th of a pixel (and all lower variations).

This is demonstrated in SHAKER from version 2.1.

It should be possible to further increase this precision to 1/128th pixel, with the help of another technique described in this document...

16.7 THE RIGHT MOMENT...

If R7 is updated without "precautions", a new VSYNC may occur during or after the current frame. In this situation, the monitor must handle multiple VSYNC or the absence of VSYNC after the update, and this results in frame breaks.

The monitor is simply trying to set its image to the new position of C4=R7. This operation can take place faster or slower depending on the "v-hold" setting of the monitor.

It is possible to avoid this synchronization stall of the monitor by acting so that C4 returns to the new value of R7 in the same place as the old value of C4=R7. This requires modifying R4 so that it goes back to 0 earlier if the new C4=R7 to be reached is higher than the old one, or conversely to enlarge R4 so that C4 goes back to 0 later if the new C4=R7 to be reached is lower than the old one. Just imagine that it is the counters which end up where they should be.

This little gymnastics of repositioning the counters prevents the monitor from losing the VSYNC, which can be annoying for a program that cannot suffer this type of visual artifact.

Otherwise, if your code has time, it is always possible to hide this stall by acting on R1 and / or R6, or even by putting all the black inks a "certain time", but there are some perverts who titillate the V-Hold of their monitor in order to track down the deviant’s insubordinate of the diktat of fundamentalist purists.

17 DISPLAY : REGISTER R1

17.1 GENERAL

The function of this register is to define the number of horizontal characters displayed on a line.
Its value is expressed by the number of CRTC characters (as a reminder, each CRTC character occupies 2 bytes in RAM).

It also plays an important role in memorizing the current video pointer.

In general, the CRTC generates R0+1 characters on a line, of which R1 characters will be displayed. When R1 Characters are displayed (DISPLAY ENABLE ON) then the display is inhibited, and the "Border" is displayed on CPC (DISPLAY ENABLE OFF) via the GATE ARRAY.

Note that the DISPLAY ENABLE Pins have various names, and we can find it under other exotic names such as DISPTMG or DE, depending on the different CRTC documentation.

To display all of the characters that the CRTC can generate for a line, R1 should in principle be R0+1. Indeed, the DISPLAY ENABLE "Flag" is set to OFF to generate border when C0=R1.

However, C0 can never exceed R0.
To display all programmed characters with R0, it is possible to prevent C0 from reaching R1, either by positioning R1> R0, or by changing the value of R1 during the line.

In this situation, a problem arises because the equality between C0 and R1 is used to update the video pointer when C9=R9 (last line of a "character").

This defect in updating therefore causes a repetition of the character lines, under conditions specific to each CRTC.

Note :
This problem, caused by the dynamic counter management is a defect because it does not allow the video counter to reach 128 bytes for 1 line (64 µsec), which can lead some unfortunate people to use 65 µsec frames by line by programming R0 at 64 with a single horizontal synchronization position for these 65 µSec lines ...
This allows for the addition of a hiss to the music played in the demos and to verify that the monitor is poorly adjusted.
Without artifice this does not allow the possibility of getting a video pointer where the more significant address byte does not vary during the display of a line. This "trick" saves CPU when it comes to displaying data, because you just increment an 8-bit register instead of 16-bit for the video pointer. Which is why, unfortunately, so many frames are formatted with lines 64 bytes wide in "minimized frame", 16 less than the standard width.

When C0 goes to 0 (following a condition C0=R0) then the display is authorized (DISPLAY ENABLE ON). Note that if C0 returns to 0 because it reached 255 having overflowed, this does not authorize the display (at least on the CRTC 0 but is yet to be verified on the other CRTC's).
Note also that **this condition is not managed by the CRTC 2 during the HSYNC**.

If R1 is zeroed then no more characters are displayed, regardless of the CRTC of a CPC.

The border begins on position C0=R1 (considering, for CRTC 0, that the SKEW DISP function is not used because this function modifies this rule).

Note :
This is not the case on the BBC with the Hitachi HD6845SP (type 0) either with Samsung KS68C45S or a VLSI VL68C45S23PC. (But why am I talking about this in a document dedicated to the CPC, me?). (Why not the PD7220 NEC as long as I'm here?)

In the diagrams below, and in some introductory words, I refer to two memory pointers in the CRTC, which I named **VMA** and **VMA'**.

When the CRTC displays characters, it always uses **VMA** pointer.

This pointer is incremented each time a character is treated by the CRTC, whether displayed or not. In some "pre-release" versions of the CRTC, this pointer was not managed during the VSYNC, but that does not concern the CRTC's of the CPCs to my knowledge.

When C0=R1 and C9=R9, then the **VMA** current pointer is transferred to the **VMA'** pointer.

When C0=0, at the start of the line, the **VMA'** line pointer is transferred to the **VMA** current pointer, except for the first character according to the CRTC type (C4=0).

17.2 DISPLAYS ACCORDING TO R1

17.2.1 DISPLAY WITH R1 <= R0

The following diagram describes R1 management in "standard" programming of the CRTC on a CPC which has just been switched on with a standard ROM BASIC.
Initial data: CRTC-R0=63 / **CRTC-R1=40 / CRTC-R9=7 /** CRTC-R12=0 / CRTC-R13=0

C0=0	**C0=R1 & C9=R9**	**VRAM-C9-Bit 0..2**	
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7

C0:	0	1	2	3	...	37	38	39	**40**	41	42	43	44	45	...	63
	0	1	2	3	...	37	38	39	DISP-OFF	**BORDER**						
	0	1	2	3	...	37	38	39	DISP-OFF							
	0	1	2	3	...	37	38	39	DISP-OFF							
	0	1	2	3	...	37	38	39	DISP-OFF							
	0	1	2	3	...	37	38	39	DISP-OFF							
	0	1	2	3	...	37	38	39	DISP-OFF							
	0	1	2	3	...	37	38	39	DISP-OFF							
	0	1	2	3	...	37	38	39	DISP-OFF							
	40	41	42	43	...	77	78	79	DISP-OFF							
	40	41	42	43	...	77	78	79	DISP-OFF							
	40	41	42	43	...	77	78	79	DISP-OFF							
	40	41	42	43	...	77	78	79	DISP-OFF							
	40	41	42	43	...	77	78	79	DISP-OFF							
	40	41	42	43	...	77	78	79	DISP-OFF							
	40	41	42	43	...	77	78	79	DISP-OFF							
	40	41	42	43	...	77	78	79	DISP-OFF							

17.2.2 DISPLAY WITH R1 > R0

The following diagram describes the display when R1 is programmed with a value greater than 63.
Initial data: CRTC-R0=63 / CRTC-R1=**64 /** CRTC-R9=7 / CRTC-R12=0 / CRTC-R13=0

C0=0	**C0=R1 & C9=R9**	**VRAM-C9-Bit 0..2**	
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	0
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	1
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	2
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	3
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	4
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	5
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	6
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	7
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	0
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	1
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	2
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	3
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	4
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	5
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	6
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9:	7

																	R0	**R1**
C0:	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		
	0	1	2	3	…	53	54	55	56	57	58	59	60	61	62	63		

If R1 > R0, then C0 is never equal to R1 (when C9=R9), and **VMA'** is therefore not updated with **VMA**.

This causes a character repetition because the current address is not updated during the change of one "line-character" (when C9=R9).

Note 1 :
Only the first 10 bits are not updated in this context.
The bits that determine the block number (Character line) (C9 or C5) continue to "participate" in the address. See Chapter 20, page 234.

Note 2 :
Insofar as the conditions allow the address update, modifying R12/R13 allows for this repetition to be avoided. In practice, this allows you to put the "lines" when C0 returns to 0 several times during a "line". However, CRTC's 0 and 2 generate a border byte, knowing that CRTC 0 can still prevent the generation of this byte by consuming CPU. See Chapter 19.2, page 187.

Note 3 :
According to the CRTC, the initial update of CRTC-VMA'/CRTC-VMA via R12/R13 is not the same.
See Chapter 20.3, page 235 .

17.3 DYNAMIC R1 UPDATE

The condition C0=R1 is considered immediately on a line.
It can occur several times on the same line if R1 is reprogrammed.

When the first condition C0=R1 is true, there is no longer any display of characters, and the BORDER is displayed.

However, even if only the BORDER is displayed, the **VMA pointer continues to count** (the CRTC being only a large field of flowering counters,...).

If R1 is updated again during the line to meet the condition C0=R1 when C9=R9 then **this will cause an update of the video pointer**.

In other words, the modification of R1 during the BORDER R1 display allows for the video pointer to be updated **without the data being displayed**.

The diagrams below show these behaviours when C4 > 0.

C0=0	C0=R1 & C9=R9	VRAM-C9-Bit 0..2		Upd C0:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0	R1=26	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	DISP-OFF					
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1	R1=25	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	DISP-OFF						
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2	R1=24	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	DISP-OFF							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3	R1=23	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	DISP-OFF								
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4	R1=22	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	DISP-OFF									
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5	R1=21	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	DISP-OFF										
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6	R1=20	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	DISP-OFF											
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7	R1=19	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	DISP-OFF												
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0	R1=18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	DISP-OFF													
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1	R1=17	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	DISP-OFF														
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2	R1=16	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	DISP-OFF															
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3	R1=15	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	DISP-OFF																
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4	R1=14	19	20	21	22	23	24	25	26	27	28	29	30	31	32	DISP-OFF																	
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5	R1=13	19	20	21	22	23	24	25	26	27	28	29	30	31	DISP-OFF																		
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6	R1=12	19	20	21	22	23	24	25	26	27	28	29	30	DISP-OFF																			
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7	R1=11	19	20	21	22	23	24	25	26	27	28	29	DISP-OFF																				
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0	R1=12	30	31	32	33	34	35	36	37	38	39	40	41	DISP-OFF																			

R0

C0=0	C0=R1 & C9=R9	VRAM-C9-Bit 0..2		Upd C0:	0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	**63**
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0	R1=64	0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1		0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2		0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3		0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4		0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5		0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6		0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7	R1=40	0	1	2	3	...	37	38	39	DISP-OFF																									
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0		40	41	42	43	...	77	78	79	DISP-OFF																									
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1		40	41	42	43	...	77	78	79	DISP-OFF																									
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2		40	41	42	43	...	77	78	79	DISP-OFF																									
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3		40	41	42	43	...	77	78	79	DISP-OFF																									
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4		40	41	42	43	...	77	78	79	DISP-OFF																									
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5		40	41	42	43	...	77	78	79	DISP-OFF																									
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6		40	41	42	43	...	77	78	79	DISP-OFF																									
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7		40	41	42	43	...	77	78	79	DISP-OFF																									

C0=0	C0=R1 & C9=R9	VRAM-C9-Bit 0..2		Upd C0:	R1																																R0
					0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	**63**
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0	R1=40	0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	7	R1=64	0	1	2	3	...	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6		0	1	2	3	...	37	38	39	DISP-OFF																							
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7		0	1	2	3	...	37	38	39	DISP-OFF																							

C0=0	C0=R1 & C9=R9	VRAM-C9-Bit 0..2		Upd C0:	R1																																R0
					0	1	2	...	32	33	34	35	36	37	38	39	40	41	42	43	44	45	...	51	52	53	54	55	56	57	58	59	60	61	62	**63**
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0	R1=40	0	1	2	...	32	33	34	35	36	37	38	39	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1		0	1	2	...	32	33	34	35	36	37	38	39	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2		0	1	2	...	32	33	34	35	36	37	38	39	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3		0	1	2	...	32	33	34	35	36	37	38	39	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4		0	1	2	...	32	33	34	35	36	37	38	39	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5		0	1	2	...	32	33	34	35	36	37	38	39	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6		0	1	2	...	32	33	34	35	36	37	38	39	DISP-OFF																			
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7		0	1	2	...	32	33	34	35	36	37	38	39	DISP-OFF																	**OUT R1,0**		
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0		DISP-OFF			**CRTC-VMA +++++**					**OUT R1,40**				**CRTC-VMA +++++**																**OUT R1,0**		
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1		DISP-OFF			**CRTC-VMA +++++**					**OUT R1,40**				**CRTC-VMA +++++**																**OUT R1,0**		
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2		DISP-OFF			**CRTC-VMA +++++**					**OUT R1,40**				**CRTC-VMA +++++**																**OUT R1,0**		
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3		DISP-OFF			**CRTC-VMA +++++**					**OUT R1,40**				**CRTC-VMA +++++**																**OUT R1,0**		
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4		DISP-OFF			**CRTC-VMA +++++**					**OUT R1,40**				**CRTC-VMA +++++**																**OUT R1,0**		
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5		DISP-OFF			**CRTC-VMA +++++**					**OUT R1,40**				**CRTC-VMA +++++**																**OUT R1,0**		
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6		DISP-OFF			**CRTC-VMA +++++**					**OUT R1,40**				**CRTC-VMA +++++**																**OUT R1,0**		
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7		DISP-OFF			**CRTC-VMA +++++**					**OUT R1,40**				**CRTC-VMA +++++**																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	0		80	81	82	...	112	113	114	##	##	##	##	##	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	1		80	81	82	...	112	113	114	##	##	##	##	##	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	2		80	81	82	...	112	113	114	##	##	##	##	##	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	3		80	81	82	...	112	113	114	##	##	##	##	##	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	4		80	81	82	...	112	113	114	##	##	##	##	##	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	5		80	81	82	...	112	113	114	##	##	##	##	##	DISP-OFF																			
CRTC-VMA=CRTC-VMA'		CRTC-VMA C9 :	6		80	81	82	...	112	113	114	##	##	##	##	##	DISP-OFF																			
CRTC-VMA=CRTC-VMA'	**CRTC-VMA'=CRTC-VMA**	CRTC-VMA C9 :	7		80	81	82	...	112	113	114	##	##	##	##	##	DISP-OFF																			

17.4 VMA'/VMA WHEN C4=0

There are differences between CRTC's for the assignment of the video pointer with the values programmed in R12 and/or R13 when C4=0 and C0=0.

17.4.1 CRTC 0, 3, 4

The first line character begins with the address defined by R12/R13, whatever the value of R1.
On the first line of the first character of a "frame" (C4=C9=C0=0), **the VMA' pointer is updated using the content of R12/R13.**
This update is followed by the **VMA pointer update with VMA'**.

If R1>R0, the VMA' pointer is no longer updated, and the lines are repeated.

In this circumstance, when a new frame begins, **VMA' is updated and all the lines displayed become identical and equal to pointer R12/R13**.

17.4.2 CRTC 1

The first line character begins with the address defined by R12/R13, whatever the value of R1.
When you are on the first character of a "frame" (C4=C0=0), **the VMA pointer** (and not VMA' as on CRTC 0) is updated **using the content of R12/R13.**

Note: This update particularity allows for the modification of the offset via R12 and/or R13 on each line (C9 from 0 to R9) of a character while C4=0.

Note: If C4 is 0 and additional row management begins (because R4=0 and R5>0), then C4 will be 1 for the first additional rows (R9 or R5 raws (if R5<R9+1)). It is possible to modify the offset (R12/R13) on each line of this C4, but not of the following ones if R5>R9+1 and C4 exceeds 1.

This VMA update, however, has a consequence when R1> R0 throughout the frame.

Indeed, the condition C0 = R1 no longer occurs and the **VMA' pointer is no longer updated**.
VMA' is "frozen" on the last known pointer when C0 reached R1 when C9=R9.
When C9 = 0, the VMA pointer is reloaded with VMA' when C4> 0.

We have therefore, when R1> R0, a first line character which contains the pointer defined in R12/R13 and on the following, the last pointer updated in VMA'.

The video pointer continues to be increased however, even when BORDER is displayed.

This is true both for the horizontal level (R1 management) and the vertical (R6 management).

The VMA' pointer is 14 bits. When it continues to increment and exceeds the 10-bits definition limit, it can modify the Overscan Bits ™ (See Chapter 20.5, page 237) and cause video page changes.

When R1 becomes higher than R0, it should be noted, however, that if the R1 update takes place exactly when C0=R1 (R1.JIT), then the condition C0=R1 is no longer true and VMA' is not updated. If the modification of R1 occurs when C0=R1+1 (C0>R1), then the condition C0=R1 is satisfied and VMA' is updated.

Example :

If the display width of a line is 40 characters (&28), the CRTC will be able to display 1024/40 different character lines, i.e. 25 complete lines.

When C4=R6=25, the CRTC stops displaying the character lines, but it nevertheless continues to increment its pointer and to manage the updates of VMA' when C0=R1 & C9=R9.

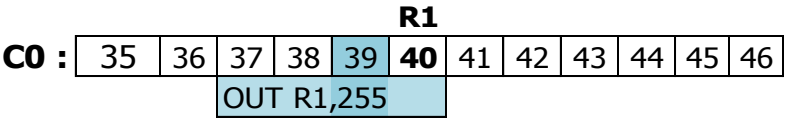
If the modification of R1 (>R0) occurs during this period, the video pointer will have overflowed.

If R1 is modified after C4=24, C9=R9 and C0>R1, then the pointer will be equal to 40 x 25=1000 (the repeated line will contain the characters 1000 to 1023, followed by 0000 to 0015).

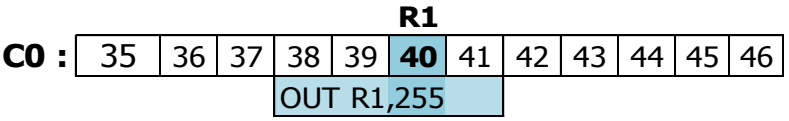
If R1 is modified after C4=25, C9=R9 and C0>R1, then the pointer will be equal to 40 x 26=(1040 and 1023)=16 (the repeated line will contain the characters 0016 to 0055).

VMA' pointer repeated = 960 :

C4=24 / C9=R9
CRTC-VMA'=40x24=960

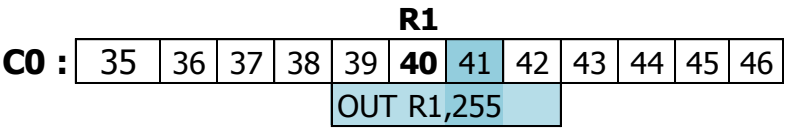


C4=24 / C9=R9
CRTC-VMA'=40x24=960



VMA' pointer repeated = 1000 (40 x 25) :
Condition C0=R1 occurred, pointer VMA' was loaded with VMA.

C4=24 / C9=R9
CRTC-VMA'=40x24=960



17.4.3 CRTC 2

On the first line, when C0=0, VMA is affected by VMA'.
VMA' itself is affected by R12/R13 when C0 reach R1 on the last frame line.

There are nevertheless some important particularities linked to this logic:
The frame's last-line state evaluation takes place when C0=0 but it occurs **after** processing the C0=R1 evaluation when R1=0. The last line status determines whether VMA' will be assigned with VMA (false status) or R12/R13 (true status). This processing priority prevents the offset from changing at the start of the last line of the frame. When the new frame starts, if R1 is still 0, the evaluation C0=R1 takes place while the last line state is still true (since it changes after the processing of R1). VMA' then becomes equal to R12/R13, then the last line state is evaluated.

The VMA'=VMA or R12/R13 assignment (depending on status) therefore takes place before VMA=VMA' assignment on position C0=0.

If R12 and/or R13 are modified after C0=0 of the last line of a frame when R1=0, they will be considered on the new frame... But not completely.

There is indeed a bug when updating VMA' with R12/R13 at position C0=0.
To update VMA' with R12/R13, the CRTC in principle carries out two logical operations:
VMA'=VMA' **AND** (R12 x 256 + R13)
VMA'=VMA' **OR** (R12 x 256 + R13)

In the situation described above, **only the first logical operation (AND) is performed**.
All bits that are 1 in R12/R13 "keep" the corresponding bits of VMA'.
And all the bits that are 0 in R12/R13 set the corresponding bits of VMA' to 0.

Conversely, if R1 is modified when C0>0 on the last line of a frame when it was 0, and the new value no longer satisfies the condition C0=R1 until the last line state becomes false again (on position C0=0 of the first line of the new frame, after the evaluation C0=R1), then the offset is not considered. Indeed VMA' will not be affected with R12/R13 but with the value of VMA' (which contained the value that VMA had on position C0=0 of the last line of the previous frame).

In other words, the first line of the new frame is identical to the last line of the previous frame.

When R1>R0, neither pointer is updated with R12/R13 anymore.
All lines are identical in this situation.

The replicated address follows the same logic as described for CRTC 1.

So, depending on when R1 becomes greater than R0, it's the pointer when the BORDER R1 is handled which is considered.

However, there is a difference with the CRTC 1 when modifying R1 on the exact position of C0=R1. CRTC 1 is faster than CRTC 2 in considering the update of R1 when C0=R1. On CRTC 2, an update of R1 on position C0=R1 arrives too late. VMA' update has already taken place using the old value of R1.

VMA' pointer repeated = 960 :

C4=24 / C9=R9
CRTC-VMA'=40x24=960

						R1						
C0 :	35	36	37	38	39	**40**	41	42	43	44	45	46
			OUT R1,255									

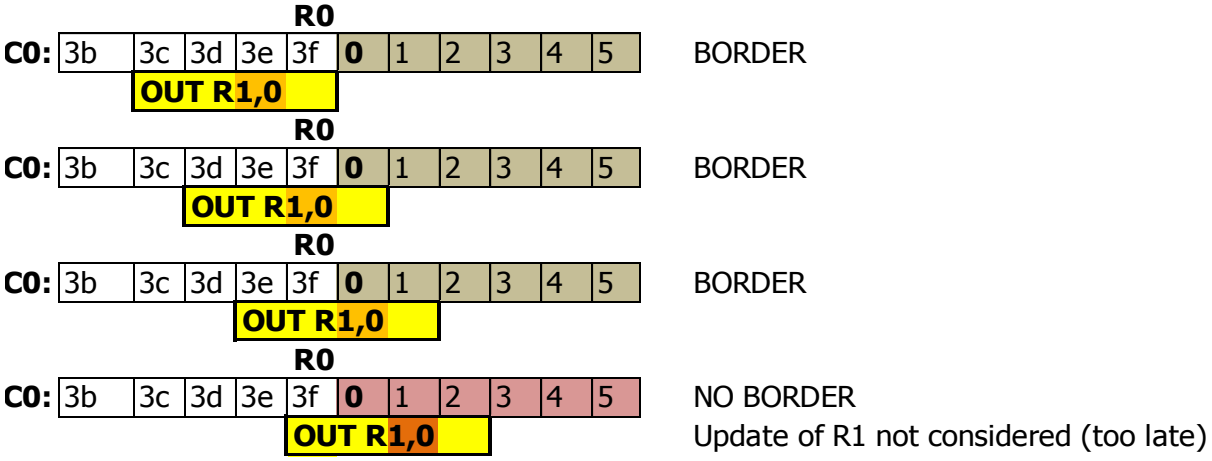
VMA' pointer repeated = 1000 (40 x 25) :
Condition C0=R1 occurred, pointer CRTC-VMA' was loaded with CRTC-VMA.

C4=24 / C9=R9
CRTC-VMA'=40x24=960

						R1						
C0 :	35	36	37	38	39	**40**	41	42	43	44	45	46
				OUT R1,255								

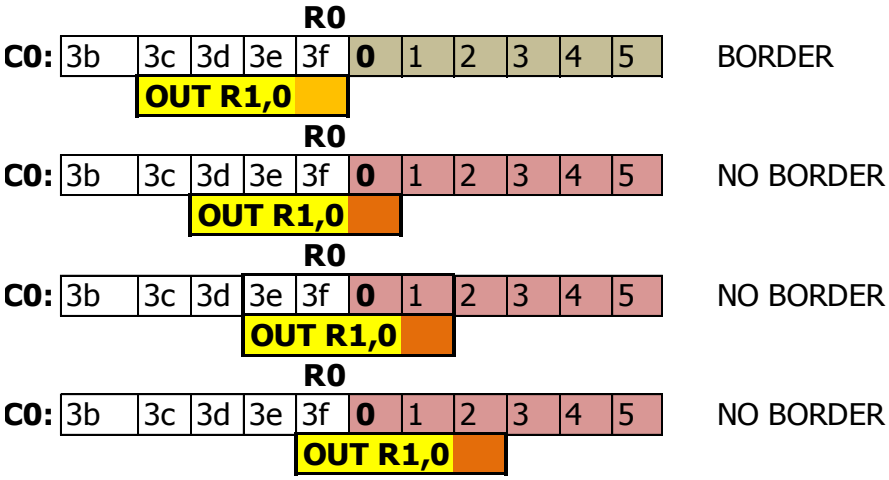
17.5 ACKNOWLEDGMENT R1=0

17.5.1 CRTC 0, 1, 2



Update of R1 not considered (too late)
Update of R1 ok (just in time)

17.5.2 CRTC 3, 4



Update of R1 not considered (too late)
Update of R1 ok (just in time)

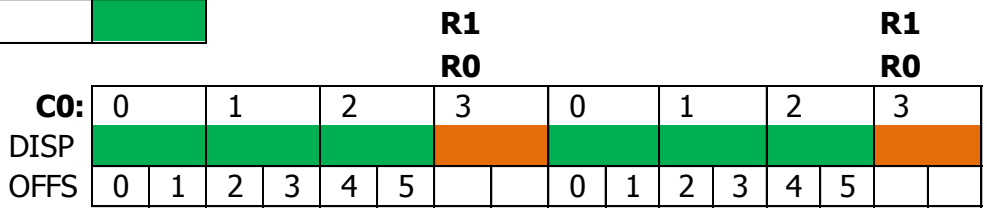
17.6 INTERLINE BORDER

17.6.1 R1=R0 AND C0=R0

CRTC 0, 1, 2, 3, 4

If R1=R0, all CRTC's will display 1µs of BORDER on the last character (C0=R0).
CRTC-R0=3
CRTC-R1=3
CRTC-R12/R13=0

DISP OFF (BORDER)
DISP ON



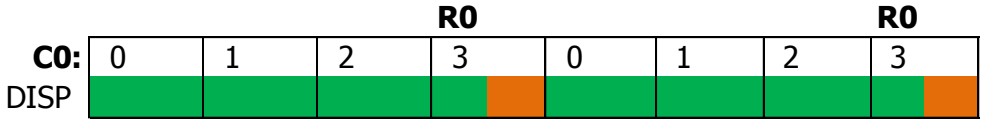
Note : The pointer in VRAM will continue at offset 6 on the next "character line" (when C9 returns to 0)

17.6.2 R1>R0 AND C0=R0

CRTC 0, 2

The consideration of the BORDER is anticipated.
1 byte of BORDER is generated (for exactly 0.5 µsec) before C0 goes to 0.
CRTC-R0=3
CRTC-R1=4
CRTC-R12/R13=0

DISP OFF (BORDER)
DISP ON



Note 1 : This behaviour remains true whatever the value of R0.
If R0=0, then the display alternates between 1 DISP ON byte, and 1 DISP OFF byte.
When C0 reaches R0 without R1 having been reached, CRTC's 0 and 2 send a "BORDER ON" signal to the GATE ARRAY, which picks it up immediately. These CRTC's are "ahead" of the characters displayed by the GATE ARRAY and send the BORDER signal 0.5 µsec too early. The "BORDER OFF" signal is sent on the character following C0=R0.

Note 2 : The VRAM pointer will get "stuck" on VMA'. If the conditions allow it (according to the CRTC and the counters values), VMA' can be reloaded by updating R12 and/or R13.

Note 3 : The CRTC 0 has a function to generate conditions for later consideration of the BORDER via its R8 register. By playing on the modification of the conditions at the time of their application, it is possible to avoid the generation of this BORDER byte. See Chapter 19.2, page 187.

Note 4 : CRTC 2 does not have the SKEW DISP (R8) function and the BORDER byte cannot be "cancelled" this way. But does LOGON really have to reveal all its secrets?

CRTC 1, 3, 4

CRTC-R0=3
CRTC-R1=4

On these CRTC's, no BORDER bytes are generated between the "frames".
Frames can therefore be created in the displayable area without a byte appearing between the frames. This allows for the possibility of carrying out aggressive plagiarisms…

DISP OFF (BORDER)	
DISP ON	

				R0				**R0**
C0:	0	1	2	3	0	1	2	3
DISP								

18 DISPLAY : REGISTER R6

18.1 GENERAL

The function of this register is to set the number of character lines displayed vertically.
When this number is reached, the BORDER is displayed.

The BORDER is displayed when C4=R6 (1st line-character R6).
This rule is true whatever the value of C9.

Except on CRTC 3 and 4, R6 is considered immediately on the current C0.

The DISPLAY ENABLE pin of the CRTC goes to "OFF" when C4=R6, in the same way as for the R1 register when C0=R1.

In general when the condition is fulfilled to display the BORDER, it is necessary to wait for a new frame (C4=C9=C0=0) to restore the displaying of characters.

With register R1, the BORDER is disabled when C0=0 and enabled when C0=R1.

Whether the border is generated via R1 or R6, the pointer in VRAM continues to be updated.
In this case, if the BORDER was activated by the condition on R6, the pointer VMA' continues to be updated.

18.2 BORDER R6 DEADLINES AND PRIORITIES

18.2.1 GENERAL

The state of the DISPLAY ENABLE pin depends on 2 groups of internal CRTC conditions.

As long as the "R6 conditions" are not satisfied, the "R1 conditions" define the state of the DISPLAY ENABLE pin.

When the "R6 conditions" are satisfied, the "R1 conditions" are no longer considered.
The common condition for restoring the bottom display is C4=C9=C0=0.

In general, the BORDER R6 condition takes precedence over the BORDER R1 condition, however there are some differences between CRTC's.

18.2.2 CRTC 0, 2

Except for the first line of a frame (C4=C9=0), positioning R6 with C4 causes the immediate and definitive activation of the BORDER until the next frame.

The condition C4=R6 is considered immediately (regardless of the value of C0), and BORDER R6 has priority over BORDER R1.

This is also true during the C4 character(s) generated during a vertical adjustment.

The value R6=0 used on the first line (C4=C9=0) is treated specifically (See Chapter R6 CONFLICTS) and it is possible, under conditions, to use this conflict to **cancel BORDER R6**.
See Chapter 18.3.

18.2.3 CRTC 1

Positioning R6 with C4 causes the immediate activation of the BORDER, which becomes final until the next frame.

The condition C4=R6 is considered immediately (regardless of the value of C0), and BORDER R6 has priority over BORDER R1.

As with other registers of this CRTC (for example R3) the value 0 is specifically considered and triggers a BORDER without the condition C4=R6 being required.

However, if the condition C4=R6 is also fulfilled when C4=0, then this BORDER becomes definitive until C4=C9=C0=0.

However, there is an exception to this rule: if the condition C4=R6=0 is true on the last character of the frame, then the border is not irreversible. An update of R6<>C4 allows to cancel it. Indeed, if the condition C4=R6=0 has already been evaluated on the previous frame, **it is no longer re-evaluated on C4=R6=0 of the new frame**. Switching to the new frame has however re-authorized the background and the border is then only displayed because of R6=0 (and no longer because of the equivalence C4=R6). Note that the update of R6=0 on the previous frame is considered soon as C0=R0 which precedes the new frame.

In the other cases, when R6 is set to 0 while C4<>0, the BORDER is activated while R6=0, and it is deactivated as soon as R6>0 and its new value is different from C4 (in which case, the BORDER R6 is activated for the frame)

18.2.4 CRTC 3, 4

The R6 test is done at the beginning of the line only.

The update of R6 **during the line is therefore not considered**.
If R6 is set to 0 when C4=0 but C0>0, the BORDER is not activated.

The BORDER is activated only when C0 goes to 0 when C4=R6.

This is also true during a vertical adjustment.

Note: C4 is not incremented on these 2 CRTC's during vertical adjustment. This implies that if R6=R4, then the BORDER will concern the last character as well as the vertical adjustment lines defined with R5.

The value of R6=0 is not specifically processed and does not temporarily activate the BORDER as on CRTC 1.

18.3 R6 CONFLICTS

18.3.1 GENERAL

In order to correctly manage some conflicts, special processing is often carried out when a register has the value 0 (particularly on CRTC 1, the domain specialist).

This is particularly the case when R1=0, because this value creates a potential conflict: deactivation of the BORDER (C0=0 following condition C0=R0), activation of the BORDER (C0=R1=0))
[In this situation, it is the deactivation of the BORDER which is activated]

For R6, the situation is much less well managed, according to the CRTC's.

18.3.2 CRTC 0, 2

Except for the first line of a frame (C4=C9=0), positioning R6 with C4 causes the immediate and definitive activation of the BORDER until the next frame.

When C4=R6=0 and C9=0 on CRTC's 0 and 2, a conflict occurs (this conflict does not exist when R6>0).

When R6 is equal to 0 in this situation, the state of DISPLAY ENABLE changes to ON at the beginning of the CRTC character and returns to OFF 0.5 µsec later.

In other words, on the first line of the frame, there is an alternation of bytes of BORDER and displayable characters (the video pointer continuing to count normally).

At each CRTC character:

- BORDER R6 passes to true because C4=R6 and C9=0
- BORDER R6 returns to false because for a new frame (C4=C9=0)

Note :
This alternation only takes place when the condition R1 is fulfilled (BORDER R1 is false) and the conditions of the conflict exist (C4=R6=C9=0).

When the conflict is cancelled (R6 updated with a value > 0), the BORDER does not remain activated as on the other lines (and the other CRTC's).

In this situation however, since R6 has been updated with 0 at least 1 time, the BORDER becomes final when C0=R1.

Still in this situation, if we prevent C0=R1 on the line C4=C9=0 (for example R1=R0+1), and R6 is no longer equal to 0, then the BORDER is deactivated on the following line.

In other words, **the condition C4=R6=0=BORDER is cancellable on the first line**.

Note :
Considering for each value of C0 the value of R6 makes it possible to precisely target areas or create this alternation BORDER/CHARACTERS by byte.

By means of a nice line-to-line rupture, this allows you to add nice colours from time to time, via BORDER, to graphic mode 2, for example, which will be called **IIMPSIIA** mode for the most modest, otherwise directly called Mode <write your nickname here>.

Note that it is possible to achieve a BORDER/CHARACTER alternation on CRTC 0 with R0=0, but without the counters C4 and C9 being able to count anymore.

18.3.3 CRTC 1

When R6 is updated with 0, the BORDER is activated as long as the register value is 0.

The consideration of each value of C0 of the value of R6 makes it possible to precisely target zones or create this BORDER.

However, if C4=R6=0 (1st line-character of a new frame) during this update, BORDER R6 becomes true first for all the rest of the frame, until the new frame (C4=C9=C0 =0).

18.3.4 CRTC 3, 4

No conflict exists since the management of R6=0 does not exist during the line and is tested only once.

Setting R6 to 0 when C4 and C9 are 0, but C0>0 will have no effect before the new frame (or the BORDER will be activated).

19 DISPLAY : REGISTER R8

19.1 GENERAL

The R8 register contains parameters for the management of the INTERLACE mode for all the CRTC's.

On CRTC's 0, 3 and 4, an additional function exists, which allows for the delaying of the management of activation/deactivation of the BORDER, or to deactivate the display as R6=0 does on CRTC 1 (except C4=0 on CRTC 1)).

CRTC	7	6	5	4	3	2	1	0
0	Sc	Sc	Sd	Sd	x	x	i	i
1	x	x	x	x	x	x	i	i
2	x	x	x	x	x	x	i	i
3	x	x	Sd	Sd	x	x	i	i
4	x	x	Sd	Sd	x	x	i	i

Other CRTC	7	6	5	4	3	2	1	0
MC6845*1	Sc	Sc	Sd	Sd	x	x	i	i
UM6845E	UpdM	US	Sc	Sd	Vdrac	Vdrad	i	i

Interlace		
0	0	No interlace
0	1	Interlace Sync Mode
1	0	No interlace
1	1	Interlace Sync & Video Mode

Skew DISPTMG		
0	0	Non Skew
0	1	One-character skew
1	0	Two-character skew
1	1	Non-output

Skew CUDISP		
0	0	Non Skew
0	1	One-character skew
1	0	Two-character skew
1	1	Non-output

19.2 FUNCTIONS « SKEW-DISPTMG »

These functions are only available on CRTC's 0, 3 and 4.
They allow for the activation of the BORDER or to generate a delay on the BORDER R1 management.

Note: If the BORDER ON function is activated, the INTERLACE function on the 2 least significant bits is not considered. (This point requires further investigation).

19.2.1 BORDER ON

This function is activated with **00110000** on R8.
It allows you to deactivate the display of characters.
The GATE ARRAY generates BORDER in this situation.
The update of the DISPEN signal (DISPLAY OFF) is immediately considered.
The VRAM pointer nevertheless continues to be incremented and the current pointer is updated when C0=R1 and C9=C0=0.
This function does not affect the BORDER R6 state (when C4=R6) and it is therefore possible to switch to one of the 3 other available states.

Note: On CRTC 1, this direct assignment of DISPEN is possible by setting R6 to 0.
However, if C4=0, then the condition C4=R6 is fulfilled, and this causes the end of display until C4=C9=C0=0. This problem does not exist with the BORDER ON function.

19.2.2 BORDER OFF

This function is activated with **00000000** on R8.
It indicates to stop managing the other "SKEW" functions.

To be tested: Set the 001100xx function, then wait for C4=R6 to activate the BORDER, and then set R8 to 000000xx and see if this can cancel a BORDER R6.

19.2.3 BORDER DELAY +1 / +2

These functions are activated with **00010000** & **00100000**
They provide a mechanism to delay the BORDER R1 management.

Without this function being activated, we have:

- The BORDER deactivation condition (beginning of line) is performed on the character following C0=R0 (C0=0).
- The BORDER activation condition (end of line) is performed on character C0=R1.

The delay can be 1 or 2 CRTC characters (thus 1 or 2 µsec) depending on the function used.
The BORDER management conditions **remain the same** as those described above.
However, an acknowledgment counter from C0 is applied according to the function:

When the programmed "delay" is 1 µsec:

- The BORDER is deactivated on the 2nd character after that or C0=R0 (C0=1).
- The BORDER is activated on the 1st character after the one where C0=R1.
  Note: If R1=R0, then the BORDER is activated on C0=0.

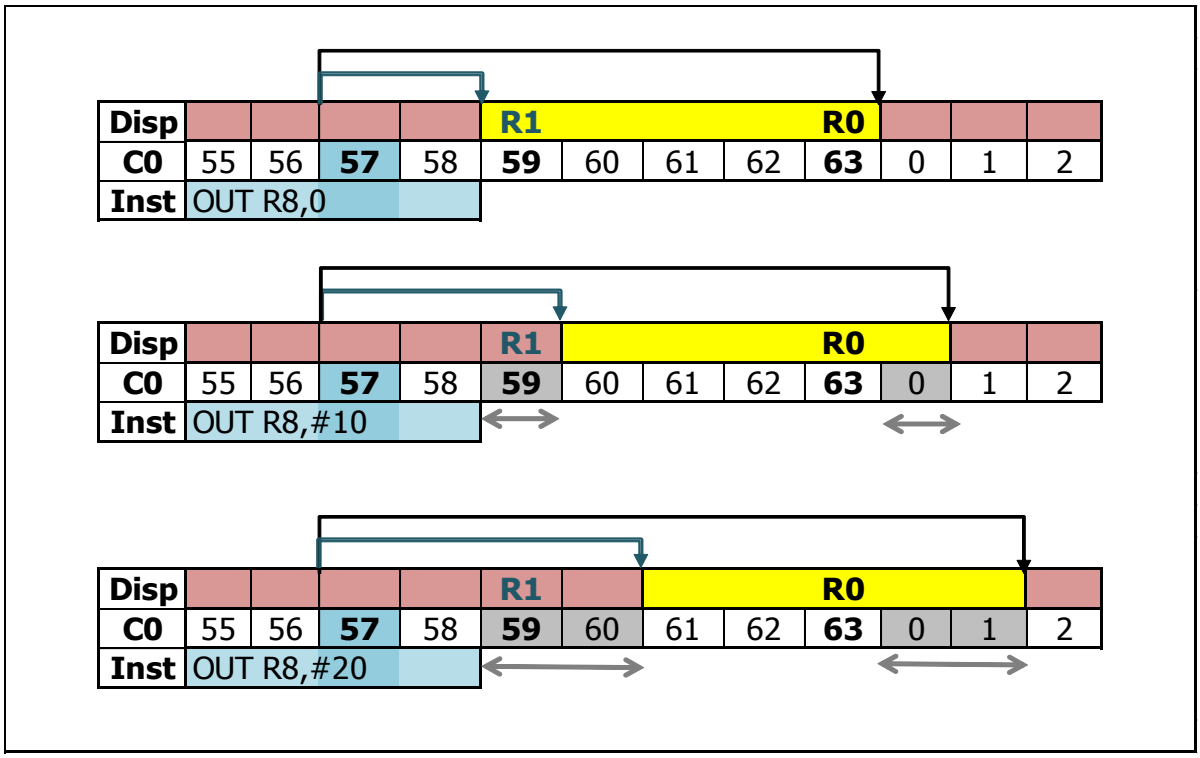
When the programmed "delay" is 2 µsec:

- The BORDER is deactivated on the 3rd character after that or C0=R0 (C0=2).
- The BORDER is activated on the 2nd character after the one where C0=R1.

Note: If R1=R0, then the BORDER is activated on C0=1.
The assignment of the video pointer, when C9=R9, follows the same logic when C0 reaches the relative value of C0 with respect to R1.
In other words, the video pointer is stored in relation to the new position of the BORDER.

The consideration of these new test "rules" when R8 is modified is immediate during the line.

The diagram below describes the position of the BORDER according to the programming of the registers, for a standard case. R1=59, R0=63, with a modification of R8 occurring elsewhere than on the corresponding positions of C0.



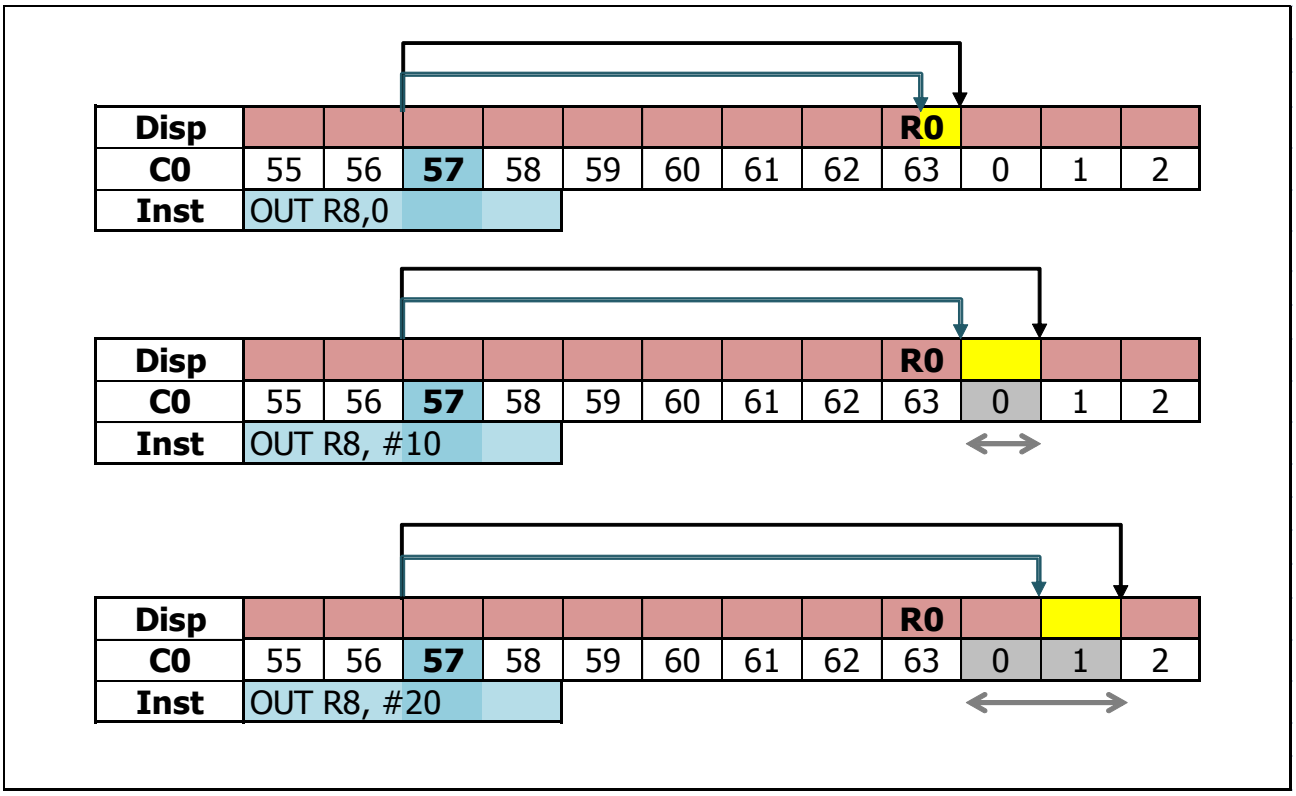
	Character Display
	Border Display
	Gap Skew ⟷
⟶	Border ON on C0=R1 applied
- - - - ⟶	Border ON on C0=R1 not applied
⟶	Border OFF on C0=R0 applied
- - - - - - ⟶	Border OFF on C0=R1 not applied

19.2.4 NO CONDITION C0=R1

In the chapter devoted to the R1 register (17.2) we saw that on CRTC's 0 and 2, a BORDER byte is generated between "2 lines" when R1 is greater than R0.

The condition C0=R1 not being met during the line, the BORDER signal is then sent 0.5 µsec after the condition C0=R0 is satisfied. The BORDER is disabled on the next character, 0.5 µsec later.
The condition C0=R0 therefore replaces the condition C0=R1.

Note: On CRTC's 1, 3 and 4, in the same context (R1>R0, with for example R0=63, R1=64), the condition C0=R1 is not met, and the CRTC does not send BORDER ON signal to GATE ARRAY.
When a delay is programmed using the SKEW DISP functions, the delay is counted based on the transitions of C0. In other words, the "deferred" BORDER will be displayed at the beginning of the character, as it would have been on the condition C0=R1.



19.2.5 DISINTEGRATION OF THE BORDER ON CRTC 0

There is a period of 1 or 2 µs, depending on the function used, during which it is possible to "cancel" the 2 conditions. The CRTC 0 has the possibility of removing this byte from BORDER by cancelling the 2 conditions. The CRTC 2 does not have the SKEWDISP functions and therefore cannot use this method to make this BORDER byte disappear between 2 lines.

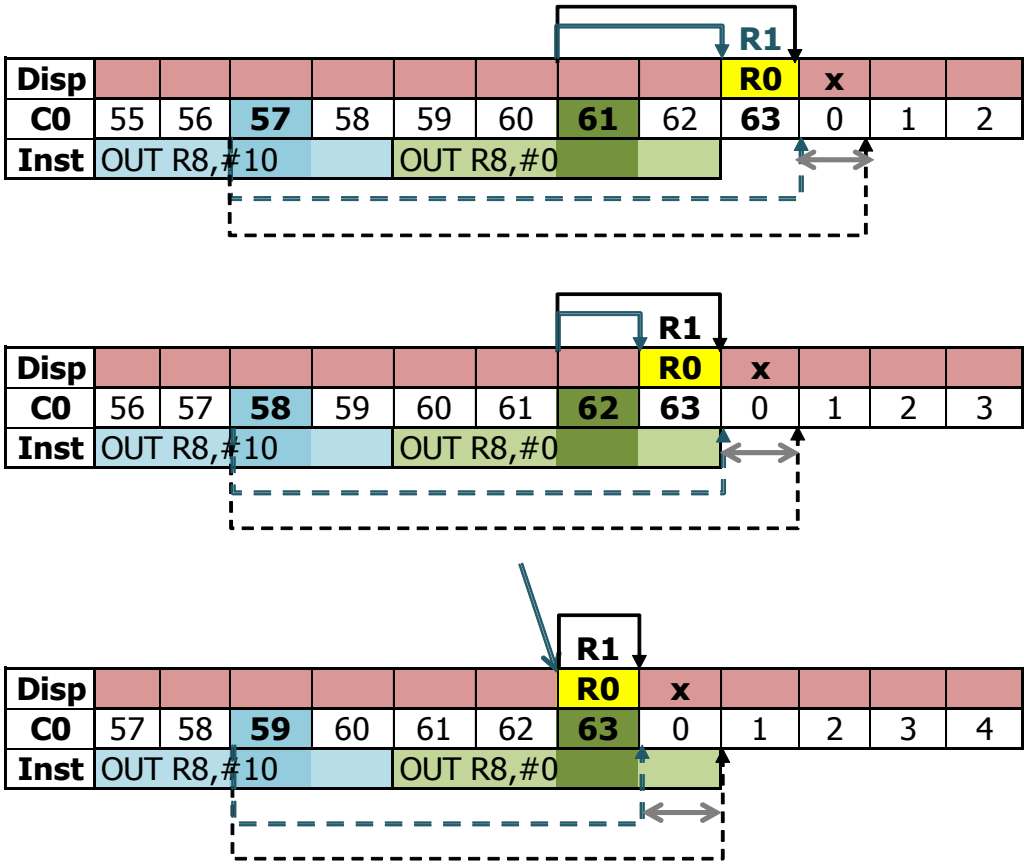
Note: This method can also be used when R1>R0. The condition C0=R1 is just replaced by the condition C0=R0 in this case.

For the different situations presented below, it is considered that:

- R0=R1=63.
- The programming of R8 is carried out on several lines. Therefore, the last value programmed on the line is the one present when the 1st OUT is performed on the following line.

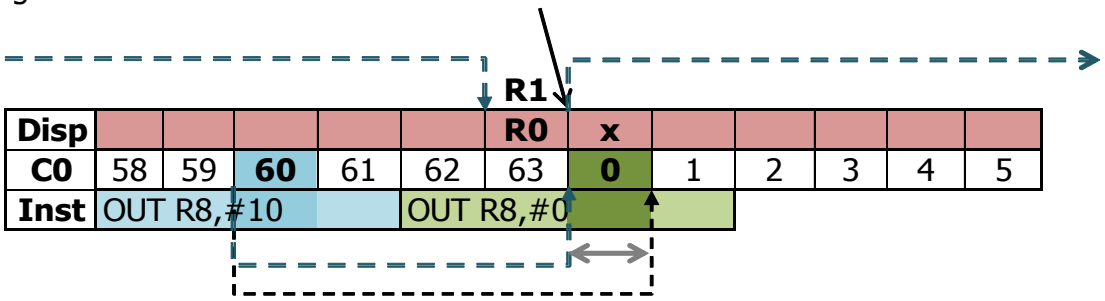
19.2.5.1SKEW DISP+1 is cancelled on C0<=63.

BORDER management is handled with the 2nd OUT.
The programming of the 1st OUT (R8=#10) is ignored



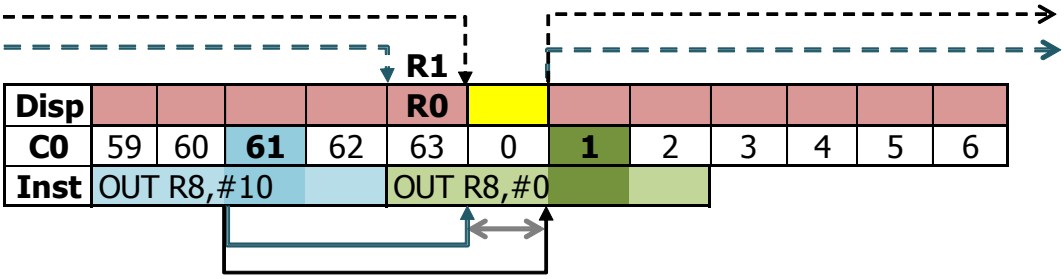
19.2.5.2SKEW DISP+1 is cancelled on C0=0 : Y GN'EST GNOU ?

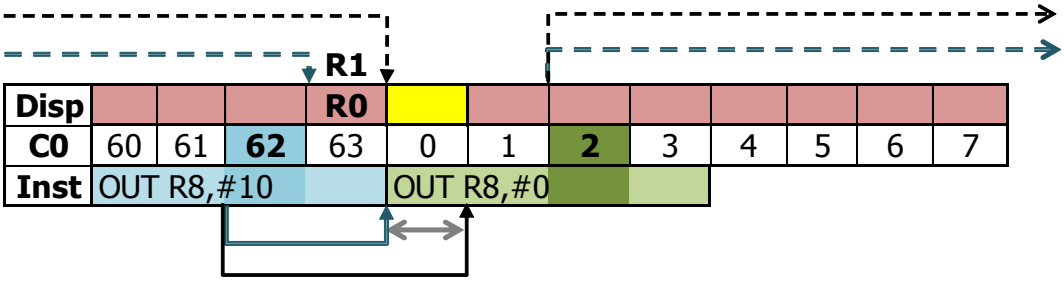
The first OUT (R8=#10) prevents the BORDER from being activated on C0=63 (Condition C0=R1 which reports the BORDER for C0=0). However, the 2nd OUT (R8=#00) on C0=0 defines a BORDER OFF on the next character C0=R0. It is considered immediately and cancels the C0=R1 condition programmed on the first OUT.



19.2.5.3SKEW DISP+1 is programmed on C0<63.

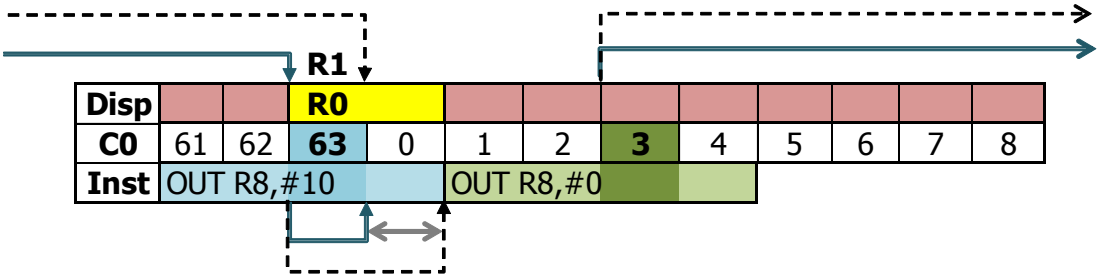
The second OUT (R8=#0) occurs too late to prevent the BORDER programmed with the first OUT from being activated on the following character C0=63 (Condition C0=R1 which reports the BORDER for C0=0).





19.2.5.4SKEW DISP+1 is programmed on C0=63 : COMBO BORDER !

Before being programmed with #10, R8 is equal to 0, and the BORDER was programmed for C0=R1 (i.e. C0=63). Programming R8 with #10 is not fast enough to prevent the BORDER from being activated. However, this programming is considered for the following character, which results in an additional BORDER byte



19.3 INTERLACE FUNCTIONS

19.3.1 GENERAL

The first thing that can be said is that the method of approach on this subject by the various CRTC manufacturers is quite... varied.

This first chapter concerns the principle of operation as it is described (very superficially) in the technical guides of the components. The chapters devoted to this function will be completed in more advanced versions of this document (having not yet finished the analysis of certain results for substantive reasons (ah ah)).

There are 2 programmable interlaced modes on all CRTC's.

The main objective of the interlaced mode is to create a "frame" composed of 2 frames at 50Hz, interlaced, in order to display an image with a doubled vertical resolution. A "complete" image is made up of 2 "frames" of 50 Hz in order to display a complete image at 25 Hz in 625 lines (30 Hz in 525 lines).

To proceed, the CRTC must:

- Be able to create an image composed of 625 distinct lines (on a 50Hz screen)
- Display two frames of 312 lines, plus an additional line.
  The length of a frame then being 19968 + 32 = 20000 µsec.

In order to increase the vertical resolution, the CRTC, during VSYNC, delays the signal by half a line when C4 changes to R7 for the frame displayed with the even lines. When C4 reaches R7, the VSYNC signal from the CRTC is generated taking C0=R0/2 as the new reference

If the CRTC communicated directly with the CTM screen, **which is not the case on CPC**, this would have the effect of raising the electron gun half a line higher (-0.5). But an additional line is added at the end of the first frame, which cause to moving the start of the odd frame down one line (-0.5+1=0.5) thus allowing the lines to be ordered.

The ability to handle an interlace image depends heavily on the monitor:

- On the one hand on the persistence capacity of phosphors, to limit flickering (which some would call "flashouille" in certain cold and humid French regions).
- On the other hand, on the stability of the high voltage elements of the monitor to limit distortions on the edges of the screen.

On the CPC, the GATE ARRAY "uses" the HSYNC and VSYNC signals to produce a composite signal sent after the 2nd HSYNC to the CTM. However, it is the starting position of the VSYNC during the line that allows to raise the electron gun more or less high (the later the VSYNC starts in the line, the higher the beam goes up).

Since the HSYNC is (in principle) always located in the same place, all the lines displayed on a CTM monitor have a definition of 1 pixel, which contradicts the logic stated above. In the case of a "Standard" CPC frame with R2=46 and R0=63, the VSYNC generated by the CRTC (on C0=0 on the first frame, on C0=31 on the second) is retrieved by the GATE ARRAY, which will wait for the end of the 2nd HSYNC to send the signal to the monitor. The electron gun then goes up exactly to the same place as on the previous frame (-0.0).

The additional line generated at the end of the first frame positions the odd frame 1 complete line down (-0.0+1=1). The first line of the even frame then alternates with the additional line, which is quite far from the expected effect.

So in this situation we have:

- 1 even frame with a duration of 19968 µsec, the **VSYNC** not being "shifted".
- 1 odd frame, which inherits the additional line from the even frame, and lasts 20032 µsec.

Fortunately, this is far from being irremediable (See Chapter 16.6, page 166).

19.3.2 THE TWO INTERLACE MODES

19.3.2.1 INTERLACE SYNC MODE : FUNCTION 00xx0001

This mode is used when you want the CRTC to display the same information for both the even and the odd frame.

The objective of this mode is to improve the quality of the characters displayed by filling the spaces between the lines in "non-interlace" mode.

In this mode, the CRTC only increases the resolution by moving the position of the VSYNC signal by 1/2 line, as indicated in the previous paragraph.

The image in this situation uses the same video memory, displaying the same thing twice.
In principle, it is not necessary in this mode to reprogram the CRTC registers.
Line 0 of the even lines frame is displayed first, followed by line 0 of the odd lines frame, and so on.

Screen:	Even	Odd	Beautiful drawing							
C9=	0									
C9=		0								
C9=	1									
C9=		1								
C9=	2									
C9=		2								
C9=	3									
C9=		3								
C9=	4									
C9=		4								
C9=	5									
C9=		5								
C9=	6									
C9=		6								
C9=	7									
C9=		7								

19.3.2.2 INTERLACE SYNC & VIDEO MODE : FUNCTION 00xx0011

This **IVM** mode is called "**Video Mode**" because it can be used to display images for which each of the 625 lines is different.

Contrary to the "Interlace Sync Mode", no line from the 2 frames is repeated.

On the first frame, the CRTC displays the lines for which C9 is even.
On the second frame, the CRTC displays the lines for which C9 is odd.

At the end of the display of the 2 frames (~0.04 second), the lines follow each other in even/odd order.

Given that it is necessary to display twice as many lines, it is therefore necessary to program the registers of the CRTC as if we were building a frame of 624 lines.

This logic has been abused by circuit designers.

The addition of the 625th line is managed automatically by the CRTC, as a reminder.

Note: In the UM6845R documents, the figure (7) used to describe this mode is incorrect (description of the line numbers displayed in the even/odd frames).
This error does not exist in the UM6845 documents from UMC (figure 13), which is a "confirmed" copy of the HD6845S from HITACHI (and detected as a CRTC 0).

Screen:	Even	Odd	ArtWork from MOMA							
C9=	0									
C9=		1								
C9=	2									
C9=		3								
C9=	4									
C9=		5								
C9=	6									
C9=		7								
C9=	8									
C9=		9								
C9=	10									
C9=		11								
C9=	12									
C9=		13								
C9=	14									
C9=		15								

Due to the repositioning of the sending of the C-VSYNC by the GATE ARRAY, the even frame is displayed a full line earlier than the odd frame.

In this context, line 0 alternates with a line of BORDER, while line 1 alternates with line 2, line 3 with line 4, and so on…

The image described above thus appears as on the diagram below, the "common" pixels then appearing in their original colours, rather than alternating with the white ink.

C9	Beautiful Drawing Even Screen							
0								
2								
4								
6								
8								
10								
12								
14								

C9	Beautiful Drawing Odd Sceen							
	BORDER or PREVIOUS LINE							
1								
3								
5								
7								
9								
11								
13								
15								

19.3.3 LIMITATIONS

There are specification differences on the registers to be updated (and their value) in Interlace mode between different circuits. These differences are mainly related to the counting mode of C9 and C4 initially defined by the designers of the circuits.

Of course, these restrictions are made not to be respected, but they nevertheless give interesting clues to the inner workings of the counters.

This is particularly the case for the R9 register on CRTC's 0, 3 and 4, which must be filled require with the number of lines of a vertical character, minus 2 for IVM mode. This should facilitate the end comparison of even lines, because it suffices to ignore bit 0 to perform the end of character comparison and to manage this bit 0 as that of frame parity.

The display of one line out of two causes the faster increment of C4 (on CRTC's 0, 1, 3, 4) and and VSYNC therefore occurs earlier when C4=R7.. The construction of a frame in the IVM mode therefore requires adapting R4 and R7 to the total size of the desired frame (except for CRTC 2, which adopts another methodology).

When a large frame is defined, it is possible to use "Overscan Bits"™ to avoid having to reprogram R12/R13 between even and odd frames, but I digress.

19.3.4 UNLOVED FEATURE

"Interlace" mode is an unpopular feature with CPC developers.
Note: I should have released demo 4 :-)

However, **this function is managed in real time** and its interest is not only related to its ability to "properly" process an Interlace image but also by producing, via the GATE ARRAY, a useful signal for the monitor.

As soon as the function is activated on a given C9 line, it can affect **the C9 and/or C4 count for the following lines**.

According to the value of C9 and its parity when R8 is modified, then the following line is immediately calculated according to a recipe specific for each CRTC.

In other words, from the perspective of creating a "complete" Interlace frame, **R8 should in principle only be activated when a frame starts** (when C4=C9=0).

From the perspective of using only certain functions of the IVM interlace, it is possible to activate and deactivate this mode before certain functions take place.

Until its activation the frame contains even and odd lines. As soon as IVM mode is activated, the frame contains even and/or odd lines (according to rules specific to the CRTC) until IVM mode is deactivated, in order to return to "classic" count.

Activating Interlace mode, in addition to modifying the C9 and C4 count, activates two other distinct managements:

- The **management of one additional line at the end of the first frame**, before C9=C4=C0=0.

- **Management of activation of the VSYNC signal on C0=R0/2** (**therefore on C0=31 in the case of a frame or R0=63**) when C4=R7 (**Mid-VSYNC**).

19.4 VERTICAL INTERLACE PROGRAMMING

19.4.1 CRTC 0

In **INTERLACE SYNC**, R9 must be programmed with the value N-1 if N represents the number of lines(s) of a vertical C4 character. R4, R5, R6 and R7 must be programmed by considering that C4 characters contain n line(s) each.

In **INTERLACE VIDEOMODE** (IVM), R9 must be programmed with value N-2 if N represents the number of lines(s) of a vertical character. The CRTC 0 shares this feature with CRTC's 3 and 4. R4, R6 and R7 must be programmed by considering that the characters contain twice less lines. When N is odd, a particular logic of balancing takes place (see chapter 19.5.2). R5 still contains a finite number of lines without considering the "Interlace" mode.

If a vertical character C4 makes 8 lines, R9 must contain 6.
If R9 is programmed with 0, this means that there are at least 2 lines displayed (1 per frame).

19.4.2 CRTC 1

If N represents the number of line(s) of a vertical character, then R9 must be programmed with the value N-1 in the 2 interlace modes.

In **INTERLACE SYNC**, R4, R5, R6 and R7 must be programmed as if C4 characters contain N lines.

In **INTERLACE VIDEOMODE (IVM)**, R4, R6 and R7 must be programmed as if C4 characters twice less lines. When N is odd, a particular logic of balancing takes place (see chapter 19.5.3). R5 still contains a finite number of lines without considering the "Interlace" mode.

19.4.3 CRTC 2

If N represents the number of line(s) of a vertical character, then R9 must be programmed with the value N-1 in the 2 interlace modes.

In **INTERLACE SYNC**, R4, R6 and R7 must be programmed by considering that C4 characters contain n lines each. R5 still contains a finite number of lines without considering the "Interlace" mode

In **INTERLACE VIDEOMODE**, R4, R6 and R7 must be programmed by considering that C4 characters contain n lines each. R5 still contains a finite number of lines without considering the "Interlace" mode.

For each C4 value, two characters of N/2 lines are treated.
A specific C9 counter (C9.IVM) is managed for display. This counter is initialized when C9 goes to 0, but also when it reaches the value of R9 "outside parity", in order to update the VMA video pointer without C4 being incremented. (see Chapter 19.8.3).

When R9 = 7, we have C4 characters of 8 lines for each frame with an update of video pointer every 4 lines. This greatly simplifies the programming of R4 and R7 in the most common situations, since the total number of lines to display during a frame is equivalent to a "no-interlaced" mode.

Thus R6 concerns 1 "double" character of 8 lines and not 1 character of 4 lines.

One more than expected line is therefore displayed on odd frames.
If R9 is 6, for example, line C9 = 7 will still be displayed on odd frames.
In addition, the update of the video pointer (when C0 == R1) no longer takes place when C4 is included because it takes place only when C9.IVM = R9 (excluding parity). (see Chapter 19.8.3).

C4 continues to be managed normally, but the display is incorrect.

The other CRTC's allows to manage characters indiscriminately with a number of even or odd lines, with a tricky management of the parity of the frame, counters and registers.

It is therefore an urban legend to consider that the CRTC 2 manages the "Interlace" mode better than the others CRTC's. It is certainly simpler to modify only R8 without having to modify R4, R5, R6 or R7, but it is impossible to define C4 characters composed of odd lines without this counting being aligned with an even number during the odd frames and this causes a serious problem with the assignment of the video pointer to each new C4.

19.4.4 CRTC 3 & 4

In **INTERLACE SYNC**, R4, R5, R6 and R7 must be programmed as if C4 characters contain N lines (and N represents the number of line(s) of a vertical character).

In **INTERLACE VIDEOMODE** (IVM), R9 must be programmed with value N-2 if N represents the number of lines(s) of a vertical character. The AMSTRAD' CRTC's shares this feature with CRTC 0. R4, R6 and R7 must be programmed by considering that the characters contain twice less lines. When N is odd, a particular logic of balancing takes place (see chapter 19.5.5). R5 still contains a finite number of lines without considering the "Interlace" mode.

If a vertical character is 8 lines long, R9 must contain 6.
If R9 is programmed with 0, this means that there are at least 2 lines displayed (1 per frame).

19.5 PARITY

19.5.1 GENERAL

When it comes to interlace mode, the frame parity plays an important role because it participates in:

- The addition of the extra line when the construction of the even frame is completed.
- The generation of a **MID-VSYNC** when C4=R7 on the even frame.
- Calculation of even and odd C9s.

The common point between all CRTC's is to have an internal state that switches from even to odd and vice versa according to different conditions. Note that on CRTC's 0, 3 and 4, there is not one, but several distinct parity management states (see chapter 19.5.5)

19.5.2 CRTC 0

There are several parity management states:

- **ParityFrame=ParityR6** when C4=C9=C0=0. This state defines the parity of the first C9 of the frame.

- **ParityC9** which defines bit 0 of the C9s in "Interlace" mode

- **ParitéR6=ParitéFrame xor 1** when **C4 reach R6**

The **ParityR6** state allows the CRTC to anticipate the parity of the next frame. If **ParityFrame** is even, then parity will be odd and vice versa. If C4 cannot reach R6 (because R6> R4) this state is no longer updated and **ParityFrame** remains frozen with the last value of **ParityR6**.

The management of **ParityR6** is independent of the value of R8.

At the start of the frame, **ParityFrame=ParityR6.**
C9 parity is managed only when R8 = 3 to update C9.

The **ParityR6** state also lets you know if an additional line will be generated at the end of the frame (see chapter 19.6.1)

In IVM mode, R9 is programmed with an **even** number to define an **even** number of lines in a character (For example R9=6 to obtain 2 x 4 line/char=8 lines). This is related to the method used to test the end of a character composed only of even lines or odd lines depending on the current parity. In this case, the parity is identical regardless of the value of C4.

When R9 is **odd** in IVM (unlike CRTC 1), this implies a difference between the number of even and odd lines for a character between 2 frames (the total number of lines of a vertical character is odd). In this circumstance, the designers of the circuit balanced the number of lines between 2 frames in order to avoid that there is a frame with R4+1 lines more on an even frame than on an odd frame.

The adopted solution consists in alternating characters composed of even lines with characters composed of odd lines on the same frame **each time C4 evolves**. Thus, when **R9 is odd, the parity of the lines depends on that of C4 and on the current parity at the start of the frame**.

C9 parity is calculated so when R8 is 3:
ParityC9 = C4.0 xor ParityFrame
(0 = even/1 = odd)

Thus, on an even frame, the parity of C9 is equal to that of C4.
On an odd frame, the parity of C9 is the opposite of that of C4.

For each even C4, the balance of lines is preserved.

Example :
R9 = 7. On an even frame with C4 = 0, the CRTC generates 5 even lines (0.2.4.6.8) followed by 4 odd lines (1.3.5.7). On an odd frame with C4 = 0, the CRTC generates 4 odd lines followed by 5 even lines. For each even C4, the line balance is thus maintained at 9 lines.

To be fully functionaly, this method involves specific management of **VSYNC** in IVM mode. Indeed, if R7 is scheduled on an odd C4, then the VSYNC is delayed by 1 line. It occurs when C4= R7 and C9.VMA=2 on the odd C4s. This avoids phase shift with the **VSYNC** generated on this same C4 on an even frame.

	PARITYFRAME=ODD				PARITYFRAME=EVEN		
	C4	C9			C4	C9	
	0	0	R8=3		0	0	
	0	3			0	2	
	0	5			0	4	
	0	7			0	6	
	1	0			0	8	
VSYNC	1	2	R7=1		1	1	VSYNC
	1	4			1	3	
	1	6			1	5	
	1	8			1	7	
VSYNC	2	1	R7=2		2	0	VSYNC
	2	3			2	2	
	2	5			2	4	
	2	7			2	6	
	3	0			2	8	
VSYNC	3	2	R7=3		3	1	VSYNC
	3	4			3	3	
	3	6			3	5	
	3	8			3	7	
VSYNC	4	1	R7=4		4	0	VSYNC
	4	3			4	2	

Note :
If R8 goes to 3 on an odd C4, this can cause a phase of 1 line between the **VSYNC** of even and odd frames. Indeed, this **VSYNC** shift technique used by engineers who designed the circuit only works properly to manage the line imbalance between an even C4 and an odd C4.

For example, if R9=7 and R8 goes to 3 on C4=1 (C9=0), the number of lines on C4=0 will be identical whatever the framework of the frame (namely 8 lines), but it will be different on C4=1 (or 5 even lines on an odd frame, or 4 odd lines on an even frame).

This phase shift in the line of lines between the 2 frames phase the **VSYNC**:

	PARITYFRAME=ODD			PARITYFRAME=EVEN		
	C4	C9		C4	C9	
	0	0		0	0	
	0	1		0	1	
	0	2		0	2	
	0	3		0	3	
	0	4		0	4	
	0	5		0	5	
	0	6		0	6	
	0	7		0	7	
	1	0	R8=3	1	0	
	1	2		1	3	
	1	4		1	5	
	1	6		1	7	
	1	8	R7=2	2	0	VSYNC
VSYNC	2	1	R7=2	2	2	
	2	3		2	4	
	2	5		2	6	
	2	7		2	8	
	3	0	R7=3	3	1	VSYNC
VSYNC	3	2	R7=3	3	3	
	3	4		3	5	
	3	6		3	7	
	3	8	R7=4	4	0	VSYNC
VSYNC	4	1	R7=4	4	2	
	4	3		4	4	
	4	5		4	6	
	4	7		4	8	
	5	0	R7=5	3	1	VSYNC
VSYNC	5	2	R7=5	3	3	

Note: On an even frame, the **VSYNC** occurs in the middle of the line on C0 = R0/2.

There are several methods to select a specific parity.

It is possible to know current parity since there is a counting bug when the IVM mode is activated on C9 = R9 with an odd parity, or when the IVM mode is disabled on C9.VMA=R9+1 (See Chapter 19.8.1).

Once parity is identified, it suffices to let C4 reach R6 once for the parity to reverse.

19.5.3 CRTC 1

There are two parity states:

• **ParityFrame** switch between each frame when **C4 = C9 = C0 = 0**

• **ParityC9** switch between each C4 if **R9 is even**.

When a frame starts, **ParityFrame** switch off even to odd and vice versa, **whatever the value of R8**.

If **ParityFrame** is even, then an **additional line** and a **MID-VSYNC** are scheduled.
If **ParityFrame** is odd, then **no additional line** and **no MID-VSYNC**.

When R9 is **even**, the number of lines of a vertical character is **odd**. In this circumstance, the designers of the circuit balanced the number of lines between 2 frames in order to avoid that there is a frame with R4+1 lines more on an even frame than on an odd frame.

The adopted solution consists in alternating characters composed of even lines with characters composed of odd lines in the same frame. Thus, when **R9 is even, the parity of the lines depends on that of C4 and on the current parity at the start of the frame.**

However, contrary to what was done on CRTC's 0, 3 and 4, there is no specific management of the **VSYNC** in IVM mode when the number of lines of a character is odd. The **VSYNC** is not delayed from a line on odd C4s when R9 is even. So, it is impossible to position R7 on an odd C4 without creating a gap of 1 line on the **VSYNC** between 2 frames:

	EVEN FRAME				ODD FRAME	
	C4	C9			C4	C9
+32	0	0	VSYNC	VSYNC	0	1
	0	2			0	3
	0	4			0	5
	0	6			0	7
	0	8		VSYNC	1	0
+32	1	1	VSYNC		1	2
	1	3			1	4
	1	5			1	6
	1	7			1	8
+32	2	0	VSYNC	VSYNC	2	1
	2	2			2	3
	2	4			2	5
	2	6			2	7
	2	8		VSYNC	3	0
+32	3	1	VSYNC		3	2
	3	3			3	4
	3	5			3	6
	3	7			3	8
+32	4	0	VSYNC	VSYNC	4	1
	4	2			4	3
	4	4			4	5
	4	6			4	7
	4	8		VSYNC	5	0
+32	5	1	VSYNC		5	2
	5	3			5	4

When R8 is modified (R8 changes from 0 to 3 or vice versa), the parity and/or bit 0 of C9 are updated according to rules which involve the current parity, the parity of C9 before the change to IVM and the parity of C4 if R9 is even.

These updates are performed on the 3rd and 4th µseconds of the OUT(C),C instruction (on R8).

<u>On the 3rd µsecond when R8 changes from 3 to 0 or vice versa:</u>

- If R9 is even, the parities of C4 and C9 are used to set the parity of the new C9.
- If R9 is odd, only the parity of C9 is used to set the parity of the new C9.

ParityC9 = C9.0
ParityC9 = ParityC9 xor (C4.0 and not (R9.0))

It should be noted that the current parity of the frame is not modified.

<u>On the 4th µsecond when IVM becomes active (OUT R8,3):</u>

- If **ParityFrame** is even and R9 is even, then the parity is equal to that of C4.
- If **ParityFrame** is even and R9 is odd, then the parity is even.
- **Parityframe** changes to even, except for cases where **ParityFrame** and **ParityC9** were odd before the request for IVM.

If (ParityFrame==EVEN)
Then
    ParityC9=C4.0 and (not R9.0)
End If
ParityFrame=ParityFrame and (ParityC9 xor (C4.0 and (not R9.0)))

<u>On the 4th µsecond when IVM goes idle (OUT R8.0):</u>
When IVM mode is disabled (OUT R8,0), then **ParityFrame** is equal to **ParityC9**.

ParityFrame=ParityC9

If IVM mode is toggled on and off on an even C9 line, regardless of the value of R9, the parity is set to EVEN. It is thus possible to fix the parity quite easily on this CRTC.

Parityframe is reversed with each new frame.

ParityC9 is reversed with each C4 increasing when R9 is peer.

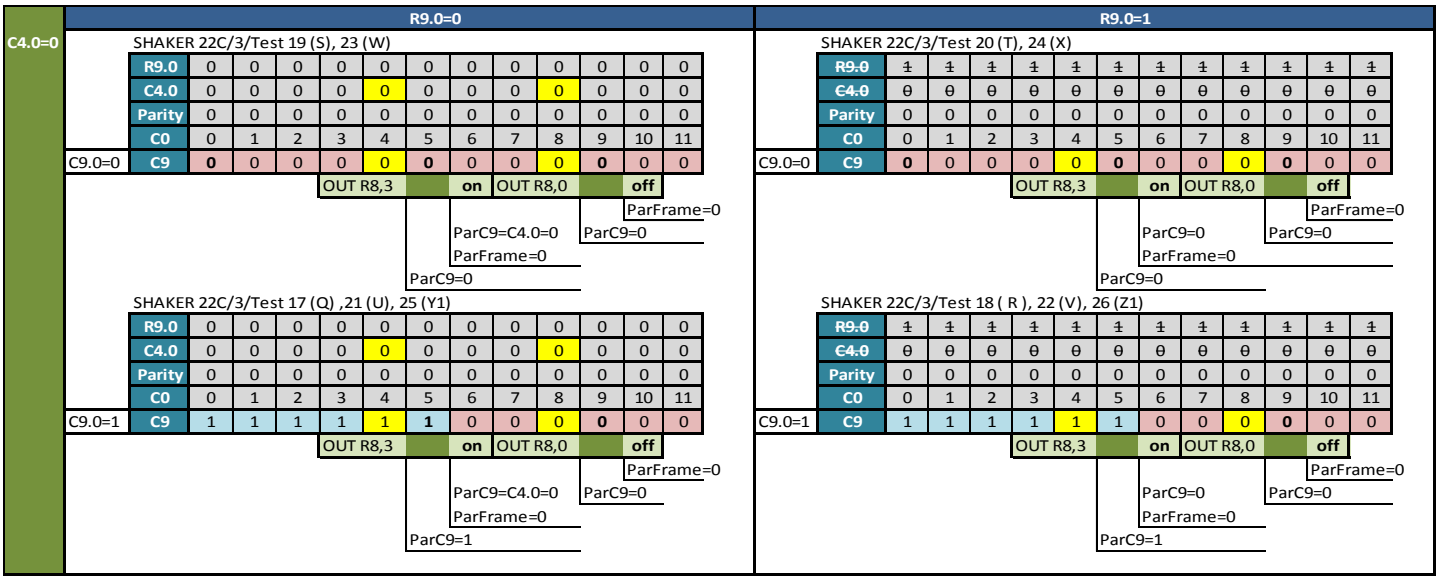
It is necessary to take these permutations into account if the IVM mode is left active for several frames.

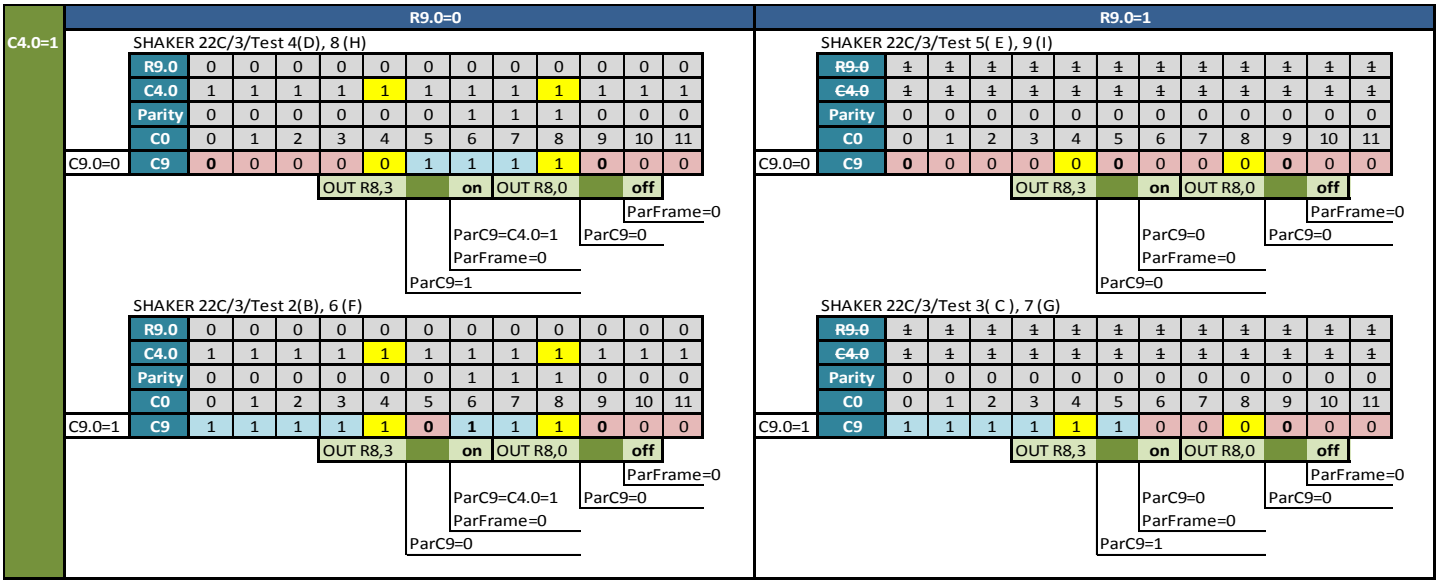
The R9 bit 0 is considered immediately by IVM activation. Modifying it before or after the activation of the IVM mode therefore affects C9. Conversely, deactivate the IVM mode can also modify C9, and modify the end condition of character C4.

The diagrams below describe all the scenarios that can be encountered when IVM mode is activated, then deactivated.

Initial parity: EVEN

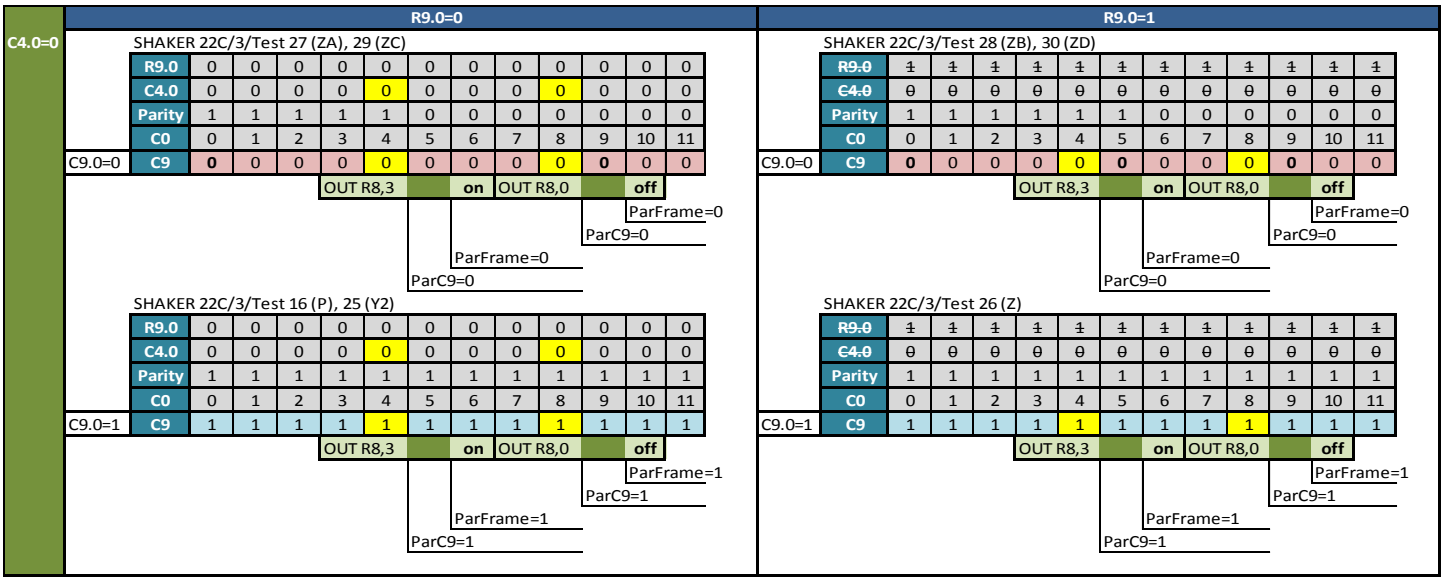
The diagrams below describe the frame parity and the parity of C9 for a current even parity, according to the parity of the previous C9 and that of C4 (when R9 is even), when there is an out R8, 3 followed by an OUT R8,0

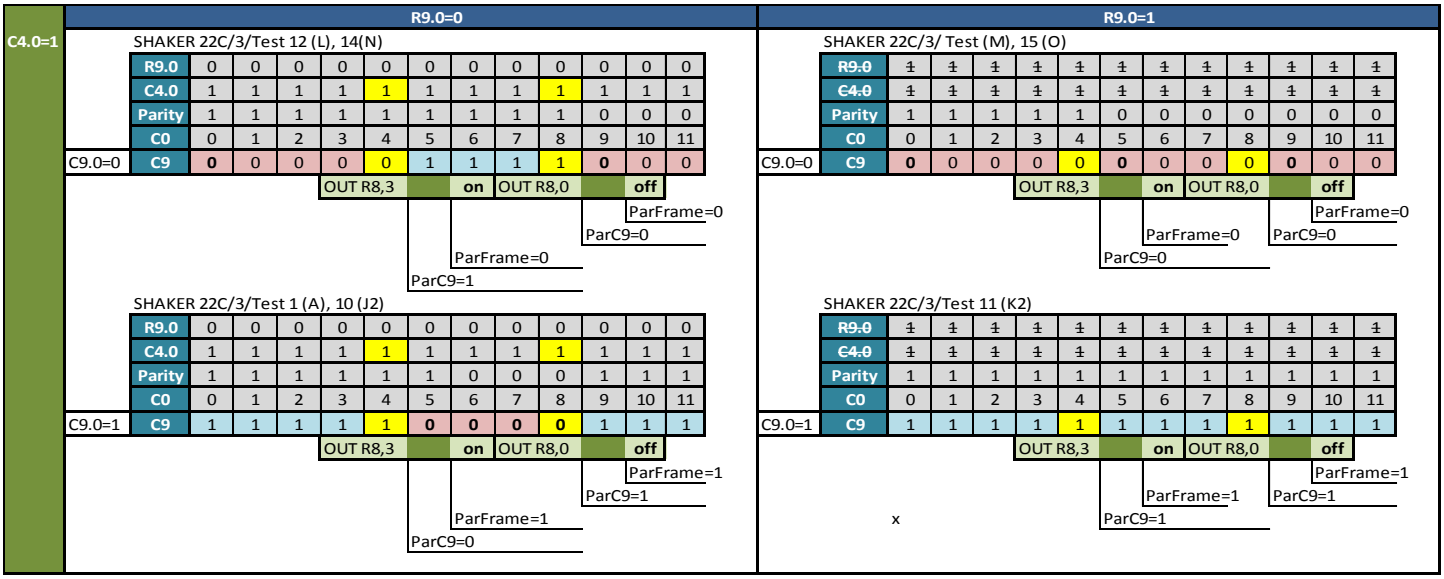




Initial parity: ODD

The diagrams below describe the frame parity and the parity of C9 for an odd current parity, according to the parity of the previous C9 and that of C4 (when R9 is even), when there is an out R8, 3 followed by an OUT R8,0





19.5.4 CRTC 2

There are several parity management states:

- **ParityFrame=ParityR6** when C4=C9=C0=0. This state defines the parity of the first C9 of the frame.

- **ParitéR6=ParitéFrame xor 1** when **C4 reach R6**

The **ParityR6** state allows the CRTC to anticipate the parity of the next frame. If **ParityFrame** is even, then parity will be odd and vice versa. If C4 cannot reach R6 (because R6> R4) this state is no longer updated and **ParityFrame** remains frozen with the last value of **ParityR6**.

The management of **ParityR6** is independent of the value of R8.

The **ParityR6** state also lets you know if an additional line will be generated at the end of the frame (see chapter 19.6.3)

At the start of the frame, **ParityFram**e = **ParityR6.**
C9 parity is managed only when R8 = 3 to update C9.

Unlike other CRTC's, C9 management has been carried out in a simple way, not without introducing some constraints (R9 odd for example).

Parity is respected whatever the values of R9 and C4.
On the other CRTCs, R9 defines a total number of lines to share between 2 frames, which is a problem when this number of lines is odd, and requires some adjustments in order to balance the lines between 2 frames and properly treat the **VSYNC**.

Parity management for the "calculation" of C9 is considered **immediately** from the 3rd NOP of OUT(C) on R8:

EVEN PARITY FRAME

C0	0	1	2	3	4	5	6	7	8	9	10	11
C9	3	3	3	3	3	3	3	3	3	3	3	3
C9.Vma	3	3	3	3	3	6	6	6	6	3	3	3
				OUT R8,3			**on**	OUT R8,0			**off**	

ODD PARITY FRAME

C0	0	1	2	3	4	5	6	7	8	9	10	11
C9	3	3	3	3	3	3	3	3	3	3	3	3
C9.Vma	3	3	3	3	3	7	7	7	7	3	3	3
				OUT R8,3			**on**	OUT R8,0			**off**	

CRTC 2 may not be able to do **SPLITBORDER**, but it can do **SPLITC9**!

It is possible to test the existence of the additional line to determine parity. If the IVM mode is activated on the first line of an odd frame, then this line will become an additional line, and a new line 0 will follow the old line 0, which will extend the size of the frame with R0 µsec.

19.5.5 CRTC 3 & 4

There are two parity states:

• **ParityFrame** switch between each frame when **C4=C9=C0= 0**

• **ParityC9** switch between each C4 if **R9 is even**.

When a frame starts, **ParityFrame** switch off even to odd and vice versa, **whatever the value of R8**.

If **ParityFrame** is even, then an **additional line** and a **MID-VSYNC** are scheduled.
If **ParityFrame** is odd, then **no additional line** and **no MID-VSYNC**.

At the beginning of the frame, **Parityc9=ParityFrame**.

When R8 changes to 1 or 3, **Parityc9**=**C9.0** (**Parityc9** is fixed with the parity of the current C9). This C9 parity is managed only when R8 = 3 to update C9.

ParityC9 switches to each new C4 when R9 is odd (as on CRTC 0) (odd number of lines for a character in IVM mode).

This means that it is possible to have even C9's on a first odd frame, or odd C9's on a first even frame. **The parity of C9 then aligns with the parity of the frame**.

In other words, the parity of the C9 is defined **immediately** when R8 goes to 3 in opposition to the parity of the frame (which then acts only at the level of the **additional line** and the **MID-VSYNC**). In this case, the C9 parity will then be identical on the following frame. Unlike CRTC's 1 and 2, and as CRTC 0, C9 does not change during the line.

In IVM mode, R9 is programmed with an **even** number to define an **even** number of lines in a character (For example R9=6 to obtain 2 x 4 line/char=8 lines). This is related to the method used to test the end of a character consisting of only even lines or odd lines according to the **ParityC9** state. In this case, the parity is identical regardless of the value of C4.

When R9 is **odd** in IVM (as on CRTC 0), this implies a difference between the number of even and odd lines for a character between 2 frame (the total number of lines of a vertical character is odd). In this circumstance, the designers of the circuit balanced the number of lines between 2 frames in order to avoid that there is a frame with R4+1 lines more on an even frame than on an odd frame.

The adopted solution consists in alternating characters composed of even lines with characters composed of odd lines on the same frame **each time C4 evolve**s. Thus, when R9 is **odd, the parity of the lines depends on that of C4 and on the current parity of C9**. (The parity of C9 being reloaded from the parity of the frame at each new frame).

To be fully functional, this method involves specific management of **VSYNC** in IVM mode. Indeed, if R7 is programmed on an odd C4 during an odd frame, then the **VSYNC** is delayed by 1 line. This avoids phase shift with the **VSYNC** generated on this same C4 on an even frame. The following diagram shows this balancing after 1 frame.

	PARITYFRAME=ODD			PARITYFRAME=EVEN		
	C4	C9		C4	C9	
	0	1		0	0	
	0	3		0	2	
	0	5		0	4	
	0	7		0	6	
	1	0		0	8	
VSYNC	1	2	R7=1	1	1	VSYNC
	1	4		1	3	
	1	6		1	5	
	1	8		1	7	
VSYNC	2	1	R7=2	2	0	VSYNC
	2	3		2	2	
	2	5		2	4	
	2	7		2	6	
	3	0		2	8	
VSYNC	3	2	R7=3	3	1	VSYNC
	3	4		3	3	
	3	6		3	5	
	3	8		3	7	
VSYNC	4	1	R7=4	4	0	VSYNC
	4	3		4	2	

However, according to the parity of C9 on which R8 changes to 3, this can lead to a contradiction between the parity of the frame and the parity of C9 on the first frame. There may also be a counting imbalance if R8 goes to 3 on an odd C9.

Example 1: R9=7. If R8 goes to 3 on C9=1 (odd) on an odd frame, line C9=0 has already been displayed and there are therefore 1 more line on C4=0. The frame being odd, the VSYNC is delayed by 1 line on the odd C4s. On the following frame (even), the **VSYNC** will not be delayed at each odd C4. This leads to a **VSYNC** imbalance between the first 2 frames on all the values of R7:

	PARITYFRAME=ODD			PARITYFRAME=EVEN		
	C4	C9		C4	C9	
	0	0		0	0	
R8=3	0	1		0	2	
	0	3		0	4	
	0	5		0	6	
	0	7		0	8	
	1	0	R7=1	1	1	VSYNC
VSYNC	1	2	R7=1	1	3	
	1	4		1	5	
	1	6		1	7	
	1	8	R7=2	2	0	VSYNC
VSYNC	2	1	R7=2	2	2	
	2	3		2	4	
	2	5		2	6	
	2	7		2	8	
	3	0	R7=3	3	1	VSYNC
VSYNC	3	2	R7=3	3	3	
	3	4		3	5	
	3	6		3	7	
	3	8	R7=4	4	0	VSYNC
VSYNC	4	1	R7=4	4	2	
	4	3		4	4	

Note: On an even frame, the **VSYNC** occurs in the middle of the line on C0 = R0/2.

Example 2: R9=7. If R8 changes to 3 on C9=1 (odd) on an even frame, line C9=0 has already been displayed and there are therefore 1 more line on C4=0. The frame being even, the **VSYNC** is not delayed by 1 line on the odd C4s. On the following frame (odd), the **VSYNC** will be delayed at each C4 odd. This leads to a **VSYNC** imbalance between the first **2 frames** on the even R7's:

	PARITYFRAME=EVEN			PARITYFRAME=ODD		
	C4	C9		C4	C9	
	0	0		0	1	
R8=3	0	1		0	3	
	0	3		0	5	
	0	5		0	7	
	0	7		1	0	
VSYNC	1	0	R7=1	1	2	VSYNC
	1	2		1	4	
	1	4		1	6	
	1	6		1	8	
	1	8	R7=2	2	1	VSYNC
VSYNC	2	1	R7=2	2	3	
	2	3		2	5	
	2	5		2	7	
	2	7		3	0	
VSYNC	3	0	R7=3	3	2	VSYNC
	3	2		3	4	
	3	4		3	6	
	3	6		3	8	
	3	8	R7=4	4	1	VSYNC
VSYNC	4	1	R7=4	4	3	
	4	3		4	5	

To prevent the transition from creating a problem on the first 2 frames, R8 must be programmed with 3 on the first line of a peer frame. The problem being that knowing the parity of a frame requires going into IVM mode (R8 = 3). So there is one in two chance that the VSYNC is worthy of the first two frames.

19.6 ADDITIONAL INTERLACE LINE

19.6.1 CRTC 0

The additional line is added at the end of the frame (after the R5 lines if necessary) if one of the two "Interlace" modes is activated (R8=3 or 1) and the **ParityR6** state is odd.

As a reminder, **ParityR6** state becomes odd when C4 reaches R6 on an even frame, and even when C4 reaches R6 on an odd frame, because it is used to anticipate the parity of the following frame.

If R6>R4, **ParityR6** state is no longer updated and this has the effect of freezing the parity of the frame and the addition of the additional line.

This management principle implies that if we are on an even frame, and that C4 reaches R6, then **ParityR6** becomes odd. An additional line will then be generated at the end of this even frame. The new frame becomes odd (equal to **ParityR6**). But if, during this new frame (odd) C4 can no longer reach R6 (because R6 has been modified to be greater than R4) then an additional line will be generated for each frame as long as R6>R4 (and R8=3 or 1), whatever the parity of the C9's. The opposite is true: **ParityR6** becomes even, and there is therefore no additional line at the end of the frame. The following frame becomes even, but if R6 becomes greater than R4 during this frame, then all the frames will remain even and without additional line.

C4 is incremented only once for all additional lines (R5 and interlace) and is equal to C4=R4+1.

19.6.2 CRTC 1

The additional line is added at the end of the frame (after the R5 lines if necessary) if one of the two "Interlace" modes is activated (R8=3 or 1) and if **ParityFrame** is even.

C4 is incremented each time C9=R9. When the interlace mode is active, if R9+1 is a multiple of R5 then C4 is incremented once again on all even frames. So the CRTC completes the C4/C9 lines according to the counting mode.

19.6.3 CRTC 2

The additional line is added at the end of the frame (after the R5 lines if necessary) if one of the two "Interlace" modes is activated (R8=3 or 1) and the **ParityR6** state is odd.

As a reminder, **ParityR6** state becomes odd when C4 reaches R6 on an even frame, and even when C4 reaches R6 on an odd frame, because it is used to anticipate the parity of the following frame.

If R6>R4, **ParityR6** state is no longer updated and this has the effect of freezing the parity of the frame and the addition of the additional line.

This management principle implies that if we are on an even frame, and that C4 reaches R6, then **ParityR6** becomes odd. An additional line will then be generated at the end of this even frame. The new frame becomes odd (equal to **ParityR6**). But if, during this new frame (odd) C4 can no longer reach R6 (because R6 has been modified to be greater than R4) then an additional line will be generated for each frame as long as R6>R4 (and R8=3 or 1), whatever the parity of the C9's.

The opposite is true: **ParityR6** becomes even, and there is therefore no additional line at the end of the frame. The following frame becomes even, but if R6 becomes greater than R4 during this frame, then all the frames will remain even and without additional line.

C4 is incremented each time C9 = R9. When the interlace mode is active, if R9+1 is a multiple R5, then C4 is incremented once again on all even frames. If R7 is scheduled with this C4 value, there is no more VSYNC one in 2.

There is a **noticeable bug** on the management of the additional line.

If the IVM mode is activated **on the first line of an odd frame**, **then this line will become an additional line**, and a new line 0 will follow the old line 0, which will extend the size of the frame by R0 µsec. This is true whatever the value of C0 (0 to R0) on which the IVM mode is activated.

Conversely, if the IVM mode is disabled during the additional line (C4 being then greater than R4), then C4 will not be automatically reset to 0 on the next line. C9 will count until it reaches R9, and C4 will increment until it reaches R4.

Example: if R4=38 (R5=0) and an additional line is generated because IVM is active at the end of the even frame (R8=3 when C4=R4, C9=R9, C0=R0) then an additional IVM line is added on C4=39, C9=0. If IVM is disabled on this line (R8=0), then C9 will count to R9, then C4 will be incremented again to 40 (because R4=38).

If the parity was odd when the IVM mode is activated on line 0, this line, now considered as an additional line, immediately becomes odd. Thus the C9 displayed as soon as R8=3 on this line will be odd (i.e. C9=1). The additional line cannot be triggered with a fixed even parity, because **ParityR6** is false on an odd frame when C4=R6. This last condition is necessary to obtain an even parity.

19.6.4 CRTC'S 3 & 4

The additional line is added at the end of the frame (after the R5 lines if necessary) if one of the two "Interlace" modes is activated (R8=3 or 1) and if **ParityFrame** is even.

The addition of the additional line does not depend on the C4=R6 equivalence, unlike CRTC's 0 and 2.

When the additional line is added, C4 is not incremented (unlike all other CRTC's). The last value of C4 carries all the lines programmed in R9, plus those of R5 plus the additional Interlace line generated during the even frames.

Note: The additional line generated does not consider parity states as on other CRTC's. C9 will always be 0, even if the other lines are odd on C4=R4.

Note: The update of the video pointer is considered on C9=R9 of C4=R4 without C4 being incremented. The first additional line begins with the pointer that VMA was worth at the time when C9=R9 and C0=R1.

19.7 MID-VSYNC

19.7.1 GENERALITIES

MID-VSYNC defines a **VSYNC** that occurs in the middle of a line.

In general, **VSYNC** occurs when C4 is equal to R7 on any position of C0 (except on CRTC's 3 and 4, which dictate that C4=C9=C0=0).

There is also an exception on CRTC's 0, 3 and 4 when the line count of a C4 character is odd on an odd frame and an odd C4. The **VSYNC** can then be delayed by one line. **MID-VSYNC** is not cumulative with this line because it cannot occur on an odd frame with an odd C4.

When the state of **MID-VSYNC** is active, then the **VSYNC** only triggers when C0 reaches R0/2.

This is what in principle allows to create half-lines, in conjunction with the additional line which allows the monitor to compensate for the difference between 2 frames in "Interlace" (each frame then occupying 32 µs more (with R0=63)).

19.7.2 CRTC 0, 1, 2

The **MID-VSYNC** is generated when C4=R7 if **ParityFrame** is even (and R8 is 3 or 1).

If R7=0, the management of the **VSYNC** coincides with the start of the frame.
ParityFrame management takes priority over VSYNC management.

Thus, if R7=0, and **ParityFrame** switch and becomes even when C4=C9=C0=0, the C4/R7 comparison will then be processed to trigger a **VSYNC**.

If R8=3 or R8=1, and **ParityFrame** has change to even, then **VSYNC** will occur when C0 reaches R0/2.

MID-VSYNC therefore always takes place when **ParityFrame** is even, including when R7=0.

19.7.3 CRTC's 3, 4

The **MID-VSYNC** is generated when C4=R7 if **ParityFrame** is even (and R8 is 3 or 1).

In the particular case where R7=0, the management of the **VSYNC** coincides with the start of the frame. In this case, **the management of the VSYNC has priority over the assignment of ParityFrame.**

Thus, if R7=0 (and R9=3 or 1), the C4/R7 comparison is processed before **ParityFrame** switch.

If **ParityFrame** was odd, then there will be no **MID-VSYNC**, and **VSYNC** will start on C4=C9=C0=0, although **ParityFrame** has change to Even.
Conversely, if **ParityFrame** was even, then there will be a **MID-VSYNC**, and the **VSYNC** will start when C0 reaches R0/2 and even though **ParityFrame** has become odd.

MID-VSYNC therefore always takes place when **ParityFrame** is even, except when R7=0.
And it never occurs when **ParityFrame** is odd, except when R7=0.

19.8 COUNTING IN INTERLACE VIDEOMODE

19.8.1 CRTC 0

When **IVM** mode is activated, counter C9 continues to increment normally.

However, when C9 is used in building up the VMA address, C9 is considered shifted left by 1 bit, and bit 0 represents parity. Each increment therefore "corresponds" to an addition of 2 to the counter C9-VMA considering **ParityC9**.

The increment management of C9 continues to be processed on the normal value of C9.

From the line C9 which follows that where R8 goes to 3, the value of C9 is multiplied by 2 to be tested with R9 in order to allow C9 to return to 0. The capacity of R9 being 5 bits, the more significant bit is "lost" in the multiplication operation.

We can formulate it like this on the lines located after the one where R8 has gone to 3:

```
If ((C9 x 2) or ParityFrame)== (R9+ParityFrame)
Then
        If C4==R4 (end of frame (last line true))
        Then
                C4=0
                If ParityR6==true (true if C4==R6 is true on even frame)
                        ParityFrame=ParityFrame xor 1
                End if
        Else
                C4++
        End If
        If R9.0=0 (C9 parity switched if R9 is odd)
        Then
                ParityC9=C4.0 xor ParityFrame
        End of
        C9=0
        C9.VMA=(C9 x 2) or ParityC9
Else
        C9=C9+1
        C9.VMA=(C9 x 2) or ParityC9
End If
```

When R8 changes to 3, a status indicates that the calculation of the value compared to **'R9 or ParityFrame**' will be performed on the next C0=0, after the C9/R9 test of the line. This is most certainly done in this way to prevent the C9 used for the display from switching mid-line. The test value for R9 is therefore the old C9, but the **parity is however considered immediately for R9**.

Thus, on the line where R8 goes to 3, it is C9 (and not C9.VMA) which is compared to "**R9 or ParityFrame**". **If C9=R9 and the parity is odd, then the test C9=R9+1 is false: C9 is not reset to 0 and overflows**. In other words, C9=R9+1.

The value of C9 when R8 goes to 3 can lead to an overflow of C9.

For example, if R9=6 and C9=4 when R8 goes to 3, C9 is different from "**R9 or ParityFram**e" (6 or 7) and the following C9 will therefore be C9+1=5. The C9.VMA displayed (and tested until the end of the line) will be (C9x2)+ParityFrame, i.e. 10 or 11 depending on the current parity. C9 will then continue to increment until **C9x2+ParityFrame** equals "**R9 or ParityC9**".

When R8 returns to 0, the same mechanics apply. The state set with R8 is considered when C0 returns to 0. The test takes place with C9.VMA (which includes parity) and R9: ParityC9 is no longer considered for R9. The limit test is therefore carried out with 'C9x2+ParityFrame' (C9.VMA) and R9 (without parity).

Note: The C9/R9 or C9.VMA/'R9 or ParityC9' test also takes place when C0=R1 (which determines the activation of the BORDER) and the assignment of VMA' with VMA when C9=R9 (or C9.VMA= R9 or ParityC9). Considering parity with R9 as soon as R8=3 (and not taking it into account as soon as R8=0) in the test with C9/C9.VMA can therefore also affect the update of VMA'.

If C9 was 3 and R9=6 on an odd frame, then C9.VMA=7 at the beginning of the line. When R8 goes to 0, C9.VMA is compared to R9 (the consideration of parity is cancelled). C9.VMA=7 <> R9=6, which leads C9 to be incremented (and therefore go to 4). If we wanted C9 to return back to 0, we would for example have to program R9 with C9.VMA before the end of the line (i.e. 7) so that the comparison between C9.VMA and R9 without parity allows C9 to return back to 0.

The diagrams on the following pages describe different counting situations, when switching to IVM mode or when exiting IVM mode. They indicate the value of C9 considered and that available for the constitution of the video pointer.

The IVM period for the following tests covers only a few lines of the frame.
The values of the registers for the indicated results are:
R9=6 (even) and R8=0 before going to 3 (or returning to 0).

Switching to IVM mode on CRTC 0 :

EVEN FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	**R8=3**
0	1	2	
0	2	4	
0	3	**6**	
1	0	0	
1	1	2	
1	2	4	
1	3	**6**	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	**R8=3**
0	2	4	
0	3	**6**	
1	0	0	
1	1	2	
1	2	4	
1	3	**6**	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	**R8=3**
0	3	**6**	
1	0	0	
1	1	2	
1	2	4	
1	3	**6**	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	**R8=3**
0	4	8	
0	5	10	
0	6	12	
0	7	14	
0	8	16	
0	9	18	
0	10	20	
0	11	22	
0	12	24	
0	13	26	
0	14	28	
0	15	30	
0	16	0	
0	17	2	
0	18	4	
0	19	**6**	
1	0	0	
1	1	2	
1	2	4	
1	3	**6**	

ODD FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	**R8=3**
0	1	3	
0	2	5	
0	3	**7**	
1	0	1	
1	1	3	
1	2	5	
1	3	**7**	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	**R8=3**
0	2	5	
0	3	**7**	
1	0	1	
1	1	3	
1	2	5	
1	3	**7**	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	**R8=3**
0	3	**7**	
1	0	1	
1	1	3	
1	2	5	
1	3	**7**	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	**R8=3**
0	4	9	
0	5	11	
0	6	13	
0	7	15	
0	8	17	
0	9	19	
0	10	21	
0	11	23	
0	12	25	
0	13	27	
0	14	29	
0	15	31	
0	16	1	
0	17	3	
0	18	5	
0	19	**7**	
1	0	1	
1	1	3	
1	2	5	
1	3	**7**	

EVEN FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	R8=3
0	5	10	
0	6	12	
0	7	14	
0	8	16	
0	9	18	
0	10	20	
0	11	22	
0	12	24	
0	13	26	
0	14	28	
0	15	30	
0	16	0	
0	17	2	
0	18	4	
0	19	**6**	
1	0	0	
1	2	2	
1	4	4	
1	6	**6**	

ODD FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	R8=3
0	5	11	
0	6	13	
0	7	15	
0	8	17	
0	9	19	
0	10	21	
0	11	23	
0	12	25	
0	13	27	
0	14	29	
0	15	31	
0	16	1	
0	17	3	
0	18	5	
0	19	**7**	
1	0	1	
1	2	3	
1	4	5	
1	6	**7**	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	R8=3
0	6	12	
0	7	14	
0	8	16	
0	9	18	
0	10	20	
0	11	22	
0	12	24	
0	13	26	
0	14	28	
0	15	30	
0	16	0	
0	17	2	
0	18	4	
0	19	**6**	
1	0	0	
1	2	2	
1	4	4	
1	6	**6**	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	R8=3
0	6	13	
0	7	15	
0	8	17	
0	9	19	
0	10	21	
0	11	23	
0	12	25	
0	13	27	
0	14	29	
0	15	31	
0	16	1	
0	17	3	
0	18	5	
0	19	**7**	
1	0	1	
1	2	3	
1	4	5	
1	6	**7**	

EVEN FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	
0	6	**6**	**R8=3**
1	0	0	
1	2	2	
1	4	4	
1	6	**6**	
2	0	0	
2	2	2	
2	4	4	
2	6	**6**	
3	0	0	
3	2	2	
3	4	4	
3	6	**6**	
4	0	0	
4	2	2	
4	4	4	
4	6	**6**	
5	0	0	

ODD FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	
0	6	6	**R8=3**
0	7	15	
0	8	17	
0	9	19	
0	10	21	
0	11	23	
0	12	25	
0	13	27	
0	14	29	
0	15	31	
0	16	1	
0	17	3	
0	18	5	
0	19	**7**	
1	0	1	
1	1	3	
1	2	5	
1	3	**7**	

Exit of IVM mode on CRTC 0 :

EXIT IVM MODE ON EVEN FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	**R8=3**
0	1	2	
0	2	4	
0	3	6	
1	0	0	**R8=0**
1	1	1	
1	2	2	
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	**R8=3**
0	1	2	
0	2	4	
0	3	6	
1	0	0	
1	1	2	**R8=0**
1	2	2	
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	**R8=3**
0	1	2	
0	2	4	
0	3	6	
1	0	0	
1	1	2	
1	2	4	**R8=0**
1	3	3	
1	4	4	
1	5	5	
1	6	6	
2	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	**R8=3**
0	1	2	
0	2	4	
0	3	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	**R8=0**
2	0	0	
2	1	1	
2	2	2	
2	3	3	

EXIT IVM MODE ON ODD FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	1	R8=3
0	1	3	
0	2	5	
0	3	7	
1	0	1	R8=0
1	1	1	
1	2	2	
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	1	R8=3
0	1	3	
0	2	5	
0	3	7	
1	0	1	
1	1	3	R8=0
1	2	2	
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	1	R8=3
0	1	3	
0	2	5	
0	3	7	
1	0	1	
1	1	3	
1	2	5	R8=0
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	1	R8=3
0	1	3	
0	2	5	
0	3	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	R8=0
1	4	4	
1	5	5	
1	6	6	
1	7	7	

19.8.2 CRTC 1

The calculation method of C9 is based on two principles. On the one hand the equivalence in number of lines between the even and odd frames, and on the other hand a conservation of the parity of lines.

Parity is fixed when R8=3 according to the parity of R9 and C9. In addition, if R9 is even (odd number of lines of a character), then the parity of C4 is also considered.

When the IVM mode is selected, the counting is carried out in 2 stages and depends on the R9 parity to allow for the treatment of all cases.

When C0 goes to 0:

- **C9 = C9+not (R9.0) (C9 is incremented if R9 is even)**
- **If (C9 and %11110) == (R9 and %11110) (test C9/R9 excluding parity)**
  Then
  C4 management (C4 ++ or C4 = 0 if C4 == R4)
  ParityC9= ParityC9 xor (not r9.0) (inversion of parity if r9 is even)
  C9 = ParityC9
  Else
  C9 = C9+1+(R9.0)
  End if

As soon as R8 returns to 0, the counting logic normally resumes.

If C9 == R9
Then
C9 = 0
C4 management (C4++ or C4=0 if C4==R4)
Else
C9 = C9+1
End if

19.8.3 CRTC 2

It is not necessary to reprogram R4 when R8 changes to 3. This is also the case when the IVM mode is programmed on a vertical fraction of the frame. C4 is not skewed when R8 is updated, even temporarily. R5, R6 and R7 also do not need to be reprogrammed for the frame to be synchronized.

In "Interlace" mode, C9 is compared with R9 in a conventional way to process C4.
Another counter, **C9.IVM**, is used for displaying and managing video pointer updating.

When R9 is odd, the video address is updated 2 times for each value of C4.

We can formulate it like this on the lines located after the one where R8 has gone to 3:

```
If (C9 == R9)
Then
        C9.IVM=ParityFrame
        C9=0 ; Management of C4 (C4++ or C4=0)
Else
        C9.IVM=C9.IVM+2
        C9=C9+1
End if

If (C9.IVM and &1e==R9 and &1e)
Then
        If (C0==R1)
        Then
                VMA'=VMA
        End If
        C9.IVM=ParityFrame
Fin Si
```

This management of **C9.IVM** takes place all the time, including when the IVM mode is not activated. Activation of the IVM mode during the line **immediately uses C9.IVM for the display**.

The specific management of assignment of VMA' with VMA when C0==R1 is only processed when R8 is equal to 3. When R8 is equal to 0 or 2, this assignment takes place only when C9==R9. In other words, C4 can increment without **VMA'** being updated in IVM mode.

C9.IVM is reset (0 or 1 depending on **ParityFrame**) twice during a C4 character, because it counts twice as fast as C9. Once when **C9.IVM** reaches R9 (out of parity) and once when C9 reaches R9. If R9 is even, C9 reaches R9 before **C9.IVM** reaches R9 (out of parity) and the VMA' video pointer is not transferred into VMA.

If, for example, C4=1 and R9 is 6 when R8=3, C9 will count from 0 to 6 whatever the parity of the frame. On even frames, **C9.IVM** will be 0, 2, 4, 6, 0, 2, 4 and on odd frames, **C9.IVM** will be 1, 3, 5, 7, 1, 3, 5. However, "**C9 .IVM and &1E**" will only equal **"R9 and &1E**" (6) once (when **C9.IVM**=6 or 7) and C4 will change to 2 without the video pointer being updated.

It should also be remembered that if the IVM mode is activated on the first line of a frame (When C4=C9=0) while the parity was odd, then the **C9** and **C9.IVM** are cleared on the 2nd line (**C9.IVM**=**ParityFrame** and **C9=0**).

The limit value of **C9.IVM** is always treated "out of parity", excluding bit 0 of **R9** and **C9.IVM** for the comparison test.

If R8 goes from 0 to 3 when C9=5 and R9=11, then **C9.IVM**=10 on an even frame (**C9.IVM**=11 on an odd frame).

If R8 goes from 3 to 0 when **C9.IVM**=8 and R9=11, then C9 goes to 4 if we were on the 5th line or 10 if we were on the 11th line.

This translation between **C9 and C9.IVM** is immediately considered during the creation of the displayed address, including during the line, from position C0 where R8 is modified.

The **BORDER** is activated when C4=R6. If we consider that a value of C4 allows to define 2 displayed characters, then the **BORDER** works in pairs.

The diagrams on the following pages describe different counting situations, when switching to IVM mode or when exiting IVM mode.

R9=7 and R8=0 before going to 3 (or returning to 0).

Switching in IVM Mode on CRTC 2 :

EVEN FRAME

C4	C9	C9-IVM	R8 UPDATE
0	0	0	R8=3
0	1	2	
0	2	4	
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	
1	7	6	

ODD FRAME

C4	C9	C9-IVM	R8 UPDATE
0	0	0	R8=3
0	1	3	
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	
1	7	7	

C4	C9	C9-IVM	R8 UPDATE
0	0	0	
0	1	1	R8=3
0	2	4	
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	
1	7	6	

C4	C9	C9-IVM	R8 UPDATE
0	0	0	
0	1	1	R8=3
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	
1	7	7	

C4	C9	C9-IVM	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	R8=3
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	
1	7	6	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	R8=3
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	
1	7	7	

EVEN FRAME

C4	C9	C9-IVM	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	R8=3
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	
1	7	6	

ODD FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	R8=3
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	
1	7	7	

C4	C9	C9-IVM	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	R8=3
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	
1	7	6	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	R8=3
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	
1	7	7	

C4	C9	C9-IVM	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	R8=3
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	
1	7	6	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	R8=3
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	
1	7	7	

EVEN FRAME

C4	C9	C9-IVM	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	
0	6	6	R8=3
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	
1	7	6	

ODD FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	
0	6	6	R8=3
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	
1	7	7	

C4	C9	C9-IVM	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	
0	6	6	
0	7	7	R8=3
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	
1	7	6	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	
0	1	1	
0	2	2	
0	3	3	
0	4	4	
0	5	5	
0	6	6	
0	7	7	R8=3
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	
1	7	7	

Exit IVM Mode on CRTC 2 :

EXIT IVM MODE

EVEN FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	2	
0	2	4	
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	R8=0
1	1	1	
1	2	2	
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

ODD FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	3	
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	R8=0
1	1	1	
1	2	2	
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	2	
0	2	4	
0	3	6	
0	4	8	
0	5	0	
0	6	2	
0	7	4	
1	0	0	
1	1	2	R8=0
1	2	2	
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	3	
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	R8=0
1	2	2	
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	2	
0	2	4	
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	R8=0
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	3	
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	R8=0
1	3	3	
1	4	4	
1	5	5	
1	6	6	
1	7	7	

EXIT IVM MODE

EVEN FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	2	
0	2	4	
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	R8=0
1	4	4	
1	5	5	
1	6	6	
1	7	7	

ODD FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	3	
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	R8=0
1	4	4	
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	2	
0	2	4	
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	R8=0
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	3	
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	R8=0
1	5	5	
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	2	
0	2	4	
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	R8=0
1	6	6	
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	3	
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	R8=0
1	6	6	
1	7	7	

EXIT IVM MODE

EVEN FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	2	
0	2	4	
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	R8=0
1	7	7	

ODD FRAME

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	3	
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	R8=0
1	7	7	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	2	
0	2	4	
0	3	6	
0	4	0	
0	5	2	
0	6	4	
0	7	6	
1	0	0	
1	1	2	
1	2	4	
1	3	6	
1	4	0	
1	5	2	
1	6	4	
1	7	6	R8=0
2	0	0	
2	1	1	

C4	C9	C9-VMA	R8 UPDATE
0	0	0	R8=3
0	1	3	
0	2	5	
0	3	7	
0	4	1	
0	5	3	
0	6	5	
0	7	7	
1	0	1	
1	1	3	
1	2	5	
1	3	7	
1	4	1	
1	5	3	
1	6	5	
1	7	7	R8=0
2	0	0	
2	1	1	

19.8.4 CRTC 3, 4

There is no documentation on the implementation of Interlace on these AMSTRAD-emulated CRTC's, however the logic is quite simple.

Remember that when R9 is updated with a value less than C9, then C9 goes to 0. As on CRTC's 0 and 1, activating IVM mode modifies the counting of C4, and therefore requires adapting the values of R4, R6 and R7 according to changes in C4.

In order for the IVM mode to follow a logic of even/odd lines, it is necessary to program R9 **in the same way as on a CRTC 0**. It is therefore necessary that R9 contains the number of character lines of character less 2. So 6 If a character is made up of 8 lines. If R9 is odd, C9's parity switches each time C4 changes.

The following algorithm describes the management of C9 and C4:

```
If C9 >= R9
Then
        If C4==R4
        Then
                C4=0
                ParityFrame=ParityFrame xor 1
                ParityC9=ParityFrame
        Else
                C4++
                If R9.0==1 (R9 is odd)
                Then
                        ParityC9=ParityC9 xor 1
                End If
        End If
        C9=ParityC9
Else
        If R8<3
        Then
                C9=C9+1
        Else
                C9=C9+2
                C9=C9 or ParityC9
        Enf If
End If
```

Example 1 :

On line N, R9=7, C9=6, C4=0, R4=10, ParityC9=even(0), ParityFrame=even(0)

Line N+1 C9<R9 (6<7)
C9=C9+2 (C9=8)

Line N+2 C9>=R9 (8>7)
C4=C4+1 (C4=1)
ParityC9=1 because R9 is odd (R9=7)
C9=ParityC9 (C9=1)

In this example, frame parity and C9 parity are even (because Parityc9=ParityFrame at the start of the frame). The C9 lines displayed at the start of the frame are aligned on the parity of the frame and for C4=0, we obtain the lines C9 = 0, 2, 4, 6, 8. When C4 is incremented (to go to 1), ParityC9 state is reversed (because R9 is odd, this in order to balance the lines between 2 frames, and obtain C4 composed of 5 even lines/ 4 odd lines and C4 of 4 even lines / 5 odd lines). On the character C4 = 1, we will therefore have the C9=1, 3, 5, 7.

Example 2 :

On line N, R9=7, C9=7, C4=9, R4=9, ParityC9=odd(1), ParityFrame=even(0)

Line N+1 C9>=R9 (7>=7)
C4=0 because C4==R4
ParityFrame=1
ParityC9=ParityFrame=1
C9=ParityC9=1

Line N+2 C9<R9 (1<7)
C9=C9+2 (C9=3)

In this example, we are on the last C4 of the frame. The parity of the frame is even and R9 is odd. During this frame, the C9s were even on the even C4 and the C9 were odd on the odd C4s. R4 being odd (number of even C4's, C4= 0 to 9) we have even C9 on C4=0 and odd C9 on C4=9. But on the new frame, the **ParityC9** state lines up with the **ParityFrame** state. On the new frame, we then have odd C9s with an even C4, and even C9 for an odd C4.

When R8 goes from 0 to 3 (mode IVM on), ParityC9 state is immediately assigned with the parity of the current C9: ParityC9 = C9.0 The parity of the C9 can therefore be in contradiction with the parity of the current frame until the next frame , where parity is then aligned with the parity of the frame.

The diagrams on the following pages describe different counting situations, when switching to IVM mode or when exiting IVM mode.

Note: The test IVM period covers only a few lines of the frame, after VSYNC, before VSYNC and before C4 reaches R6. R8 is 0 during VSYNC.

R9=6 or R9=7, and R8=0 before going to 3 (or going back to 0).

Switching to IVM mode on CRTC 3 & 4 :

R9=7, ODD OR EVEN FRAME

C4	C9	R8 UPDATE
0	0	R8=3
0	2	
0	4	
0	6	
0	8	
1	1	
1	3	
1	5	
1	7	
2	0	
2	2	
2	4	
2	6	
2	8	

R9=6, ODD OR EVEN FRAME

C4	C9	R8 UPDATE
0	0	R8=3
0	2	
0	4	
0	6	
1	0	
1	2	
1	4	
1	6	
2	0	
2	2	
2	4	
2	6	
3	0	
3	2	

C4	C9	R8 UPDATE
0	0	
0	1	R8=3
0	3	
0	5	
0	7	
1	0	
1	2	
1	4	
1	6	
1	8	
2	1	
2	3	
2	5	
2	7	

C4	C9	R8 UPDATE
0	0	
0	1	R8=3
0	3	
0	5	
0	7	
1	1	
1	3	
1	5	
1	7	
2	1	
2	3	
2	5	
2	7	
3	1	

C4	C9	R8 UPDATE
0	0	
0	1	
0	2	R8=3
0	4	
0	6	
0	8	
1	1	
1	3	
1	5	
1	7	
2	0	
2	2	
2	4	
2	6	

C4	C9	R8 UPDATE
0	0	
0	1	
0	2	R8=3
0	4	
0	6	
1	0	
1	2	
1	4	
1	6	
2	0	
2	2	
2	4	
2	6	
3	0	

R9=7, ODD OR EVEN FRAME

C4	C9	R8 UPDATE
0	0	
0	1	
0	2	
0	3	R8=3
0	5	
0	7	
1	0	
1	2	
1	4	
1	6	
1	8	
2	1	
2	3	
2	5	

R9=6, ODD OR EVEN FRAME

C4	C9	R8 UPDATE
0	0	
0	1	
0	2	
0	3	R8=3
0	5	
1	7	
1	1	
1	3	
1	5	
1	7	
2	1	
2	3	
2	5	
2	7	

C4	C9	R8 UPDATE
0	0	
0	1	
0	2	
0	3	
0	4	R8=3
0	6	
0	8	
1	1	
1	3	
1	5	
1	7	
2	0	
2	2	
2	4	

C4	C9	R8 UPDATE
0	0	
0	1	
0	2	
0	3	
0	4	R8=3
0	6	
1	0	
1	2	
1	4	
1	6	
2	0	
2	2	
2	4	
2	6	

Exit IVM Mode on CRTC 3 & 4 :

EVEN FRAME
EXIT IVM MODE, R9=7

C4	C9	R8 UPDATE
0	0	R8=3
0	2	R8=3
0	4	R8=3
0	6	R8=3
0	8	R8=3
1	1	**R8=0**
1	2	
1	3	
1	4	
1	5	
1	6	
1	7	
2	0	

ODD FRAME
EXIT IVM MODE, R9=6

C4	C9	R8 UPDATE
0	1	R8=3
0	3	R8=3
0	5	R8=3
0	7	R8=3
1	1	**R8=0**
1	2	
1	3	
1	4	
1	5	
1	6	
2	0	
2	1	
2	2	

C4	C9	R8 UPDATE
0	8	R8=3
1	1	R8=3
1	3	R8=3
1	5	R8=3
1	7	R8=3
2	0	**R8=0**
2	1	
2	2	
2	3	
2	4	
2	5	
2	6	
2	7	

C4	C9	R8 UPDATE
0	1	R8=3
0	3	R8=3
0	5	R8=3
0	7	R8=3
1	1	R8=3
1	3	**R8=0**
1	4	
1	5	
1	6	
2	0	
2	1	
2	2	
2	3	

C4	C9	R8 UPDATE
0	0	R8=3
0	2	R8=3
0	4	R8=3
0	6	R8=3
0	8	**R8=0**
1	0	
1	1	
1	2	
1	3	
1	4	
1	5	
1	6	
1	7	

C4	C9	R8 UPDATE
0	1	R8=3
0	3	R8=3
0	5	R8=3
0	7	R8=3
1	1	R8=3
1	3	R8=3
1	5	**R8=0**
1	6	
2	0	
2	1	
2	2	
2	3	
2	4	

EVEN FRAME
EXIT IVM MODE, R9=7

C4	C9	R8 UPDATE
0	0	R8=3
0	2	R8=3
0	4	R8=3
0	6	**R8=0**
0	7	
1	0	
1	1	
1	2	
1	3	
1	4	
1	5	
1	6	
1	7	

EVEN FRAME
EXIT IVM MODE, R9=6

C4	C9	R8 UPDATE
0	4	R8=3
0	6	R8=3
1	0	R8=3
1	2	R8=3
1	4	R8=3
1	6	**R8=0**
2	0	
2	1	
2	2	
2	3	
2	4	
2	5	
2	6	

C4	C9	R8 UPDATE
0	2	R8=3
0	4	R8=3
0	6	R8=3
0	8	R8=3
1	1	R8=3
1	3	**R8=0**
1	4	
1	5	
1	6	
1	7	
2	0	
2	1	
2	2	

C4	C9	R8 UPDATE
0	0	R8=3
0	2	R8=3
0	4	R8=3
0	6	R8=3
1	0	R8=3
1	2	**R8=0**
1	3	
1	4	
1	5	
1	6	
2	0	
2	1	
2	2	

C4	C9	R8 UPDATE
0	4	R8=3
0	6	R8=3
0	8	R8=3
1	1	R8=3
1	3	R8=3
1	5	**R8=0**
1	6	
1	7	
2	0	
2	1	
2	2	
2	3	
2	4	

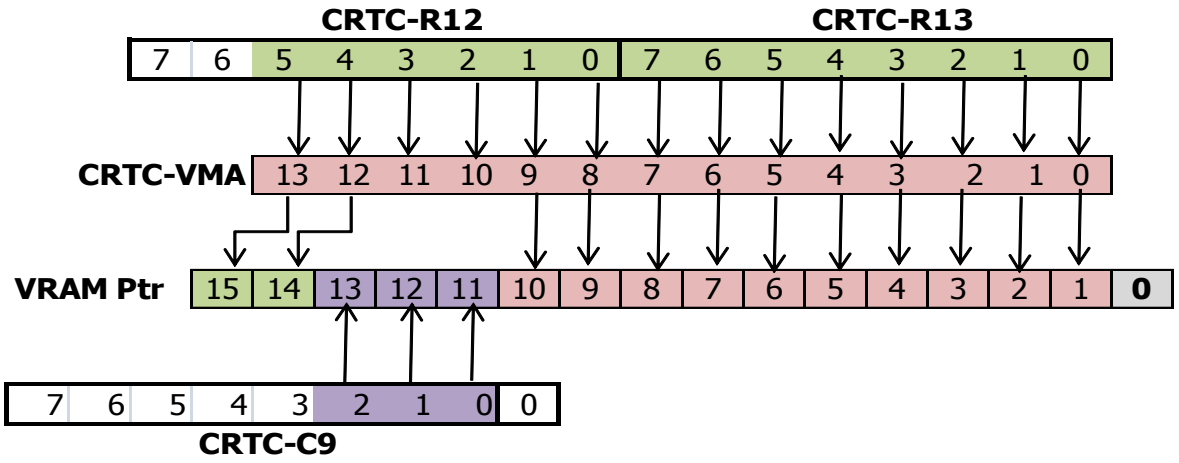
C4	C9	R8 UPDATE
0	2	R8=3
0	4	R8=3
0	6	R8=3
1	0	R8=3
1	2	R8=3
1	4	**R8=0**
1	5	
1	6	
2	0	
2	1	
2	2	
2	3	
2	4	

20 VIDEO POINTER:REGISTERS R12/R13

20.1 GENERAL

These two registers allow for the definition, in conjunction with C9, of the base memory address given by the CRTC to the GATE ARRAY for it to display its characters.

20.2 VIDEO POINTER CALCULATION



Bit 0	**Always at 0 because the CRTC works in words**
Bits 1 to 10	**From bits 0 to 9 of CRTC-VMA**
Bits 11 to 13	**From bits 0 to 2 of C9**
Bits 14 and 15	**From bits 12 and 13 of CRTC-VMA**

When the VMA/VMA' update conditions are met, then VMA or VMA'=R12/R13.
See Chapter 17.4, page 176.

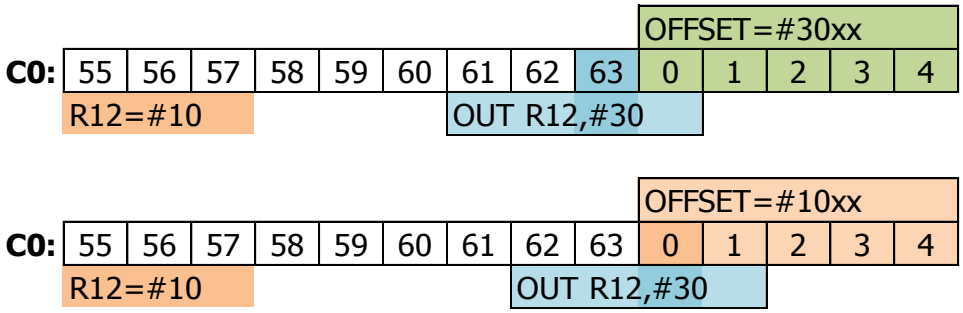
20.3 UPDATE CONDITIONS

20.3.1 CRTC 0

When the counters C4, C9 and C0 change to 0, the video pointers (VMA' & VMA) are initialized with R12/R13.

Note that if the "frames" are 2 µsec (R4=R9=0 and R0=1) then C4 passes through the values 0 and 1 alternately, allowing R12 and/or R13 to be considered every 4 µs.
(This is the triggering of the "R5" additional line management, which generates a line despite R5=0, and therefore increments C4 despite R4=0).

The updates of R12 and R13 are considered immediately.



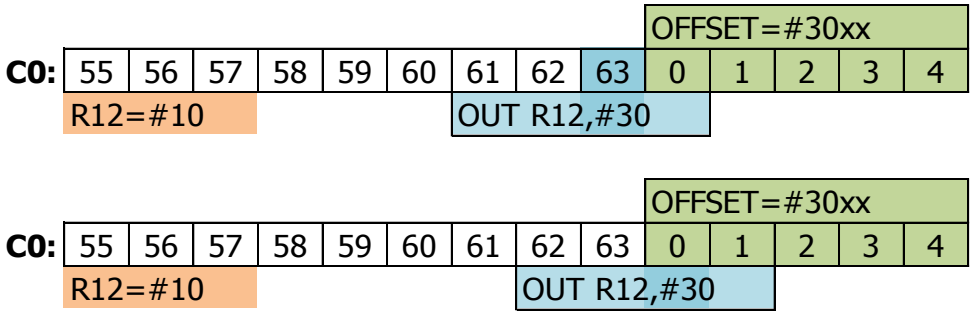
On CRTC 0, **VMA' & VMA are loaded with R12/R13 if C4=0 and C0=0.**

20.3.2 CRTC 1

When C0 and C9 go to 0 and C4=0, the VMA video pointer is initialized with R12/R13.

Note that the update of both R12 and R13 are immediate.

If the conditions are met, it is possible to change the offset on frames of 1 µs (R0=0).



On CRTC 1, **VMA is loaded with R12/R13 while C4=0.**

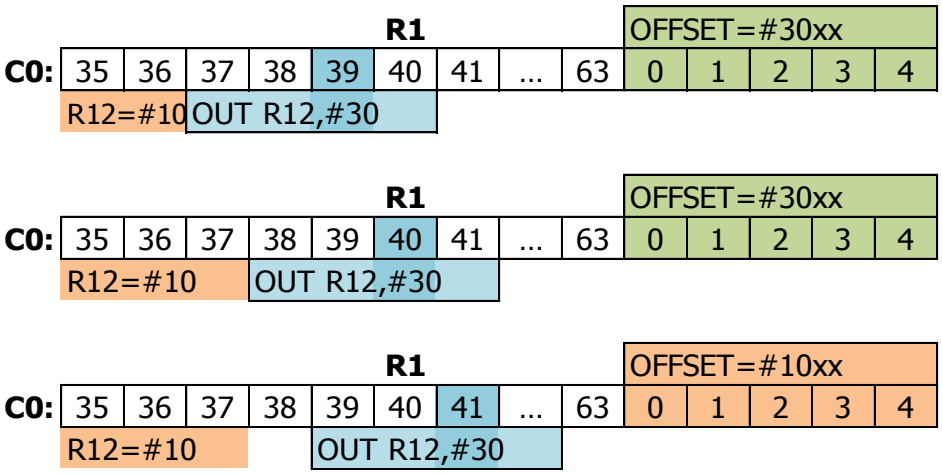
This particularity gives a certain flexibility because it is possible to modify the address over a period much greater than all the other CRTC's. To allow for compatibility with CRTC 1, it is therefore advisable not to change the address for the next frame too early, when C4=0. We can for example see this bug in terms of the display of vegetation in the game "**007 The Living Daylights**" (Domark, 1987). The address of the score zone is scheduled too early and replaces the address of the decor area containing the vegetation.

20.3.3 CRTC 2

When counters C4, C9 and C0 change to 0, pointers (VMA' & VMS) are initialized with R12/R13.

VMA' is a transient pointer updated with VMA when C0 reaches R1 (and C9=R9). It allows "to move forward" in the video ram when all the lines of a C4 character have been displayed. When the conditions of the last line of the frame are met (C4=R4/C9=R9 at the start of the line on C0=0), VMA'=R12/R13 (and not VMA'=VMA).

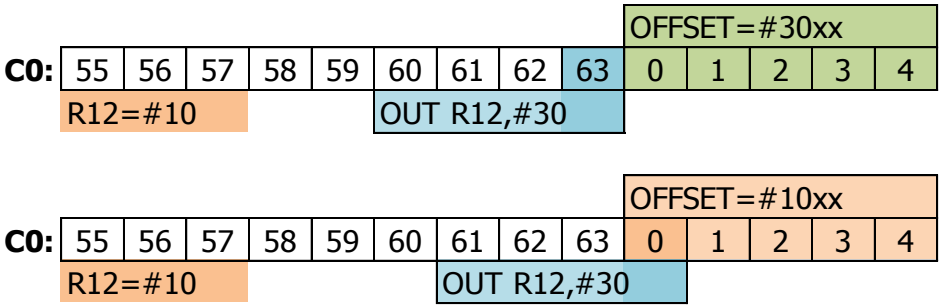
If the update of the R12/R13 registers is immediate, the consideration of the new line can no longer take place when C0 exceeds R1



The end of frame state is however evaluated after processing C0=R1 on position C0=0. Consequently, if R1 is 0 on the last line of the frame, VMA'= VMA at the beginning of this line. On the other hand, if the condition C0=R1=0 occurs on the first line of the new frame, the end of frame state is still true and VMA'=R12/R13 at the beginning of this line.

20.3.4 CRTC 3 & 4

When the counters C4, C9 and C0 change to 0, the video pointers (VMA' & VMA) are initialized with R12/R13.



On CRTC 3 and 4 VMA' & VMA are loaded with R12/R13 if C4=0 and C0=0.

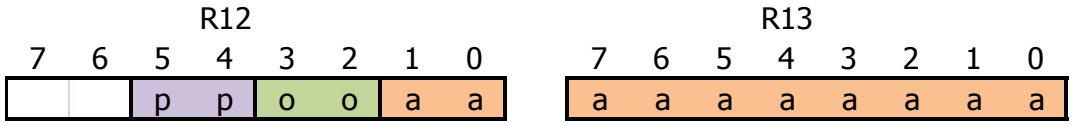
20.4 DEADLINES

See the chapter on the R0 register for the precise detail of the update of the video pointer.

20.5 OVERSCAN-BITS

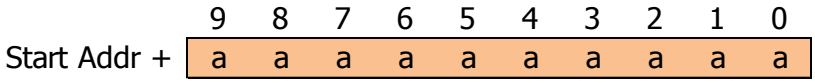
The construction of the memory address is specific insofar as:

- Bits from C9 are "immovable" in the final pointer.
- The internal address counter counts on 14 bits where bits 10, 11 and 12 are not "used" (at least in appearance).



p	p	Start address
0	0	&0000
0	1	&4000
1	0	&8000
1	1	&C000

o	o	Managed size
0	0	16 k
0	1	16 k
1	0	16 k
1	1	32 k



VMA always represents the character address.

Although bits 11, 12 and 13 of the final pointer come from C9, the VMA counter counts on 14 bits. Also, when the counting reaches the end of 10 address bits, bits 2 and 3 coming from R12 take part in the counting, even if they are not used directly for the video pointer.

If these bits are both at 1, this causes a report on bits 4 and 5 of R12.
These 2 bits represent bits 14 and 15 of the video pointer.

The consequence is therefore a change of page when the end of the 10 bits is reached.

It is therefore possible to display an image of more than 16k without resorting to a rupture with this method.

21 READ REGISTERS.

21.1 GENERAL

There are two types of reading possible on the CRTC's.

The first type aims to retrieve the value contained in certain CRTC registers. According to the CRTC's, this reading does not concern the same registers and the same ports.

The purpose of the second type is to retrieve the status of certain internal CRTC operations in order to communicate on the fulfilment of various conditions. Reading statuses does not exist on all CRTC's and does not use the same type of reading.

21.2 READING THE CONTENTS OF THE REGISTERS

In order to be able to read the contents of a register, it must first of all be selected.
As a reminder, an I/O port is selected with the I/O address &BC00.

21.2.1 CRTC 0

Only the 5 least significant bits of the register number are considered. The other 3 most significant bits are ignored. This CRTC is able to read the contents of the following registers on the port located at &BF00:

Register	Definition	Unit	r/w	7	6	5	4	3	2	1	0
R12	Display start address (High)	Pointer	r/w	0	0						
R13	Display start address (Low)	Pointer	r/w								
R14	Cursor address (High)	Pointer	r/w	0	0						
R15	Cursor address (Low)	Pointer	r/w								
R16	Light Pen (High)	Pointer	r	0	0						
R17	Light Pen (Low)	Pointer	r								

Note : The cursor is not managed on the CPC. However, it is perfectly possible to store values in R14 and R15 and then read them back.
An attempt to read another register (0 to 255) returns the value 0.

21.2.2 CRTC 1, 2

Only the 5 least significant bits of the register number are considered. The other 3 most significant bits are ignored. These CRTC's can read the contents of the following registers on the port located at &BF00:

Register	Definition	Unit	r/w	7	6	5	4	3	2	1	0
R14	Cursor address (High)	Pointer	r/w	0	0						
R15	Cursor address (Low)	Pointer	r/w								
R16	Light Pen (High)	Pointer	r	0	0						
R17	Light Pen (Low)	Pointer	r								

Note : The cursor is not managed on the CPC. However, it is perfectly possible to store values in R14 and R15 and then read them back.

For CRTC 2, an attempt to read another register (0 to 255) returns the value 0.
For CRTC 1, an attempt to read another register returns 0, except for register 31 (and all values whose bits 0-4 are 1) which returns a non-zero value (I got 127 or 255). This register was probably defined by UMC but not used on this model.

21.2.3 CRTC 3, 4

For CRTC's 3 and 4, only the 3 least significant bits of the selected register number are considered to read a register according to the following table.

Reading register 4 therefore also means reading register 12 (8+4) or 20 (16+4)
Reading a CRTC register is possible on two I/O addresses indistinctly: &BE00 and &BF00

Note that the "Register Status" port has the same function as the "Register Read" port. These CRTC's nevertheless manage an impressive collection of statuses. The designers used the R10 and R11 registers for this purpose (see next chapter on the statuses).

These CRTC's can read the contents of the following registers:

Nb	Register	Definition	Unit	r/w	7	6	5	4	3	2	1	0
0	R16	Light Pen (High)	Pointer	r	0	0						
1	R17	Light Pen (Low)	Pointer	r								
2	R10	Asic CRTC Status 1	Function	r								
3	R11	Asic CRTC Status 2	Function	r								
4	R12	Display start address (High)	Pointer	r/w								
5	R13	Display start address (Low)	Pointer	r/w								
6	R14	Cursor address (High)	Pointer	r/w	0	0						
7	R15	Cursor address (Low)	Pointer	r/w								

Note : The cursor is not managed on the CPC. However, it is perfectly possible to store values in these registers and then read them back, including both CRTC's 3 and 4.

21.3 READING STATUS

21.3.1 GENERAL

Only CRTC 1 has a status register present on the specific port &BE00.
This port is a mirror of the read port for CRTC's 3 and 4, which handle status differently.

CRTC	7	6	5	4	3	2	1	0
0	x	x	x	x	x	x	x	x
1	x	L	V	x	x	x	x	x
2	x	x	x	x	x	x	x	x
3	d	d	d	d	d	d	d	d
4	d	d	d	d	d	d	d	d

High impedance

High impedance

L	
1	Light pen reading
0	R16 / R17 registers can be read
V	
1	BORDER R6 is true
0	BORDER R6 is false

d : Mirror of BF00 port (read of CRTC reg)

Other CRTC	7	6	5	4	3	2	1	0
UM6845E	U	L	V	x	x	x	x	x
R6545E	U	L	V	x	x	x	x	x

U	
1	Update event
0	Reg R31 has been read / written by the MPU

21.3.2 CRTC 0, 2

These two CRTC's **do not have** a status register.
Therefore, it is not recommended to use the value read on port &BE00 on CRTC's 0 and 2, particularly to test the type of CRTC (risk of Candy Crush).
My CPC CRTC 2 always returns 255 read on this port.
My CPC CRTC 0 randomly returns 255 or 127 on this port.

21.3.3 CRTC 1

The input/output address of the status register on this CRTC is &BE00.

The UMC documentation specifies that bits 0 to 4 and bit 7 are unused, without further details.
Repeated readings of this register return 0 on these bits.

On CRTC 1, bit 5 of the Status register is updated when C0=R0 according to the BORDER R6 conditions (False: C4=C9=C0=0 / True: C4=R6 & C9=C0=0).

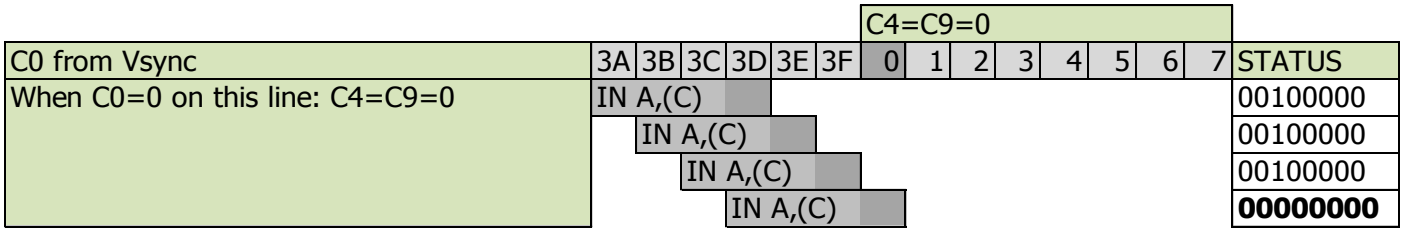
The bit is 1 when the BORDER R6 condition is true.
The bit is 0 when the BORDER R6 condition is false.

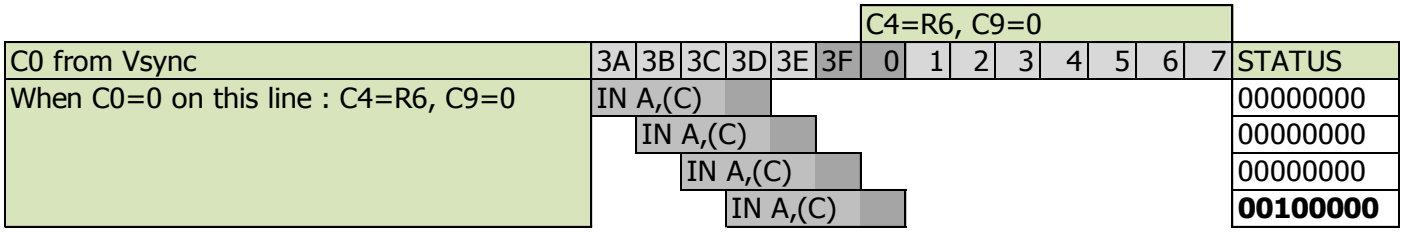
This does not necessarily mean that BORDER or CHARACTERS are displayed, because this evaluation is done when C0=R0.

But BORDER can already be displayed if the BORDER R1 condition is true.

Note also that if R6 is positioned at 0 (while C4>0) to generate BORDER, this state is not detected. The value 0, like in other CRTC 1 registers, is handled in a special way. In other words, if R6=0 while C4>0 and the status is 0 (Characters displayed), bit 5 of the status register will continue to be 0.

The diagrams below describe the exact transition to 1 or 0 of this status bit.





21.3.4 CRTC 3, 4

The designers of these ASIC's used R10 and R11 as status registers in order to trace a large number of events. It is therefore possible to wait for very precise events and to calculate quite simply the value of all the CRTC registers which are read-only.

The first identification of these statuses was made on CPC PLUS by Kevin THAKER (ArnoldEmu) who refers to them here: http://cpctech.cpc-live.com/docs/cpcplus.html
However, it was a mistake to state that these statuses do not exist on CRTC 4.

Reading these statuses requires great precision because several bits only change state for 1 µsec. In this case, if the status register is read with an **INI** or **IND** instruction, the I/O is read and written in RAM during the 4th µsecond of the instruction (which takes 5). When the register is read with the instruction **IN reg8,(C)**, the I/O is read during the 4th µsecond of the instruction (which takes 4). It is therefore necessary to perfectly calibrate the read instructions so that the 4th µsecond is located at the exact moment when the status is set by the CRTC.

Note : Since bit 3 of the register number is forced to 1, reading registers R2 and R3 is equivalent to reading R10 and R11.

21.3.4.1 Definition of STATUS 1

STATUS 1 (CRTC-R10)		
Bit number	**Bit Value**	**Event**
0	1	C0=R0
1	0	C0=R0/2
2	0	C0=R1-1 (if R0>=R1)
3	0	C0=R2
4	0	C0=R2+R3
5	0	R3h>0 : C0=0..R0 on the line R3h from Vsync (C4=R7)
	1	R3h=0 : C0=0..R0 over 15 lines from Vsync (C4=R7)
6	1	Always 1
7	0	C0=0..R0-1 : VMA.Lsb=0xFF
	0	C0=R0 : VMA'.Lsb=0x00 (same cond if C0=R0=0)

As a reminder, VMA' is loaded with VMA when C0==R1 and C9==R9.
Bit 7 is used to indicate that on the next CRTC character, the less significant byte of the video pointer will be reset to 0 (either from an overrun on the current VMA pointer, or when this pointer is going to be reloaded from VMA' at the end of the line)

21.3.4.2 Definition of STATUS 2

STATUS 2 (CRTC-R11)		
Bit number	**Bit Value**	**Event**
0	0	C4=R4 and C9=R9 and C0=R0 : Last char of screen
1	0	C4=R6-1 and C9=R9 and C0=R0 : Last char displayed
2	0	C4=R7-1 and C9=R9 and C0=R0 : Last char before Vsync
3	0/1	Timer 16 CRTC frames. See below
4	1	Always 1
5	0	C9=R9 : C0=0 to R0
6	0	Always 0
7	1	(C9=R9 and C0=R0) or (C9=0 and C0=0 to R0-1)

Bit 3 of status 2 toggles from 1 to 0 and vice versa over the entire frame every 16 frames.
With a RLAL, for example, this status toggles every 16 lines.

21.4 DUMMY REGISTER

Among some of the myths and legends about CPC CRTC's is the existence of Register 31, called the DUMMY REGISTER.

If register 31 does exist on the CRTC UM6845E from the company UMC, **it does not exist** on the CRTC UM6845R (type 1), nor on the CRTC UM6845 (type 0).

On the CRTC UM6845R, bit 7 of the status register which refers to it is simply unused.
On the CRTC UM6845, the status register itself does not exist (hence finding its bit 7…).

However, it should be noted that reading this register on a CRTC UM6845R (type 1) returns a non-zero value, unlike the other read-only registers.

On the CRTC UM6845E (excluding CPC), register 31 is related to the transparent mode.

The management of this mode also uses bits 6 and 7 of register 8 in addition to bit 7 of the status register. I will not dwell on this subject, which is as interesting as the programming of the EF9345P…

22 FULLSCREEN & CENTERING

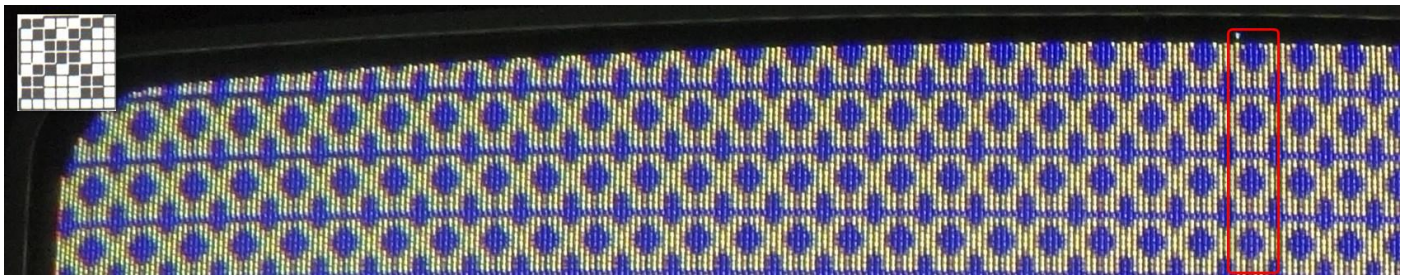
22.1 FOREWORD

CPC CRT screens are convex with quite a noticeable screen curvature.
Because of this shape and the deflection angle of the electron beam, the pixels at the edges are larger than those at the centre.

In addition, the case that surrounds the cathode ray tube has rounded corners and edges that "roughly" follow the shape of the cathode ray tube, but however masks a few additional pixels in passing. The data given in this chapter corresponds to the general case. Due to the analog nature of the screens, there may be some small variations in size or positioning between 2 screens. Let's not forget that the CPC is over 30 years old.

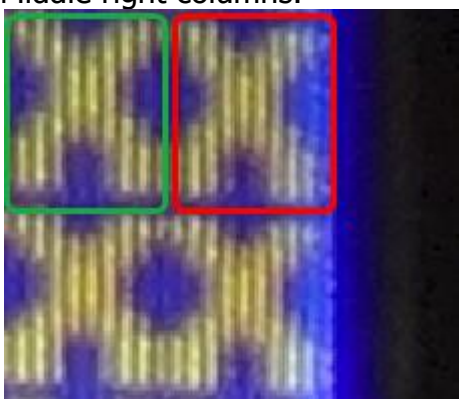
You can see the differences of several horizontal pixels and vertical lines between the centre of the screen and the edges of the latter.

Vertically, a difference of 5 lines can exist between the first line visible in a corner and the first line visible in the middle of the horizontal axis of the screen:

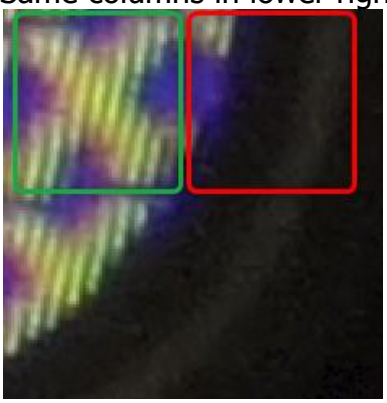


Horizontally, a difference of almost 1 CRTC character can be observed between a column located at the top or bottom side corner of the screen and this same column in the middle of the vertical axis of the screen (line 136):

Middle right columns.



Same columns in lower right corner.



It is by counting the number of horizontal characters and vertical lines from the middle of the horizontal and vertical axes of the screen that we can determine the values to be programmed to ensure that all the pixels visible on the screen are "filled". However, keep in mind that many pixels in these rows and columns will not be visible.

22.2 HORIZONTAL FULLSCREEN

On a CTM monitor, one can visualize a horizontal line of **48 CRTC characters**, which represents 96 bytes (192 pixels mode 0, 384 pixels mode 1, 768 pixels mode 2).

The centering of a horizontal line depends on the HSYNC signal (See Chapter 15, page 139).

The first character visible on the left is the 15th character from position C0=R2.

To obtain an exact centering on a line of 64 µsec (R0=63), R2 must be set to 50 (with R3>=6) to display C0=0 at the extreme left of the screen.

Depending on the H-HOLD setting of the monitor (accessible via a small flat screwdriver in the small hole at the rear left of the monitor), the BORDER may appear on the right or on the left if the line size is set to 48 characters.

If R3 is less than 6, then the frame is shifted to the right on the screen (See Chapter 14.4, page 126).

22.3 VERTICAL FULLSCREEN

On a CTM monitor, you can see **272 vertical lines** at 50 Hz (and much more with "homemade" interlace by lowering the frequency, but that's another subject).

The image begins to be visible from the **34th line**, which is the second line of the 5th character of 8 lines from the start of the VSYNC. (See Chapter 15, page 152).

If R7 is positioned with 35 while R4=38 (and R5=R8=0), then 33 “not visible” lines will be generated from the VSYNC, and the line C4=0/C9=1 will be partially visible.

23 TIPS AND TRICKS

As soon as it is necessary to modify the registers of several circuits, the 64 µsec of a line is often an important constraint to respect. Several tricks save valuable µsec. The main optimizations reside in the economical use of Z80A registers, the method of access to Inputs-Outputs or the management of iterations. These 3 approaches can be combined at will.

23.1 R12/R13 UPDATES

When it is a question of frequently modifying the values of the registers of certain circuits, it is for example not always necessary to modify the 2 registers R12/R13 when certain tricks are time critical, in general for demos.

It is possible to affect only one or the other, or even separately depending on the memory architecture of what must be displayed.

The access mode to the CRTC, which involves the prior selection of a register, is very CPU intensive. (Everything being relative…). An option to minimize the access time to these two registers is to use the **OUT (nn),A** instruction with all the reservations surrounding the use of this malicious instruction on a CPC

Example 1 :
The following code, for example, is used to initialize the pair R13/R12 quickly (considering of course the preloading of the registers). The principle is to allow the selection of a CRTC register without manipulating B several times to switch to the data writing function of the circuit.
Prerequisite : B=&BC, A=12, C=13, H=R12, L=R13
(optionally, SP=&BABA)

```
OUT (C),C       ; Select R13
INC B           ; B=Crtc Data Address
OUT (C),L       ; R13=L
OUT (#FF),A     ; Select R12 (12="combo" CRTC Reg Select & Value Reg Select)
OUT (C),H       ; R12=H
```

It is possible to select CRTC registers 8, 4 and 0 with this method. However, the last two values also send the value of register A to port A of the PPI

Example 2 : R0=0

```
LD B,#BD        ; B=Crtc Data Address
XOR A
OUT (#FF),A     ; Select R0
OUT (C),A       ; or OUT(C),0 (See Chapter 23.4)
```

In the context where only registers 12 and 13 of the CRTC are updated several times a tip consists in swaping the order of updating registers between each line in order to save the selection of one register. This allows to select one register by line and updare this register 2 times. (R12, R13 / R13, R12 / R12, R13 / ...).

This is the case for example when you modify the 2 registers on each line in the context of a line to line rupture (RLAL) on the CRTC 0, 1, 3 and 4:

Initial context : Selection R12
First line : Update R12 / Selection R13--Update R13
Second line : Update R13 / Selection R12--Update R12
Third line : Update R12 / Selection R13--Update R13
And so on...

Note: This principle of CPU gain for the selection of an index when several registers of the same circuit are sequentially updated is applicable to other circuits which manage a register selection index, such as the Gate Array or the AY- 3-8912.

Thus, for example, if it is necessary to change the color of several pen on each line, it is wise to order the order of the "common" Pen between each line:

First line : Select Pen 1--Color Pen 1 / Select Pen 2--Color Pen 2 / Select Pen 3--Color Pen 3
Second line: Color Pen 3 / Select Pen 1--Color Pen 1 / Select Pen 2--Color Pen 2
Third line : Color Pen 2 / Select Pen 3--Color Pen 3 / Select Pen 1--Color Pen 1
And so on...

23.2 COMMON USE OF REGISTER(S)

In an effort to save valuable Z80A registers, a few tricks are to put the value of the Input/Output ports or the value of the select registers into registers that also serve as pointers to tables. However, this limits access to tables whose most significant byte of the address, for example, is a valid value for the circuit.
However, this poses a significant constraint on the organization of memory.
For example, it is possible to place in memory at addresses &BC00, &BD00, &BE00, &7F00 tables containing useful values.

Example 1 :

```
LD B,#BD        ; BC is both CRTC write port
LD A,(BC)       ; and pointer to the value to write (indexed by C here)
OUT (C),A
```

Example 2:

```
LD B,#BE        ; BE is CRTC write port (via OUTI)
LD H,B          ; But also pointer to the value to write (indexed by L)
OUTI            ; HL being also incremented in the operation
```

This sport can also apply to selection registers, such as 12 when it comes to modifying R12 for example.

```
LD B,#BC
LD H,#0C        ; HL is a pointer to a table between #0C00 and #0CFF
OUT (C),H       ; But H also serves as a CRTC register selector
INC B
INC B
OUTI            ; The value present at address #0Cxx is sent to R12
```

23.3 WAITING VSYNC

It is sometimes interesting that B is already preloaded at the end of a "traditional" VSYNC wait. It is possible to use the **IN A,(#FF)** instruction to avoid B being involved, but this involves reloading A with the I/O address of Port B (PPI) at each iteration:

```
WSYNC   LD A,#F5                    LD B,#F5
        IN A,(#FF)          WSYNC   IN A,(C)
        RRA                         RRA
        JR NC,WSYNC                 JR NC,WSYNC
```

By loading A using another preloaded 8-bit register, the loop then occupies the same time as a "standard" loop with B because IN A,(#FF) executes in 3 µseconds

23.4 ZERO VALUE

There is an interesting undocumented instruction (#ED,#71) that sends the value 0 to the I/O address present in register B: **OUT (C),0**.

In principle, the value sent by this instruction depends on the type of MOS logic of the Z80A. On a NMOS variant that is used in the CPCs, 0 is sent, however on the Z80 CMOS variant, it is the value 255 which is sent.

Example :

```
LD BC,#BC09     ; See Chapter 12.4.2, page 94
OUT (C),C
INC B
OUT (C),C
OUT (C),0
```

23.5 OUTI/OUTD AND STATUS REGISTER

The official documentation is incorrect regarding the N and C bits of the Z80A F register for the OUTI and OUTD instructions. (This is probably also the case for OTIR and OTDR, but of limited interest on the CPC). When the sum of the value sent (in HL) to the circuit and the register L (post processing (incremented/decremented)) is greater than 255, then N=C=1. This potentially allows to test the end of a table without a counter, but it creates a constraint on the address of the table.

23.6 SELF-MODIFIED CODE

When a code is executed in RAM, it has an advantage compared to a code executed in ROM: it can modify its own object code. This has advantages in terms of code length and CPU. This is particularly the case when it comes to storing the value of a counter, which requires a variable.

Example of a function managing a counter:

Ram/Rom:

```
                LD A, (MyCOUNTER)       ; Loads the value of the counter in A
                DEC A                   ; Modifies the counter
                LD (MyCOUNTER), A       ; Back up the counter value
                RET
MyCOUNTER       DB 100
```

Ram:

```
MyCOUNTER       LD A, 100               ; Loads the value of the counter in A
                DEC A                   ; Modifies the counter
                LD (MyCOUNTER+1), A     ; Back up the counter value
                RET
```

The address of the **MyCOUNTER** variable is here present directly in the **LD A,N** operand (this instruction is coded **#3E, N**).
Operand is therefore 1 byte from the start of the opcode.

Depending on the instructions, the operand's offset to be modified may vary:
For an **LD H,value** the operand is located at Offset+1 of the start of the instruction.
For an **LD IXH, value**, the operand is located at Offset +2 of the start of the instruction.

Self-modification therefore involves considering the relative offset of operands in the instructions concerned.

Methodologies exist to avoid using "MyCOUNTER + offset" everywhere in a source because it can cause maintenance problems because it involves to modify the source everywhere where **MyCOUNTER** is used. (for example an **LD H, value** is relpaced by **LD IXH, value**).

The most used solution is to use an **EQU** directive which defines the variable relating to the code. In the previous example, we would have:

```
                LD A, 100           ; Loads the value of the counter in A
MyCOUNTER       EQU $ -1
                DEC A               ; Modifies the counter
                LD (MyCOUNTER), A   ; Back up the counter value
                RET
```

MyCOUNTER then points to the assembly address (indicated conventionally by the $) less 1. Assemblers can undoubtedly go further by offering a directive combined with **EQU** capable of providing the relative offset of the instruction operands according to their type.
Another example of an operand update on 16 bits is indicated in chapter 23.8 with the backup of the SP register.

The self-modification also has the interest of being able to modify the instructions themselves.
There are many possibilities.

Example 1: Inhibit or activate treatment.

```
                ;
                ; Activate treatment
                ;
                LD A,#37    ; "SCF" opcode
                LD (F_BitC),A
                ;
                ; Deactivate treatment
                ;
                LD A,#B7    ; "OR A" opcode
                ld (F_BitC),A

                ...
                ...
F_BitC          SCF
                JR C,Treatment1 / CALL C, Treatment1 / RET NC
```

A modern assembler should allow to recover the opcodes of an instruction.
For example with an "Opcode (" instruction ") directive.
Managing an instruction permutation could then be written:

```
                LD A,(F_BitC)                              ; 4 µs
                XOR (Opcode(« SCF ») xor Opcode(« OR A»))  ; 2 µs
                LD (F_BitC),A                              ; 4 µs
                ...
                ...
F_BitC          SCF/OR A                                   ; 1 µs
                JR C, Treatment                            ; 3/2 µs
```

The **XOR** is used here to swap from #37 to #B7 and vice versa.
It is also perfectly possible to modify the branching instruction itself.
For example transform a **JR C** (#38) into **JR NC** (#30).
Or even the branching value located in the operand behind the Opcode for a branching instruction (JR, JP, CALL).

If the permutation is a flip-flop treatment, we can also write:

```
                LD A,%01010101          ; 2 µs
FlipFlopSt      EQU $-1
                RRCA                    ; 1 µs
                LD (FlipFlopSt),A       ; 4 µs
                JR C,Treatment          ; 3/2 µs
```

It is possible for example to reverse the counting direction of a counter by transforming an INC into DEC, or even by deactivating the count via a NOP:

```
                LD A,100            ; Load the value of the counter in A
MyCounter       EQU $-1
CounterFunc     DEC A               ; DEC A / INC A / NOP
                LD (MyCounter),A    ; Save the counter value
                RET
```

Self-modification allows to easily reverse counting instructions, in order to avoid tests or the writing of a 2nd function:

- A XOR 1 on the opcodes INC A, INC B, INC C, INC D, INC E, INC H, INC L allows to transform them respectively into DEC A, DEC B, DEC C, DEC D, DEC E, DEC H, DEC L and Vice-Versa.

- A XOR 8 on the opcodes INCBC, INC DE, INC HL, INC SP allows to transform them respectively into DEC BC, DEC DE, DEC HL, DEC SP and Vice-Versa.

In a tense context with I/O, it is common to have multiple OUT (C),reg8.
It is interesting to remember here that these instructions takes 2 bytes in RAM and all start with the #ed prefix. It is possible to easily modify the affected register by modifying the byte which follows the prefix #ed (#79=A/#41=B/#49=C/#51=D/#59=E/#61=H /#69=L).
The #71 value allows to send the value 0.((with the Z80A mounted in the CPCs).

In the particular case of the instruction OUT (C),C self-modification allows to transform the instruction. It is for example possible to neutralize this instruction by modifying the prefix with a neutral instruction (NOP or LD Reg8, Reg8) but this presents the disadvantage of modifying the duration of the instruction, which then goes from 4 to 2 µsec.

In a fixed time context, the complete instruction can be replaced by #232B (INC HL/DEC HL). It is not recommended to modify the B register with #FF, because some CPC extensions had the brilliant idea of using only C to identify.

Replacing an OUT(C),C with an IN(C) (#ED70) is not recommended since an IN on CPC can cause an OUT with the value on the bus.

Finally, to close this chapter of the modified instructions, it is important to cite the ADD HL, BC (OPCODE #09) instruction that it is possible to transform into ADD HL, SP (OPCODE #39) and Vice-Versa. This can be very useful for a "unrolled" sprite routine when BC=#800 and SP=#C800+(R1X2). This avoids creating 8 different sprite display routines to manage all cases of a sprite higher than 8 lines and likely to be displayed on any line of a frame.

23.7 ITERATIONS AND UNROLLED CODE

When no further optimization is possible, a widely used method is to eliminate loop branch tests and instructions. However, this can lead to a significant increase in the ram required. The "unrolling" of the code can take place at the very level of the Z80A instructions, such as LDIR or LDDR.

Example:

```
LD BC,5                 ;  3 µs
LD HL,Source_pointer    ;  3 µs
LD DE, Dest_pointer     ;  3 µs
LDIR                    ; 29 µs ((6x4)+5)
                        ------
                         38 µs
```

```
LD HL,Source_pointer  ;  3 µs
LD DE,Dest_pointer    ;  3 µs
LDI                   ;  5 µs
LDI                   ;  5 µs
LDI                   ;  5 µs
LDI                   ;  5 µs
LDI                   ;  5 µs
                      -----
                       31 µs
```

Each "LDI" uses 1 µs less than if the operation had taken place within a "LDIR".
But each µSec gained thus costs 2 code bytes (an LDI is coded "#ED,#A0").
This can therefore be very consumer in memory (Rom/Ram) available.

Note: When the code works in RAM, it is however possible to create a "code that generates code" rather than using assembler "macros" that do this work. In the example above, some code could generate the LDIs to run in ram. This allows to obtain less voluminous object codes, and to put into perspective the notion of memory resource sharing. The area dedicated to generated code that can be reused by other functions.

An alternative to "all or nothing" consists in minimizing the amount of RAM necessary for the "unrolled" code by agreeing to lose a little CPU to "loop" a little less often. It is therefore a question here of making a ratio between the consumed RAM and the won CPU.

Example :

```
LD BC,2000              ; 3 µs                      ; 3 bytes
LD HL,Source_pointer    ; 3 µs                      ; 3 bytes
LD DE,Dest_pointer      ; 3 µs                      ; 3 bytes
LDIR                    ; 11999 µs ((6x2000)–1)     ; 2 bytes
                        ; ----------                ; ---------
                        ; 12008 µs                  ; 11 bytes
```

The above code occupies 11 bytes, and is carried out in 12008 µs.

If the LDIR had been fully unrolled for the 2000 occurrences defined in BC, the corresponding code would have lasted 10006 µs ((2000 x 5) +6).
But in return, this code would have occupied 4006 bytes in RAM or ROM. (2000x2+6).
Example of a mixed method where the periodic "repetition" of LDIR is simulated by a loop:

```
        LD HL,Source_pointer    ; 3 µs          ; 3 bytes
        LD DE,Dest_pointer      ; 3 µs          ; 3 bytes
        LD A,200                ; 2 µs          ; 2 bytes
LOOP
        LDI                     ; 5 x 200 µs    ; 2 bytes
        LDI                     ; 5 x 200 µs    ; 2 bytes
        LDI                     ; 5 x 200 µs    ; 2 bytes
        LDI                     ; 5 x 200 µs    ; 2 bytes
        LDI                     ; 5 x 200 µs    ; 2 bytes
        LDI                     ; 5 x 200 µs    ; 2 bytes
        LDI                     ; 5 x 200 µs    ; 2 bytes
        LDI                     ; 5 x 200 µs    ; 2 bytes
        LDI                     ; 5 x 200 µs    ; 2 bytes
        LDI                     ; 5 x 200 µs    ; 2 bytes
        DEC A                   ; 1 x 200 µs    ; 1 byte
        JR NZ,LOOP              ; (3 x 200)-1 µs ; 2 bytes
                                ; ------------   ; ------------
                                ; 10799 µs      ; 31 bytes
```

In the example above, the code occupies 31 bytes and consumes 10799 µsec.

If we had added 10 LDI (bringing their number to 20) by reducing the loop to 100, we would have a longer code (31 + 20 = 51 bytes) but faster (100 loops of 104 µs + 7µs = 10407 µs).

This principle can be applied to largely repeated portions of code.
It can be the case for example for code that deals with the display of sprites.
And especially during masking operations which consists in making a "hole" in the background (for example with AND operation) to place the sprite (with OR operation) before display.

23.8 UNCONDITIONAL BRANCHING

Instead of dealing with conditional branches, another common method is to use a list of execution pointers via the stack. Each **RET** instruction (3 µsec) causing the PC register to switch to a new routine (and SP to point to the next address).

This method generally requires preparing the table upstream, particularly to manage exit cases if this is not provided for elsewhere. This method nevertheless prohibits the use of interrupts (which of course you have foreseen).

Example :

```
EXEC_LIST1      DW FUNCTION1        EXEC_LIST2      DW FUNCTION2
                DW FUNCTION2                        DW FUNCTION1
                DW FUNCTION3                        DW FUNCTION3
;--------------------------------
FUNCTION1
                ...
                RET
;--------------------------------
FUNCTION2
                ...
                RET
;--------------------------------
FUNCTION3
                ...
SP_BACKUP1      LD SP,0                 ; Self modified code
                RET
;--------------------------------
MAIN
        DI
        LD (SP_BACKUP1+1),SP            ; Self modification of code
        LD SP,EXEC_LIST1                ; or EXEC_LIST2
        RET
```

In a situation where the time is critical, the instructions **JP HL, JP IX** and **JP IY** are valuables.
The first of these 3 instructions execute in 1 µsec (the other 2 with 1 µsec more).
L, LX or LY can be used as an index provided they remain in a 256 byte page defined by H, IXH, IYH.

Note: The ZILOG JP (HL/IX/IY) notation is misleading, because the jump takes place at the address contained in the HL/IX/IY register and not at the address contained at the address pointed to by HL/IX /IY. Recent assemblers accept these new notations.

Finally, and this is very anecdotal, but it is always possible to use CPC interrupts and arrange for R52 to generate an interrupt in order to interrupt a loop, considering that a non-repetitive instruction is not divisible. However, it must be considered that interrupts at this stage are not reliable between different CPCs when long instructions must be interrupted. (see chapter 26.7.2)

23.9 PAGES TREATMENTS

The Z80A remains an 8-bit processor, even if it is able to handle 16-bit values.

However, the instructions that use this possibility are slower than those that handle 8-bit values. It is thus faster to make an "INC L" (1 µs) than an "INC HL" (2 µs).

In addition, access to RAM are faster in indirect addressing.
LD A, (BC) or LD (HL), E will last 2 µsec for example.

An absolute addressing will be costly ('LD (address), A' 4µs and 'LD (address),DE' 6µs) and has no interest when it comes to treating a table or a structure.

The indexed addresses are quite indigestible because the index value "N" is stored in the instruction (LD A, (IX+N) 5µs). These instructions are also very slow because the pointer is calculated on 16 bits.

In a performance context, the architecture of the Z80A promotes the organization of data by pages of 256 bytes. Most assemblers integrate directives that align tables on page frontiers (when the less significant byte (LSB) of the address is 0).

It is thus possible to easily access the content of a table by handling the less significant byte of the address contained in BC, DE or HL. It is then considered that C, E or L are the indexes of a table of 256 bytes pointed out respectively by B, D or H.

Thus to define a data structure containing several values, the architecture by page offers very good access performance. Imagine (for example) that we want to define a structure which contains 2 types of data: Mxx of type Word (16 -bit) and Nxx of type Byte (8 bits).

And that's good, we need to have less than 256 times of this structure.

In principle we can define things as well as

```
MNStruct    DW M00          ; Index 0
            DB  N00
            ;
            DW M01          ; Index 1
            DB N01
            ;
            DW M02          ; Index 2
            DB N02
            …
            …
            DW M85          ; Index 85
            DB N85
```

Access to one of the **MNStruct** structures consists in multiplying the index of the structure to be achieved by the size of the structure (here 3 bytes) and adding the offset obtained to **MNStruct**. Then just read (or write) the values stored at the calculated address. Calculation and access to these structures involve working on 16 bits.

If **MNStruct** is aligned at the start of the page (for example the table begins in #4000), it is necessary to calculate the address on 16 bits if the total size of the structures exceeds 256 bytes.
In our example, this is the case because the index exceeds 84.
But it is also necessary to consider when the pointer on the structure calculated in HL, BC or DE is updated.

In our example the index structure 85 is located in #4000+(85x3) = #40ff
The LSB (less significant byte) of M85 is located on page #40, and the other 2 values of the structure on page #41. If contained #40ff, you must have a INC DE and not an INC E to access the following value with LD A,(DE) or LD (DE),A. Note however that if the table is less than 2 pages of 256 bytes, the exceeding of the page will only occur for one of the values of the structure. In the example, the INC DE will then only calculate the pointer on the MSB (Most signfificant byte) of M85, access to N85 requiring only a INC E.

If 256 is a multiple of the size of the structure, then it is no longer necessary to manage the increment of the current structure with 16-bit instructions, because no structure is then between 2 pages. For example, if the structure was 4 bytes, there would be exactly 64 structures per page of 256 bytes. It can therefore be relevant to add 1 byte to the structure to avoid having to use a 16-bit INC or ADD instruction.

Another way to organize the data is to define 3 tables each starting out on page frontier for each 8 -bit value declared in the structure:

```
TabMLow     DB M00Low
            DB M01Low
            ...
            DB M85Low

TabMHigh    DB M00High
            DB M01High
            ...
            DB M85High

TabN        DB N00
            DB N01
            ...
            DB N85
```

This organization greatly facilitates things and is very efficient.

If for example, **TabMLow** is located at #4000, **TabMHigh** is located in #4100 and **TabN** is located in #4200, then the LSB of the pointer designates the index and then the MSB of the pointer allow to switch between data.

With H=#40 and L=index, we can write:

```
LD E,(HL)   ; 2 µs ; Load TabMLow[Index] in E
INC H       ; 1 µs ; Switch to TabMHigh
LD D,(HL)   ; 2 µs ; Load TabMHigh[Index] in D
INC H       ; 1 µs ; Switch to TabN
LD C,(HL)   ; 2 µs ; Load TabN[Index] in C
```

This kind of organization is ideal when 256 structures are defined because there is no "waste" of memory. But if this is not the case, nothing prevents accommodation several structures within the same page. In our example, the first 86 bytes of each page are used, but nothing prevents managing another structure from offset 86 on each page.

We can also consider that a value contained in an indexed structure is the index of another table of structures. It's a pretty common thing. Always with the previous example, suppose that N is an index on another table located in #4300 (TAB2).
If B contains #43, we can write in the continuity of the previous example:

```
LD C,(HL)   ; 2 µs ; Load TabN[Index] in C
LD A,(BC)   ; 2 µs ; Load Tab2[TabN[Index]] in A
```

If by a lucky coincidence, it is possible to coexist within one page another index on this same page, we can then write:

```
;
LD L,(HL)   ; Load in L the R0 value, as an index to find R9
LD H,(HL)   ; Load in H the R9 value
```

On a smaller scale than a table aligned on a page of 256 bytes, we can consider that a data structure can start on an **even or odd address**. Assembler directives allows to respect this constraint. This allows to optimize the access of the data pointed to by HL, for example, by successively using **INC L / INC HL** if the start address of the data is **even**, and I**NC HL / INC L** if the start address of the data is **odd**. Indeed, it is certain that the page boundary overflow only occurs on an odd address (when L is 255).

This notion of alignment can be extended to quantities that are multiples of 256 bytes per page, in order to minimize the number of 16-bit increments. Thus if a table is aligned on a border of **16 bytes in a page of 256 bytes**, then one can use **15 times INC L and 1 time INC HL** (for a pointer defined in HL), since there will only be 16 times the probability that the zone thus defined is at the border of a page of 256 bytes. These techniques allows to minimize the use of 16-bit increments, which are twice as long as 8-bit increments.

23.10 BULK TIPS

23.10.1 LOOPS

When writing a loop that tests whether a 16-bit pointer has reached a certain value, one solution is to compare the MSB (More significant byte) and LSB (Less significant byte) respectively with the values to be reached. If the comparison is carried out with the MSB order first, the LSB of the pointer will be compared in a 2nd longer delay, and this until the condition of equality with the LSB is satisfied. On the other hand, if the comparison is carried out with the LSB, it is the MSB of the pointer that will be compared in the 2nd part of the test as many times as the value of the LSB will be reached. Depending on the value of the 16-bit pointer to be reached, it is therefore appropriate to reverse the order of testing the MSB and LSB:

```
LOOP
        INC HL
        LD A,H
        CP HIGH(pointer) ; MSB
        JR NZ,LOOP
        CP LOW(pointer); LSB 2nd test
        JR NZ,LOOP
```

```
LOOP
        INC HL
        LD A,L
        CP LOW(pointer) ; LSB
        JR NZ,LOOP
        CP HIGH(pointer) ; MSB 2nd test
        JR NZ,LOOP
```

If we want to test that a 16-bit register has reached 0, we can test it like this:

```
        LD BC, Nb_Occurrences
LOOP
        ...
        DEC BC          ; 2 µs      ; 1 byte
        LD A,B          ; 1 µs      ; 1 byte
        OR C            ; 1 µs      ; 1 byte
        JR NZ,LOOP      ; 3/2µs     ; 2 bytes
```

But we can also do this, which is just as fast but takes up less space, and doesn't modify A:

```
        LD BC, Nb_Occurrences+255
LOOP
        ...
        DEC BC          ; 2 µs      ; 1 byte
        INC B           ; 1 µs      ; 1 byte
        DJNZ LOOP       ; 4/3 µs    ; 2 bytes
```

In the event that HL needs to be incremented, it is also possible to write:

```
        LD BC, Nb_Occurtences
        LD HL, My_Pointer
LOOP
        ...
        CPI             ; 4 µs      ; 2 bytes
        JP PE,LOOP      ; 3 µs      ; 3 bytes
```

23.10.2 CALCULATION OF THE VIDEO POINTER

A great classic on CPC is to calculate the address of the following line in a "standard" formatted frame, with lines of 64 µs and R9=7.

C9's 3 less significant bits correspond to the 11.12.13 bits of the video pointer.
So, the offset of the video pointer in a page is C9 x #800.
To move to the next line on a frame which is not forced by a demonic rupture, it is therefore enough to add #800 to the current pointer.

When pointer is on the 8th line of a vertical character (C9 = 7), a specific calculation must be made for the pointer to return to the first line of the following caractor line (C9 = 0): (Pointer+(0 x #800)+(CRTC.R1 X 2)).

The most classic routine is as follows, with a few variants:

```
                Variant 1              Variant 2
NEXTLINE        LD HL,PtrVideo         ...
                LD A,H                 ...             ; 1
                ADD A,8                ...             ; 2
                LD H,A                 ...             ; 1
                RET NC                 ...
                LD BC,#C000+(R1x2)     LD A,L          ; BC is saved
                ADD HL,BC              ADD A,R1x2
                RET                    LD L,A
                                       LD A,H
                                       ADC A,#C0
                                       LD H,A
                                       RET
```

The page overflow test, which lets you know if the last line has been reached, is valid for a page located at #C000, but not on another page. Indeed, adding 8 to the MSB of the video pointer of a page located at #0000, #4000 or #8000 does not cause the pointer to overflow and does not set the C flag to 1. The table below describes the possible tests according to the different pages.

#0000	#4000	#8000	#C000

LD A,H
ADD A,8
LD H,A
ADD A,A
RET P
	LD A,H
	ADD A,8
	LD H,A
	RET P
		LD A,H
		ADD A,8
		LD H,A
		ADD A,A
		RET P
			LD A,H
			ADD A,8
			LD H,A
			RET NC

It is also possible to calculate the offset on a given page and then apply an offset correction according to the desired 16k page.

One solution is also to precompute row start pointers in a table.
An index then corresponds to the Y line number allowing the access to these pointers. But that implies to add position X to the pointer thus calculated.

Another method consists in modifying H thanks to the instruction **RES (2 µs)** and **SET (2 µs)**. However, we cannot access line in a linar way with this method using 1 single instruction each line. It is however possible by not respecting the order of the lines, but then requires organizing the graphic data in the order of their writing.
Example :

```
LD HL,#C000          ; HL=#C000 (C9=0)
LD (HL),%00000001    ;
SET 3,H              ; HL=#C800 (C9=1)
LD (HL),%00000011    ;
SET 4,H              ; HL=#D800 (C9=3)
LD (HL),%00001111    ;
RES 3,H              ; HL=#D000 (C9=2)
LD (HL),%00000111
SET 5,H              ; HL=#F000 (C9=6)
LD (HL),%01111111
SET 3,H              ; HL=#F800 (C9=7)
LD (HL),%11111111
RES 4,H              ; HL=#E800 (C9=5)
LD (HL),%00111111
RES 3,H              ; HL=#E000 (C9=4)
LD (HL),%00011111
```

Each line calculation takes 2 µs/2 bytes versus 3 µs/3 bytes considering a solution that sacrifices a register and A. With B = 8 offered in sacrifice, we would have between each line:

```
LD A,H      ; 1 µs
ADD A,B     ; 1 µs (with B=8)
LD H,A      ; 1 µs
```

One of the peculiarities of hardware scrolls is that video memory is cycling as long as the bits of pages are not affected (see chapter20.5). For example, if the memory is led to cycler on the 16K page located in #C000 (because each of the "Overscan Bits ™" are not 1), this means that the CRTC will display the byte located at #C000 after that displayed in #C7FF. And when it is necessary to recalculate the pointer which had reached the last line of a character (C9 = 7 in the example) it is necessary to consider when R1 x 2 is added to this pointer. In this case if the video pointer had exceeded #10000-(R1 x 2), it is necessary to correct the calculated pointer by anihiliating its bit 3:

```
NEXTLINE    LD HL,VideoPtr
            LD A,H
            ADD A,8
            LD H,A          ; VideoPtr+#800
            RET NC          ; Test Page Overflow for #C000 page
            LD A,L
            ADD A,R1x2
            LD L,A
            LD A,H
            ADC A,#C0
            RES 3,A         ; Video Pointer Correction
            LD H,A
            RET
```

Things can become particularly complicated for browsing the video ram when the pointer is at the border of the cycling zone. Since the memory is not linear, it is not always possible to simply correct the pointer, and in particular on odd C9s, as can be seen in the following table:

C9	Current pointer	Ptr+1	Calculated pointer	Correction	Corrected pointer
0	C7FF	INC HL	C800	RES 3,H	C000
1	CFFF	INC HL	D000	SET 3,H RES 4,H	C800
2	D7FF	INC HL	D800	RES 3,H	D000
3	DFFF	INC HL	E000	SET 3,H SET 4,H RES 5,H	D800
4	E7FF	INC HL	E800	RES 3,H	E000
5	EFFF	INC HL	F000	SET 3,H RES 4,H	E800
6	F7FF	INC HL	F800	RES 3,H	F000

7	FFFF	INC HL	0000	LD A,H
				OR #F8
				LD H,A
					F800

Performing a correction at each increment of an odd pointer to consider a situation that occurs once is extremely penalizing when it comes to displaying sprites on a scrolling hardware, for example. In addition, the most efficient sprite display methods require being able to browse the Video RAM in both directions, which represents a new complexity to manage.

A solution is to create specific display routines when the pointer on the data to be covered in the video RAM is on the cycling frontier. It is then necessary to identify that the processed line-char "contains" this frontier, and call a specific code to treat the transition.

For example, if it is a question of displaying a sprite 4 bytes wide (8 Pixels Mode 0, symbolized by color zones) on a line C9 = 0, then it should be provided that this Sprite can occupy the following locations in Video RAM:

C7FC	C7FD	C7FE	C7FF	C000	C001	C002	C003
C7FC	C7FD	C7FE	C7FF	C000	C001	C002	C003
C7FC	C7FD	C7FE	C7FF	C000	C001	C002	C003
C7FC	C7FD	C7FE	C7FF	C000	C001	C002	C003
C7FC	C7FD	C7FE	C7FF	C000	C001	C002	C003

For special cases indicated in orange, it is for example possible to manage two pointers from a table indexed by C9, in order to meet the requirements of a specific code. In the example, we would have, for example, HL=C7FF and HL '=C000 for C9 = 0, allowing the special code to switch between the two current addresses with the EXX instruction (1 µs). The particularity of the frontier would thus be circumscribed, allowing to avoid any pointer correction in the general case.

23.10.3 FLAG SETTINGS

More of a reminder than a real "trick".
The following table describes the instructions available according to some states of the register F

Asm	Définition	Flag	JR	JP	CALL	RET
Z	Zero	Z=1				
NZ	Not Zero	Z=0				
C	Carry	C=1				
NC	Not Carry	C=0				
PE	Even Parity	P=1				
PO	Odd Parity NPE	P=0				
P	Positive	S=0				
M	Negative (NP)	S=1				

The most often used states of F are C and Z.
Here are some instructions for positioning them:

Z Flag	C Flag	Instruction(s)	Update
1	0	CP A	
0	0	OR #F6	A=#F6
--	1	SCF	
--	0	OR A	
1	1	CP A + SCF	
0	1	SCF + SBC A,A	A=#FF

23.10.4 RATHER THAN...

NEG (2 bytes/2 µs) is equivalent to **CPL:INC A** (2 bytes/2 µs)

Rather than '**NEG :ADD A,d'** (4 bytes / 4 µs)
Prefer '**CPL :ADD A,d+1**' (3 bytes / 3 µs)

Rather than '**BIT 6,A :JP NZ,Kloug**' (5 bytes / 5 µs)
Prefer '**ADD A,A : JP M,Kloug**' (4 bytes / 4 µs)

Rather than **'LD A,reg8 : NEG'** (3 bytes / 3 µs)
Prefer **'XOR A :SUB reg 8'** (2 bytes / 2 µs)

Rather than **'LD A,0**' (2 bytes / 2 µs)
Prefer **'XOR A**' (1 byte / 1 µs) but only if you want to set Z to 1.

Rather than **'CP 0**' (2 bytes / 2 µs)
Prefer **'OR A**' (1 byte / 1 µs)

Rather than **'XOR #FF**' (2 bytes / 2 µs)
Prefer **'CPL**' (1 byte / 1 µs)

24 A BRIEF HISTORY OF FIXED TIME

24.1 INTRODUCTION

On CPC, the duration of each instruction is stable and linearized by the timing imposed by the GATE ARRAY on the Z80A when it needs to access memory. Unlike other processors, the speed of instructions does not depend on the order of the instructions or the type of ram from which they are read by the processor.

This particularity makes it quite easy to calculate the time taken by a function, since it suffices to know the number of µseconds taken by each instruction.
The shortest instructions are 1 µs. This is particularly the case of the NOP instruction, which is often used to define the duration of a set of instructions.

However, the duration of certain instructions may be different depending on the fulfillment of a condition processed by the instruction. This is particularly the case for relative conditional branches (JR cond,offset or DJNZ offset), function calls (CALL cond,address) or conditional function returns (RET cond). This is also the case for repetitive instructions (LDIR, LDDR, OTIR, OTDR) during the last iteration.

You can consult Chapter 25, page 272 to obtain on a single page the list of delays in µseconds of each instruction of the Z80A.

The duration of the code of a function can therefore vary according to the different conditions satisfied during its execution. Whether it is to make nice demos or certain demanding games, it is necessary to be able to create code for which we know the duration, and to ensure that this duration does not vary.

Indeed, torturing certain circuits requires them to be periodically solicited very precisely, often to the nearest µsecond. This is the case for the CRTC when it comes to certain barbaric techniques mentioned in this document which require precision of the order of one µsecond. This is also true for the GATE ARRAY when called upon to quickly change its color scheme or its graphic mode. Or even for the AY-3-8912 (the sound generator of the CPC) when trying to create new sounds.

24.2 METHODS

Writing a code in fixed time requires considering the elements likely to vary the duration of this code. In general, it is conditional branching, or loops with a number of variable iterations which are an obstacle to the achievement of this objective.

Without entering into a shared time logic, if you have a Z80A function whose time is variable, its passage in "fixed time" will require that it is the maximum period of this function which is the reference.

In other words, it is necessary here to introduce the concept of **compensation**.

For example we have a function A which last 40, 60 or 95 µs, depending on the situations.
This same fixed time function will always have to last 95 µs.
When the function will last 40 µs it will have to compensate with 55µs, and it will have to compensate 35 µs when it lasts 60 µs.

The writing of a compensation code is not always easy.
Some tips and tools can facilitate the task.

Example 1 :

```
FUNCTION_01             ; Not fixed time function
            OR A        ;  1          ; Test if A==0
            JR Z,ITSZERO ;  2  / 3
            LD HL,1234  ;  3  / 0
            LD A,2      ;  2  / 0
            RET         ;  3  / 0
ITSZERO
            LD A,1      ;  0  / 2
            RET         ;  0  / 3
                        ; 11 / 8
```

On the right, two columns symbolize the duration of the 2 execution branches of **FUNCTION_01**.

When the condition A>0 is filled **JR Z,ITSZERO** does not jump to **ITSZERO**.
The instruction nevertheless lasts 2 µs. The duration of each instruction is then indicated on the right in the first column, until the **RET**.
In this situation, the function lasted **11 µs**.

When A=0, **JR Z** jumps to **ITSZERO** in 3 µs.
In this situation, the function lasted **8 µs**.

The solution here is easy to understand.
The compensation must be 11-8 = 3 µs.
It is therefore possible to add 3 NOPs before the **RET**, and voila.
The function will then last **11 µs**.

The previous example is quite simple because the function uses 2 completely distinct branches of code. But things get complicated a bit if the jump instructions come back in common branches.

Example 2 :

```
FUNCTION_02             ; Not fixed time function
          LD A,(COUNTER) ;  4  / 4 ; Reading counter
          DEC A          ;  1  / 1 ; Counter - 1
          JR NZ,NOZERO   ;  2  / 3 ; Counter reach 0 ?
          LD A,10        ;  2  / 0 ; Yes reload with 10
NOZERO
          LD (COUNTER),A ;  4  / 4
          RET            ;  3  / 3
                         ; 16 / 15
```

In this example, the code from **NOZERO** is common.
The compensation here is more delicate to make because we cannot do it from **NOZERO**.

It is the code not executed when the condition is false (A=0) which extends the duration.

A technique consists in using a single instruction of 1 µs when the condition is not carried out. These include instructions such as **EXX / EX DE,HL / LD r8,R8 / INC r8**

```
FUNCTION_02             ; Fixed time function
          LD A,(COUNTER) ;  4  / 4 ; Reading counter
          DEC A          ;  1  / 1 ; Counter - 1
          LD B,10        ;  2  / 2
          JR NZ,NOZERO   ;  2  / 3 ; Counter reach 0 ?
          LD A,B         ;  1  / 0 ; Yes Counter reloaded with 10
NOZERO
          LD (COUNTER),A ;  4  / 4
          RET            ;  3  / 3
                         ; 17 / 17
```

Register B was sacrificed on the fixed time altar.
But **FUNCTION_02** will always last 17 µs. Mission accomplished !

We see here that a solution consists in preparing a **context** whose duration is greater than 1 µs (**LD B,10**) and which will be exploited by the instruction of 1 µs which follows the **JR Condition** (**LD A, B**).

Thus, **EXX** and **EX DE,HL** are particularly effective instructions.

```
FUNCTION_03             ; Not fixed time function
          OR A           ;  01 / 01
          LD HL,ADRESS2  ;  03 / 03
          LD DE,#4321    ;  03 / 03
          JR Z,ITSZERO   ;  02 / 03
          LD HL,ADRESS1  ;  03 / 00
          LD DE,#1234    ;  03 / 00
ITSZERO
          LD (HL) ,E     ;  02 / 02
          INC HL         ;  02 / 02
          LD (HL),D      ;  02 / 02
          RET            ;  03 / 03
                         ;  24 / 19
```

Here, the solution is also quite simple to manage with **EXX**.

```
FUNCTION_03             ; Fixed time function
            OR A        ;  01 / 01
            LD HL,ADRESS1 ;  03 / 03
            LD DE,#1234 ;  03 / 03
            EXX         ;  01 / 01
            LD HL,ADRESS2 ;  03 / 03
            LD DE,#4321 ;  03 / 03
            JR Z,ITSZERO ;  02 / 03
            EXX         ;  01 / 00 Context swap
ITSZERO
            LD (HL) ,E  ;  02 / 02
            INC HL      ;  02 / 02
            LD (HL),D   ;  02 / 02
            RET         ;  03 / 03
                        ;  26 / 26
```

In the case where only one context is not enough, it is possible to chain them as long as no instruction changes the condition state.
If we take the previous example, before the **RET**, we could have:

```
            ...
ITSZERO
            LD (HL) ,E  ;  02 / 02
            INC HL      ;  02 / 02
            LD (HL),D   ;  02 / 02
            LD A,10     ;  02 / 02
            LD B,20     ;  02 / 02
            JR Z,ZEROAGAIN ; 02 / 03
            LD A,B      ; 01 / 00 2nd context
ZEROAGAIN
            RET         ;  03 / 03
                        ;  36 / 36
```

Sometimes there are situations or the content of a register will determine a treatment:

```
            CP 0
            JR Z,TREATMENT0
            CP 1
            JR Z, TREATMENT1
            CP 2
            JR Z, TREATMENT2
            ...
```

If the values contained in A are close, a method consists in creating a treatment table indexed by A.

```
TREATMENT_TAB
            DW TREATMENT0
            DW TREATMENT1
            DW TREATMENT2
            ...
```

The connection code is quite simple and its execution is in fixed time :

```
LD L,A                  ; Index x 2
LD H,0
ADD HL,HL
LD BC, TREATMENT_TAB
ADD HL,BC               ; Ptr to TREATMENT_TAB[Index]
LD A,(HL)               ; HL=TREATMENT_TAB[Index].PtrExec
INC HL
LD H,(HL)
LD L,A
JP (HL)
```

If the values tested are not consecutive and cannot serve as an index, it is possible to create a table of values associated with addresses, and browse the entire table, **even when the value is found before having reached the end of the table**. For the value found, it is enough to retain the address found. The index search function must be in fixed time.

Creating a fixed-time generic index search function is a valuable ally in this quest

24.3 FIXED TIME AND INTERRUPTIONS

In general, it is rather better to deactivate the interruptions if you have not planned to count them. But it is possible to do with it..

This is not complex, insofar as the call in #38 (Mode IM 1) lasts 5 µs, regardless of the CPC. It takes 7 µs in IM 2. See chapter 26.4 for more information.

Many programs synchronize thanks to the interruptions, and in particular thanks to the HALT instruction. This method is often used to allows to define the code periods in "free time" before the interruption occurs, and then reserve the periods sensitive to the "fixed time" which occurs after the HALT instruction.

A difficulty is that many CPU optimization methods imply that the stack serves as a pointer to read data. However, the stack is used in particular by the instructions for calling functions (CALL, RST). And an interruption achieves an RST. Also the content of the stack is modified with the address of the instruction where the interruption was accepted by the Z80A.

The interruption code can plan to "correct" the RAM "perverted" to return to the interrupted code, but that requires knowing exactly what will be destroyed and where:

```
MY_INTER
                LD (BACKUP_HL+1),HL; Save HL
                POP HL              ; Get adress of interrupted code
                LD (RET_INT_ADR+1),HL ; Prepare end of interruption
                LD HL,xxxx          ; Corrective value of ram pointed by SP
                LD (yyyy),DE        ; Ram correction
BACKUP_HL       LD HL,#2121         ; Load HL
                EI                  ; Acq. interruption
RET_INT_ADR     JP #C3C3            ; Return to interrupted code
```

It is easier to deactivate interruptions with a simple DI.

However, this causes a bias, because if the interruptions should be authorized again, this cannot be done anywhere if they had been disabled for more than 52 lines. In this case, the reactivation of interruptions during the last 20 lines of the groups of 52 lines following leads to a discrepancy of the interruptions, and it is therefore necessary to consider it.

Finally, to finish this chapter, it is useful to specify that an interruption occurs at the end of a HSYNC. On CRTC 3 and 4, the HSYNC occurs 1 µs later than on CRTC 0, 1 and 2.

With the same R2 and R3 programming the interruptions therefore occurs 1 µs later on the CRTC's 3 and 4. A commonly used method consists in reducing R3 by 1 to allow compatibility. In addition, it is necessary to add to this discharge 1 µs more because the I/O CRTC of the instructions OUT(C),C on the CRTC's 3 and 4 occur 1 µs later (OUTI instruction not being concerned by this difference).

24.4 COMPENSATORY TOOLS

The examples of the previous chapter show that fixed time programming requires calculations. This can significantly increase the maintenance of a code.

Some assembler can incorporate directives to calculate the CPU of the instructions located between 2 labels and use the value thus found to "**compensate**".

Imagine for example the **CPU_START** and **CPU_END:<Variable>** directives.

Example of **compensation** for a function whose **period** is 64 µs.

```
            LD HL,NICE_COLOR_TABLE  ;
            LD B,#7F
            OUT (C),0               ; Select Pen 0
            LD A,20                 ; 20 lines of 64 µs each
LOOP_START
CPU_START:
            LD A,(HL)               ; 2 µs
            OUT (C),A               ; 4 µs
            INC HL                  ; 2 µs
            DEC A                   ; 1 µs
CPU_END: DelayCPU
            DEFS 64-DelayCPU-3,[NOP]
            JP NZ,LOOP_LINE         ; 3 µs
LOOP_END
```

In this example, the objective is that the code present between **LOOP_START** and **LOOP_END** always lasts 64 µs, whatever the updates likely to be carried out then between the **CPU_START** and **CPU_END** directives by the programmer.
The assembler will calculate the CPU in **DelayCPU** variable.
In our example, **DelayCPU is worth 9 µs.**
It is necessary to consider the 3 µs of the loop (the **JP NZ**), to define the number of NOPs to create for compensation (via **DEFS**) so that the duration is always equal to 64 µs.

Fixed time programming requires being able to "waste" time.

We can wait "for a long time", as is the case with the **wait_usec** function detailed at the end of chapter 7.2. This function is provided to wait for a quantity of µsec defined on 16 bits, but does not allow to define a waiting value of less than 49 µs.

It is possible to write a more precise function:

```
Wait_64     nop
...         ...
...         ...
Wait_13     nop
Wait_12     nop
Wait_11     nop
Wait_10     nop
Wait_09     nop
Wait_08     ret
```

Thus a **CALL Wait_08** will take very exactly 8 µs (the CALL and the RET).
If **Wait_64** is considered to be the reference period of this function, then the execution pointer for a waiting period between 8 and 64µs is **Wait_64+64-delay**.

For shorter time than 8 µs, it is possible to use neutral instructions.
Several of these instructions are described in chapter 24.7.

The calculation by an assembler of the CPU between 2 labels remains **very limiting to manage compensation**. It's useful, but it doesn't consider the reality of the global code execution. In addition, even by adopting programming reflexes in fixed time, the updates of a source can be tedious, even unmanageable.

There is a tool that allows to calculate the CPU located between two addresses in order to **automate compensation**. It was created to allow the democratization of this type of programming. Its implementation allows to design code in fixed time on the entire frame (19968 µs) without having to make tedious calculations. Moreover, a fixed time code over a period of 19968 µs makes waiting for VSYNC obsolete.

There is a big difference between the CPU calculated by an assembler directive (in order to create the code) and a tool that calculates the exact CPU of an already assembled code.
Indeed, the CPU depends on many factors that an assembler cannot manage (self-modified code, generated code, loops, conditional calls, ...).

It is possible to download the Z80A source from this tool here:
http://www.logonsystem.eu/html/downloadlogon.htm

To use it, you have to pass a start address (in HL) and an end address (in DE) for the code concerned. The function returns to BC the calculated number of µs. It is thus simple to be able to calculate the necessary compensation over a given fixed period of time.

In order to generalize the application of this tool to several "portions" of code, we can define a structure pointed out by IX and containing the call and compensation parameters:

```
CCPU_UNITY                  ; Unity Compensation of CPU
            ld l,(ix+0)     ; HL=start address of code
            ld h,(ix+1)
            ld e,(ix+2)     ; DE=end address of code
            ld d,(ix+3)
            ;
            call CalcCPU    ; Calculation of code duration in µs
            ;
            ld l,(ix+4)     ; HL=Expected time fixed duration of code
            ld h,(ix+5)
            or a
            sbc hl,bc       ; From which we subtract the calculated duration
            ex de,hl        ; DE= compensation duration (in µs)
            ld l,(ix+6)     ; HL=start adress of compensation context
            ld h,(ix+7)
            ld (hl),e       ; In which we store the compensation duration
            inc hl
            ld (hl),d
            ret
```

Example of code using this function:

```
                ld ix,SCCPU_FRAME     ; Définition of code to compensate
                call CCPU_UNIT        ; Update of « ld de,0 » at FRAME_COMP
                ;
                call sync_vbl         ; Waiting Vsync
FRAME_START                           ; 1ère µsec after vsync (C0vs)
                ...
                your code here
                ...
FRAME_COMP      ld de,0               ; xxxx µs (compensatory duration)
FRAME_END
                call wait_usec        ; 5 µs
                jp FRAME_START        ; 3 µs

SCCPU_FRAME     dw FRAME_START, FRAME_END
                dw 19968-8, FRAME_COMP+1
```

In this example, we want **FRAME_START** to be called all 19968 µs, regardless of the code in fixed time which is entered between the **FRAME_START** and **FRAME_END** labels.

The **CCPU_UNIT** function will do the calculation, then subtract the found result from 19968-8 (the 8 µs correspond to the duration of the **call wait_usec** and the **JP FRAME_START**). We cannot "integrate" the expectation of compensation in the calculation since the objective of the calculation is precisely to define the duration.

At the end of the function, the compensation value modified the operand of **"le de,0".**

In this example, the use of this function is simplified. In particular it does not manage the case where the code lasts longer than the maximum duration provided in order to alert the programmer that its code is too long..

In addition, in development mode, it is possible to manage compensation to the "**run**", but provide a **compensation manager** for the "delivered" code which calculates all the compensations of the project. This is how SHAKER is developed.

It is possible to manage **several code areas** within the same frame, **each with their compensation zones**. It is possible to encapsulate this function in a much more efficient compensation manager, capable in particular of using cascading compensations, starting with the lowest level functions. Sub-functions can thus be compensated, while it will itself be integrated into the compensation of higher-level functions.

It should never be forgotten that the cpu calculator "simulates" the operation of the code, and therefore updates the RAM (data, code) during its calculation. It is therefore necessary to think of resetting the data likely to have been modified during the simulation. As part of a compensation manager, this requires providing an initialization function in the event that code has been self-modified, to be called between each compensation of sub-level functions.

24.5 FREE FIXED TIME!

In a context of "cyclic" or "short" code, as can be demos, it is possible to overcome the programming rules in fixed time using the CPU calculator mentioned in the previous chapter.

For a demo whose duration would be 4 minutes with parts created in "free time", it is possible to calculate its compensation value for each frame.

Each minute requires 3000 compensation values. (50 frame/sec x 60 sec).

In order to avoid defining 3000 16-bit values (6000 bytes), it is possible to redefine the unit of time. If the maximum duration to be compensated is 19968 µs, then 256 CPU units of 78 µs (19968/256) can be defined. Finally, these values can also be compressed.

The dynamic calculation of this table is not possible as it is, because the calculator simulates the code and is therefore slower than direct execution.

It is possible to bypass this problem by segmenting the code with the different areas in fixed time, in order to allow to assemble, like a lego, the calculation of the values of the compensation table with the different bricks already calculated.

24.6 FROM FREE TIME TO FIXED TIME...

Transforming a program written "freely" into a fixed time can be complex.

Here is a method that I used to transform an AYC music player into a fixed time.
The source is available here: http://www.logonsystem.eu/html/downloadlogon.htm (TFixAYC.rar)

The general idea is to calculate the CPU in real time consumed by the Player, and to compensate for this duration at the end of the execution.

This implies on the one hand to define measurable time units in the code, even if it means transforming it a little to adapt certain sections to the available CPU units.

The higher the amount of Z80A registers allocated to this task, the more it increases the precision of calculation.

In the example, 5 registers are used to measure the CPU:
IXL=UT 24µs / C=UT 16µs / IYH=UT 10 µs / IXH=UT 8µs / IYL=UT 4µs

The minimum time unit is 4 µs, which is ideal for **compensation**.

Thus for a portion of code that uses 10µs, we will find an INC IYH, for example.

On the other hand, this implies calculating the **longest CPU** performed by the player according to the music used, in order to be able to compensate for the CPU from this value. This is why the player incorporates an option which allows to previously calculate the compensation value to be implemented in the final source (in the source this is the **CalcCpuMax** equivalence).

Once the maximum duration is known, it is enough to compensate for this value per unit of 4µs:

```
ccpu_wait04
                inc a                   ; 1 µs
                jr nz,ccpu_wait04       ; 3 µs
```

The real time calculation of the CPU degrades the overall performance of the player a little, but this degradation is nevertheless attenuated by the **compensation necessity**.

Thus, for the technical information, this player capable of managing 14 sound registers from an unprocessed AYC standard format, the cost of decompression with 8-bit buffers and the sending of one register to the sound generator is 61.21 µsec.

24.7 WASTING TIME...

When it is a question of waiting in a precise way, it is possible to use a series of NOP's (1 µsec) but this can considerably lengthen the code present in ram. If the goal is to minimize the size of a loop in ram, there are a few more or less "register neutral" instructions that take up little space and wait longer than the NOP instruction.

Beware of interrupts and stack contents (or registers) when data neutrality is handled with stack pointer associated instructions.

Instruction (s)	Duration (in µsec)	Size (in bytes)	Be careful !
NOP	1	1	
CP (HL)	2	1	Update F
JR $+2	3	2	
INC HL + DEC HL	4	2	
INC (HL) + DEC (HL)	6	2	Update F
PUSH HL + POP HL	7	2	Update stack content
EX (SP),HL + EX (SP),HL	12	2	

25 DURATION OF INSTR. ON THE CPC

The Z80A is interrupted by the GATE ARRAY via its READY pin when it reads the 2 bytes of the ram addressed by the CRTC every µ-second. The Z80A tests the state of its WAIT pin according to the definition of the different cycles of the instruction when it needs to access the ram or perform an input-output. The WAIT pin (#24) of the Z80A is driven low for 3 out of 4 cycles, causing each memory access to align when the WAIT signal is high again. The following table describes the Z80A instruction duration in µseconds.

Instruction	µs	Size		Instruction	µs	Size
ADC A,(HL)	2	1		DI	1	1
ADC A,(IX/IY+d)	5	3		DJNZ	4/3	2
ADC A, A/B/C/D/E/H/L	1	1		EI	1	1
ADC A,HX/LX/HY/LY	2	2		EX (SP),HL	6	1
ADC A,d	2	2		EX (SP),IX/IY	7	2
ADC HL,BC/DE/HL/SP	4	2		EX AF,AF’	1	1
ADD A,(HL)	2	1		EX DE,HL	1	1
ADD A,(IX/IY+d)	5	3		EXX	1	1
ADD A, A/B/C/D/E/H/L	1	1		HALT	1	1
ADD A, HX/LX/HY/LY	2	2		IM m	2	2
ADD A,d	2	2		IN A/B/C/D/E/H/L,(C)	4	2
ADD HL,BC/DE/HL/SP	3	1		IN A,(d)	3	2
ADD IX,BC/DE/HL/SP	4	2		IN F	4	2
ADD IY,BC/DE/HL/SP	4	2		INC (HL)	3	1
AND A,(HL)	2	1		INC (IX/IY+d)	6	3
AND A,(IX/IY+d)	5	3		INC A/B/C/D/E/H/L	1	1
AND A, A/B/C/D/E/H/L	1	1		INC HX/LX/HY/LY	2	2
AND A, HX/LX/HY/LY	2	2		INC BC/DE/HL/SP	2	1
AND A,d	2	2		INC IX/IY	3	2
BIT x,(HL)	3	2		IND	5	2
BIT x,(IX/IY+d)	6	4		INDR	6/5	2
BIT x, A/B/C/D/E/H/L	2	2		INI	5	2
CALL cond,aa	5/3	3		INIR	6/5	2
CALL aa	5	3		JP aa	3	3
CCF	1	1		JP cond,aa	3	3
CP A, (HL)	2	1		JP HL	1	1
CP A, (IX/IY+d)	5	3		JP IX/IY	2	2
CP A, A/B/C/D/E/H/L	1	1		JR a	3	2
CP A, HX/LX/HY/LY	2	2		JR cond,a	3/2	2
CP A,d	2	2		LD (BC/DE),A	2	1
CPD	4	2		LD (HL),A/B/C/D/E/H/L	2	1
CPDR	6/4	2		LD (HL),d	3	2
CPIR	6/4	2		LD (IX/IY+d), A/B/C/D/E/H/L	5	3
CPI	4	2		LD (IX/IY+d),d’	6	4
CPL	1	1		LD (aa),A	4	3
DAA	1	1		LD (aa),BC/DE/SP/IX/IY	6	4
DEC (HL)	3	1		LD (aa),HL	5	3
DEC (IX/IY+d)	6	3		LD A,(BC/DE)	2	1
DEC A/B/C/D/E/H/L	1	1		LD A/B/C/D/E/H/L,(HL)	2	1
DEC HX/LX/HY/LY	2	2		LD A/B/C/D/E/H/L,(IX/IY+d)	5	3
DEC BC/DE/HL/SP	2	1		LD A,(aa)	4	3
DEC IX/IY	3	2		LD A/B/C/D/E/H/L, A/B/C/D/E/H/L	1	1

Instruction	µs	Size	I/O	Instruction	µs	Size
LD A/B/C/D/E/H/L,d	2	2		RLD	5	2
LD HX/LX,A/B/C/D/E/HX/LX	2	3		RR (HL)	4	2
LD HY/LY,A/B/C/D/E/HY/LY	2	3		RR (IX/IY+d)	7	4
LD BC/DE/HL/SP,dd	3	3		RR (IX/IY+d),A/B/C/D/E/H/L	7	4
LD IX/IY,dd	4	4		RR A/B/C/D/E/H/L	2	2
LD SP,IX/IY	3	2		RRA	1	1
LD SP,HL	2	1		RRC (HL)	4	2
LD HX/LX/HY/LY,d	3	3		RRC (IX/IY+d)	7	4
LD BC/DE/HL/SP/IX/IY,(aa)	6	4		RRC(IX/IY+d),A/B/C/D/E/H/L	7	4
LD HL,(aa)	5	3		RRC A/B/C/D/E/H/L	2	2
LD I,A / LD A,I	3	2		RRCA	1	1
LD R,A / LD R,A	3	2		RRD	5	2
LDD	5	2		RST 0/8/10h/18h/28h/30h/38h	4	1
LDDR	6/5	2		SBC A,d	2	2
LDI	5	2		SBC A,(HL)	2	1
LDIR	6/5	2		SBC A,(IX/IY+d)	5	3
NEG	2	2		SBC A, A/B/C/D/E/H/L	1	1
NOP	1	1		SBC A, HX/LX/HY/LY	2	2
OR A,(HL)	2	1		SBC HL,BC/DE/HL/SP	4	2
OR A,(IX/IY+d)	5	3		SCF	1	1
OR A, A/B/C/D/E/H/L	1	1		SET x,(HL)	4	2
OR A, HX/LX/HY/LY	2	2		SET x,(IX/IY+d)	7	4
OR A,d	2	2		SET x,(IX/IY+d),A/B/C/D/E/H/L	7	4
OTDR	6/5	2	5*	SET x,A/B/C/D/E/H/L	2	2
OTIR	6/5	2	5*	SLA (HL)	4	2
OUT (C),A/B/C/D/E/H/L	4	2	3	SLA (IX/IY+d)	7	4
OUT (C),0	4	2	3	SLA (IX/IY+d),A/B/C/D/E/H/L	7	4
OUT (d),A	3	2	3	SLA A/B/C/D/E/H/L	2	2
OUTD	5	2	5*	SLL (HL)	4	2
OUTI	5	2	5*	SLL (IX/IY+d)	7	4
POP AF/BC/DE/HL	3	1		SLL (IX/IY+d),A/B/C/D/E/H/L	7	4
POP IX/IY	4	2		SLL A/B/C/D/E/H/L	2	2
PUSH AF/BC/DE/HL	4	1		SRA (HL)	4	2
PUSH IX/IY	5	2		SRA (IX/IY+d)	7	4
RES x,(HL)	4	2		SRA A/B/C/D/E/H/L	2	2
RES x,(IX/IY+d)	7	4		SRL (HL)	4	2
RES x,(IX/IY+d),A/B/C/D/E/H/L	7	4		SRL (IX/IY+d)	7	4
RES x, A/B/C/D/E/H/L	2	2		SRL (IX/IY+d),A/B/C/D/E/H/L	7	4
RET	3	1		SRL A/B/C/D/E/H/L	2	2
RET cond	4/2	1		SUB A,(HL)	2	1
RETI	4	2		SUB A,(IX/IY+d)	5	3
RETN	4	2		SUB A,A/B/C/D/E/H/L	1	1
RL (HL)	4	2		SUB A, HX/LX/HY/LY	2	2
RL (IX/IY+d)	7	4		SUB A,d	2	2
RL (IX/IY+d),A/B/C/D/E/H/L	7	4		XOR A,(HL)	2	1
RL A/B/C/D/E/H/L	2	2		XOR A,(IX/IY+d)	5	3
RLA	1	1		XOR A,A/B/C/D/E/H/L	1	1
RLC (HL)	4	2		XOR A, HX/LX/HY/LY	2	2
RLC (IX/IY+d)	7	4		XOR A,d	2	2
RLC (IX/IY+d),A/B/C/D/E/H/L	7	4		x=[0..7] d=[0..ff] m=[0..2]		
RLC A/B/C/D/E/H/L	2	2		aa=[0..ffff] a=[0..ff]		
RLCA	1	1		* some exceptions exist		

26 INTERRUPTS

26.1 GENERAL

Interrupts are generated on the CPC by the GATE ARRAY, according to parameters defined by the CRTC. The GATE ARRAY has a counter for this purpose whose objective is to count from 0 to 51 before returning to 0, which is often called **R52**.

This counter is incremented at the end of a HSYNC which is produced according to the parameters defined by the R0, R2 and R3 registers of the CRTC. There is no minimum HSYNC size for R52 to support. It is even if R3=1. However, due to the re-entrance protection mechanism of the HSYNC's, the minimum delay between 2 HSYNC's is 2 µsec (because it takes at least 1 µsec between 2 HSYNC's). This implies that the counter can loop in 104 µsec minimum.
Note that the re-entrance prevention mechanism is bugged on CRTC's from 1 to 4, which allows the creation of an infinite HSYNC depending on the method used (for example R0=R2=0 and R3=1).

In "standard" operation, with a HSYNC programmed every 64 µsec, 1 interrupt can occur every 3328 µsec (300 Hz).
On a European standard CPC, this represents 6 interrupts per frame:
6 x 52 lines = 312 lines.

26.2 MANAGEMENT OF R52 COUNTER

There are several particularities of management of the value of counter **R52** by the GATE ARRAY.

It can return to 0:

- When it exceeds 51.
- By setting bit 4 of the RMR register of the GATE ARRAY to 1.(10xIRrmm on #7f00)
- At the end of the 2nd HSYNC after the start of the VSYNC.

Bit 5 of counter R52 changes to 0:

- When an interrupt was armed (pending) and it is authorized while **R52** had continued to evolve.

Note: If the request to reset R52 counter to 0 is made via the RMR function of the Gate Array and this request takes place on the last µsecond of the HSYNC (C0=R2+R3-1), then the reset to 0 has priority over incrementation. If the request comes 1 µsecond before the end of the HSYNC (C0=R2+R3-2), then R52 is zeroed, then incremented on R2+R3-1.

26.3 TRIGGER CONDITIONS

For an interrupt to occur, it must be triggered.

26.3.1 TRIGGER WHEN R52 LOOPS TO 0

The GATE ARRAY sends an interrupt request when R52 reaches 51 and loops to 0.

- If the interrupts were authorized at the time of the request, then the interrupt takes place and bit 5 of R52 is set to 0 (which is not very useful because it was already equal to 0).
- If interrupts are not authorized, the counter continues to increment, but the interrupt remains armed (the GATE ARRAY then maintains its INT signal). When the interrupt is enabled (via Z80A EI instruction) then:
    - An interrupt occurs **after the instruction following the EI**. A HALT following an EI in this circumstance will be considered as 1 NOP.
    - Bit 5 of counter R52 is reset to 0 at the "real" end of this instruction. If bit 5 of the counter had reached 1 (R52>=32), then this is equal to "removing" 32 rows from the current counter.
      This has the effect of shifting the moment when the next interrupt can occur.
      It prevents less than 20 lines separating 2 interrupts.

26.3.2 TRIGGERING ON VSYNC

Two HSYNC's after the start of VSYNC:

An interrupt is requested by the GATE ARRAY from the Z80A only if bit 5 of R52 is 1. In general, R52 "returns" to 0 when 312 HSYNC's have occurred since the last VSYNC.
But if the frame is not formatted correctly, then the value of R52 may be different from 0.
As described in the previous Chapter, this rudimentary mechanism prevents two interrupts from occurring too closely together.

R52 is set to 0 unconditionally.

26.3.3 Z80A AND INTERRUPTIONS

The Z80A **EI** instruction is used to enable an interrupt to occur as soon as the interrupt is armed. When an interrupt occurs, interrupts are disabled until a new EI occurs. The Z80A's IFF flags are set to 0, which has the same effect as a **DI**.
However, the R52 counter continues to evolve.
If R52 returns to 0, a new interrupt is then pending, and will occur as soon as interrupts are authorized again. Only one interrupt can be pending.
In order to avoid re-entrancy problems in an interrupt code, the designers of the Z80A have imposed that a new interrupt cannot occur on the instruction following the EI.
This was done to allow the **RET, RETI, RETN** instructions to execute right behind the EI instruction, to prevent the stack from overflowing.

A repetitive sequence of the EI instruction will not allow an interrupt to occur before the end of that sequence, each EI deferring the interrupt.

The Z80A **HALT** instruction is used to wait for an interrupt to occur by repeating NOP's of 1 µsec indefinitely. If interrupts are not allowed when this instruction is executing, the Z80A will get stuck on this instruction. To my knowledge, few games have used the HALT instruction.

However, we can note that the game ***Trailblazer*** (published by "Gremlin Graphics" in 1986) stalls its "split rasters" with a HALT. Indeed, this instruction is interesting insofar as it makes it possible to wedge an interruption to the nearest microsecond.

Indeed, an interrupt cannot cut an instruction (except a repetitive instruction like LDIR or OTIR). So, if the instruction is several microseconds long, the interrupt will occur after the instruction completes.

26.4 INTERRUPT MODE 1

Most code written for the "OLD" AMSTRAD CPC uses the Z80A's IM 1 mode.
We switch to this mode using the **IM 1** instruction.

In this mode, when an interrupt occurs, the code is interrupted with an instruction equivalent to **RST #38.**

The address following the interrupted instruction (or the address of the instruction if it is a repetitive instruction) is put on the stack and PC=#38.

The Z80A RST #38 instruction lasts 4 µsec when called by code.
When an interrupt occurs, the call in #38 lasts 5 µsec.
[to test it, just compare the time taken by an RST #38 and the time taken by an interrupt with a fixed time code on 19968 NOP's]

26.5 INTERRUPT MODE 2

IM 2 mode, also called vectorized mode, is designed to allow several peripherals to generate interrupts, via a table of pointers to interrupt routines.

The address of the table is defined, for the most significant address, by the **I register of the Z80A**. The low order byte of the address is normally defined by one of the 128 possible peripherals by indicating its number on the 7 most significant bits, bit 0 equalling 0.

On the "OLD" AMSTRAD CPC, the least significant value of the address is undetermined (High Impedance value) and can vary from one CPC to another.
[I haven't checked if this value can vary while the CPC is on, but the interest in gaining ram is moderate and it's something to avoid]

In the perspective of moving the interrupt address elsewhere than in #38, it is nevertheless possible to use this mode with a few precautions:

It is necessary to create an interrupt vector table containing 257 bytes of the same value.
Indeed, bit 0 of the address of the selected table being unpredictable, this can cause the Z80A to read its vector on the 256th and 257th byte of the table.

If bit 0 is equal to 0, the high and low byte of the vector read from the table will be reversed with respect to a vector read from a table where bit 0 is equal to 1.
It is therefore advisable to create an interrupt vector where the most significant byte of the vector is equal to its less significant byte.

Example: Vector table created in #2000, which contains between #2000 and #2100, 257 times the #CA value. The interrupt then occurring in #CACA (fart!)

The game "***The Great Escape***", published by Ocean Software in 1986, used this mode to free up the first page of 256 bytes. The vector table starts at #100 (I=1) and occupies 257 bytes with the value #BC (for a vector located at #BCBC).

When an interrupt occurs, the call to the pointer located in the table lasts 7 µsec.

26.6 CRTC & INTERRUPTS...

26.6.1 GENERAL

A HYSYNC begins when the internal condition C0=R2 is met.

Let us recall once again that on CRTC's 0, 1 and 2, the HSYNC visually begins approximately 1 µsec before the display of the corresponding CRTC character.

The display of characters by the GATE ARRAY on these CRTC's starts again from C0 Disp=R2+R3-1 (except for R3=0 for CRTC's 0 and 1).

The measurement of the moment when the interrupt begins is carried out by considering C0vs from the VSYNC signal returned by the PPI.

On CRTC's 3 and 4, the HSYNC begins at the start of the display by the GATE ARRAY of the CRTC character corresponding to C0=R2.

As a general rule, an interrupt always starts 1 µsec after the end of the HSYNC regardless of the CRTC.

Given that the HSYNC starts 1 µsec earlier than expected on a CPC without ASIC (CRTC's 0, 1, 2) there is therefore a delay of 1 µsec with an interruption on a CPC with ASIC (CRTC's 3, 4).

With the same programming of R2 and R3, an interrupt occurs 1 µsec later on CRTC's 3 and 4 than on the other CRTC's, because the ASIC manages a HSYNC synchronous with the display.

HSYNC generated by an ASIC is more compliant with display management.
The following diagrams describe this management according to the CRTC's.

26.6.2 CRTC 0, 1, 2

R3=14					R2																								
C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C3:					0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Code interrupted 15 µsec after C0vs=R2									
																			R3										

R3=8					R2																								
C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C3:					0	1	2	3	4	5	6	7	8	Code interrupted 9 µsec after C0vs=R2															
													R3																

R3=1					R2																								
C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C3:					0	1	Code interrupted 2 µsec after C0vs=R2																						
						R3																							

26.6.3 CRTC 0, 1

R3=0					R2																								
C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

No interruption

26.6.4 CRTC 2

R3=0					R2																								
C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C3:					0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	Code interrupted 17 µsec after C0vs=R2							
																					R3								

26.6.5 CRTC 3, 4

R3=14					R2																								
C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C3:					0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Code interrupted 16 µsec after C0vs=R2									
																			R3										

R3=8					R2																								
C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C3:					0	1	2	3	4	5	6	7	8	Code interrupted 10 µsec after C0vs=R2															
													R3																

R3=1					R2																								
C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C3:					0	1	Code interrupted 3 µsec after C0vs=R2																						
						R3																							

R3=0					R2																								
C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C3:					0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	Code interrupted 18 µsec after C0vs=R2							
																					R3								

Note :

Since it is possible to generate "not active" HBLs for the monitor (when R3<=2), this implies the possibility of generating interrupts in the visible area.

26.6.6 PERSPECTIVE

The diagram below provides a perspective of a scheduled interrupt under the same conditions according to the different CRTC's.

It should provide an understanding of the mechanisms implemented from the reference **C0vs** character, considering the different deadlines implemented by the CRTC's.

When it comes to compatibility between the CRTC's, it is therefore necessary to check the differences relating to the delays to consider the registers and the effect of certain values, particularly if these registers are modified when a HSYNC is in progress.

	R3=14					R2																						
	C0 from Vs	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
CRTC 0, 1, 2	C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	C3:					0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Code interrupted 15 µsec after C0vs=R2									
CRTC 3, 4	C0 from GA	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	C3:					0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Code interrupted 16 µsec after C0vs=R2									
						1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16									

26.7 THREESOME...

26.7.1 R52 IN TIME ...

As we have seen in the previous Chapters, the GATE ARRAY "requests" an interrupt from the Z80A according to different conditions.

It is also responsible for managing its internal R52 counter by resetting it to 0, by incrementing it or by wildly eliminating its bit 5.
The incrementation of R52 depends on the CRTC which signals to the GATE ARRAY the end of the HSYNC.

The setting to 0 of bit 5 of R52 depends on the Z80A, which informs the GATE ARRAY that an interrupt has occurred. It is after the instruction following the EI of a pending interruption that the Gate Array will kill bit 5.

If R52 is 31, and the GATE ARRAY receives an end of HSYNC from the CRTC to increment R52, but at the same time it receives the order to eliminate bit 5, what happens?
The answer is not going to please emulator writers at the NOP.

Depending on the actual execution length of the instruction following the EI instruction, resetting bit 5 to 0 may take more or less time. The consequence is that R52 can be incremented before or after setting bit 5 to 0.

We can therefore have the following two situations:

- R52 goes from 31 to 32, then its bit 5 is eliminated, and R52 goes to 0.
- Bit 5 of R52=31 is eliminated (which has no effect) and R52 changes to 32.

In the first situation, the next interrupt cannot occur before 52 lines.
In the second situation, the next interrupt cannot occur before 20 lines.

Concretely, the GATE ARRAY activates the INT signal to indicate to the Z80A its interrupt request, which accepts it according to the state of an internal flag of the Z80 set by EI/DI (this flag is called IFF1, for "Interrupt Flip-Flop No. 1").

If the interrupt is accepted, the Z80A activates the IORQ and M1 signals, also connected to the GATE ARRAY.
Note that the end of the M1 signal during an interrupt occurs after the TWait cycles of the Z80A, in other words after the execution of the instruction following the EI.

26.7.2 RELIABILITY OF INTERRUPTIONS

The Gate Array is clocked at 16 MHz, and he is the conductor of the other circuits of the machine. It notably cadences the Z80A at 4 MHz.

The Gate Array manages certain signals from the CRTC, including its HSYNC signal. When the state of this signal changes, it allows the Gate Array to manage its R52 counter in order to position the INT signal of the Z80A when R52 has reached its limit. This treatment is a priori carried out on the first cycles of treatment of the Gate Array.

If the interruptions are authorized and the Int signal is activated, the Z80A will deal with an interruption after the end of the investigation.

Unfortunately, **CRTCs are not reliable concerning the management of their HSYNC signal**. In particular, the CRTC 1 for which disparities exist between models of the same type and different types. Thus, with equivalent programming, the end of the HSYNC signal can occur more or less late on a 1/16th MHZ scale. Even if the CRTC bus is 1 MHz, direct programming of registers or internal processing of signals is not necessarily aligned with this frequency.

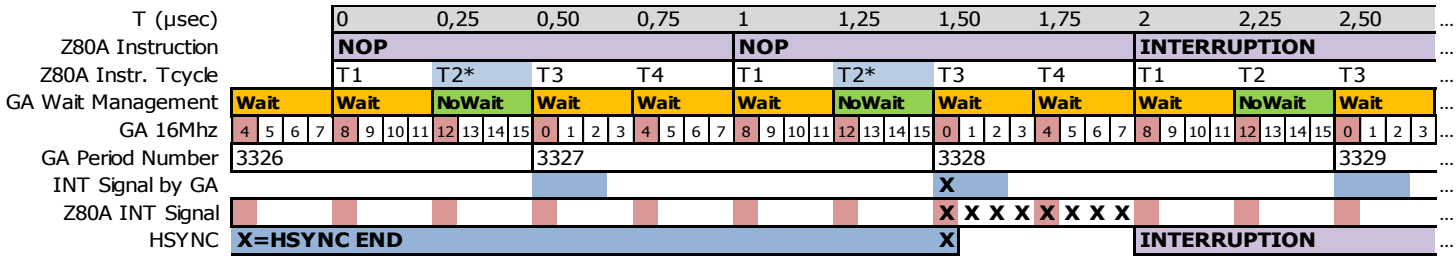
As a reminder, the Gate Array also positions the Wait signal of the Z80A for ¾ of its frequency (e.g. 12/16 MHz) and constraints the Z80A to add waiting cycles on the cycles provided for this within the M Cycles of the Z80A instructions directory. The Z80A is "free" of the motif imposed by the Gate Array for only 4/16th of MHz (0.25 µsec, 1 cycle T). This motif causes a time stretching of the instructions. (See Chapter 4.4.2).

If the CRTC activates the end of HSYNC during the last cycle T of an instruction (0.25 µsec), the delay is very short to allow the Gate Array to activate the INT signal early so that the Z80A consider it. If the end of HSYNC arrives too late, then the Z80A is not warned in time, and the interruption does not take place. According to the CRTC, the Z80A can therefore generate additional instruction before generating its interruption.

The timing logic of the Gate Array to align the cycles of the Z80A instructions allows for the avoidance of the consequences of these discordant HSYNC ends for certain instructions whose first cycles of the Gate Array are not on the last cycle T of the instruction of Z80A.

This is the case with NOP instruction (and by extension of its official supplier, HALT instruction).

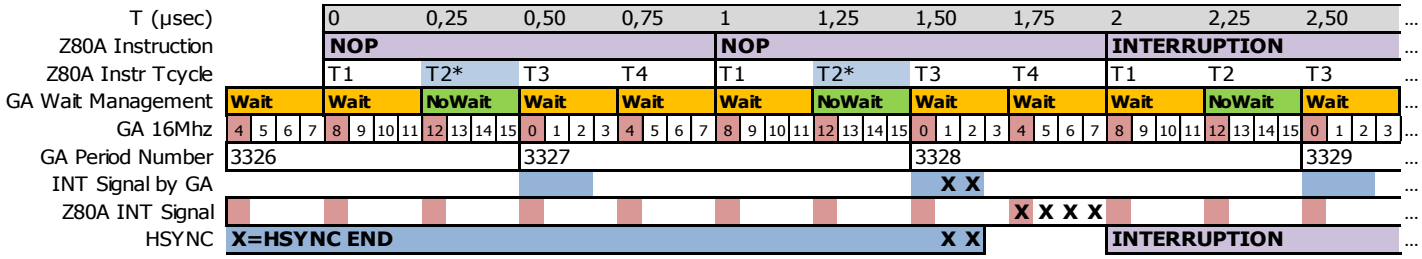
The end of HSYNC occurs before or on the first cycle of 1/16 MHz which begins under T3:



The Gate Array was on its last cycle before the end of HSYNC.
With R0 = 63, 64 µsec x 52 lines = 3328 cycles have passed since the last end of HSYNC.

The INT signal of the Z80A is positioned at T3 level and an interruption will take place at the end of the NOP.

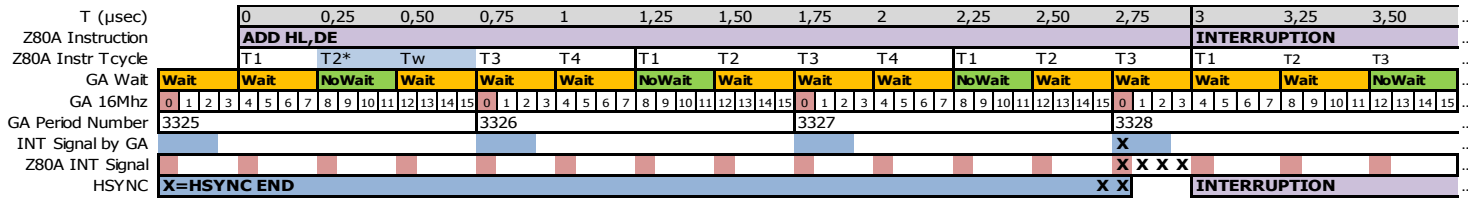
The end of HSYNC occurs after the first cycles of 1/16 MHz which begins under T3:



The end of the HSYNC arrived too late. The GA has processed the signal, but the Z80A will be informed about this at T4 cycle. The interruption will still occur at the end of the NOP instruction.

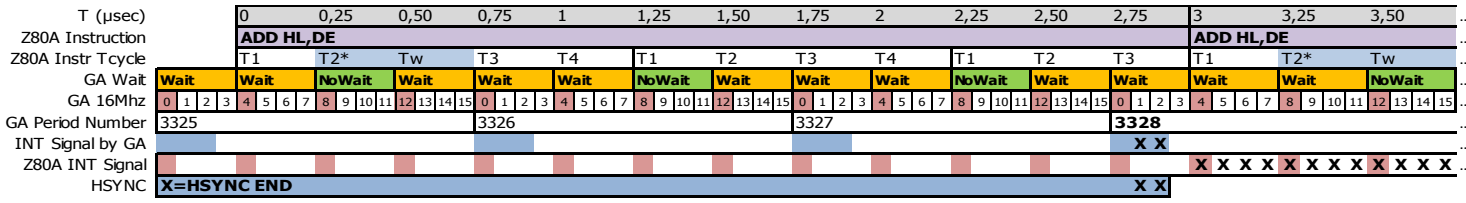
Unfortunately, there are many instructions whose last cycle T is positioned in front of the first cycles of the Gate Array. A simple delay of some 1/16th of MHz is then enough to scuttle an interruption.

Example with two consecutives ADD HL,DE instructions:
The end of HSYNC occurs before or on the first cycle of 1/16 MHz which begins under T3:



The Gate Array had time to consider the end of HSYNC to activate the INT signal of the Z80A, considered at the start of T3 cycle. The Z80A "knows" that it will have to generate an interruption and not to read the next instruction.

The end of HSYNC occurs after the first cycles of 1/16 MHz which begins under T3:



The Gate Array did not have time to consider the end of HSYNC so that the INT signal is transmitted to the Z80A in T3. Then, the Z80A will not be able to consider this signal and it will deal with the following instruction and will generate its interruption after this 2nd instruction.

An instruction like **set n,(ix+n')** officially lasts 23 cycles-T without being stretched. On CPC the alignment caused by the Gate Array lengthens this instruction to 27 or 28 cycles-T.
If an interruption is missed, it will then occur 7 µsec later according to the CRTC.

No difference's were noted between different CRTC type 0's, which led to this model being used as a benchmark. The CRTC's 2, 3 and 4 generally give results identical to the CRTC 0, with a few small differences.

The CRTC 1 has significant differences, even between different types 1.
In a 464, for example, it can behave like a CRTC 0.

The Gate Array 40008 can here have an impact here since this is an exchange problem between three circuits and that there are differences in the order of 1/16MHz between certain models of Gate Array.

However, tests were carried out on 6128 Multi-CRTC, which demonstrate these differences from the Gate Array 40010, also observed on unmodified machines.

In conclusion, it is risky to rely on the temporal position of an interrupt as it can occur at the very end of an instruction.

27 CRTC IDENTIFICATION

There are a plethora of methods for identifying a CRTC, most with code, others only visually.

Some methods sometimes involve frame desynchronization on the screen and are therefore less well regarded by some purists.

It is also possible to identify a CRTC by being able to identify the machine, as is the case with the CPC+, which has management differences in other circuits (and which have extended functions).

27.1.1 VIA C4 AND/OR C9 OVERFLOW

It is possible to overflow C4 in many different situations depending on the CRTC, whether by programming R0, R9, R4, R7 or even R8.
In general, on CRTC 0, for example, C4 is incremented by default.
In additional management, the value of C4 exceeds the value programmed in R4.
On a CRTC 0, this overrun will occur once.
On CRTC's 1 or 2, this overrun will take place several times depending on the content of R9 and R5.
On CRTC's 3 or 4, there will be no overtaking.
If C4 reaches R7, then a VSYNC may occur.

Example : Frame of 312 lines with R4=36, R9=7, R5=16
312=((36+1)x(7+1))+16
On a CRTC 3 or 4, if R7>36, then the VSYNC does not occur anymore.
On a CRTC 0, if R7>37, then the VSYNC does not occur anymore.
On a CRTC 1 or 2, if R7>39, then the VSYNC does not occur anymore.

27.1.2 VIA VSYNC MANAGEMENT DURING HSYNC

On a CRTC 2, setting R2 and R3 so that VSYNC occurs during the HSYNC period "turns off" VSYNC for the remainder of the C4=R7 period.
It is therefore sufficient to place R2+R3 so that it exceeds the value of R0

27.1.3 VIA CONSIDERATION OF VSYNC ACTIVATION

When the VSYNC is triggered by positioning R7 with C4, when C0>0, the line counter of the VSYNC is different according to the CRTC's. It starts from 0 on a CRTC 0, and from 1 on a CRTC 1, while no VSYNC occurs in this condition on a CRTC's 3 or 4.

27.1.4 VIA VSYNC LENGTH

Only CRTC's 0, 3 and 4 can manage the length of the VSYNC.
A VSYNC on CRTC's 1 and 2, which starts when R7 was programmed before C4=R7, lasts 16 lines.

27.1.5 VIA HSYNC LENGTH

On CRTC's 2, 3 and 4, the HSYNC lasts 16 µsec when R3=0.
On CRTC's 0 and 1, there is no HSYNC when R3=0.
There is therefore no interruption, which is one of the methods for testing the consequences of the value 0 on R3.

27.1.6 VIA THE BORDER, VISUALY

- CRTC 2 whose C0=0 during the HSYNC can no longer deactivate its BORDER.
- CRTC's 0 and 2, of which R6=0 is in conflict when C4=C9=0, alternates background and BORDER bytes (at least on the first line without adequate programming).
- CRTC's 0 and 2 will create a BORDER byte when C0 reaches R0 (and R1>R0).
- CRTC's 0, 3 and 4 can deactivate the BORDER (or add delays on considering the BORDER) with the SKEW BIT function of R8, whereas this function does not exist on CRTC's 1 and 2.

27.1.7 VIA THE INTERLACE MODE

C4 counting methods are different depending on the CRTC.
On CRTC 2, the count of C4 is not affected.
Being the only one, it is therefore detectable once the others have been identified.
On CRTC's 0 and 1, with R9 programmed respectively with 6 and 7 without having modified R7, the VSYNC occurs 2 times faster, since C4=R7 with characters of 4 lines instead of 8.

27.1.8 VIA STATUS REGISTER &BE00

Only on CRTC 1, is it possible to test the transition of bit 6 at the appropriate time.
On CRTC's 3 and 4, the port at &BE00 behaves like the one at &BF00 (see below)

27.1.9 VIA READ REGISTER &BF00

On CRTC 0, it is possible to read R12 and R13, as well as R14/R15 (cursor) and R15/R17 (light pen). The register number value is truncated to 5 bits. Thus, selecting register 108 is equivalent to selecting register 12.
On CRTC 1, this register returns 0 on all registers, except for register 31 (and all registers whose bits 0 to 4 are at 1). Values 255 and 127 were observed.
On CRTC 2, this register is used to read registers R16 and R17. The value 0 is returned on all other register numbers. The register number value is truncated to 5 bits.
On CRTC's 3 and 4, it is possible to read registers R16, R17, R10, R11, R12 and R13. The register number read comes from a table of 8 registers on 3 bits (see Chapter 20.3.4). Reading R4 or R20 is equivalent to reading R12.
On CRTC's 3 and 4, reading R10 and R11 (or equivalent for the first 3 bits) returns values internal to the ASIC (see Chapter 21.3.4).

27.1.10 VIA R10/R11 STATUS REGISTERS

Only on CRTC's 3 and 4, it is possible to test the update of many bits at the appropriate time, such as for example bit 0 of status 1 which is worth 1 when C0=R0 (0 otherwise). Furthermore (and subject to additional tests), bit 3 of status 2 should allow to differentiate between CRTC 3 and CRTC 4.

28 CPC IDENTIFICATION

It is possible to identify CPC models from differences which are not related to the CRTC or ROM content.
This is the case for the 2 machines which have ASIC's:
CPC PLUS ASIC 40489 CRTC 3
CPC LOWCOST ASIC 40226 CRTC 4

28.1 IDENTIFICATION METHODS

28.1.1 ENABLING EXTENDED FEATURES

CPC PLUS : Activation via an unlock sequence sent to the CRTC.
CPC LOWCOST : No extended functions (without the secret sequence).
OTHER CPCS : No extended functions.

28.1.2 BUG PPI PORT C

CPC PLUS : When the command register is used to configure port C, the bits of this port are reset to 0. The ASIC poorly emulates the PPI and does not reset these bits to 0. Very annoying for the compatibility of keyboard routines between the CPC PLUS and older models.
CPC LOWCOST: The 8255 PPI is not emulated by the 40226 ASIC. These CPC's have a PPI and therefore behave like the first generation of CPC's.
OTHER CPCS: Port C bits are normally zeroed by the command register.

28.1.3 BUG PPI PORT B

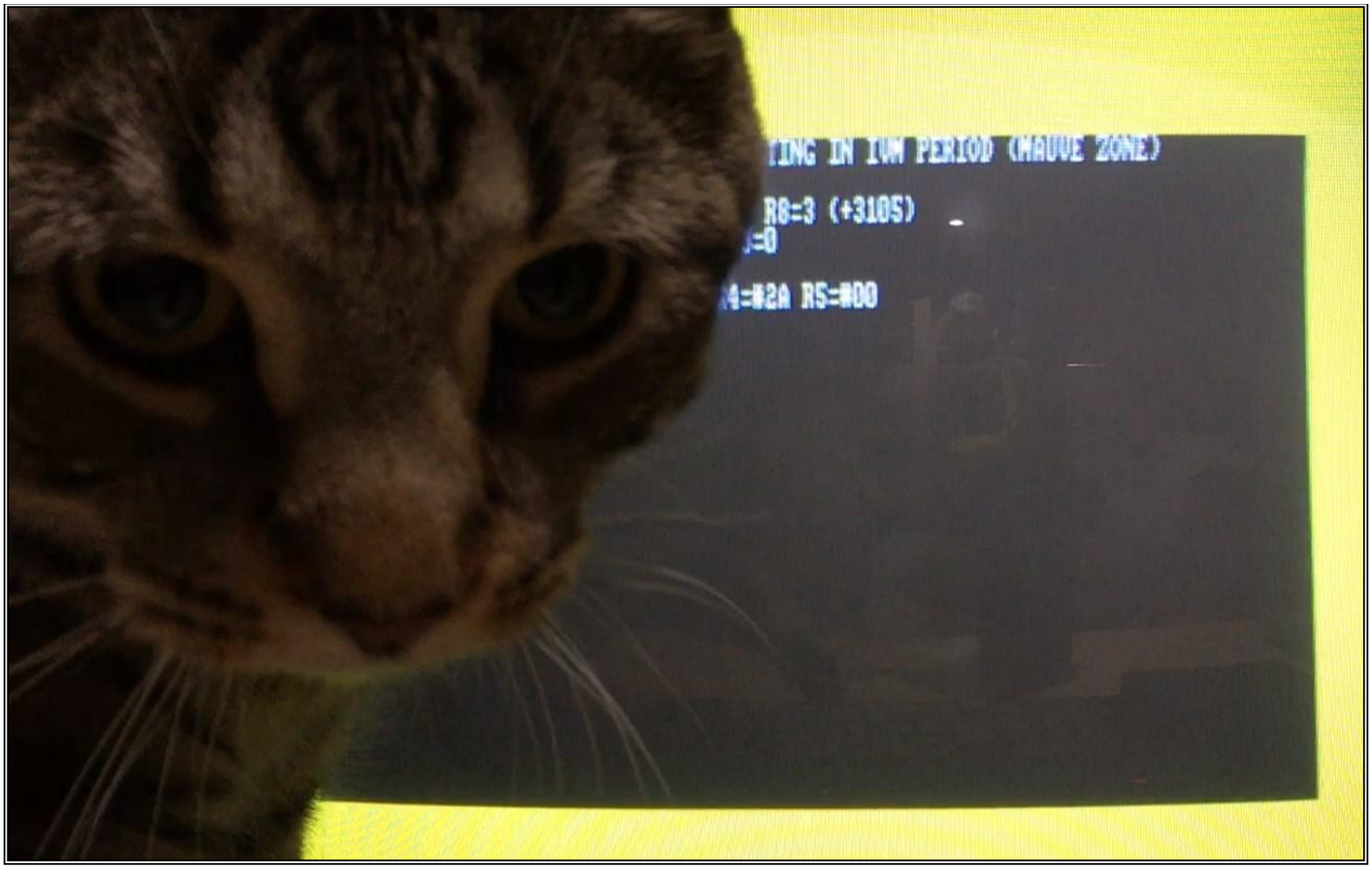
On a PPI, you can program the direction of port B (input or output).
At the output, we can therefore place a value there, which we can read again.

The PPI of CPC LOWCOST would not make it possible to read back a value stored in the port B placed in output. We therefore systematically read the value of port B as input (and therefore the peripherals connected to it). This remains to be verified however, failing to indicate which PPIs were installed in the test CPC's.

AMSTRAD used, as per the case for CRTC's, 8255 PPIs from several different manufacturers.
We can count the following circuits, used in all models without distinction (464, 664, 6128)

- NEC D8255AC-2
- NEC D8255AC-5
- TOSHIBA TMP8255AP-5

Note that the difference between "-2" and "-5" is due to the maximum frequency manageable by the circuit (4 MHz for the "-5" and 5 MHz for the "-2")



Raaahhhhhhhhh !!
Idiot Canadian Cat Curious in front of a CPC

This document is licensed under a CC BY-NC-ND 4.0 license
Attribution-Non Commercial-NoDerivatives 4.0 International