Meer Blog maken Inloggen

Security

Hacking everything, by Chris Evans / scarybeasts

Monday, November 16, 2020

Reverse engineering a forgotten 1970s Intel dual core beast: 8271, a new ISA

"As I recall, those two chips were fairly large. And fairly late -- to the marketplace. We had lots of issues with them. [...] Sometimes the elegant solution isn't the best solution." -- Dave House, digressing to the 8271 during "Oral History Panel on the Development and Promotion of the Intel 8080 Microprocessor" [link], April 26th 2007, Computer History Museum, Mountain View, California.

Introduction

Around 1977, Intel released a floppy disc controller (FDC) chip called the 8271. This controller isn't particularly well known. It was mainly used in business computers and storage solutions, but its one breakthrough into the consumer space was with the BBC Micro, a UK-centric computer released in 1981.

Intel 8271 (left) in an issue 3 BBC Micro model B

There are very few easily discovered details about this chip online, aside from the useful datasheet. This, combined with increasing observations of strange behavior, make the chip a bit of an enigma. My interest in the chip was piqued when I accidentally triggered a wild test mode that managed to corrupt one of my floppy discs even though the write protect tab was present! You can read about that here:

A wild bug: 1970s Intel 8271 disc chip ate my data!

Can we reverse engineer a detailed understanding of how it works? What wonders will we find?

Credits

The work described here represents the efforts of a virtual team that came together in an impromptu way to investigate the chip. There are many critical players, and special thanks go to:

- **Nigel Barnes**. Bridged the MAME community and BBC Micro community, locating someone with chip decapping skills.
- **BeebMaster**. Provided a couple of sacrificial 8271 chips for decapping.
- **Sean Riddle**. Decapped the chips and provided beautiful hi-resolution images.

Subscribe to my Twitter feed: @scarybeasts.

Subscribe To ScarybeastSecu

Posts

Comments

Blog Archive

▼ 2020 (6)
▼ November (1)
Reverse engineering a forgotten 1970s Intel dual c...
► July (1)
► June (3)
► April (1)
► 2017 (10)
► 2016 (7)
► 2015 (1)
► 2014 (5)
► 2013 (2)
► 2012 (9)
► 2011 (10)
► 2010 (11)
► 2009 (29)
► 2008 (20)

My other stuff

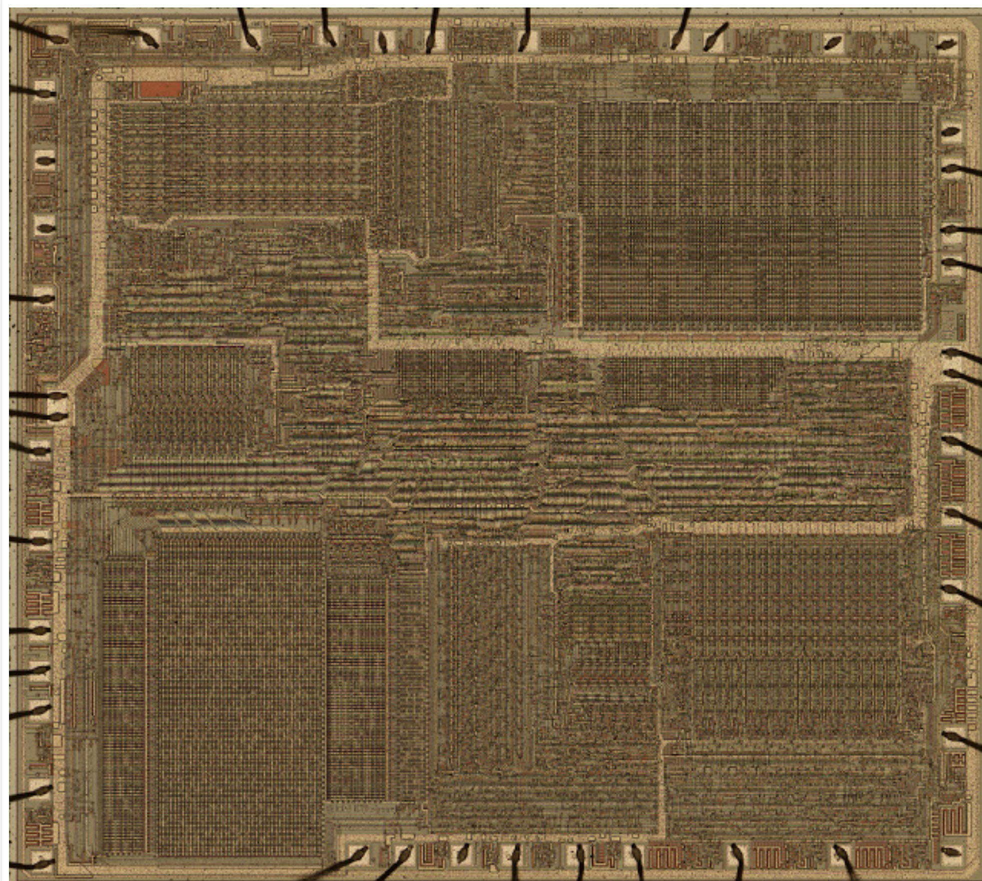
- My security advisories
- vsftpd home page
- Tavis' security advisories

- **ZXGuesser**, **Diminished**. Hardware level reverse engineering, including accurate extraction of ROM bits.
- **Ken Shirriff**. Provided notes on silicon macro structures, and located historic documents of great utility.
- **Rich Talbot-Watkins**, **Chris Evans** (me). Calculating the ISA, disassembling the ROM.

The beauty of the beast: recap and decap

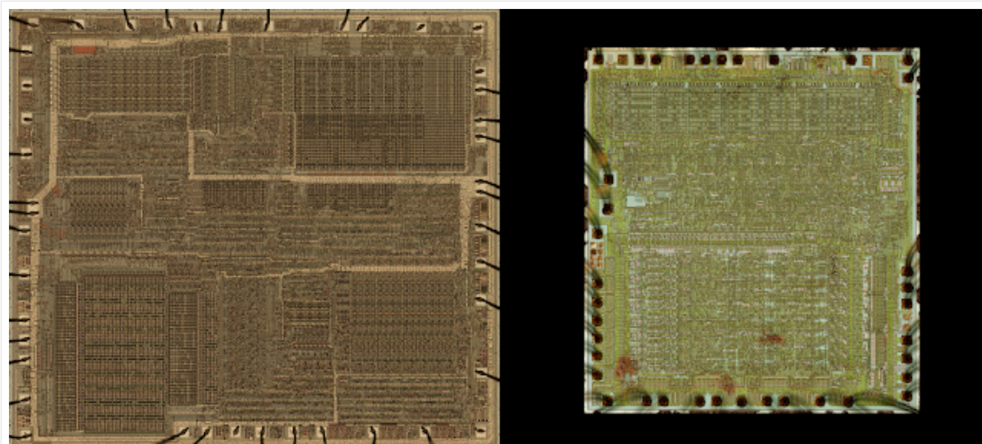
So to recap: we had a few indicators that this could be a very interesting chip. These range from the crazy test mode we found above, to the fact the data sheet hints at a large number of internal registers, with only a few documented.

And, to decap:



(See references at the end for very high resolution shots)

This a complicated chip for an FDC! There are a large number of large structures present, densely packed. To illustrate the point, we can compare this chip with the heart of the BBC Micro -- a venerable, legendary 6502, as used in iconic 80s machines and consoles such as the Apple][, Commodore 64 and NES:



8271 left, 6502 right

Not only is the 8271 larger than the main CPU by a significant amount, it also cost more by all accounts:

> *"In Acorn's wisdom, they had chosen the Intel 8271 disk controller for the BBC Micro - this controller*

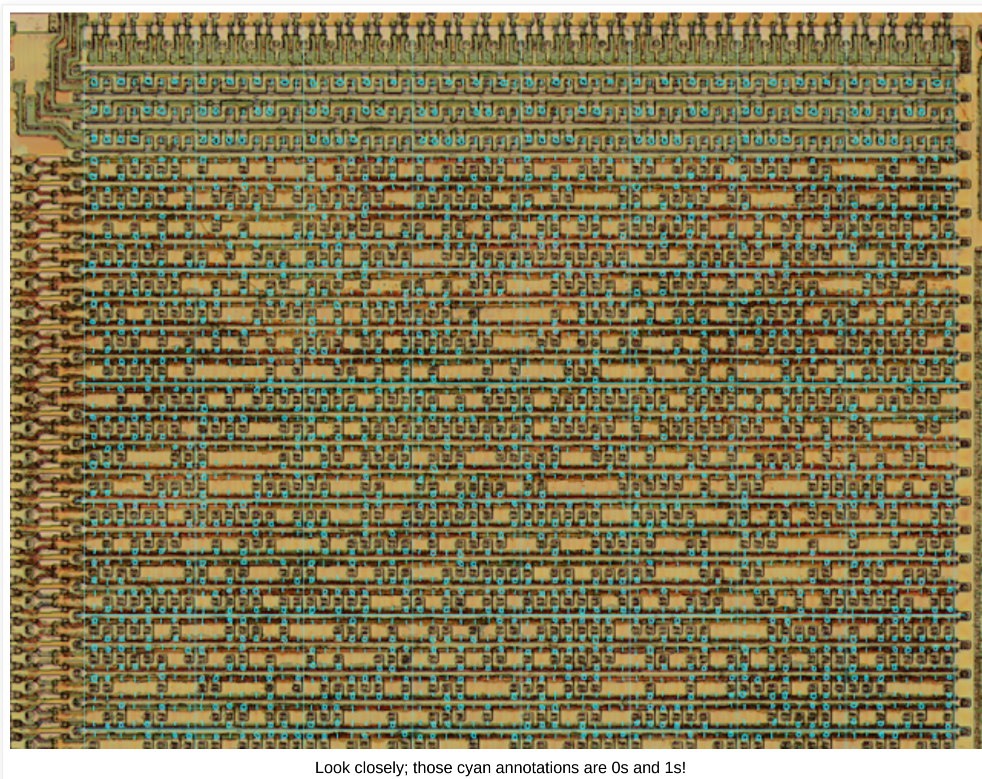
> *was probably obsolete even before the BBC Micro was launched. The Acorn disk upgrade comprised the 8271, a handful of standard TTL ICs, an Acorn DFS ROM, and the disk manual. The ICs plugged into unpopulated sockets on the motherboard.*
>
> *A few hardy folks tried sourcing the parts separately - which was fine, except that an 8271 tended to cost about £109 on its own,* if *you could find one..."* -- [source link]

By contrast, the 6502 allegedly cost around $7.45 in 1981, which is a huge difference factor.

Getting the ROM bits

From a quick eyeball of the 8271 die, there are plenty of PLA-like structures, but perhaps most promisingly, a large rectangle of ROM in the lower left. Right away, we're thinking that this *could indeed* be a general purpose microcontroller CPU if there's a ROM program. Initial focus fell immediately to extracting the ROM.

ZXGuesser and Diminished put in a Herculean effort, transcribing the ROM bits first by hand and later with some tooling assist and cross-checking. In case you're wondering what transcribing ROM bits by hand looks like, it looks a little like this:



Look closely; those cyan annotations are 0s and 1s!

The ROM matrix is 64x108 bit cells, for a ROM size of 864 bytes.

Once you've gotten the ROM bits, you still have the challenge of assembling them together into a correctly sequenced byte stream. This isn't always as easy as you might think, and is particularly rough when you don't have a reliable way of telling if the extracted bytes are ok, as is the case here. During my research, I found:

- Two 8-bit bytes interleaved.
- Bits and bytes heavily interleaved in a Channel F game cartridge.
- Byte ordering permuted fairly arbitrarily -- obfuscation?

Fortunately, the ROM reversers also looked at the row and column circuitry connecting the ROM, and gave a fairly robust opinion that the ROM bits / bytes are:

- Left-to-right, top-to-bottom.
- Bits inverted relative to the initial decode in the image above.
- Bits MSB first.
- Bytes build from 1 bit per linear 8 bit group.

This gives the first bytes of the ROM as the following:

```
0000:0000 FC 06 02 F7 00 FF B9 04 BA 00 BB 00 BC 00 BE 00 ...÷.ÿ¹.º.».¼.¾.
```

This happens to be a correct decode, although we couldn't be sure for some time.

Architecture hints

As Rich and myself engaged in efforts to try and disassemble the ROM, without any prior knowledge of the Instruction Set Architecture (ISA), I was fortunate enough to have a conversation with Ken Shirriff (righto.com) about this interesting chip. Against all odds, he found a detailed conference presentation abstract from 1977 [link to copy of full abstract].

This image from the abstract, will look familiar (imagine +90 degrees rotated).

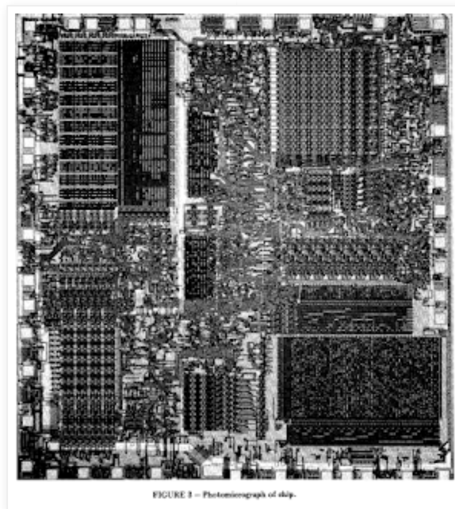


FIGURE 1 – Photomicrograph of chip.

The rest of the abstract is a gold mine. The presentation was titled "A Dual Processor Serial Data Controller Chip" and begins:

> *"A DUAL PROCESSOR microprogrammable chip that implements a specialized architecture for high-speed serial data controllers will be described. The chip measures 218 mils by 244 mils and contains 22,000 transistors..."*

22,000 transistors!! We knew this was a bit of a chonker, but for reference, the 6502 has 3,218 transistors; the 8080 6,000 transistors and the 8086 29,000 transistors. Yes, the 8271 is not that far off an 8086 in terms of transistor count.

This instructive table and diagram are also from the abstract:

	BIT	*BYTE*
Number of Instructions	Not Applicable	46
Instruction Execution Time	250 ns	1.25 µs
Width of Control Store	33	8
Number of Bits in Control Store	3300	6912
Number of Devices	4600	13,500
% Area of Total Chip	29%	71%
Special Characteristics	Horizontal Microprogram	Multitask

TABLE 1 – Comparison between bit and byte processor.

[Right]

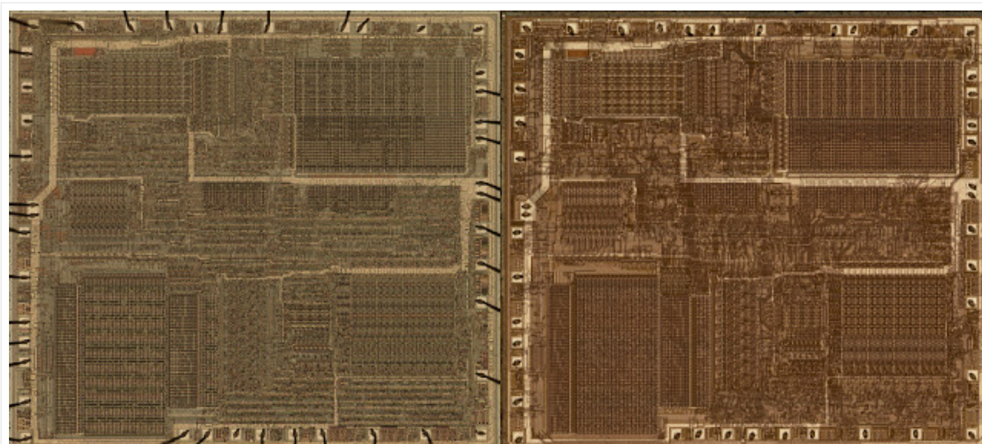
FIGURE 2 – Byte-processor block diagram.

SYSTEM BUS
ACC
REGISTER FILE
IR
ALU
TO BIT PROCESSOR
I/O PORT
TO PERIPHERAL OR MODEM
ROM
PC
DISPATCHER
STACK

This confirms our suspicions that we *are* dealing with a general purpose CPU, coupled with some acceleration for I/O. The general purpose CPU has the features you'd expect, including a PC, stack, ALU, accumulator, registers, and access to a bus. Further detail in the abstract includes "32 eight-bit registers" and "four-level stack".

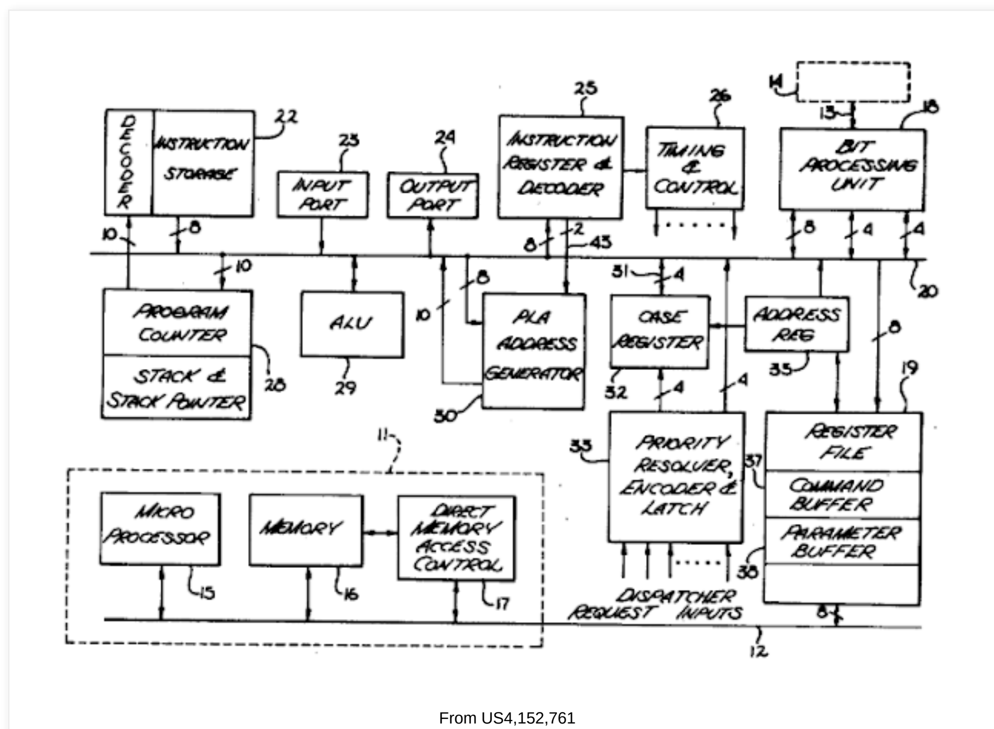
The presence of a a bit processor (a co-processor if you like) with 250ns cycle time resolves the elephant in the room with the "general CPU" theory. A general CPU of the era probably wouldn't be fast enough to calculate CRC16 at the disc data rate. However, a specialized PLA-driven bit engine bolted on to the side will do just fine.

Finally on this abstract, we see that it states "To date, two distinct controllers have been microprogrammed: a floppy disk controller and a synchronous data link controller (SDLC)". Well, we've found the FDC, the 8271. Intel's SDLC from the era was the 8273. Enter Sean Riddle once more -- what a star -- and another sacrificed chip or two later, we have our result:



8271 left, 8273 right

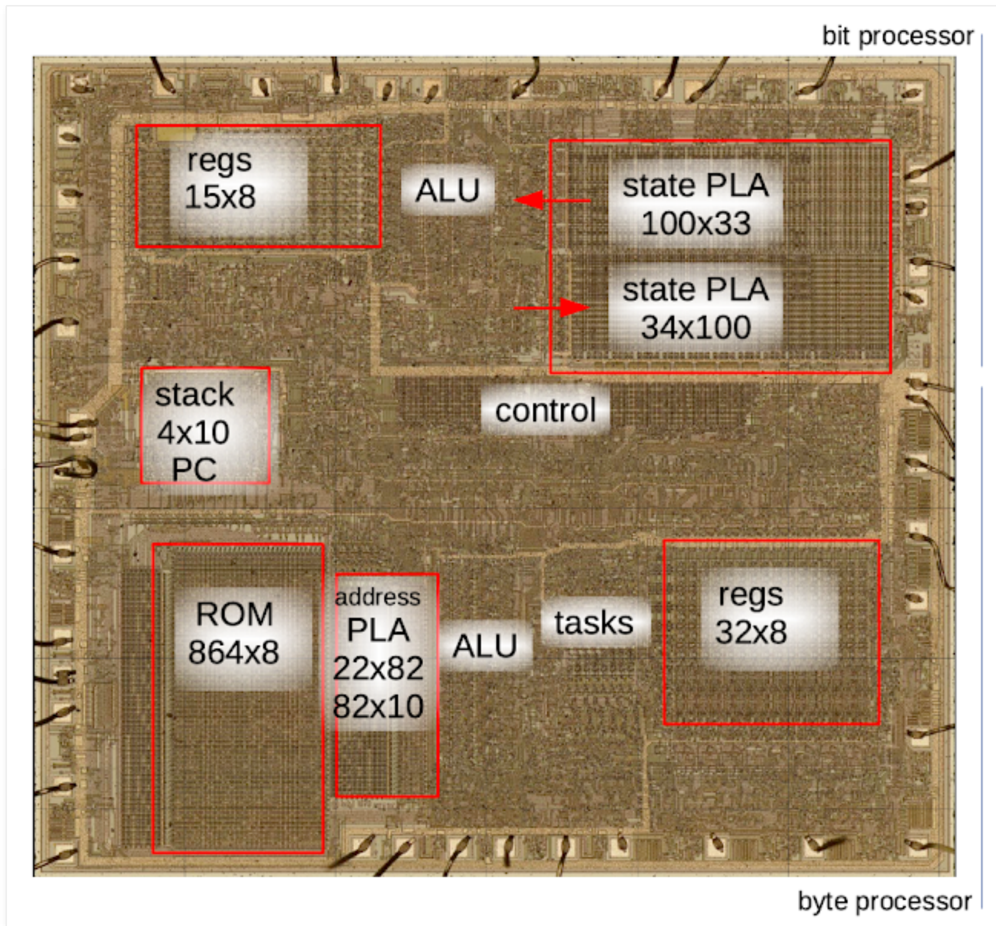
Ken also found a patent relating to this dual core design! It's US4152761. [link to a more complete version]. There's a lot of useful architectural detail in the patent, including this interesting summary of key components.



From US4,152,761

As can be seen, there's additional complexity described here that falls outside what we'd expect from a traditional CPU. There's dispatcher requests, priority resolution, a "case" and an "address" register in addition to the program counter and instruction register. This relates to some form of scheduling, which we'll encounter later.

Finally, here's a diagram (courtesy of Ken). It's an early estimation of how the architecture we've seen in the the documents so far might map to the silicon. As a dual processor behemoth, note how there's two ALUs, two sets of registers, lots of PLAs (including a couple to the left and right of the "control" label), and plenty of patches of silicon with unknown function.



From bits to bytes to an Instruction Set Architecture (ISA)

With an overview of the architecture, we have a better idea of what sort of opcodes we might find in the instruction set. The more pieces of the puzzle we have in our heads, the better chance we have of making abstract connections in our attempt to solve a problem with a lot of moving parts.

Initial attempts to disassemble the ROM centered around the theory that the CPU core might be based on the Intel MCS-48 microcontroller series. This series includes the well-known 8048 and has several variants such as the slightly cut-down 8020 (less I/O lines). And why wouldn't this core be based on a further cut-down 8020? It just fits too well: 1K ROM, 64 registers, 13 I/O lines. The timing works well too: the MCS-48 series first launched in 1976, making the core available before the release of the 8271. So the theory went, you'd be crazy as an 1970s Intel employee to not just walk down the corridor and raid the parts bin of your colleagues in order to get a headstart.

The MCS-48 theory turned out to fit extremely well.... but be false. No amount of ROM bit / byte wrangling led to sensible MCS-48 disassemblies.

Back to the drawing board, we looked again at our "probably correct" ROM decode and analyzed it for patterns we'd expect to find in an 8271 ROM, based on our understanding of how the controller works, and how disc recording works in general. Here's a section we found enlightening:

```
0000:0170 A1 79 FF 70  A0 8B 6E FD  E9 E9 E9 E9  FD 00 A6 01 | ¡yÿp .nýééééý.¦.
0000:0180 F3 04 64 FF  90 A3 24 FC  DF FC DB F4  D7 F5 FC 15 | ó.dÿ.£$üßüÛô×õü.
0000:0190 00 23 FC DF  FC D8 F5 FE  F4 C7 F0 01  15 F5 FF F0 | .#üßüØõþôÇð..õÿð
0000:01A0 00 02 F3 0A  FC E3 FC DB  F5 FB FE 98  F5 E5 F4 FF | ..ó.üãüÛõûþ.õåôÿ
0000:01B0 FC D1 98 E5  FC EF 0D EE  02 A5 8B EE  ED F0 08 02 | üÑ.åüï.î.¥.îíð..
```

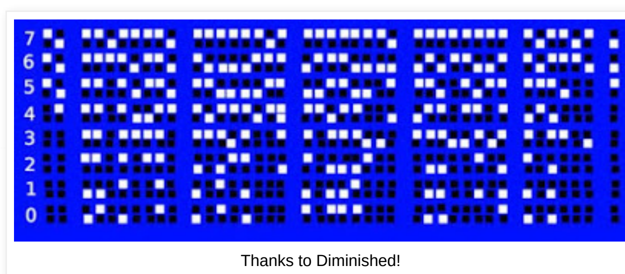
Most significant is the appearance of the constants $FE + $C7 (middle box), then $E5 + $FF (lower box) in a the same context. These constants are the data byte + clocks byte we'd expect to see in the format routine for FM (single density) formatted discs. $FE + $C7 is the sector header marker, and $E5 + $FF is the default fill byte for freshly formatted sectors. As a bonus, we speculated that $E9 (appearing four times in a row) could be a shift left or right opcode, with $FD perhaps being RET. All in all, some strong circumstantial evidence for a correct decode.

After the MCS-48 opcode list failed to match our ROM, we spent a long time trying to derive an alternate instruction set.

But it's hard; it's a bit like one of those jigsaw puzzles where every piece is the exact same color. It's very hard to find a start as there are so many different possibilities to try. Research is too often

presented as a neatly packaged result, with no mention of the struggle. I think this contributes to discouraging newcomers. So make no mistake: this was a struggle; it went on for some time; there was swearing; and the characteristics leading to success were perseverance and grit as opposed to any particular technical ability.

The breakthrough was catalyzed when the hardware investigations got a good read on the wiring and content of one of the instruction PLAs associated with the byte processor. The PLA in question is actually the darker rectangular block to the left of the "control" label in the image above. It is populated like this:



Thanks to Diminished!

With the instruction PLAs decoded, it would be possible to trace the effects of each 8-bit opcode by following activation lines to the registers, stack, ALU, etc. However, we were able to hit our breakthrough with only opcode ranges, provided by this PLA. Of specific interest is these ranges from Rich:

```
1111 1100   FC
1111 1101   FD
[...]
0010 xxxx   20-2F
[...]
0011 xxxx   30-3F
[...]
001x xxxx   20-3F
```

The specialized $FC and $FD match our theory of CALL and RET, but more pivotal is the range of $20-$3F, split into 2x 16 wide blocks. Immediately, I speculated this could be "move register to accumulator" and "move accumulator to register". This would turn out to be correct. Furthermore, I noted that opcode $0? was common before $2? or $3?, e.g.

```
05D:   04 22 03 35
```

This led to the theory that these are register bank selection opcodes. Similar to the MCS-48, the theory is that there aren't enough opcodes for all operations to be able to reference all 32 registers, so there must be a bank select. This also would turn out to be correct. In fact, with these pieces, the jigsaw started to fall in place, and fall into place faster and faster.

Some ROM code examples

For these code examples, bear in mind that the Instruction Set Architecture (ISA) presented here hasn't seen public light of day as far as we know. We've had to invent our own assembly mnemonics, although they're designed to be familiar to anyone familiar with assembly languages in general. Some specific notes:

- SEL RB is SELect Register Bank, and provides the base index for register access (multiply by 8 to get actual index).
- Note that all register references are by index, not register number. A register number can be calculated with the index and register bank.

1) READ SPECIAL REGISTER

```
.command_READ_SPECIAL_REGISTER
32E   00      SEL RB 0
32F   27      MOV A, I7             ; R7 ($07) (param 1)
330   30      MOV I0, A             ; R0 ($00) = register index
331   F8      MOV A, [I0]           ; read special register value to A
332   02      SEL RB 2
333   36      MOV I6, A             ; R22 ($16) (ext RESULT) = A
334   0E      SEL RB 14
335   EE      SYS 2, RB             ; JMP (2,E,0) => $288, .post_command_tidy_up
```

The READ SPECIAL REGISTER command is one of the simplest. As suspected, "special register" externally is just "index into the 32 registers" internally. We can also note plenty of interesting things:

- The command setup code, which we'll see in a bit, writes parameters to R7 downwards.
- Indirect reads and writes are done via special opcodes that indirect through I0 (typically but not always R0), as per the MCS-48 architecture.

- Some of the internal registers interact with the external bus registers used to interact with the host CPU. R22 is where you write values for them to appear in "ext RESULT", which is read with BBC Micro's 6502 at memory location $FE81.
- The "register bank" concept appears to be re-used to provide a lookup index to the SYS 2, RB opcode. You might consider this a minor kludge.
- In order to jump from page 3 ($3??) to page 2 ($2??) at the end, a special SYS opcode is required. Normal JMP / CALL instructions only have an 8-bit operand and can only jump to the current page. SYS opcodes look up a 10-bit PC from a hard coded table.
- The command exits by jumping to a common exit routine, `.post_command_tidy_up`. This routine disables most of the chip, including the bit processor, and events. This will be important later.

2) SPECIFY

```
.command_SPECIFY
066    00         SEL RB 0
067    27         MOV A, I7              ; R7 ($07) (param 1)
068    30         MOV I0, A              ; R0 ($00) = destination index
069    F1 03      MOV I1, #$03           ; R1 ($01) = count of 3 extra parameters
06B    14         YIELDTO 4              ; seems to set PARAM callback to
                                         ; .wakeup_PARAM_4_SPECIFY, then YIELD
; ** entry point (3, segment 9, routine 4 5 6 7 C D E F)
.wakeup_PARAM_4_SPECIFY
06C    03         SEL RB 3
06D    2E         MOV A, IE              ; R30 ($1E) (ext PARAM)
06E    00         SEL RB 0
06F    E8         MOV [I0], A
070    00         SEL RB 0
071    80         INC I0
072    A1         DEC I1                 ; R1 ($01), decrement count
073    8A 8C      BZ $08C                ; branch if done
                                         ; lands at SEL RB 14, JMP RB
                                         ; which jumps to .post_command_tidy_up
075    FF         YIELD
```

SPECIFY is also a simple command. It's not strictly a necessary command because it has the same effect as 3 WRITE SPECIAL REGISTER commands to sequential register numbers. It's unclear why it exists, given the space crunch in the ROM and on the silicon, but the datasheet does describe initializing the 8271 using a few of calls to this. Other items of note:

- SPECIFY accepts and applies the register writes immediately, YIELDing (sleeping / idling) between writes and waiting for the host CPU to supply the next register. This means the SPECIFY command might be partially completed for some time, and never fully completed, depending on how we program the chip. This will be significant later.
- SPECIFY internally uses R0 as a destination index to write register values, and R1 as a count to complete. This makes it an ideal candidate to do some black box testing to confirm the chip behaves the same as our source code disassembly. Specifically, I tried and confirmed:
  - Pass the first parameter to SPECIFY as 0, indicating we are writing register values starting at R0. Note that as per the code, R0 is used internally by SPECIFY.
  - Pass the second parameter as $22. This will get written to R0 (because R0 currently points to R0) and incremented, leaving $23 in R0. This has "corrupted" R0.
  - Pass the third parameter as $48. This will get written to MMIO register R35 ($23), and should turn on the drive motor.
  - Do not pass a fourth parameter. Despite not “completing” the full four parameters of the command, we expect the writing of the third parameter to have the effect as just described. **It does on real hardware**.

3) External command register handling

```
.wakeup_COMMAND
014    02         SEL RB 2
015    F6 00      MOV I6, #$00           ; R22 ($16) = $00 (ext RESULT)
017    CF BF      AND I7, #$BF           ; R23 ($17) (ext STATUS) !CMD_FULL
019    00         SEL RB 0
01A    F5 01      MOV I5, #$01           ; R5 ($05) = $01 (param 3 default $01)
                                         ; That’s 1x 128 byte sector in many cases.
01C    F4 01      MOV I4, #$01           ; R4 ($04) = $01 (param 4 default $01)
01E    03         SEL RB 3
01F    98 05      MOV A, #$05            ; 5 parameters expected
```

```
021    6F 18 27   TBZ IF, #$18, $027       ; R31 ($1F) (ext CMD), jump if 5 param
                       ; command
                                           ; matches SCAN, FORMAT
024    2F         MOV A, IF                ; R31 ($1F) (ext CMD)
025    9C 03      AND A, #$03              ; (CMD & 3) parameters expected
027    00         SEL RB 0
028    31         MOV I1, A                ; R1 ($01) = A = parameters expected
029    8A 3D      BZ $03D                  ; if no parameters, start command
02B    F0 07      MOV I0, #$07             ; R0 ($00) = $07 (put parameters at $07 down)
02D    BC 01      TASK 4, 1                ; select .wakeup_PARAM_1_accept
                                           ; 3 9 (1, 3, 9, B) = $035
02F    FF         YIELD
```

This is where to start reading if you want to trace the main entry point into the ROM. The 8271 byte processor wakes up here where the external host CPU writes to the external command register ($FE80 on the BBC Micro). And again some interesting things to note:

- For commands other than SCAN and FORMAT, the number of parameters expected is actually encoding in to the low order bits of the command byte. Undoubtedly, this saves space in the ROM. This is another case where a simple test can confirm the behavior claimed by the ROM code:
  - Take a command that takes 1 parameter, e.g. READ SPECIAL REGISTER.
  - Increment the command byte and supply that to the 8271 instead.
  - We'd expect the 8271 to require 2 parameters to start the command, but then ignore the second parameter and behave as if the 1 parameter version was used. **This is indeed observed on real hardware**.
- Some default parameter values are deployed. These are used in the commands the datasheet calls "128 Byte Single Record Format". Presumably it is again a win for ROM space savings.

4) Full ROM disassembly

The full ROM disassembly, as of time of writing, may be found here: [link]. This copy will remain static. It's mostly complete, and all of the "main" paths are fully traced, including SEEK, READ DATA, WRITE DATA and FORMAT TRACK.

Javascript on a (1970s!) chip

It is time to look at some of the unusual sounding instructions in the ISA:

- **YIELD**. $FF. This instruction tells the processor to switch to running the highest priority dispatcher request, or (more likely) go idle if there's no dispatcher request active.
- **TASK t, r**. $B8-$BF, 2 bytes. This instruction changes the callback routine for the specified task. Callback routines are specified as an integer that form part of a lookup key into a table of ROM addresses.
- **YIELDTO r**. $10-$1F. Change the callback routine for the currently executing task and yield.
- **SYS 0, RB, A**. $EC. Jump to the ROM address at key (0, RB, A) in the address PLA. RB is the current register bank from the SEL RB instruction, and A is the accumulator value.
- **SYS 1, RB, R**. $ED. Jump to the ROM address at key (1, RB, R) in the address PLA. RB is the current register bank from the SEL RB instruction, and R is current routine value.
- **SYS 2, RB**. $EE. Jump to the ROM address at key (2, RB, 0) in the address PLA. RB is the current register bank.

There are quite a few concepts introduced here. If you find them somewhat jumbled, you are not alone. A lot of things about the non-traditional parts of the byte processor CPU appear very ad-hoc to me. Let's look at a concrete example: the handler for the SPECIFY command. This is the `.command_SPECIFY` code in example chunk 2) above.

To get `.command_SPECIFY` to run, the host CPU provides the command $35 to the external command register, then 1 initial expected parameter to the external parameter register. The byte processor CPU wakes up at `.wakeup_PARAM_1_accept`. The context here is:

- PC == $035
- TASK == 4
- SEGMENT == 9, ROUTINE == 1

The fact that writes to the external parameter register wake up task 4 is **hardcoded**. The PC executed is determined by the routine selected for this task, which was 1 at the time. This is used to create the key (3,9,1), which looks up the address $035 in the address PLA. The address PLA is **hardcoded**. The fact that task 4 keys lookups as (3,9,x) is **hardcoded**.

`.command_SPECIFY` ends with this, because it is the one command that gets executed and then decides it

wants to yield to wait for 3 more parameters:

```
06B   14       YIELDTO 4             ; seems to set PARAM callback to
                                     ; .wakeup_PARAM_4_SPECIFY, then YIELD
```

This unusual instruction sets the callback routine for the *current* task to be 4. This means that the callback code for the next external parameter register write will be keyed (3,9,4) in the address PLA. That's PC $06C, aka. `.wakeup_PARAM_4_SPECIFY`.

A good mental model for this is Javascript. All execution is event based; the handler for a given event can be changed (by the handler itself, or an unrelated handler); there is no pre-emption until an explicit yield.

The known events wired in to the byte processor are:

- 0: Not reversed at all. Appears to be related to the SCAN command, which isn't enabled on the BBC Micro due to lack of DMA.
- 1: Bit processor event, e.g. found sync, lost sync, CRC error.
- 2: Bit processor, read byte ready.
- 3: Bit processor, write byte needed.
- 4: External parmeter register written.
- 5: appears unused. (Could be permanently connected to external command register?)
- 6: Disc drive index pulse.
- 7: appears unused.

On top of the "normal" Javascript model, there's also the concept of task priorities. These are not visible in the ISA and are presumably hardcoded. One instance this might come in handy in a floppy disc controller is when a disc drive index pulse (once per disc revolution) fires at the same time the bit processor needs a write byte. (It's not common to write across the index, but it could happen.) In this instance, providing the bit processor a data byte is a much more real-time task than handling an index pulse, so it should be handled first.

Yes, this is quite some complexity in addition to the complexity typical in a general purpose CPU. In particular, it provides many additional ways to slice and dice control flow handling beyond the standard JMP / CALL / RET. In fact -- and somewhat painfully -- the different control flow possibilities are mixed together! To briefly get a taste of the horrors, here's the command handler for many of the sector read operations:

```
.command_READ_DATA
.command_READ_DATA_AND_DEL_DATA
.command_VERIFY_DATA_AND_DEL_DATA
0A7   FC C9    CALL $0C9       ; .do_common_path_from_seek
                               ; very gnarly because this CALL does a YIELD
                               ; context on RET is from
                               ; .wakeup_BITPROC_EVENT_1_check_header_crc
                               ; if we get a matching sector header,
                               ; the RET at $255 fires
0A9   BA 0B    TASK 2, 11      ; select .wakeup_BITPROC_READ_10_11_count_GAP2
                               ; (3,5,(10,11)) = $1B8
0AB   12       YIELDTO 2       ; select
                               ; .wakeup_BITPROC_EVENT_2_check_for_data_marker
                               ; (3,4,2) = $2D0
                               ; and YIELD
```

As can be seen, a CALL subroutine is doing a YIELD without unwinding the stack. This should probably be considered an anti-pattern? The stack is a shared resource across all tasks, so you'd better hope that two different tasks can't trip over each by doing this at the same time. There's also the complexity that when a CALL returns, various aspects of the execution environment (task, routine, RB, etc.) may well have changed.

All this makes the ROM very hard to follow for the read and write paths, even with a fully disassembled and commented ROM!

Unusual features of the ISA

1) Lack of symmetry

These are the opcodes we found relating to incrementing, decrementing, adding and subtracting:

```
$80-$83:       INC Ix
$84-$87:       ADC A, Ix
[...]
$A0-$A7:       DEC Ix
```

```
[...]
$90-$93:        ADC Ix                  does INC Ix if carry
$94-$97:        SBB A, Ix
```

This is interesting because there is an asymmetry between which registers can be incremented vs. decremented. Some registers cannot be incremented at all, even with register banking, because banking operates in multiples of 8 and there are only 4 increment opcodes.

One theory is that the chip designers were short on silicon space, and trying to get away without add and subtract support in the ALU. There's only circumstantial evidence to support this, such as the hacked-out ranges in the opcode space; the code generally being add / subtract free except for the code implementing disc drive seek; no compare instruction found (yet -- there are unknowns); the use of XOR to do equality checks at $240 and $24F; and the presence of only direct equality checking or mask based checking for the most common branch opcodes (see item 2) directly below). But it's fun to speculate wildly isn't it?

2) 3-byte opcodes

An entire quarter(!) of the opcode space is devoted to four conditional branch instructions:

```
$40-$4F:        BEQ Ix, #imm, abs       branch if Ix == imm
$50-$5F:        BNE Ix, #imm, abs       branch if Ix != imm
$60-$6F:        TBZ Ix, #imm, abs       test and branch if zero
$70-$7F:        TBNZ Ix, #imm, abs      test and branch if not zero
```

These opcodes are all three bytes, which is a departure from Intel's other microcontrollers of the era. It's a lot of opcode space, but they can do a lot with a little, for example this piece of code from `.do_seek`.

```
13E   62 01 3E  TBZ I2, #$01, $13E   ; MMIO R34 ($22) (drive in), wait until CNT/OPI
141   72 01 41  TBNZ I2, #$01, $141  ; MMIO R34 ($22) (drive in), wait until !CNT/OPI
```

For a certain (less common) seek mode, these 6 bytes are sufficient to busy loop while waiting for the drive to start the seek, then acknowledge finishing it.

These three byte opcodes were found by the previously mentioned "jigsaw" analogy. Once the command entry function was fully disassembled apart from just three bytes at $021, there was only one clean possibility that fitted the required behavior.

3) No timers, port I/O or IRQs

Presumably to save silicon, the byte processor CPU has done away with timers, dedicated port I/O instructions and IRQs. The MCS-48 has all of those.

They're not needed, though. Delays, such as the millisecond range delays for seek steps, are simply timed via busy loops.

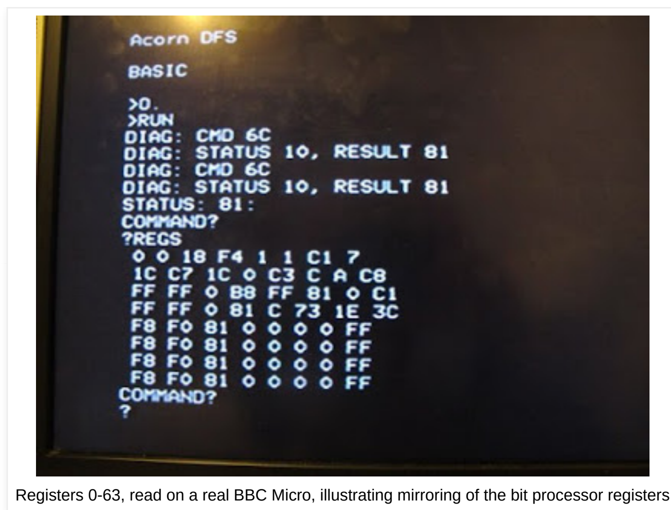
4) Decide your own adventure

At time of writing, our opcode list is here: [link]. It is not complete. The one part of the 8271 ROM we have not disassembled, SCAN handling, uses at least opcodes $9A, $9B and $8C. (We've ignored SCAN because it needs DMA wired up, which is not the case in the BBC Micro application.) Furthermore, the PLAs suggest that other opcode ranges *not* seen in the 8271 ROM might do something. This includes $A8-$AF and $B0-$B7. Feel free to go and have a look!

Interface to the bit processor

Now that we've got the byte processor understood and disassembled, it's time to turn our attention to interface to the bit processor and its behavior. After all, it's the bit processor that is wired to the disc drive control and data lines!

Our initial assumption was that the byte processor CPU would, like the MCS-48, have some form of port I/O instructions. This turned out to be false. The reality is simpler: it uses MMIO (Memory Mapped Input/Output). This means that access to certain register indexes change registers in the bit processor instead of the byte processor. It's quite simple: 0-31 references the byte processor registers. And the range 32-39 references bit processor registers. For simplicity of decoding, the bit processor's 8 register references are mirrored 4 times across the range 32-63. The entire 0-63 range is then mirrored 4 times in the entire addressable range of 0-255.



Registers 0-63, read on a real BBC Micro, illustrating mirroring of the bit processor registers

Furthermore, the context of bit processor references in the byte processor code makes it clear that the bit processor interface is very simple. It is so simple we didn't feel the need to reverse engineer the bit processor further. The bit processor register assignments are as follows:

- 0: control register, 4 bits
  - Bit 0 (0x01) => gather CRC (?)
  - Bit 1 (0x02) => finish CRC (?)
  - Bit 2 (0x04) => 1 for read, 0 for write
  - Bit 3 (0x08) => idle state
- 1: status register, indicates sync data byte type, CRC error, etc.
- 2: drive input, read for drive status
- 3: drive output, controls step / write / etc. lines
- 4: clocks output byte
- 5: data output byte
- 6: data input byte
- 7: unused? (returns 0xFF, and byte processor ROM relies on this!)

The astute reader might ask: does this mean the bit processor can be programmed directly with the WRITE SPECIAL REGISTER command, since the bit processor registers are MMIO? And the answer is yes! There are severe caveats however:

- The generic command entry code corrupts bit processor state on entry.
- The generic command exit code resets the bit processor and associated callbacks on exit (but strangely and usefully not for WRITE SPECIAL REGISTER).
- **The latency of WRITE SPECIAL REGISTER is terrible**.

That last bullet, the poor latency, is unfortunate. It's about 211us, which means there's zero chance for tricks like writing data to the disc a clocks + data byte at a time. The latency is large because the byte processor executes a large number of instructions on the command entry path, doing things like checking and caching drive status, as well as reading parameters one at a time, checking if the selected drive changed, etc.

Writing the drive output register directly is useful, though. I did this for my trick to write "weak bits" directly using the 8271! See my blog post about weak bits for more details. [link]

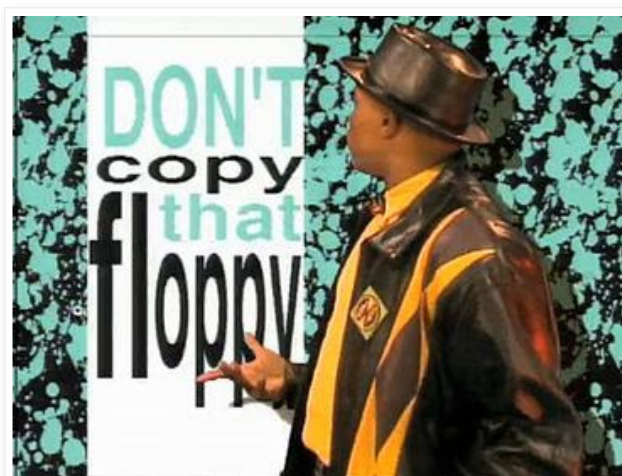
Writing the unwritable

Now that we know how this monster works, it is of course time to turn our attention to mischief. Can we make the chip do things it is "not supposed" to be able to do? Of course we can, and as usual, it involves disobeying the datasheet:

We are specifically going to disobey the sentence that states "Issuing a command while another command is in progress is illegal." We are now equipped to see exactly what happens if we do this, by reading and reasoning about the code. The callback called when the external command register is written gets on with its job without regard for whether a command is in progress, so side effects will be:

- The internal command register itself is corrupted, i.e. mismatched with the currently executing command. It is referenced from time to time so this may be useful to us.
- The illegal command will change internal register values, which may impact the execution of the current command.
- The illegal command may change or disable callbacks, or reset or reconfigure the bit processor.

Taking these things into consideration, we are going to **try and write an arbitrary FM bit stream**. Achieving this will enable us to recreate copy protected disc surfaces that are not supposed to be writeable with the 8271. Remember, kids:



We're going to attempt this by separating out how we write the data bytes and how we write the special sector mark clock bytes. Writing a full track of data bytes is easy, but useless on its own. We can do this by:

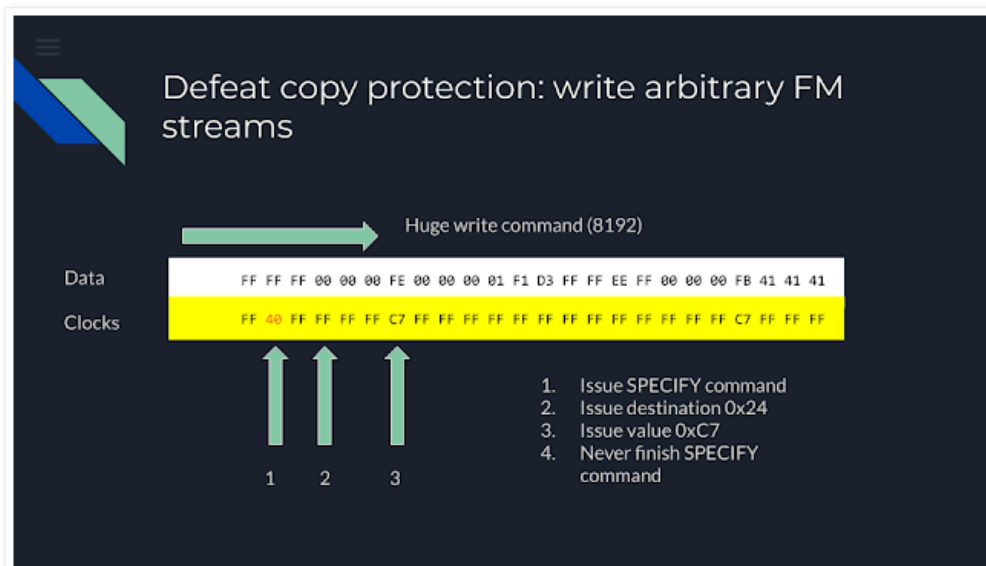
- Formatting a track with a single sector header.
- Issue a WRITE DATA command for that sector, of size 8192 bytes. A track is only 3125 bytes (with a perfectly calibrated drive), so:
    - At the first wrap around, start writing the track of bytes we want.
    - 3125 or so bytes later, at the second wrap around, abort the command by resetting the controller.

This track of data bytes alone will be useless. If you try to read a sector from it, you will get error $18, aka. "sector not found". The special clock byte markers required to identify sector headers and sector data will be missing.

The way FM encoding works is very simple: it alternates clock bit, data bit, clock bit, data bit, ... every 4us.

Normally all the clock bits are 1, to maintain timing and keep the drive electronics happy. But for a sector header or data marker, a few clock bits are left out so that the floppy disc controller can locate things in a bitstream where it is not sure where it is.

So, while the WRITE DATA command is running, we're going to sneak in some parallel SPECIFY commands to "corrupt" registers for the running WRITE DATA command, without disturbing it. Specifically, we're only going to corrupt the clocks byte register (MMIO R36) at the precise times necessary to write clocks byte values other than 0xFF. The overall operation looks like this:



By some miracle, this scheme does work, and it easily replicates discs that no BBC Micro copier back in the day could come close to. One example of a tough disc, for some reason, is The Sentinel [link]. The disc has a pretty label and packaging so of course, time for a gratuitous image:

There are plenty of quirks and caveats to get this working. To briefly note them for completeness:

- The SPECIFY command was used instead of WRITE SPECIAL REGISTER to help with latency concerns. The SPECIFY command can be "primed" by passing the command and the first parameter, such that the second parameter (first register value to write) executes the write and executes with low latency at just the right moment.
- The WRITE SPECIAL REGISTER command exits without resetting the bit processor or clearing all the I/O callbacks. Unfortunately, the same cannot be said for SPECIFY. Fortunately, we don't have to exit the SPECIFY command! It starts having the useful side effect of writing internal registers before it is complete. And then you can later restart it again and again, never completing it.
- The act of command dispatch corrupts the MMIO clocks byte register! This is very unexpected but it uses MMIO R36 as a temporary storage location while it is calculating which disc drive (0 or 1) is selected, and whether it changed. It is worth looking at:

```
.parameters_complete_launch_command
03D    FC 5D      CALL $05D        ; .read_drive_status
03F    BC 00      TASK 4, 0        ; select .wakeup_PARAM_0_no_action
                                   ; (3,9,(0,2,8,A)) => $030
041    03         SEL RB 3
```

```
042   2F      MOV A, IF      ; R31 ($1F) (ext CMD)
043   CF 3C   AND I7, #$3C   ; R31 ($1F) (select bits + param count masked out)
045   9C C0   AND A, #$C0    ; A now contains drive select bits
047   04      SEL RB 4
048   34      MOV I4, A      ; MMIO R36 ($24) (???) (temp storage?)
049   E3      XOR A, I3      ; MMIO R35 ($23) (drive out)
04A   9C C0   AND A, #$C0
04C   8A 53   BZ $053        ; only update drive out if select bits changed
                             ; .command_dispatch
04E   98 20   MOV A, #$20    ; bit for side select (drive 0 vs. 2)
050   C3      AND A, I3      ; MMIO R35 ($23) (drive out)
051   D4      OR A, I4       ; MMIO R36 ($24) merge back select bits?
052   33      MOV I3, A      ; MMIO R35 ($23) (drive out)
                             ; only drive select bits and side select kept
                             ; clears write enable, head load, and others
                             ; matches data sheet
```

The line in orange is the one in question. I annotated it as iffy with ??? when I first disassembled it as it looked wacky. But, it's correct and a real machine exhibits the corrupted clocks precisely as described by the code. The corrupt clocks value, $40 if writing to drive 0, is shown in orange in the above diagram. Once you know it's there, and why, the easiest way to navigate around it is to arrange for the clocks corruption to land where it doesn't case problems. When paired with a $FF data byte, it doesn't create weak bits on the disc surface, and the controller actually seems to skip over it on read.

- The write I/O path, by some stroke of good fortune, resets the clocks byte to $FF on every write I/O callback. This saves us a lot of trouble:

```
.wakeup_BITPROC_WRITE_3_set_clocks_and_count_host_bytes
318   F4 FF   MOV I4, #$FF   ; MMIO R36 ($24), standard clocks
[...]
```

- Careful timing is needed. There's a pipeline of bytes from the external data register to the internal bit processor data byte register to the actual output pulse machinery. This needs to be accounted for.

Combined, these quirks convince me that I'd have never gotten this going, or gotten close, without a thoroughly reverse engineered and disassembled ROM.

Other successes and failures

Other things enabled or demonstrated based on careful reading of the ROM code:

- beebjit now has a much more accurate 8271 driver.
- Unexplained weirdness trying to read a sector with logical track id $FF has been explained as an integer overflow in the bad tracks handling.
- Sectors on a non-zero physical track, but with a zero logical track id, were believed "impossible" to read. I'm able to read them by using the "command within command" trick. Once a READ DATA command is safely underway, including having processed the seek request and gone idle, it's possible to use WRITE SPECIAL REGISTER to change R7 (command param 1, which is the requested track) to zero, and have the sector read fine. The issue is that references to logical track 0 in the seek code are always treated as a mandate to find physical track 0. You need to bypass that.

Things not achieved:

- Reading or writing MFM (double density) -- looks fundamentally impossible. Does not appear to be a capability in the hard-wired bit processor.
- Executing arbitrary code on the chip. This is a shame, as the byte processor is a capable CPU! Who knows what we could use a little co-processor for? Things making this hard include:
    - Separation of code and data in separate address spaces.
    - No references to the stack possible outside of CALL and RET.
    - Indirect jump targets stored in a read-only PLA. (Microsoft CFG? :)
    - ... and yes, curiously, these accidental defenses all sound similar to defensive technologies investigated or deployed since year 2000. So, the 1970s called and...

Summary

The 8271 has exceeded our expectations! Where to start? It's a massive chip, encompassing dual cores and a Javascript like execution model. Remember, this was the mid-1970s. Its general purpose CPU runs an Intel instruction set architecture that I don't believe has been publicly documented until now. It's not every day we get the treat of a new Intel ISA.

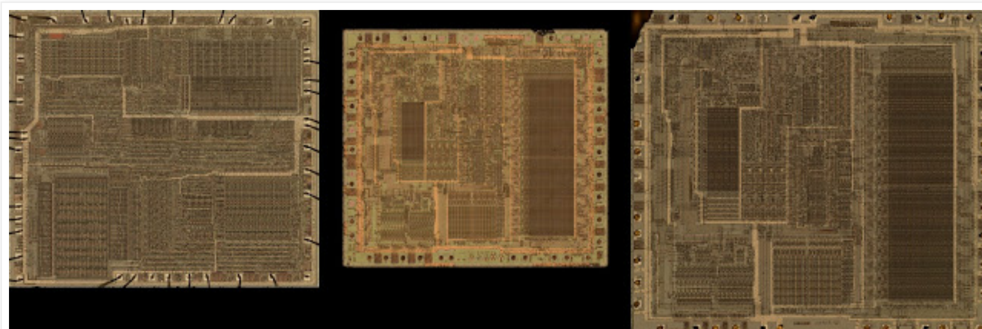
We never got to the bottom of the crazy test mode that started this whole investigation. There's no trace of it in the byte processor ROM, so it must be handled by some other component on the silicon. Something to

investigate for another day perhaps.

Having seen the complexity of the chip, I must confess to a feeling of surprise every time my BBC Micro successfully loads a disc.

Epilogue

Given the 8271 issues with cost, heat, supply chain, complexity, and lack of MFM, it wouldn't be surprising if Intel had had enough with the architecture behind the 8271 and 8273. Intel staggered bravely forward with the 8272, which introduced MFM support and... hang on, let's have a look at a decapitated one of those...



Left to right: Intel 8271, NEC D765, Intel 8272

This is very cheeky! The 8272 die may say "8272 (c) Intel 1979" but it is the same die as a NEC D765, stamped "NEC D765B". It looks like Intel may have licensed the NEC design. The NEC doesn't appear much smaller in terms of die size, but the layout looks much simpler and less busy. Bizarrely, Intel appears to have fabbed the 8272 much larger than the NEC.

Extra references

- StarDot forum thread where the investigation unfolded: [link]
- Sean Riddle's decap page: [link]
- beebjit's 8271 driver: [link]
- 8271 tests and tools (warning: rough) for the BBC Micro: [link]
- Live document for 8271 disassembly: [link]
- Very high resolution die shots of the 8271 and 8273 (beware, will hang browsers!): [link]

Posted by Chris at 11:26 AM

5 comments:

Shawn N. said...

Awesome write-up, thank you!

November 17, 2020 at 4:23 AM

Anonymous said...

Yes, thank you! Not my era, but I have great respect and love for reading about the early days! Thanks again!

November 17, 2020 at 5:21 PM

Unknown said...

Great sleuthing! Wouldn't have expected something so capable to be used as a mere disk controller in the early 70s

November 22, 2020 at 12:53 AM

Mr Z said...

What were the date codes on the D765 and 8272 you compared?

I suspect that the NEC chip is newer, and is a due shrink.

November 22, 2020 at 9:06 PM

Anonymous said...

Fascinating read, well done on all that effort. Something almost certainly sits in the Intel offices today, filed away for decades, about the design I'm sure.

Any possibility that Intel could've stolen the NEC design directly from the chip? Was decapping and imaging like this a thing in the 70s? More likely that they gained access to the original litho masks but chose a different scale.

November 27, 2020 at 5:21 PM

Post a Comment

Home Older Post

Subscribe to: Post Comments (Atom)

Simple theme. Theme images by gaffera. Powered by Blogger.