Jan
25

The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully boot up

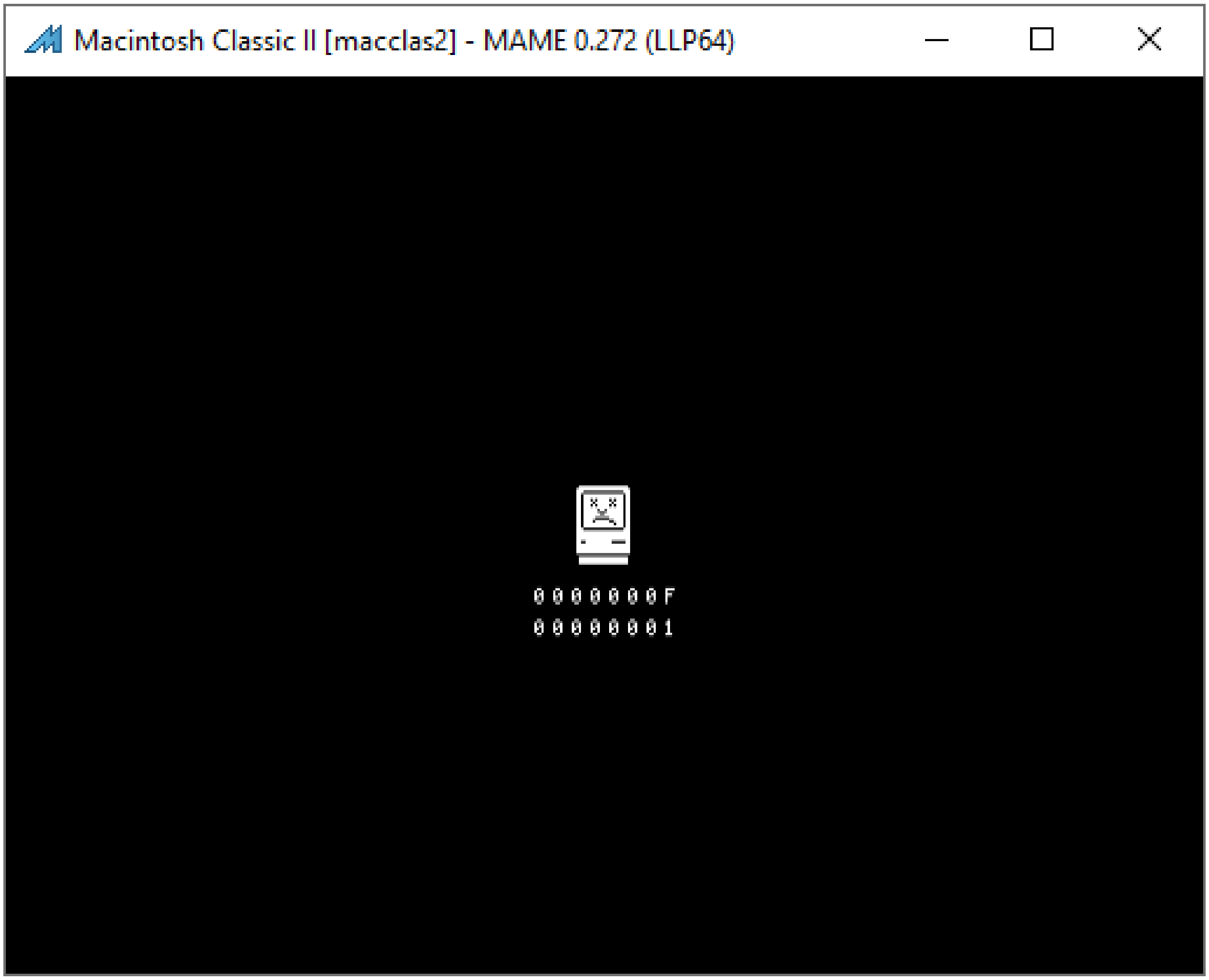
Doug Brown Classic Mac, Reverse engineering 2025-01-25

This is the story of how Apple made a mistake in the ROM of the Macintosh Classic II that probably should have prevented it from booting, but instead, miraculously, its Motorola MC68030 CPU accidentally prevented a crash and saved the day by executing an undefined instruction.

I’ve been playing around with MAME a lot lately. If you haven’t heard of MAME, it’s an emulator that is known best for its support of many arcade games. It’s so much more than that, though! It is also arguably the most complete emulator of 68000-based Mac models, thanks in large part to Arbee‘s incredible efforts. I will admit that I’ve used MAME to play a game or two of Teenage Mutant Ninja Turtles: Turtles in Time, but my main use for it is Mac emulation.

Here’s how this adventure begins. I had been fixing some issues in MAME with the command + power key combination that invokes the debugger, and decided to see if the keystroke also worked on the Classic II. Even though this Mac model has a physical interrupt button on the side, it also has an “Egret” 68HC05 microcontroller for handling the keyboard and mouse (among other things) that should be able to detect the keypress and signal a non-maskable interrupt to the main CPU. I believe the Egret disables this keystroke by default, but MacsBug contains code that sends the command to enable it.

I didn’t get very far while testing the command+power shortcut in MAME’s emulated Classic II, because I observed something very odd. It booted up totally fine in 24-bit addressing mode, but I could not get it to boot at all if I enabled 32-bit addressing, which I needed in order for MacsBug to load. It would just pop up a Sad Mac, complete with the Chimes of Death. On this machine, the death chime is a few notes from the Twilight Zone theme song.



```
0000000F
00000001
```

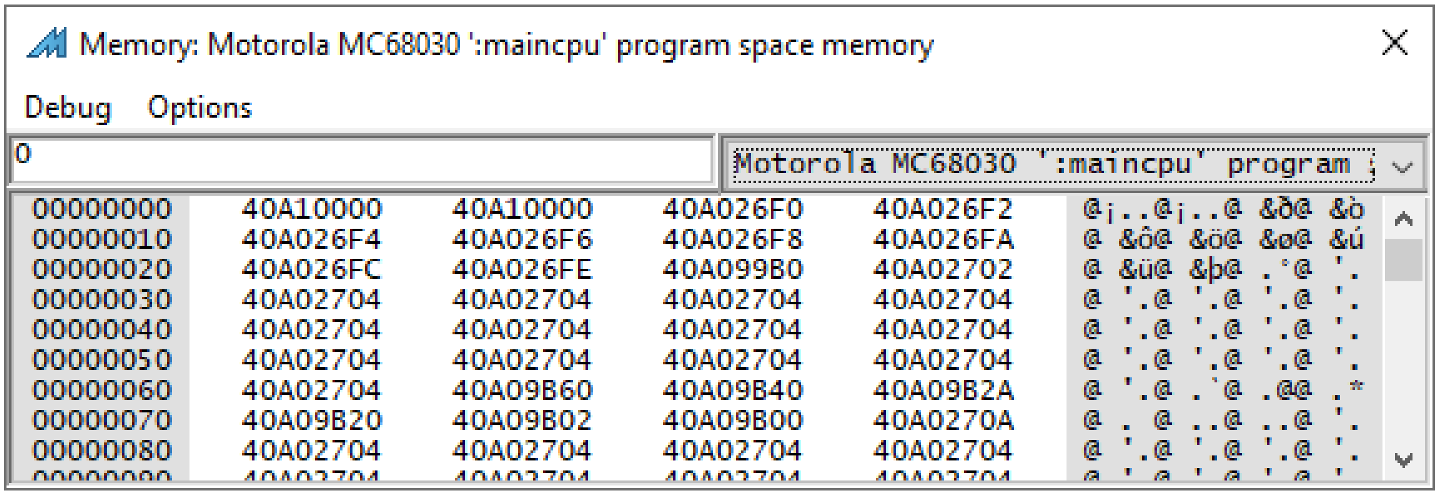
If you're not familiar with Apple's whole 24-bit versus 32-bit addressing saga, I'll briefly summarize it for you here. The original Motorola 68000 processor only had 24 address lines even though it used 32 bits internally for addresses. Apple took those eight extra otherwise unused bits and repurposed them for storing flags as a way to save on RAM, which was scarce at the time. When newer machines/processors came out that supported a full 32-bit address space, the upper byte couldn't be used for flags anymore. Because of that discrepancy, old software would have been incompatible, so newer machines had two modes: 24-bit mode for compatibility with older software, and 32-bit mode for being able to use all of your RAM.

So why was the Classic II failing to boot in 32-bit mode in MAME? What was broken? Arbee also reproduced the issue, so at least I knew I wasn't losing my mind. I assumed it was a random bug in MAME, so I started looking deeper into it to try to understand what needed to be fixed.

According to an old Apple Tech Info Library article, 0000000F means an exception occurred and 00000001 means the exception was a bus error. A bus error on 68k Macs typically means that something tried to access an invalid address, like if you try to read from or write to an expansion card when there isn't one installed.

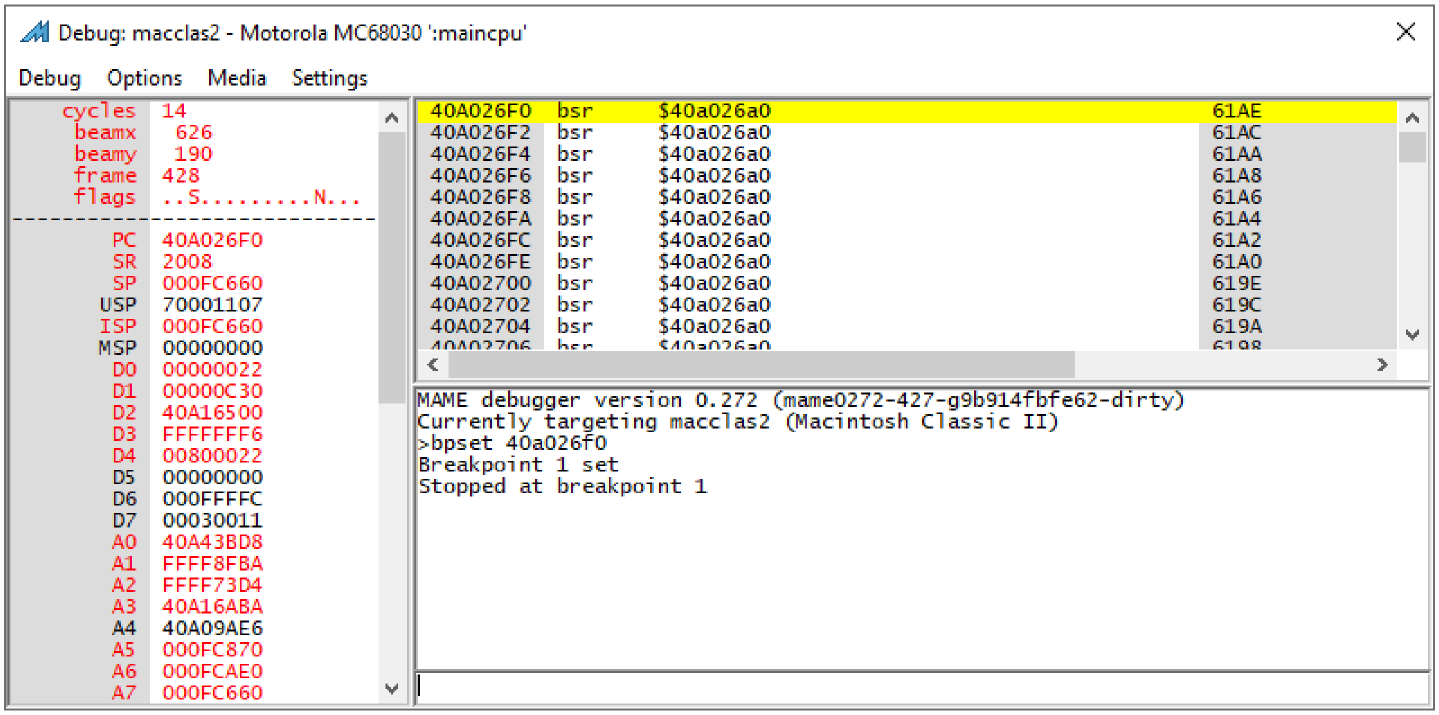
What was the invalid address being accessed? I decided to step through the code using MAME's amazing debugger to understand what was leading to the crash. Comprehending what's going on in the ROM with no context at all can be tricky, but luckily, Apple included symbol maps for a bunch of Mac ROMs with Macintosh Programmer's Workshop (MPW). MPW was Apple's development environment.

Tracing backwards from the actual Sad Mac screen would be difficult, because there is a ton of code involved in setting up and displaying the screen. To make it easier on myself, I decided I would set a breakpoint on the bus error handler and then look backwards from there. The 68030's vector table starts at the very beginning of the address space, and the bus error vector is at 0x00000008. With the Sad Mac error still on the screen, here's what memory looked like at that location:


Memory: Motorola MC68030 ':maincpu' program space memory
Debug Options
Motorola MC68030 ':maincpu' program

```
0
00000000  40A10000  40A10000  40A026F0  40A026F2  @¡..@¡..@ &ð@ &ò
00000010  40A026F4  40A026F6  40A026F8  40A026FA  @ &ô@ &ö@ &ø@ &ú
00000020  40A026FC  40A026FE  40A099B0  40A02702  @ &ü@ &þ@ .°@ '.
00000030  40A02704  40A02704  40A02704  40A02704  @ '.@ '.@ '.@ '.
00000040  40A02704  40A02704  40A02704  40A02704  @ '.@ '.@ '.@ '.
00000050  40A02704  40A02704  40A02704  40A02704  @ '.@ '.@ '.@ '.
00000060  40A02704  40A09B60  40A09B40  40A09B2A  @ '.@ .`@ .@@ .*
00000070  40A09B20  40A09B02  40A09B00  40A0270A  @ . @ ..@ ..@ '.
00000080  40A02704  40A02704  40A02704  40A02704  @ '.@ '.@ '.@ '.
```

This meant the bus error handler was at 0x40A026F0, which is also known as *GenExcps* in the ROM map. I performed a hard reset of the emulated machine, set a breakpoint on that address, and then waited until it hit the breakpoint. It looks like *GenExcps* is a big list of BSR instructions that all jump to 0x40A026A0, which is common error handling code identified in the ROM map as *ToDeepShit*. Nice name, Apple!

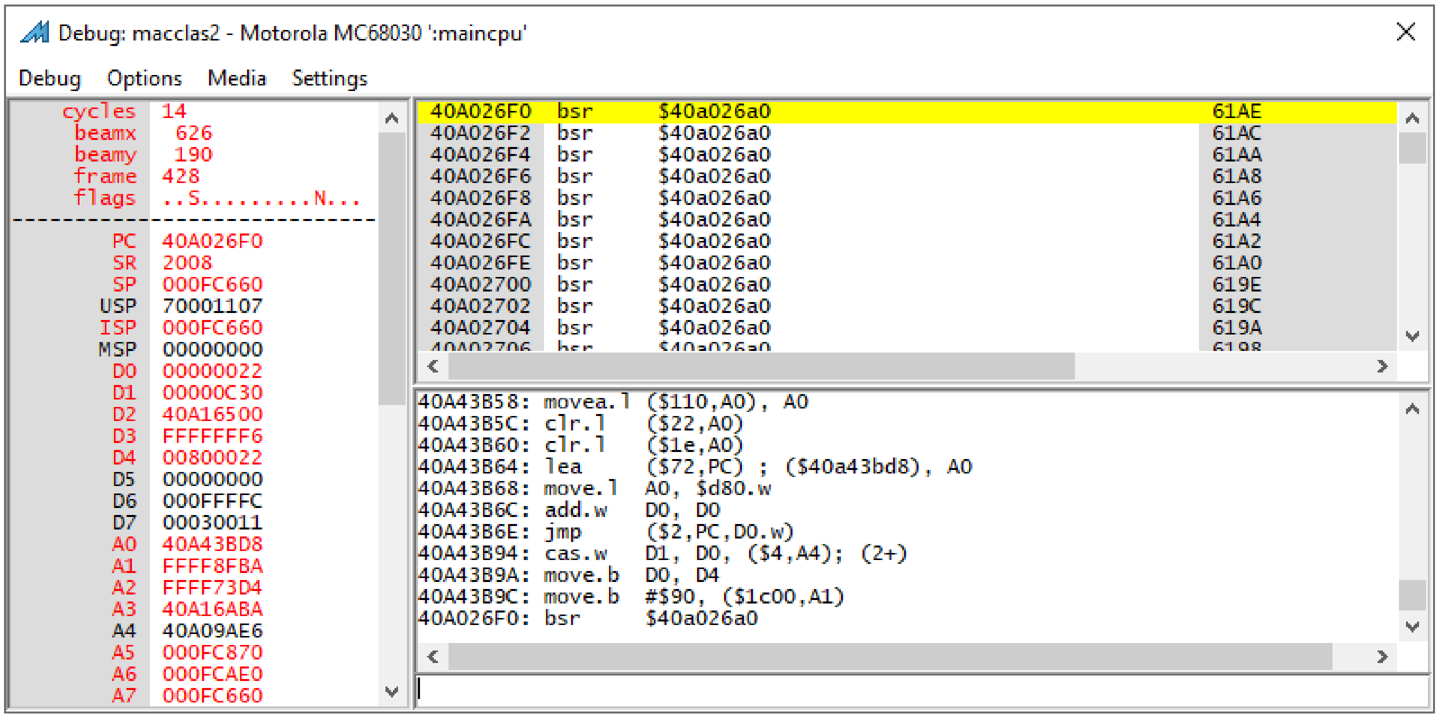

Debug: macclas2 - Motorola MC68030 ':maincpu'
Debug Options Media Settings

```
cycles  14
 beamx  626
 beamy  190
 frame  428
 flags  ..S.........N...
------------------------------
    PC  40A026F0
    SR  2008
    SP  000FC660
   USP  70001107
   ISP  000FC660
   MSP  00000000
    D0  00000022
    D1  00000C30
    D2  40A16500
    D3  FFFFFFF6
    D4  00800022
    D5  00000000
    D6  000FFFFC
    D7  00030011
    A0  40A43BD8
    A1  FFFF8FBA
    A2  FFFF73D4
    A3  40A16ABA
    A4  40A09AE6
    A5  000FC870
    A6  000FCAE0
    A7  000FC660
```

```
40A026F0  bsr    $40a026a0          61AE
40A026F2  bsr    $40a026a0          61AC
40A026F4  bsr    $40a026a0          61AA
40A026F6  bsr    $40a026a0          61A8
40A026F8  bsr    $40a026a0          61A6
40A026FA  bsr    $40a026a0          61A4
40A026FC  bsr    $40a026a0          61A2
40A026FE  bsr    $40a026a0          61A0
40A02700  bsr    $40a026a0          619E
40A02702  bsr    $40a026a0          619C
40A02704  bsr    $40a026a0          619A
```

```
MAME debugger version 0.272 (mame0272-427-g9b914fbfe62-dirty)
Currently targeting macclas2 (Macintosh Classic II)
>bpset 40a026f0
Breakpoint 1 set
Stopped at breakpoint 1
```

Anyway, since MAME hit my breakpoint, this meant Apple's technote was correct about it being a bus error. I was able to use the MAME debugger's *history* command to show a backtrace of instructions that led to this point. The end of the history output is displayed in the bottom pane of the screenshot below:


Debug: macclas2 - Motorola MC68030 ':maincpu'
Debug Options Media Settings

```
cycles 14
 beamx  626
 beamy  190
 frame 428
 flags ..S.........N...
-------------------------------
    PC 40A026F0
    SR 2008
    SP 000FC660
   USP 70001107
   ISP 000FC660
   MSP 00000000
    D0 00000022
    D1 00000C30
    D2 40A16500
    D3 FFFFFFF6
    D4 00800022
    D5 00000000
    D6 000FFFFC
    D7 00030011
    A0 40A43BD8
    A1 FFFF8FBA
    A2 FFFF73D4
    A3 40A16ABA
    A4 40A09AE6
    A5 000FC870
    A6 000FCAE0
    A7 000FC660
```

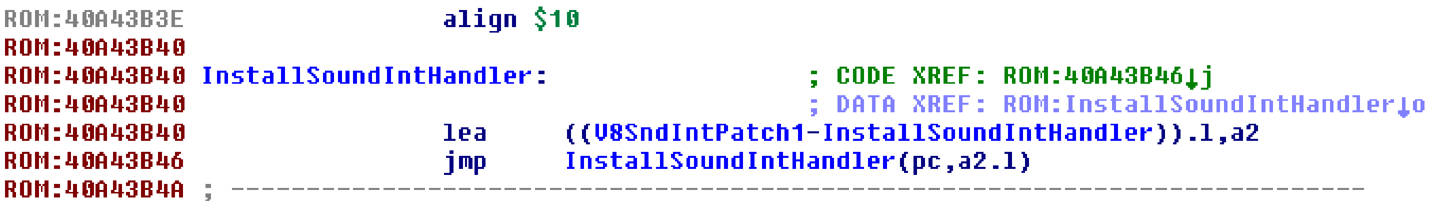
```
40A026F0  bsr      $40a026a0                     61AE
40A026F2  bsr      $40a026a0                     61AC
40A026F4  bsr      $40a026a0                     61AA
40A026F6  bsr      $40a026a0                     61A8
40A026F8  bsr      $40a026a0                     61A6
40A026FA  bsr      $40a026a0                     61A4
40A026FC  bsr      $40a026a0                     61A2
40A026FE  bsr      $40a026a0                     61A0
40A02700  bsr      $40a026a0                     619E
40A02702  bsr      $40a026a0                     619C
40A02704  bsr      $40a026a0                     619A
40A02706  bsr      $40a026a0                     6198
```

```
40A43B58: movea.l ($110,A0), A0
40A43B5C: clr.l   ($22,A0)
40A43B60: clr.l   ($1e,A0)
40A43B64: lea     ($72,PC) ; ($40a43bd8), A0
40A43B68: move.l  A0, $d80.w
40A43B6C: add.w   D0, D0
40A43B6E: jmp     ($2,PC,D0.w)
40A43B94: cas.w   D1, D0, ($4,A4); (2+)
40A43B9A: move.b  D0, D4
40A43B9C: move.b  #$90, ($1c00,A1)
40A026F0: bsr     $40a026a0
```

If we walk upwards, we can see that the instruction that caused the bus error was at 0x40A43B9C:

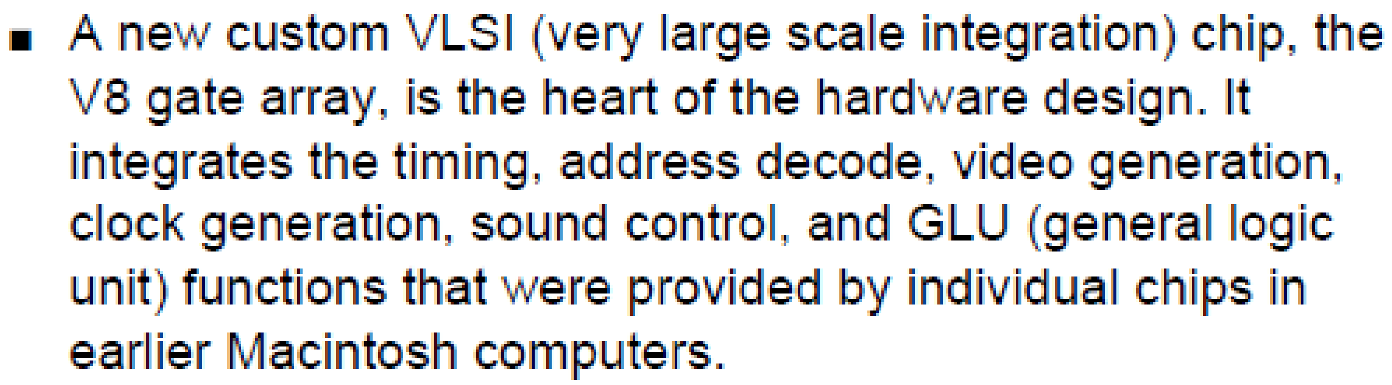
```
move.b #$90, ($1c00,A1)
```

I opened up this section of code in IDA, which I still find myself using with 68k Mac stuff because I'm used to it. It was pretty clearly part of the routine that starts at 0x40A43B40, which is helpfully labeled in the ROM map as *InstallSoundIntHandler*. Let's look at the whole function in more depth.



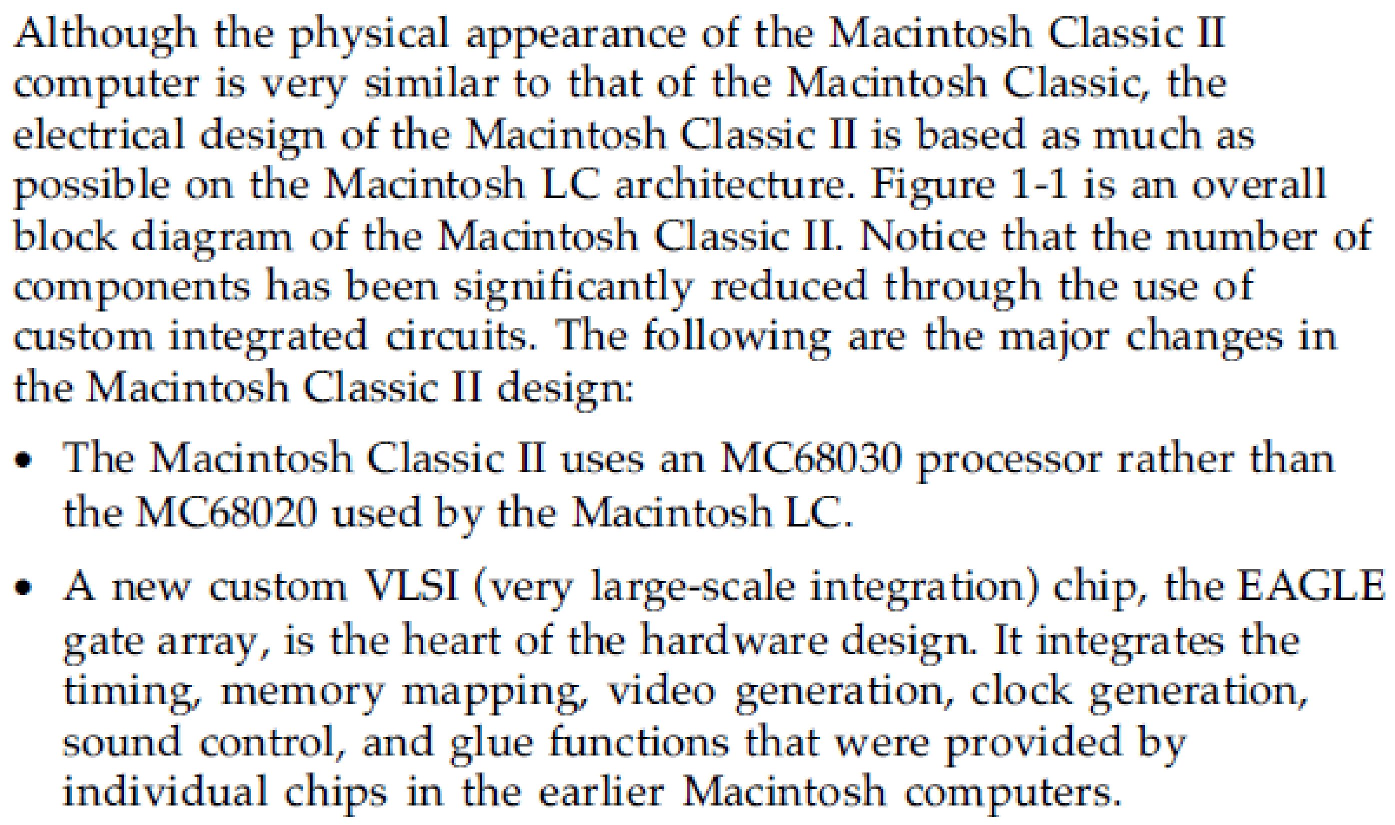
```
ROM:40A43B3E                 align $10
ROM:40A43B40
ROM:40A43B40 InstallSoundIntHandler:                 ; CODE XREF: ROM:40A43B46↓j
ROM:40A43B40                                         ; DATA XREF: ROM:InstallSoundIntHandler↓o
ROM:40A43B40                 lea     ((V8SndIntPatch1-InstallSoundIntHandler)).l,a2
ROM:40A43B46                 jmp     InstallSoundIntHandler(pc,a2.l)
ROM:40A43B4A ; ---------------------------------------------------------------------------
```

The first thing it does is immediately jump to *V8SndIntPatch1*. This appears to be something that was patched into this ROM for handling sound initialization for the V8. For some added context, the Classic II isn't powered by an eight-cylinder gasoline engine; V8 is the name of the custom chip that Apple first used in the Macintosh LC. From the LC hardware developer note:



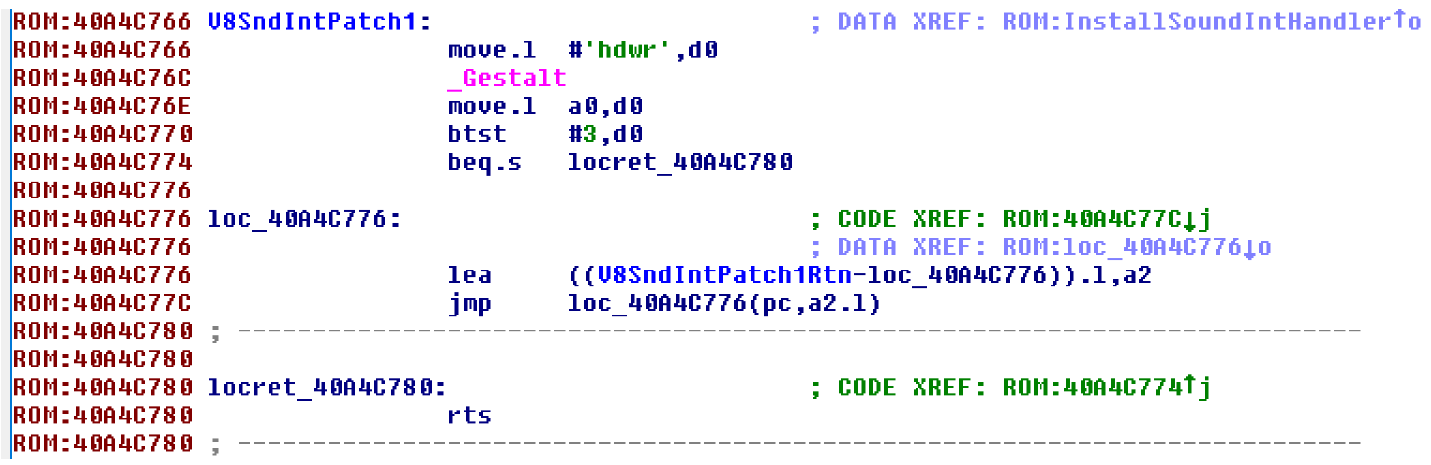
- A new custom VLSI (very large scale integration) chip, the V8 gate array, is the heart of the hardware design. It integrates the timing, address decode, video generation, clock generation, sound control, and GLU (general logic unit) functions that were provided by individual chips in earlier Macintosh computers.

Why are we talking about the LC here? Well, the reason is because the Classic II is architecturally based more on the LC than the original Macintosh Classic. Here's an explanation from the corresponding developer note for the Classic II:



> Although the physical appearance of the Macintosh Classic II computer is very similar to that of the Macintosh Classic, the electrical design of the Macintosh Classic II is based as much as possible on the Macintosh LC architecture. Figure 1-1 is an overall block diagram of the Macintosh Classic II. Notice that the number of components has been significantly reduced through the use of custom integrated circuits. The following are the major changes in the Macintosh Classic II design:
>
> - The Macintosh Classic II uses an MC68030 processor rather than the MC68020 used by the Macintosh LC.
> - A new custom VLSI (very large-scale integration) chip, the EAGLE gate array, is the heart of the hardware design. It integrates the timing, memory mapping, video generation, clock generation, sound control, and glue functions that were provided by individual chips in the earlier Macintosh computers.

The text description of the EAGLE gate array is very similar to that of the V8, so it should come as no surprise that the chips themselves are very similar too. MAME handles them both in the same source file. The point I'm trying to make here is that it makes sense that the Classic II's ROM has code referring to the V8. With that info out of the way, let's look at *V8SndIntPatch1*:



```
ROM:40A4C766 V8SndIntPatch1:                        ; DATA XREF: ROM:InstallSoundIntHandler↑o
ROM:40A4C766                 move.l  #'hdwr',d0
ROM:40A4C76C                 _Gestalt
ROM:40A4C76E                 move.l  a0,d0
ROM:40A4C770                 btst    #3,d0
ROM:40A4C774                 beq.s   locret_40A4C780
ROM:40A4C776
ROM:40A4C776 loc_40A4C776:                          ; CODE XREF: ROM:40A4C77C↓j
ROM:40A4C776                                        ; DATA XREF: ROM:loc_40A4C776↓o
ROM:40A4C776                 lea     ((V8SndIntPatch1Rtn-loc_40A4C776)).l,a2
ROM:40A4C77C                 jmp     loc_40A4C776(pc,a2.l)
ROM:40A4C780 ; ---------------------------------------------------------------------------
ROM:40A4C780
ROM:40A4C780 locret_40A4C780:                       ; CODE XREF: ROM:40A4C774↑j
ROM:40A4C780                 rts
ROM:40A4C780 ; ---------------------------------------------------------------------------
```

This chunk of code is calling the Gestalt trap, which is how you determine various info about the Mac. In particular, it's using the gestaltHardwareAttr selector, which is defined as ‘hdwr’ in Apple’s public header files.

If bit 3 (gestaltHasASC) isn’t set in the response, it bails and returns. Otherwise, it jumps to *V8SndIntPatch1Rtn* at 0x40A43B4A, which you can see in the history trace in the MAME debugger screenshot from earlier. I went pretty deep into the hardware tables for the Classic II and can confirm that gestaltHasASC is definitely set on the Classic II. After all, the EAGLE contains a stripped-down equivalent of the Apple Sound Chip (ASC).

Now, let’s take a look at *V8SndIntPatch1Rtn*:

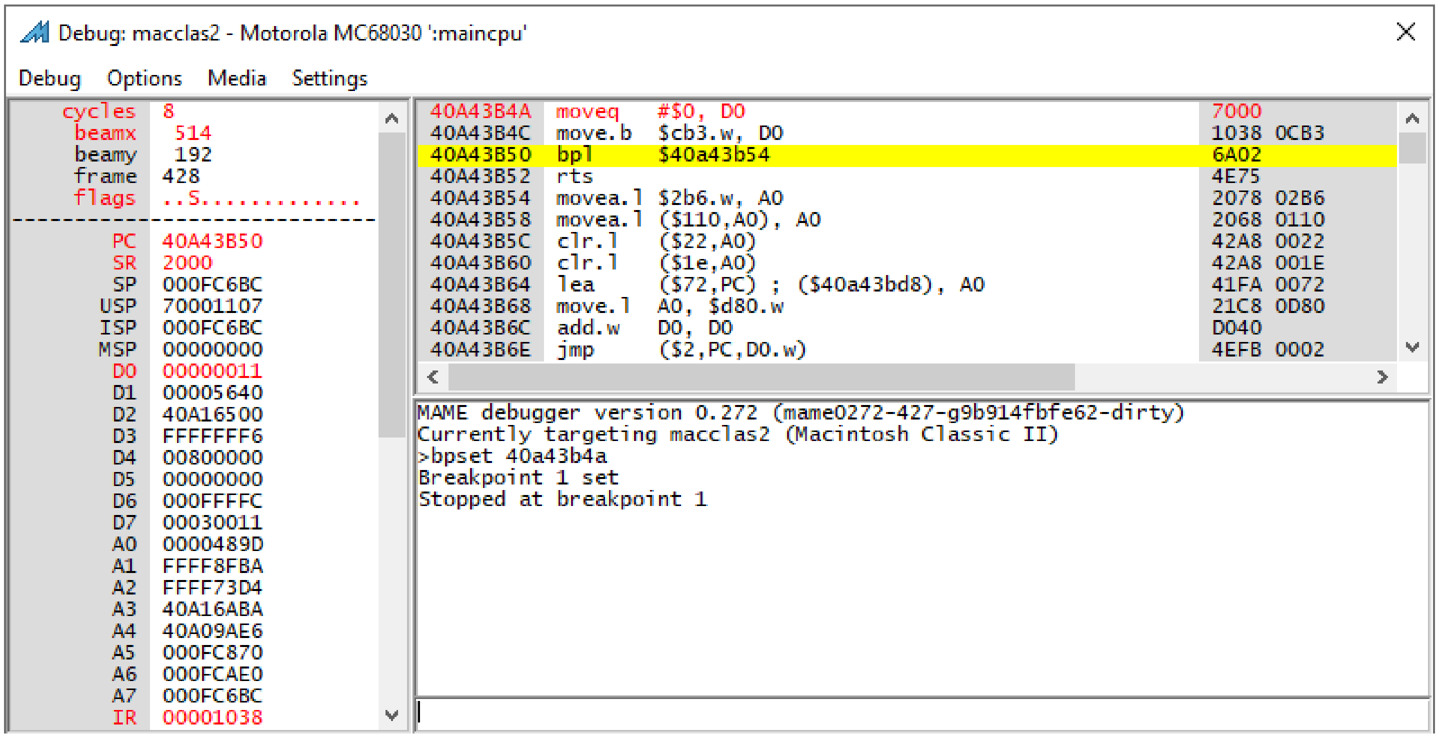
```
ROM:40A43B4A V8SndIntPatch1Rtn:                         ; DATA XREF: ROM:loc_40A4C776↓o
ROM:40A43B4A                 moveq   #0,d0
ROM:40A43B4C                 move.b  (byte_CB3).w,d0
ROM:40A43B50                 bpl.s   loc_40A43B54
ROM:40A43B52                 rts
ROM:40A43B54 ; ---------------------------------------------------------------------------
ROM:40A43B54
ROM:40A43B54 loc_40A43B54:                              ; CODE XREF: ROM:40A43B50↑j
ROM:40A43B54                 movea.l (dword_2B6).w,a0
ROM:40A43B58                 movea.l $110(a0),a0
ROM:40A43B5C                 clr.l   $22(a0)
ROM:40A43B60                 clr.l   $1E(a0)
ROM:40A43B64                 lea     loc_40A43BD8,a0
ROM:40A43B68                 move.l  a0,(dword_D80).w
ROM:40A43B6C                 add.w   d0,d0
ROM:40A43B6E                 jmp     loc_40A43B72(pc,d0.w)
ROM:40A43B72 ; ---------------------------------------------------------------------------
ROM:40A43B72
ROM:40A43B72 loc_40A43B72:                              ; CODE XREF: ROM:40A43B6E↑j
ROM:40A43B72                 bra.s   loc_40A43B92
ROM:40A43B74 ; ---------------------------------------------------------------------------
ROM:40A43B74                 bra.s   loc_40A43B92
ROM:40A43B76 ; ---------------------------------------------------------------------------
ROM:40A43B76                 bra.s   loc_40A43B92
ROM:40A43B78 ; ---------------------------------------------------------------------------
ROM:40A43B78                 bra.s   loc_40A43B92
ROM:40A43B7A ; ---------------------------------------------------------------------------
ROM:40A43B7A                 bra.s   locret_40A43BA4
ROM:40A43B7C ; ---------------------------------------------------------------------------
ROM:40A43B7C                 bra.s   loc_40A43BA6
ROM:40A43B7E ; ---------------------------------------------------------------------------
ROM:40A43B7E                 bra.s   loc_40A43B92
ROM:40A43B80 ; ---------------------------------------------------------------------------
ROM:40A43B80                 bra.s   loc_40A43BB2
ROM:40A43B82 ; ---------------------------------------------------------------------------
ROM:40A43B82                 bra.s   loc_40A43BA6
ROM:40A43B84 ; ---------------------------------------------------------------------------
ROM:40A43B84                 bra.s   loc_40A43BA6
ROM:40A43B86 ; ---------------------------------------------------------------------------
ROM:40A43B86                 bra.s   loc_40A43BA6
ROM:40A43B88 ; ---------------------------------------------------------------------------
ROM:40A43B88                 bra.s   loc_40A43BA6
ROM:40A43B8A ; ---------------------------------------------------------------------------
ROM:40A43B8A                 bra.s   loc_40A43BA6
ROM:40A43B8C ; ---------------------------------------------------------------------------
ROM:40A43B8C                 bra.s   locret_40A43BA4
ROM:40A43B8E ; ---------------------------------------------------------------------------
ROM:40A43B8E                 bra.s   loc_40A43BA6
ROM:40A43B90 ; ---------------------------------------------------------------------------
ROM:40A43B90                 bra.s   loc_40A43BA6
ROM:40A43B92 ; ---------------------------------------------------------------------------
ROM:40A43B92
ROM:40A43B92 loc_40A43B92:                              ; CODE XREF: ROM:loc_40A43B72↑j
ROM:40A43B92                                            ; ROM:40A43B74↑j ...
ROM:40A43B92                 movea.l (dword_CEC).w,a1
ROM:40A43B96                 bclr    #4,$1800(a1)
ROM:40A43B9C                 move.b  #$90,$1C00(a1)
ROM:40A43BA2                 rts
ROM:40A43BA4 ; ---------------------------------------------------------------------------
ROM:40A43BA4
ROM:40A43BA4 locret_40A43BA4:                           ; CODE XREF: ROM:40A43B7A↑j
ROM:40A43BA4                                            ; ROM:40A43B8C↑j
ROM:40A43BA4                 rts
ROM:40A43BA6 ; ---------------------------------------------------------------------------
ROM:40A43BA6
ROM:40A43BA6 loc_40A43BA6:                              ; CODE XREF: ROM:40A43B7C↑j
```

```
ROM:40A43BA6                                        ; ROM:40A43B82↑j ...
ROM:40A43BA6                movea.l (dword_CEC).w,a0
ROM:40A43BAA                move.b  #$90,$13(a0)
ROM:40A43BB0                rts
ROM:40A43BB2 ; ---------------------------------------------------------------------------
ROM:40A43BB2
ROM:40A43BB2 loc_40A43BB2:                          ; CODE XREF: ROM:40A43B80↑j
ROM:40A43BB2                movea.l (dword_CEC).w,a0
ROM:40A43BB6                move.b  #2,8(a0)
ROM:40A43BBC                rts
ROM:40A43BBE ; ---------------------------------------------------------------------------
```

Phew! This is a decent amount of code. It’s not that complicated though. I’ll explain the important stuff. You can see the instruction that leads to the Sad Mac at 0x40A43B9C. If you start at the top, what’s happening is it’s loading a byte from RAM at 0xCB3 into register D0:

```
moveq #$0,d0
move.b (byte_CB3).w,d0
```

If you know where to look out there, you can discover that this global variable is called BoxFlag and contains a value identifying which machine you have. If I step through this code in MAME, I can see that D0 ends up loaded with the value 0x11 = 17, which is correct for the Classic II.



```
  cycles  8
   beamx   514
   beamy   192
   frame  428
   flags  ..S.............
----------------------------
      PC  40A43B50
      SR  2000
      SP  000FC6BC
     USP  70001107
     ISP  000FC6BC
     MSP  00000000
      D0  00000011
      D1  00005640
      D2  40A16500
      D3  FFFFFFF6
      D4  00800000
      D5  00000000
      D6  000FFFFC
      D7  00030011
      A0  0000489D
      A1  FFFF8FBA
      A2  FFFF73D4
      A3  40A16ABA
      A4  40A09AE6
      A5  000FC870
      A6  000FCAE0
      A7  000FC6BC
      IR  00001038
```

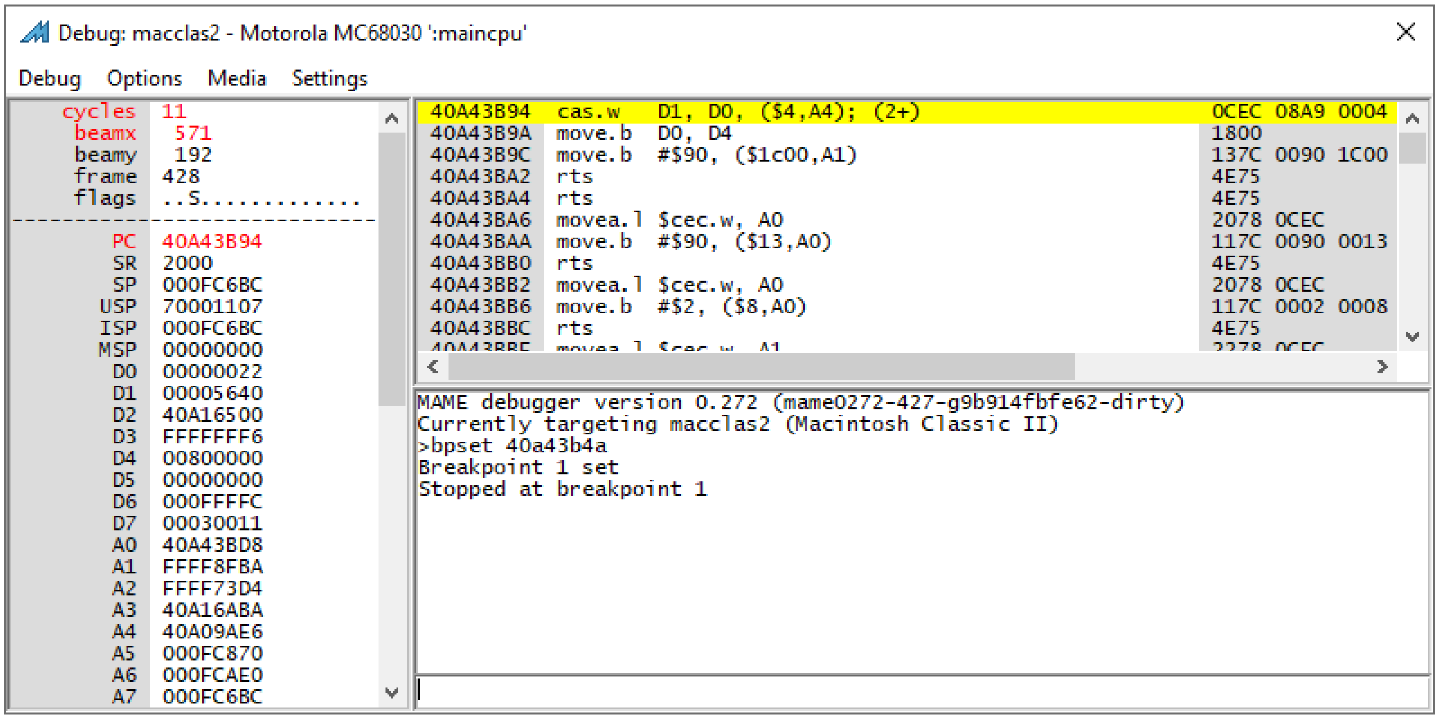
```
40A43B4A  moveq    #$0, D0                                 7000
40A43B4C  move.b   $cb3.w, D0                              1038 0CB3
40A43B50  bpl      $40a43b54                               6A02
40A43B52  rts                                              4E75
40A43B54  movea.l  $2b6.w, A0                              2078 02B6
40A43B58  movea.l  ($110,A0), A0                           2068 0110
40A43B5C  clr.l    ($22,A0)                                42A8 0022
40A43B60  clr.l    ($1e,A0)                                42A8 001E
40A43B64  lea      ($72,PC) ; ($40a43bd8), A0              41FA 0072
40A43B68  move.l   A0, $d80.w                              21C8 0D80
40A43B6C  add.w    D0, D0                                  D040
40A43B6E  jmp      ($2,PC,D0.w)                            4EFB 0002
```

```
MAME debugger version 0.272 (mame0272-427-g9b914fbfe62-dirty)
Currently targeting macclas2 (Macintosh Classic II)
>bpset 40a43b4a
Breakpoint 1 set
Stopped at breakpoint 1
```

Continuing further through the code, some other stuff happens, and then at 0x40A43B6C the value in D0 ends up being doubled (so it turns into 0x22). Immediately after this, it is used as an offset in a jump instruction. Here's IDA's syntax for the jump, because it's more intuitive than what MAME displays:

```
add.w d0,d0
jmp loc_40A43B72(pc,d0.w)
```

Since D0 ends up as 0x22 after being doubled, we jump to 0x40A43B72 + 0x22 = 0x40A43B94, and here's what that code looks like in MAME's debugger when we reach it:

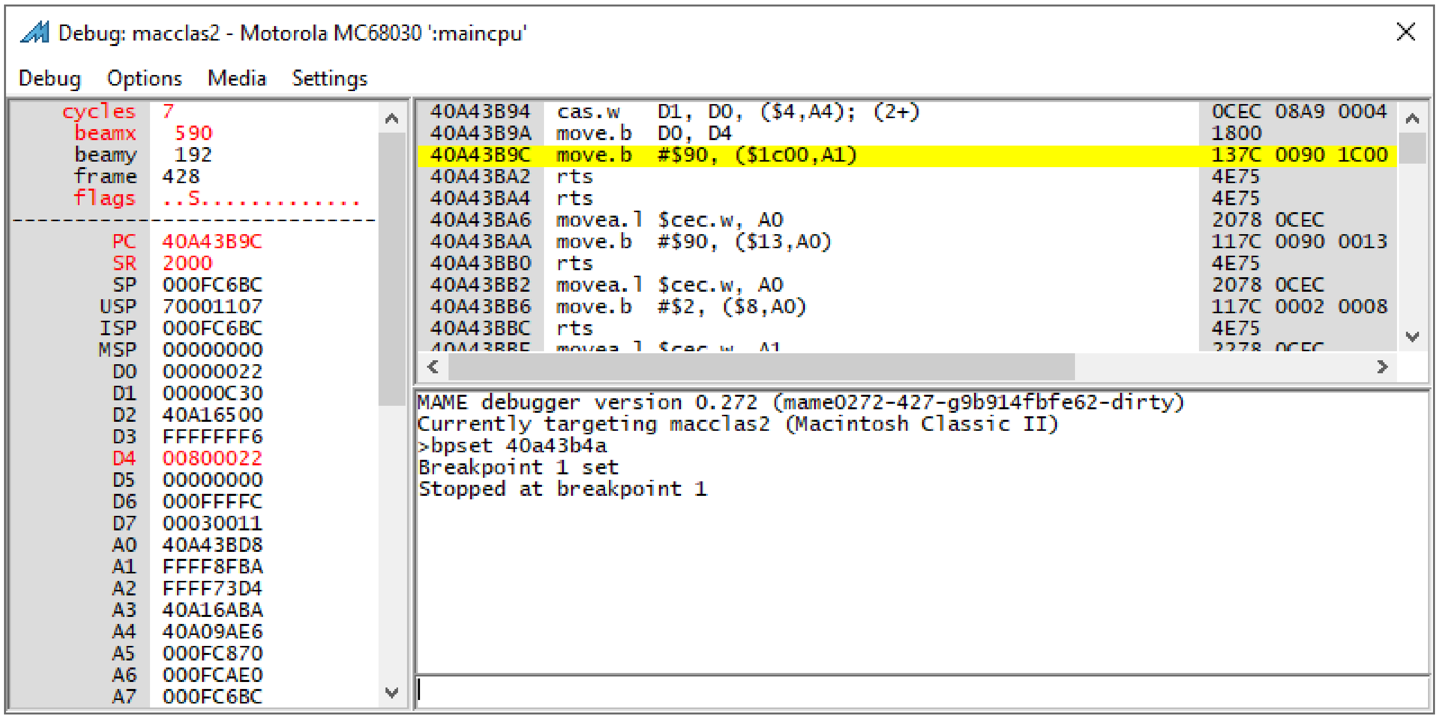


```
   cycles  11
    beamx   571
    beamy   192
    frame  428
    flags  ..S.............
----------------------------
       PC  40A43B94
       SR  2000
       SP  000FC6BC
      USP  70001107
      ISP  000FC6BC
      MSP  00000000
       D0  00000022
       D1  00005640
       D2  40A16500
       D3  FFFFFFF6
       D4  00800000
       D5  00000000
       D6  000FFFFC
       D7  00030011
       A0  40A43BD8
       A1  FFFF8FBA
       A2  FFFF73D4
       A3  40A16ABA
       A4  40A09AE6
       A5  000FC870
       A6  000FCAE0
       A7  000FC6BC
```

```
40A43B94  cas.w    D1, D0, ($4,A4); (2+)          0CEC 08A9 0004
40A43B9A  move.b   D0, D4                         1800
40A43B9C  move.b   #$90, ($1c00,A1)               137C 0090 1C00
40A43BA2  rts                                     4E75
40A43BA4  rts                                     4E75
40A43BA6  movea.l  $cec.w, A0                     2078 0CEC
40A43BAA  move.b   #$90, ($13,A0)                 117C 0090 0013
40A43BB0  rts                                     4E75
40A43BB2  movea.l  $cec.w, A0                     2078 0CEC
40A43BB6  move.b   #$2, ($8,A0)                   117C 0002 0008
40A43BBC  rts                                     4E75
40A43BBE  movea.l  $cec.w, A1                     2278 0CEC
```

```
MAME debugger version 0.272 (mame0272-427-g9b914fbfe62-dirty)
Currently targeting macclas2 (Macintosh Classic II)
>bpset 40a43b4a
Breakpoint 1 set
Stopped at breakpoint 1
```

Stepping further through the code, you can see we eventually reach the instruction that causes a Sad Mac. Let's see what all the registers look like before it's executed:

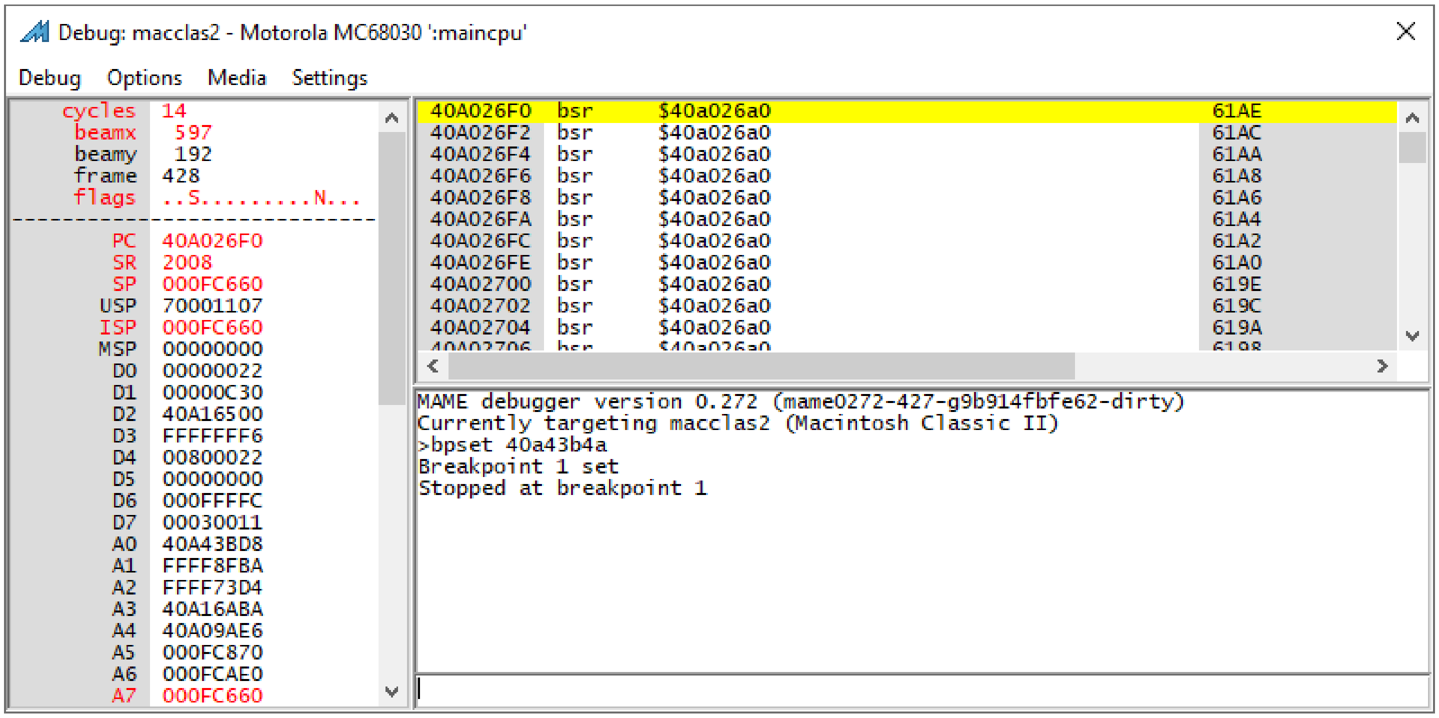

Debug: macclas2 - Motorola MC68030 ':maincpu'
Debug Options Media Settings

```
cycles  7
 beamx   590
 beamy   192
 frame  428
 flags  ..S.............
---------------------------
    PC  40A43B9C
    SR  2000
    SP  000FC6BC
   USP  70001107
   ISP  000FC6BC
   MSP  00000000
    D0  00000022
    D1  00000C30
    D2  40A16500
    D3  FFFFFFF6
    D4  00800022
    D5  00000000
    D6  000FFFFC
    D7  00030011
    A0  40A43BD8
    A1  FFFF8FBA
    A2  FFFF73D4
    A3  40A16ABA
    A4  40A09AE6
    A5  000FC870
    A6  000FCAE0
    A7  000FC6BC
```

```
40A43B94  cas.w   D1, D0, ($4,A4); (2+)        0CEC 08A9 0004
40A43B9A  move.b  D0, D4                       1800
40A43B9C  move.b  #$90, ($1c00,A1)             137C 0090 1C00
40A43BA2  rts                                  4E75
40A43BA4  rts                                  4E75
40A43BA6  movea.l $cec.w, A0                   2078 0CEC
40A43BAA  move.b  #$90, ($13,A0)               117C 0090 0013
40A43BB0  rts                                  4E75
40A43BB2  movea.l $cec.w, A0                   2078 0CEC
40A43BB6  move.b  #$2, ($8,A0)                 117C 0002 0008
40A43BBC  rts                                  4E75
```

```
MAME debugger version 0.272 (mame0272-427-g9b914fbfe62-dirty)
Currently targeting macclas2 (Macintosh Classic II)
>bpset 40a43b4a
Breakpoint 1 set
Stopped at breakpoint 1
```

Hmm, that's odd. This crashing instruction writes the value 0x90 to an offset 0x1C00 bytes past the address stored in A1. A1 is set to 0xFFFF8FBA, so the address where the write occurs at is 0xFFFF8FBA + 0x1C00 = 0xFFFFABBA. This is a totally invalid address on the Classic II! No wonder we get a Sad Mac. As expected, as soon as we step into this instruction, instead of reaching the RTS instruction just below it, we end up in the code path for displaying a Sad Mac error at 0x40A026F0. This is definitely where everything craps out.



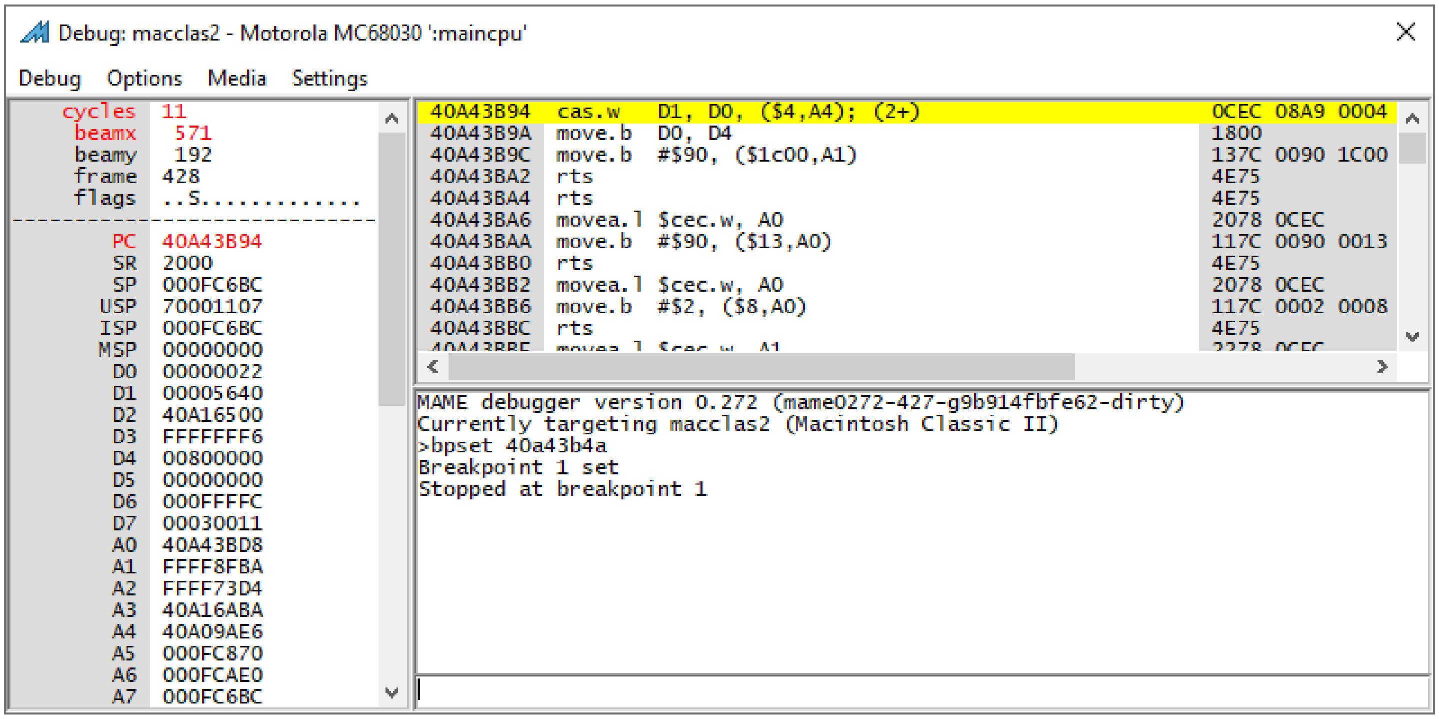
```
cycles  14
 beamx   597
 beamy   192
 frame  428
 flags  ..S.........N...
------------------------------
    PC  40A026F0
    SR  2008
    SP  000FC660
   USP  70001107
   ISP  000FC660
   MSP  00000000
    D0  00000022
    D1  00000C30
    D2  40A16500
    D3  FFFFFFF6
    D4  00800022
    D5  00000000
    D6  000FFFFC
    D7  00030011
    A0  40A43BD8
    A1  FFFF8FBA
    A2  FFFF73D4
    A3  40A16ABA
    A4  40A09AE6
    A5  000FC870
    A6  000FCAE0
    A7  000FC660
```

```
40A026F0  bsr      $40a026a0                        61AE
40A026F2  bsr      $40a026a0                        61AC
40A026F4  bsr      $40a026a0                        61AA
40A026F6  bsr      $40a026a0                        61A8
40A026F8  bsr      $40a026a0                        61A6
40A026FA  bsr      $40a026a0                        61A4
40A026FC  bsr      $40a026a0                        61A2
40A026FE  bsr      $40a026a0                        61A0
40A02700  bsr      $40a026a0                        619E
40A02702  bsr      $40a026a0                        619C
40A02704  bsr      $40a026a0                        619A
```

```
MAME debugger version 0.272 (mame0272-427-g9b914fbfe62-dirty)
Currently targeting macclas2 (Macintosh Classic II)
>bpset 40a43b4a
Breakpoint 1 set
Stopped at breakpoint 1
```

Okay, so now I had a pretty good idea of what was happening in MAME. A1 had a junk value, so the ROM code was writing to an invalid address. FFFFABBA dabba doo! I decided to investigate further to understand how A1 came to be loaded with a bad address. And that's when I discovered something really bizarre.

Let's take a closer look at one of the earlier screenshots, after we used the value of D0 (BoxFlag) to jump to the correct chunk of code for the Classic II:



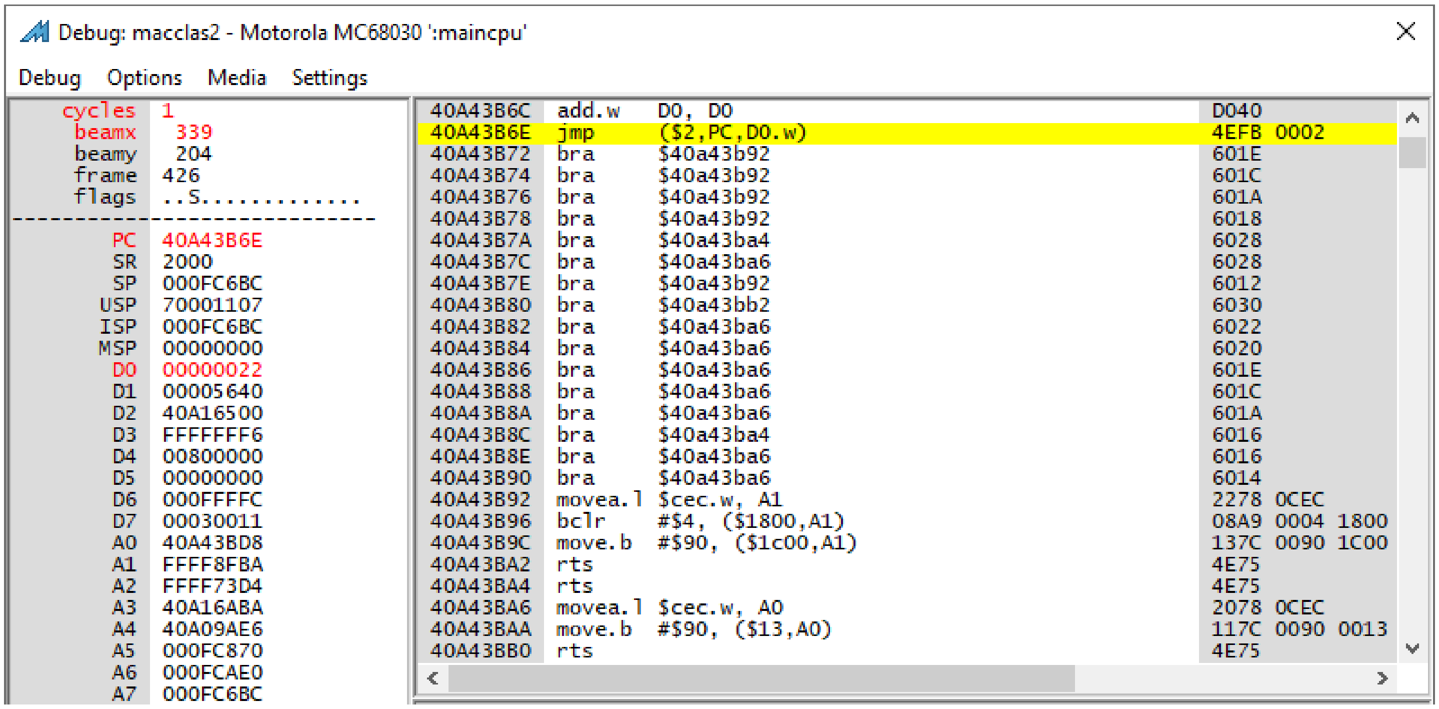
```
  cycles  11
   beamx   571
   beamy   192
   frame  428
   flags  ..S.............
------------------------------
      PC  40A43B94
      SR  2000
      SP  000FC6BC
     USP  70001107
     ISP  000FC6BC
     MSP  00000000
      D0  00000022
      D1  00005640
      D2  40A16500
      D3  FFFFFFF6
      D4  00800000
      D5  00000000
      D6  000FFFFC
      D7  00030011
      A0  40A43BD8
      A1  FFFF8FBA
      A2  FFFF73D4
      A3  40A16ABA
      A4  40A09AE6
      A5  000FC870
      A6  000FCAE0
      A7  000FC6BC
```

```
40A43B94  cas.w    D1, D0, ($4,A4); (2+)          0CEC 08A9 0004
40A43B9A  move.b   D0, D4                         1800
40A43B9C  move.b   #$90, ($1c00,A1)               137C 0090 1C00
40A43BA2  rts                                     4E75
40A43BA4  rts                                     4E75
40A43BA6  movea.l  $cec.w, A0                     2078 0CEC
40A43BAA  move.b   #$90, ($13,A0)                 117C 0090 0013
40A43BB0  rts                                     4E75
40A43BB2  movea.l  $cec.w, A0                     2078 0CEC
40A43BB6  move.b   #$2, ($8,A0)                   117C 0002 0008
40A43BBC  rts                                     4E75
```

```
MAME debugger version 0.272 (mame0272-427-g9b914fbfe62-dirty)
Currently targeting macclas2 (Macintosh Classic II)
>bpset 40a43b4a
Breakpoint 1 set
Stopped at breakpoint 1
```

I thought about this some more, and eventually realized that something absolutely crazy happened here. We were supposed to be jumping into a table of BRA.S instructions, one for each possible BoxFlag value. That's why we added D0 to itself before using it as a jump offset — each BRA.S instruction is two bytes long, so the index into the table needed to be doubled to turn it into a byte offset. Why didn't we end up pointing at a BRA.S instruction? And where did this CAS.W instruction come from?

If you look closely at the table of branches below the JMP instruction at 0x40A43B6E, there are only 16 entries in the table, corresponding to BoxFlags 0 through 15. The Classic II is BoxFlag 17!



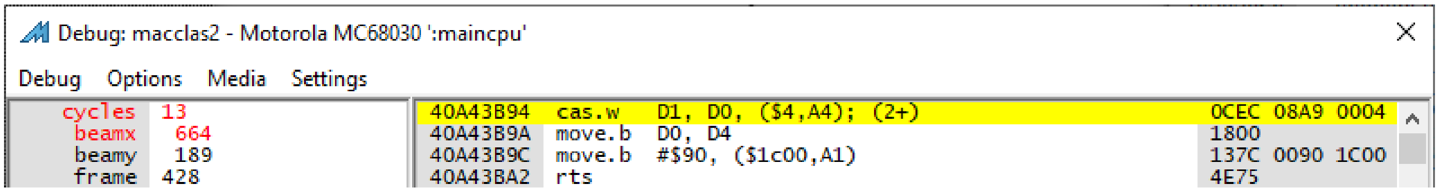
Debug: macclas2 - Motorola MC68030 ':maincpu'

Debug Options Media Settings

```
 cycles  1                    40A43B6C  add.w    D0, D0                 D040
  beamx   339                 40A43B6E  jmp      ($2,PC,D0.w)           4EFB 0002
  beamy   204                 40A43B72  bra      $40a43b92              601E
  frame  426                  40A43B74  bra      $40a43b92              601C
  flags  ..S............      40A43B76  bra      $40a43b92              601A
-----------------------       40A43B78  bra      $40a43b92              6018
     PC  40A43B6E             40A43B7A  bra      $40a43ba4              6028
     SR  2000                 40A43B7C  bra      $40a43ba6              6028
     SP  000FC6BC             40A43B7E  bra      $40a43b92              6012
    USP  70001107             40A43B80  bra      $40a43bb2              6030
    ISP  000FC6BC             40A43B82  bra      $40a43ba6              6022
    MSP  00000000             40A43B84  bra      $40a43ba6              6020
     D0  00000022             40A43B86  bra      $40a43ba6              601E
     D1  00005640             40A43B88  bra      $40a43ba6              601C
     D2  40A16500             40A43B8A  bra      $40a43ba6              601A
     D3  FFFFFFF6             40A43B8C  bra      $40a43ba4              6016
     D4  00800000             40A43B8E  bra      $40a43ba6              6016
     D5  00000000             40A43B90  bra      $40a43ba6              6014
     D6  000FFFFC             40A43B92  movea.l  $cec.w, A1             2278 0CEC
     D7  00030011             40A43B96  bclr     #$4, ($1800,A1)        08A9 0004 1800
     A0  40A43BD8             40A43B9C  move.b   #$90, ($1c00,A1)       137C 0090 1C00
     A1  FFFF8FBA             40A43BA2  rts                             4E75
     A2  FFFF73D4             40A43BA4  rts                             4E75
     A3  40A16ABA             40A43BA6  movea.l  $cec.w, A0             2078 0CEC
     A4  40A09AE6             40A43BAA  move.b   #$90, ($13,A0)         117C 0090 0013
     A5  000FC870             40A43BB0  rts                             4E75
     A6  000FCAE0
     A7  000FC6BC
```

As I said earlier, the calculated offset we jump to is 0x40A43B94, which is not even supposed to be the start of an instruction. It's smack dab in the middle of the MOVEA.L instruction at 0x40A43B92, which is the instruction that loads A1 with a real address that this code can use for enabling the sound interrupt.

When we jump to 0x40A43B94, we aren't running intended code anymore. The CPU gets out of sync with the path that the code was designed to follow. 0x0CEC was supposed to be the second half of the MOVEA.L instruction — the address in RAM to load from — but instead, it is being treated as the start of a new instruction.



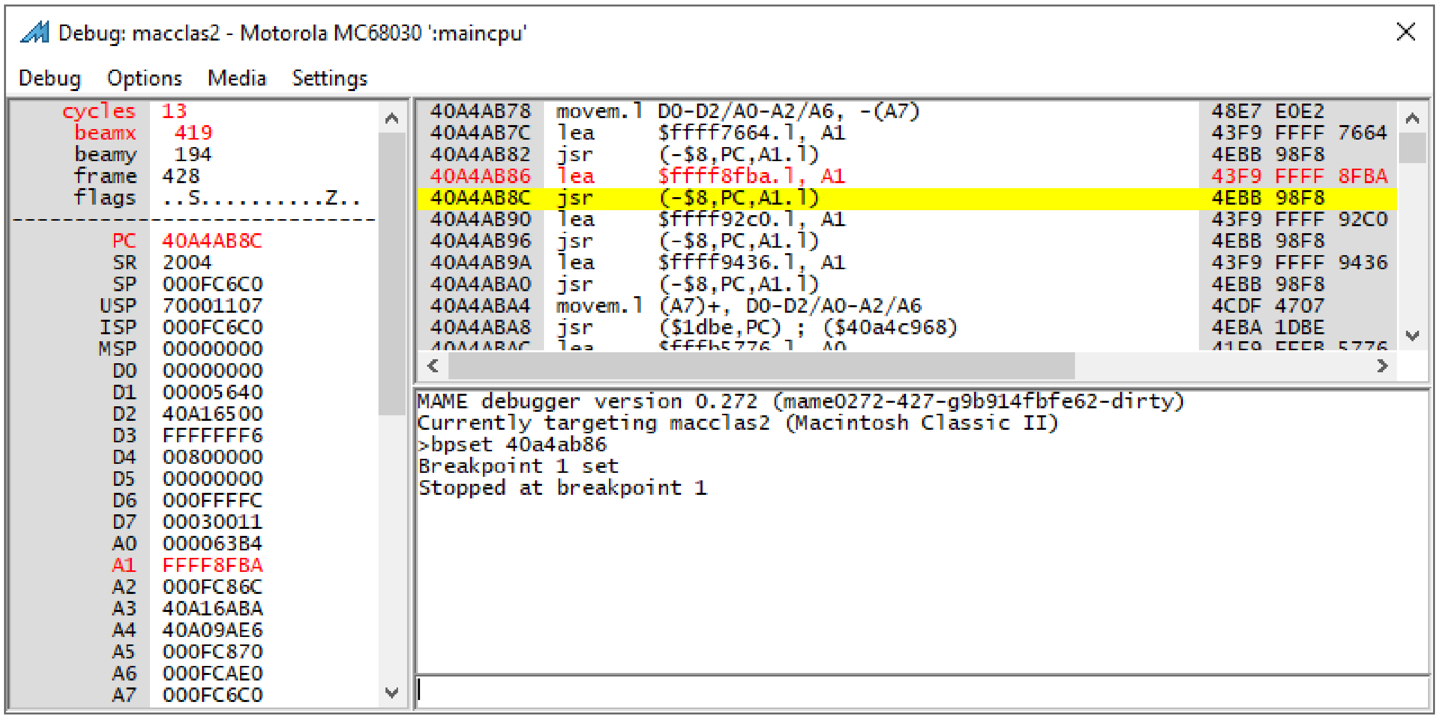
Debug: macclas2 - Motorola MC68030 ':maincpu'

Debug Options Media Settings

```
 cycles  13      40A43B94  cas.w    D1, D0, ($4,A4); (2+)      0CEC 08A9 0004
  beamx   664    40A43B9A  move.b   D0, D4                     1800
  beamy   189    40A43B9C  move.b   #$90, ($1c00,A1)           137C 0090 1C00
  frame  428     40A43BA2  rts                                 4E75
```

The CPU doesn't get back in sync right away. We execute this mystery CAS (compare and swap) instruction, and then an unintended "MOVE.B D0, D4" instruction, before finally reaching a real MOVE.B instruction at 0x40A43B9C — the instruction that crashes. That is the point where the CPU has returned to running code that Apple actually wanted it to run. But unfortunately, A1 contains an invalid address because the code that was supposed to fill it out wasn't reached, so of course everything crashes when we try to write to A1 + 0x1C00. It all makes sense.

Going back further, A1 gets loaded with the "junk" value of 0xFFFF8FBA as part of the initial jump to InstallSoundIntHandler. So, of course, it's not really junk. It's being used as an offset for a jump instruction:



```
 cycles  13
  beamx   419
  beamy   194
  frame  428
  flags  ..S..........Z..
--------------------------
     PC  40A4AB8C
     SR  2004
     SP  000FC6C0
    USP  70001107
    ISP  000FC6C0
    MSP  00000000
     D0  00000000
     D1  00005640
     D2  40A16500
     D3  FFFFFFF6
     D4  00800000
     D5  00000000
     D6  000FFFFC
     D7  00030011
     A0  000063B4
     A1  FFFF8FBA
     A2  000FC86C
     A3  40A16ABA
     A4  40A09AE6
     A5  000FC870
     A6  000FCAE0
     A7  000FC6C0
```

```
40A4AB78  movem.l  D0-D2/A0-A2/A6, -(A7)        48E7 E0E2
40A4AB7C  lea      $ffff7664.l, A1              43F9 FFFF 7664
40A4AB82  jsr      (-$8,PC,A1.l)                4EBB 98F8
40A4AB86  lea      $ffff8fba.l, A1              43F9 FFFF 8FBA
40A4AB8C  jsr      (-$8,PC,A1.l)                4EBB 98F8
40A4AB90  lea      $ffff92c0.l, A1              43F9 FFFF 92C0
40A4AB96  jsr      (-$8,PC,A1.l)                4EBB 98F8
40A4AB9A  lea      $ffff9436.l, A1              43F9 FFFF 9436
40A4ABA0  jsr      (-$8,PC,A1.l)                4EBB 98F8
40A4ABA4  movem.l  (A7)+, D0-D2/A0-A2/A6        4CDF 4707
40A4ABA8  jsr      ($1dbe,PC) ; ($40a4c968)     4EBA 1DBE
40A4ABAC  lea      $fffb5776.l, A0              41F9 FFFB 5776
```

```
MAME debugger version 0.272 (mame0272-427-g9b914fbfe62-dirty)
Currently targeting macclas2 (Macintosh Classic II)
>bpset 40a4ab86
Breakpoint 1 set
Stopped at breakpoint 1
```

IDA's disassembly is a little more readable. That value of 0xFFFF8FBA loaded into A1 represents how much you have to add to the program counter in order to reach *InstallSoundIntHandler* from where you currently are. Interpreting it as signed, it's a negative number because that function is further back in the ROM code.

```
ROM:40A4AB86 loc_40A4AB86:                          ; CODE XREF: SoundInitPatch+14↓p
ROM:40A4AB86                                        ; DATA XREF: SoundInitPatch:loc_40A4AB86↓o
ROM:40A4AB86                 lea     ((InstallSoundIntHandler-loc_40A4AB86)).l,a1
ROM:40A4AB8C                 jsr     loc_40A4AB86(pc,a1.l)
```

Overall, I felt that I totally understood what was happening. I'm probably repeating myself, but I just want it to sink in one more time: The problematic value in A1 gets loaded as part of a big relative jump to this section of ROM, and an out-of-bounds table access is jumping past code that is supposed to load A1 with an actual address of a peripheral to configure for sound interrupts. So A1 still contains that negative offset for the jump instead of a real address. Finally, it ends up being used as an address in a write operation, and boom, Sad Mac.

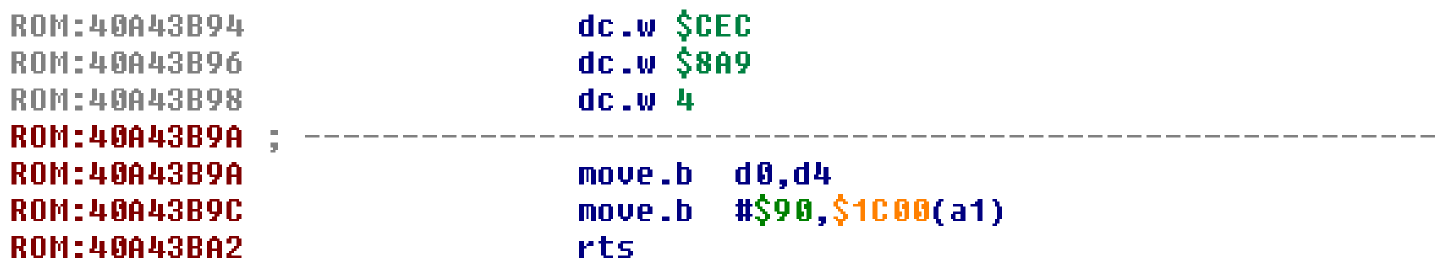
If you've followed along with me thus far, I'm sure there are some burning questions on your mind. This explains how MAME fails, but not why. Why was this happening? Also, why didn't this same failure occur on actual hardware? Obviously, the Classic II wasn't recalled because of an inability to use 32-bit addressing. There's no way that happened. It would have been all over tech news. Not to mention the fact that the people actually working on the ROM code would have quickly noticed it while they were testing. It's kind of a glaring issue.

So what gives? Was MAME doing something wrong here that didn't match hardware? This code couldn't have really been reached on hardware, right? I have the answer to these questions, but as a forewarning, the situation is way more complicated than I expected it to be.

I started out by trying to understand what the CAS instruction reached after the out-of-bounds jump was doing. Here are the bytes:

```
0C EC 08 A9 00 04
```

I quickly noticed that if I changed my disassembly in IDA so that it thought the code was supposed to start there, it refused to disassemble the instruction at all:



```
ROM:40A43B94                    dc.w $CEC
ROM:40A43B96                    dc.w $8A9
ROM:40A43B98                    dc.w 4
ROM:40A43B9A ; ---------------------------------------------------------------
ROM:40A43B9A                    move.b  d0,d4
ROM:40A43B9C                    move.b  #$90,$1C00(a1)
ROM:40A43BA2                    rts
```

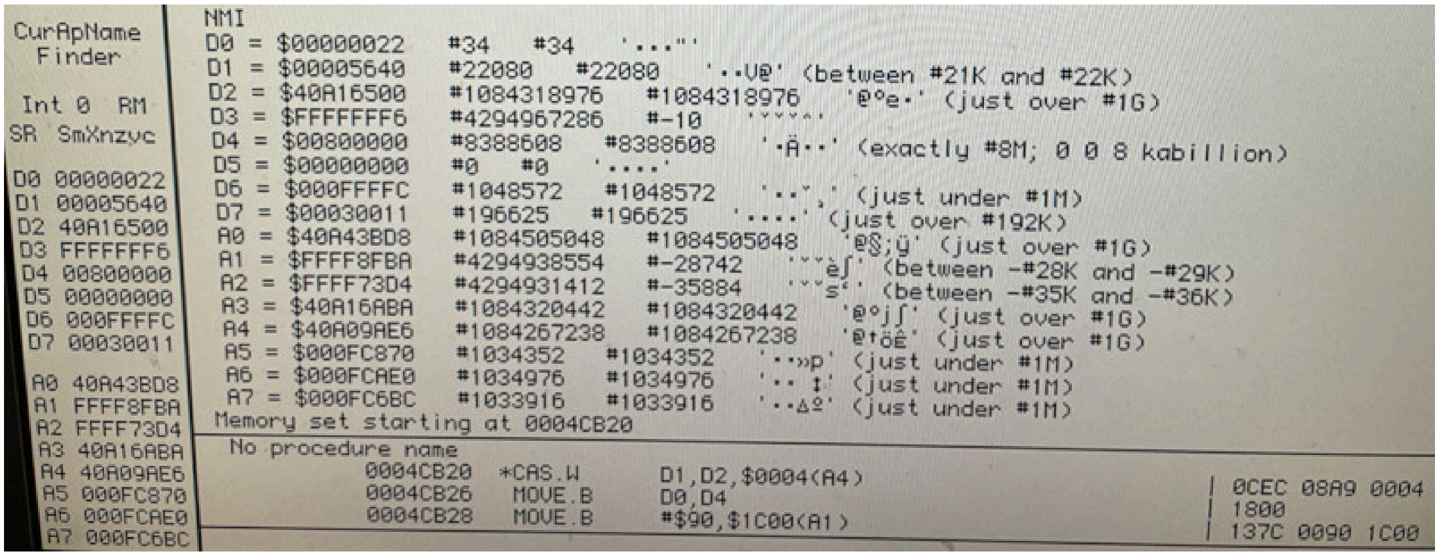
When I tried to convert it to code starting at 0x40A43B94, it said:

```
Command "MakeCode" failed
```

GNU objdump also failed to disassemble it, and then got right back in sync with the intended code:

```
40a43b94:       0cec            .short 0x0cec
40a43b96:       08a9 0004 1800  bclr #4,%a1@(6144)
40a43b9c:       137c 0090 1c00  moveb #-112,%a1@(7168)
40a43ba2:       4e75            rts
```

The fact that two well-known disassemblers balked at this instruction piqued my curiosity. I decided to use MacsBug on my Macintosh IIci, which also has a 68030 processor, to put all that code into RAM at a random location and see what MacsBug thought about it. Since I was going through all this effort, I also arranged all the other registers to be identical to what I was seeing on MAME. It wasn’t a perfect match with what I was seeing in MAME, though; I had to leave the program counter pointing into RAM instead of ROM.

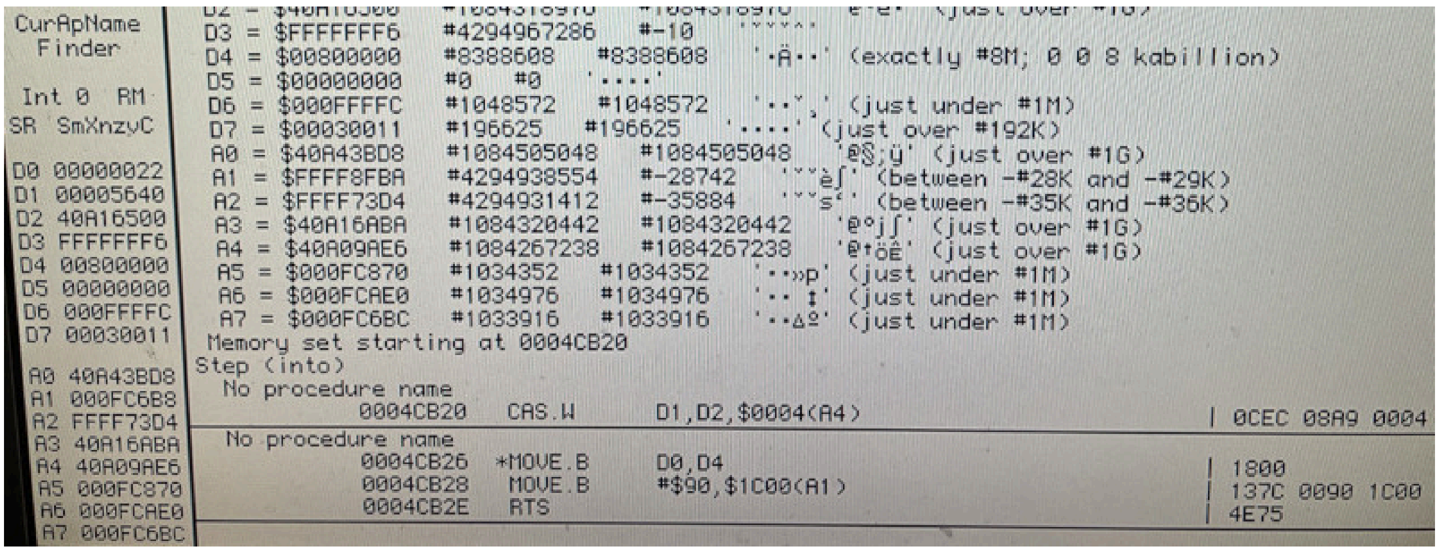


```
CurApName
 Finder

 Int 0  RM
SR  SmXnzvc

D0 00000022
D1 00005640
D2 40A16500
D3 FFFFFFF6
D4 00800000
D5 00000000
D6 000FFFFC
D7 00030011

A0 40A43BD8
A1 FFFF8FBA
A2 FFFF73D4
A3 40A16ABA
A4 40A09AE6
A5 000FC870
A6 000FCAE0
A7 000FC6BC
```

```
NMI
D0 = $00000022   #34   #34    '..."'
D1 = $00005640   #22080   #22080    '..V@' (between #21K and #22K)
D2 = $40A16500   #1084318976   #1084318976   '@°e.' (just over #1G)
D3 = $FFFFFFF6   #4294967286   #-10   'ˇˇˇˆ'
D4 = $00800000   #8388608   #8388608   '.Ä..' (exactly #8M; 0 0 8 kabillion)
D5 = $00000000   #0   #0   '....'
D6 = $000FFFFC   #1048572   #1048572   '..˘,' (just under #1M)
D7 = $00030011   #196625   #196625   '....' (just over #192K)
A0 = $40A43BD8   #1084505048   #1084505048   '@§;ÿ' (just over #1G)
A1 = $FFFF8FBA   #4294938554   #-28742   'ˇˇèʃ' (between -#28K and -#29K)
A2 = $FFFF73D4   #4294931412   #-35884   'ˇˇs“' (between -#35K and -#36K)
A3 = $40A16ABA   #1084320442   #1084320442   '@°jʃ' (just over #1G)
A4 = $40A09AE6   #1084267238   #1084267238   '@†öÊ' (just over #1G)
A5 = $000FC870   #1034352   #1034352   '..»p' (just under #1M)
A6 = $000FCAE0   #1034976   #1034976   '.. ‡' (just under #1M)
A7 = $000FC6BC   #1033916   #1033916   '..∆º' (just under #1M)
Memory set starting at 0004CB20
 No procedure name
            0004CB20  *CAS.W     D1,D2,$0004(A4)              | 0CEC 08A9 0004
            0004CB26   MOVE.B    D0,D4                        | 1800
            0004CB28   MOVE.B    #$90,$1C00(A1)               | 137C 0090 1C00
```

Interesting — MacsBug also said it was a CAS.W instruction, but it interpreted it slightly differently. It said it was CAS.W D1,D2,$0004(A4).

Of course, I couldn't resist stepping through the code in MacsBug to see what it would do on a real 68030 processor:



```
CurApName
 Finder

 Int 0  RM
SR  SmXnzvC

D0 00000022
D1 00005640
D2 40A16500
D3 FFFFFFF6
D4 00800000
D5 00000000
D6 000FFFFC
D7 00030011

A0 40A43BD8
A1 000FC6B8
A2 FFFF73D4
A3 40A16ABA
A4 40A09AE6
A5 000FC870
A6 000FCAE0
A7 000FC6BC
```

```
D3 = $FFFFFFF6   #4294967286   #-10   'ˇˇˇˆ'
D4 = $00800000   #8388608   #8388608   '.Ä..' (exactly #8M; 0 0 8 kabillion)
D5 = $00000000   #0   #0   '....'
D6 = $000FFFFC   #1048572   #1048572   '..˘,' (just under #1M)
D7 = $00030011   #196625   #196625   '....' (just over #192K)
A0 = $40A43BD8   #1084505048   #1084505048   '@§;ÿ' (just over #1G)
A1 = $FFFF8FBA   #4294938554   #-28742   'ˇˇèʃ' (between -#28K and -#29K)
A2 = $FFFF73D4   #4294931412   #-35884   'ˇˇs“' (between -#35K and -#36K)
A3 = $40A16ABA   #1084320442   #1084320442   '@°jʃ' (just over #1G)
A4 = $40A09AE6   #1084267238   #1084267238   '@†öÊ' (just over #1G)
A5 = $000FC870   #1034352   #1034352   '..»p' (just under #1M)
A6 = $000FCAE0   #1034976   #1034976   '.. ‡' (just under #1M)
A7 = $000FC6BC   #1033916   #1033916   '..∆º' (just under #1M)
Memory set starting at 0004CB20
Step (into)
 No procedure name
            0004CB20   CAS.W     D1,D2,$0004(A4)              | 0CEC 08A9 0004
 No procedure name
            0004CB26  *MOVE.B    D0,D4                        | 1800
            0004CB28   MOVE.B    #$90,$1C00(A1)               | 137C 0090 1C00
            0004CB2E   RTS                                    | 4E75
```

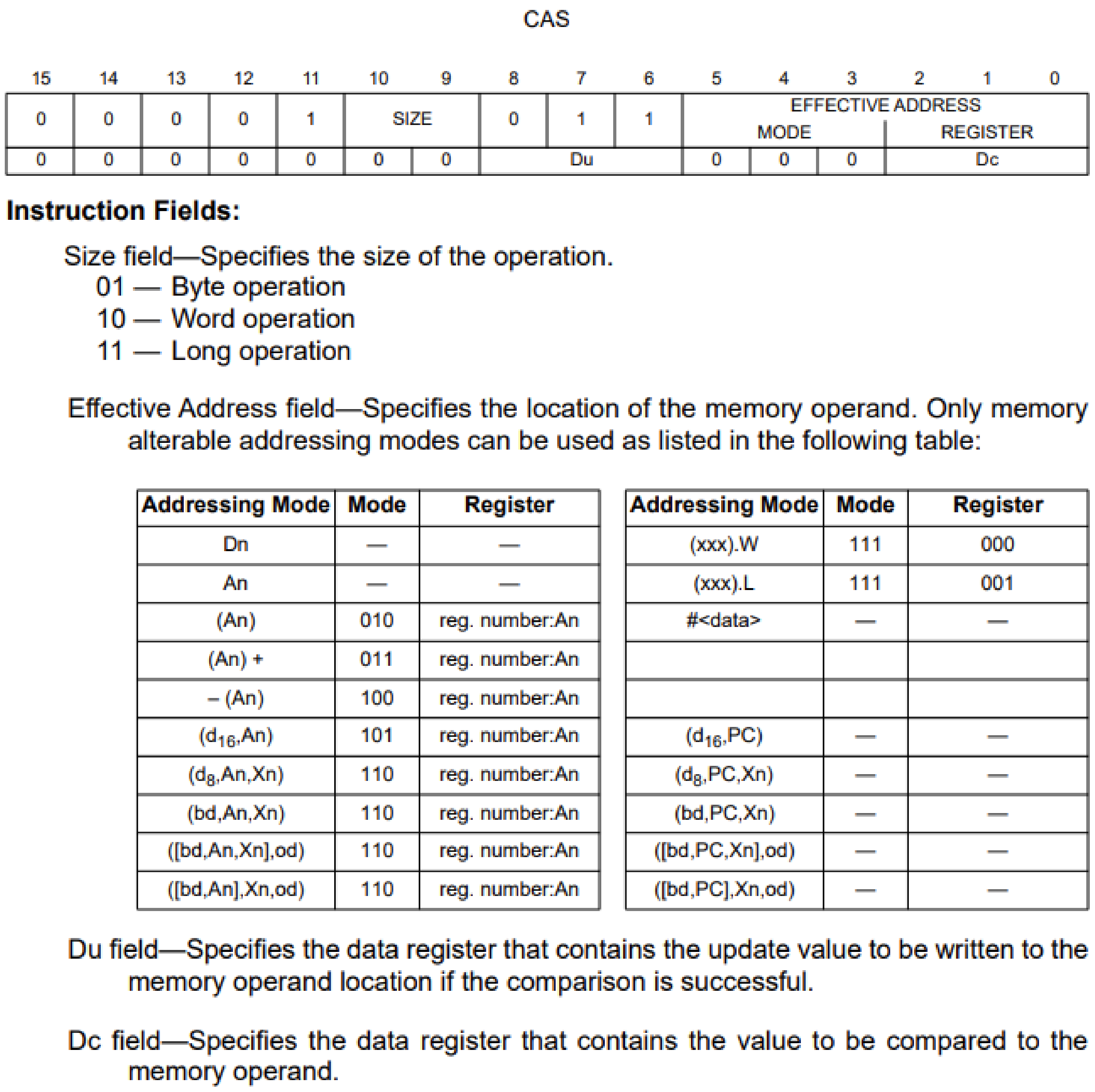
Wait…what? If you compare the register display on the left side of the screen in the first picture with the same display in the second picture, something incredibly strange has happened. Even though MacsBug and MAME both don't mention A1 in their interpretations of this CAS instruction at all, the value of A1 has changed! It started out as 0xFFFF8FBA, and ended up as 0xFC6B8. It seems to have turned into a value similar to what's in A5 through A7 — a valid RAM address.

Further tinkering with MacsBug and different register values revealed that the new value of A1 depended on the original value of A1, A7, and the program counter. I couldn't figure out exactly what it was doing, but it was definitely majorly changing A1's value.

At this point, I felt like I was onto something. The MAME-emulated Classic II was crashing because A1 didn't change, so it still contained an invalid address. On hardware, this weird instruction, which several disassemblers refused to touch, and wasn't even intended to be jumped to because it starts in the middle of an actual valid instruction, was changing A1 to a new value that was a good address. Was this crazy instruction accidentally fixing A1 and thus hiding a bug from Apple's ROM developers in the early 1990s?

This was about the time that Arbee suggested I start sharing my research on the 68kmla forums and the bannister.org forums to see if some of the incredible folks who know way more than me about the 68k instruction set might be able to chime in. I also asked around on IRC in #mac68k on Libera.

The consensus was that this is not a valid CAS instruction, and that MacsBug's interpretation of the registers being D1 and D2 is correct. Let's look at what the Motorola M68000 Family Programmer's Reference Manual says about the encoding of the CAS instruction:



CAS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	SIZE		0	1	1	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	0	0	0	0	0	0	Du			0	0	0	Dc		

Instruction Fields:

Size field—Specifies the size of the operation.
01 — Byte operation
10 — Word operation
11 — Long operation

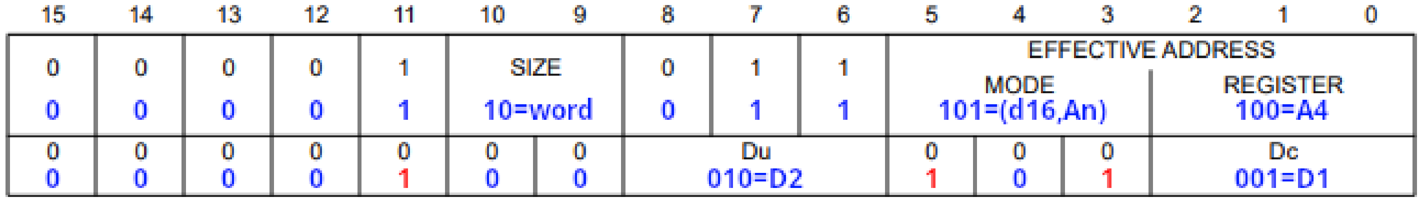
Effective Address field—Specifies the location of the memory operand. Only memory alterable addressing modes can be used as listed in the following table:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	—	—
(An) +	011	reg. number:An			
– (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	—	—
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	—	—
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	—	—

Du field—Specifies the data register that contains the update value to be written to the memory operand location if the comparison is successful.

Dc field—Specifies the data register that contains the value to be compared to the memory operand.

Comparing this with the 3 words of the instruction (0x0CEC 0x08A9 0x0004) and filling in the fields, we can see the following:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 0	0 0	0 0	0 0	1 1	SIZE 10=word		0 0	1 1	1 1	EFFECTIVE ADDRESS MODE 101=(d16,An)			EFFECTIVE ADDRESS REGISTER 100=A4		
0 0	0 0	0 0	0 0	0 1	0 0	0 0	Du 010=D2			0 1	0 0	0 1	Dc 001=D1		

The first word appears to be a valid CAS instruction. The second word, though, has a few bits that are 1 even though the instruction format specifically says they are supposed to be 0. I've marked them in red. Also, the Du and Dc fields match what MacsBug says, as opposed to how MAME interpreted it.

The third word, 0x0004, is the d16 value mentioned in the MODE field. It's the $0004 offset from A4. So according to Motorola's reference manual, this instruction is:

```
CAS D1,D2,$0004(A4)
```

…except it has three bits that are 1 in places where they are supposed to be 0. So it's not a valid instruction at all. At least, it's not documented.

Side note: I think MAME's debugger is also decoding normal CAS instructions incorrectly; if I change it to 0x0CEC 0x0081 0x0004 instead, which is the correct way to write this instruction without the three bad "1" bits, it still thinks Du is D0 instead of D2. But that's beside the point — the instruction we're dealing with in this story is completely messed up either way.

The CAS (compare-and-swap) instruction is an interesting one. It's used for accomplishing various atomic operations without requiring a lock. It is one of the few instructions in the 68000 family CPUs that perform a read-modify-write bus cycle. What this particular instruction is supposed to do is compare the word value in memory at A4 + 4 to the value of D1. If they are the same, then the value in D2 is written to memory at A4 + 4. Otherwise, the value in memory at A4 + 4 is loaded into D1.

It clearly still does some of this stuff, like the read-modify-write cycle involving A4 + 4. If I change A4 to point to an invalid address, MacsBug complains to me. For example, on my Mac IIci with the same test setup I showed earlier, if I set A4 to 0xFFFF0000 and rerun the bad instruction, MacsBug tells me this:

```
Bus Error at 0004CB20
while reading word (read-modify-write) from FFFF0004 in
Supervisor data space
```

This definitely means that this instruction still performs the RMW cycle at A4 + 4. It doesn't seem to do exactly what the CAS instruction is supposed to do though. Obviously, the normal CAS instruction wouldn't mess with the value of A1. I ran more tests after changing A4 to point to RAM. If I store the value 0xFFFF at A4 + 4, and D1 is set to 0x1111 and D2 is set to 0x2222, then after executing the instruction, memory at A4 + 4 changes to 0x2222. But that doesn't really make any sense, because it only should have written 0x2222 to memory if D1 was equal to 0xFFFF.

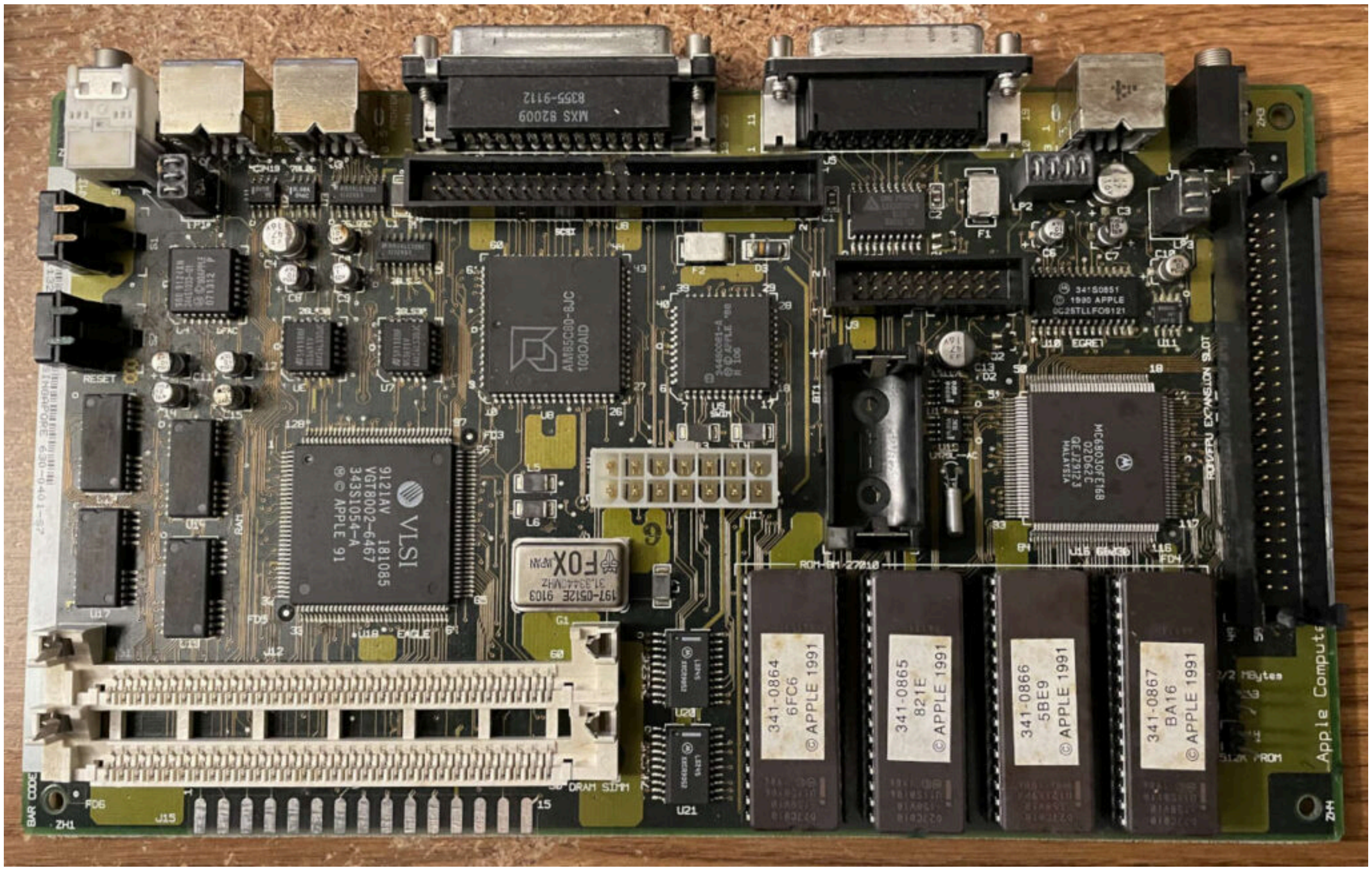
Let's summarize what we've learned so far.

- The invalid code that the ROM accidentally jumps to (0x0CEC 0x08A9 0x0004) is sort of like "CAS D1,D2,$0004(A4)", but not really, because some of the bits that are supposed to be 0 are actually set to 1.
- On another 68030-based Mac, I've observed that this instruction ends up modifying the value stored in register A1.
- MAME's 68030 CPU emulator does not change A1 like this, because the instruction is undocumented and normal code would never use it.
- The Sad Mac in MAME occurs a couple of instructions later because A1 is set to an invalid address, and code in ROM tries to write a byte to A1 + 0x1C00.

I was starting to believe something that sounded almost too crazy to be true: Apple had an out-of-bounds jump bug in the Classic II's ROM that should have caused a Sad Mac during boot, but they had no idea the bug was there because the 68030 was accidentally fixing the value of A1 by executing an undocumented instruction. How could I prove that my theory was correct?

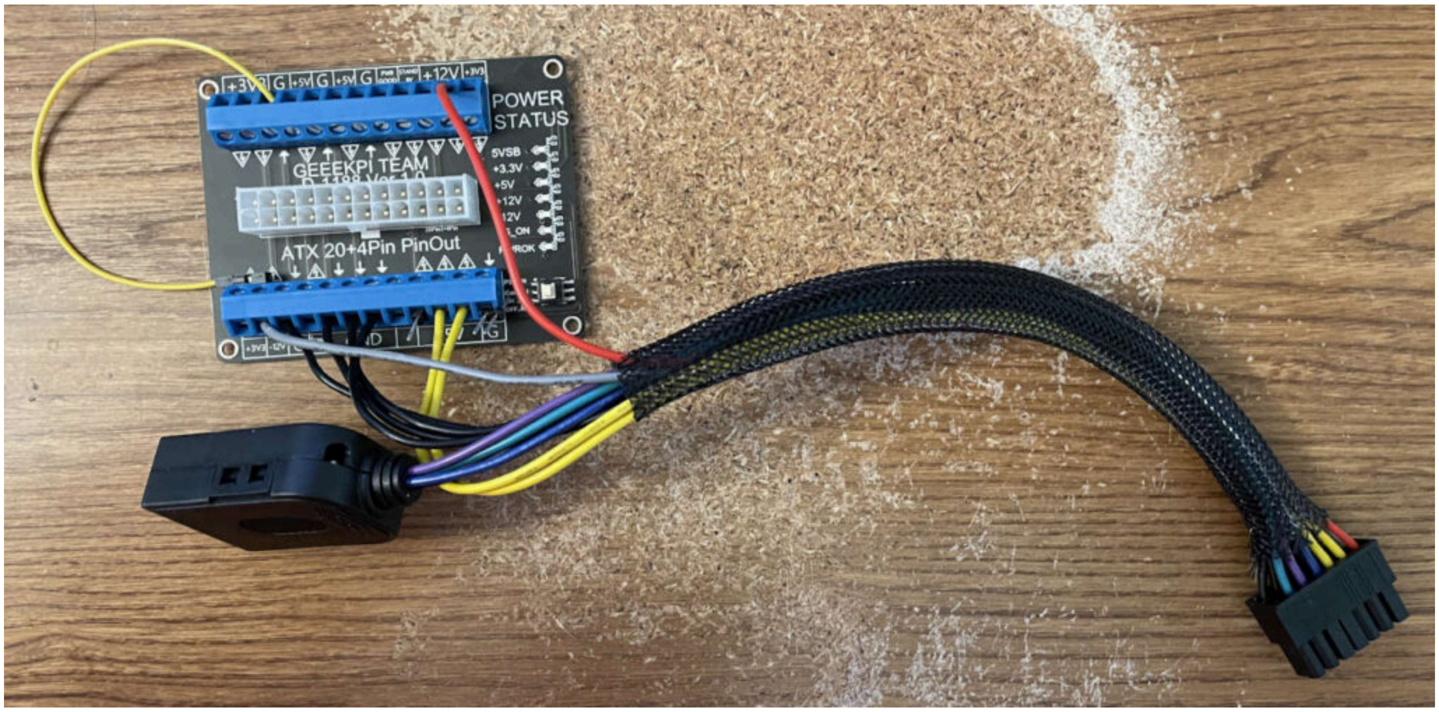
By buying a Classic II and hacking the ROM in order to see exactly what is happening on hardware, of course!

This Classic II was manufactured in 1991, so it’s about 34 years old at this point. Computers this old usually need to be repaired if nobody has already fixed them. As I mentioned in a previous post about the LC III capacitor polarization mistake, the surface mount capacitors in old Macs leak out corrosive goo over time. This machine was no exception. I didn’t even try powering it up. I immediately opened it up and removed what I believe was the original Sonnenschein 1/2 AA lithium battery with a March 1991 date code. The battery had not leaked at all, luckily! That's another unfortunate thing that happens in a lot of old Macs that have been sitting around for decades — severe battery damage.

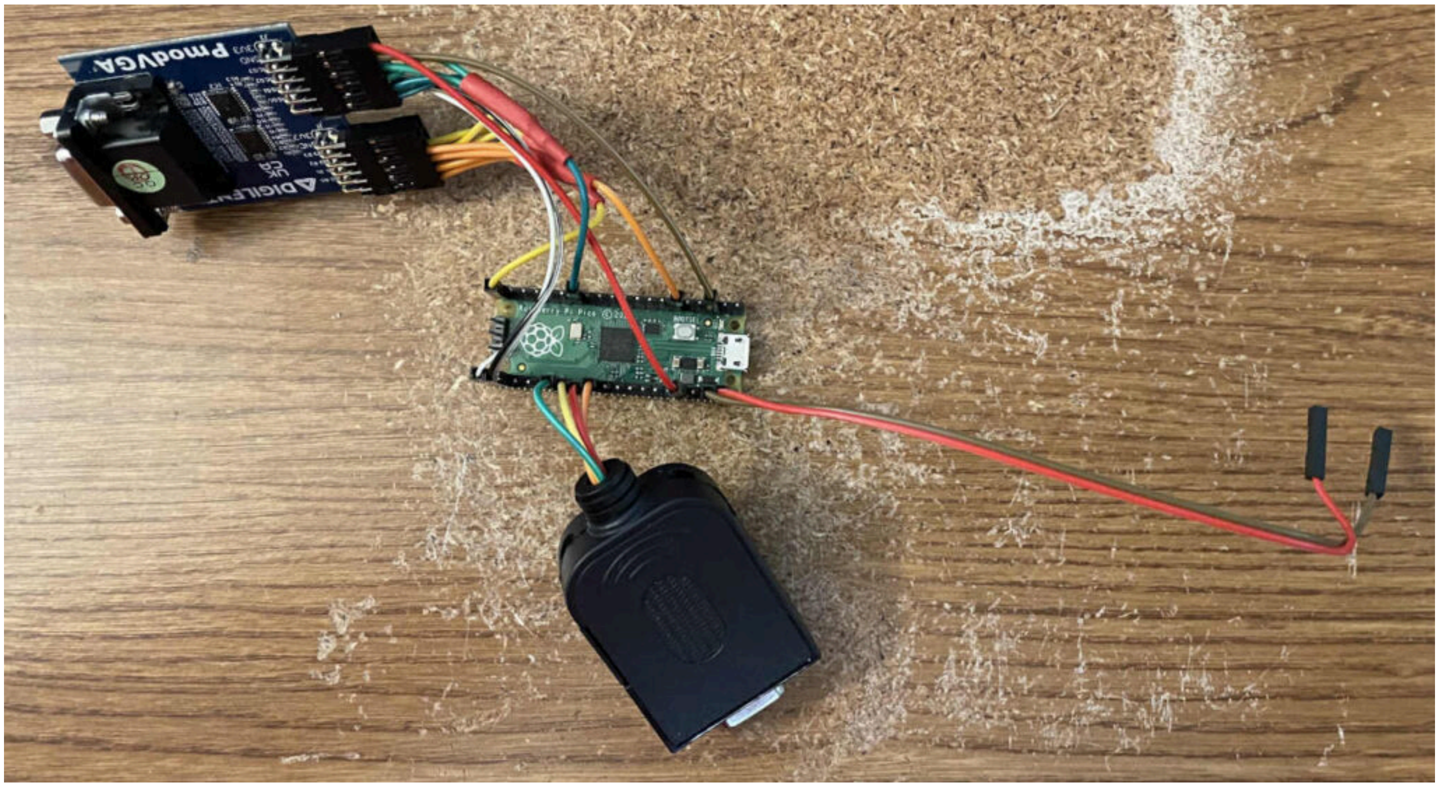


I don't want to go into excruciating detail about the repair and make this post even longer to read, but if you're interested in hearing more about that process, I posted a few updates about the repair in my 68kmla forum thread. I did accidentally short /RESET to ground while soldering in new capacitors, but I eventually got the logic board working. This was very similar to what happened to Adrian's Digital Basement on his SE/30 board a few months ago, but lucky for me, my accidental short didn't involve 12 volts and fry a bunch of chips! Adrian's channel is an excellent resource for vintage computer enthusiasts out there, if you're not already subscribed.

The Classic II has a cathode-ray tube for its screen, which scares the crap out of me, so I opted to use a different solution in order to run the logic board by itself without any dangerous voltages. Plus, the analog boards in these computers are known to be a huge pain to repair. This one had really bad capacitor leakage, so that'll probably be a future repair project. The bottom line is I knew I'd be installing and removing ROM chips a bunch, so I wanted to run the logic board out in the open, for reasons of both safety and convenience. Special thanks to 68kmla forum member davewongillies for posting his similar setup with a Classic II logic board, complete with Amazon links for a bunch of the parts. That Mini-Fit Jr. pin extractor tool on Amazon is absolute garbage though — I could never get it to work. I ended up using staples instead.

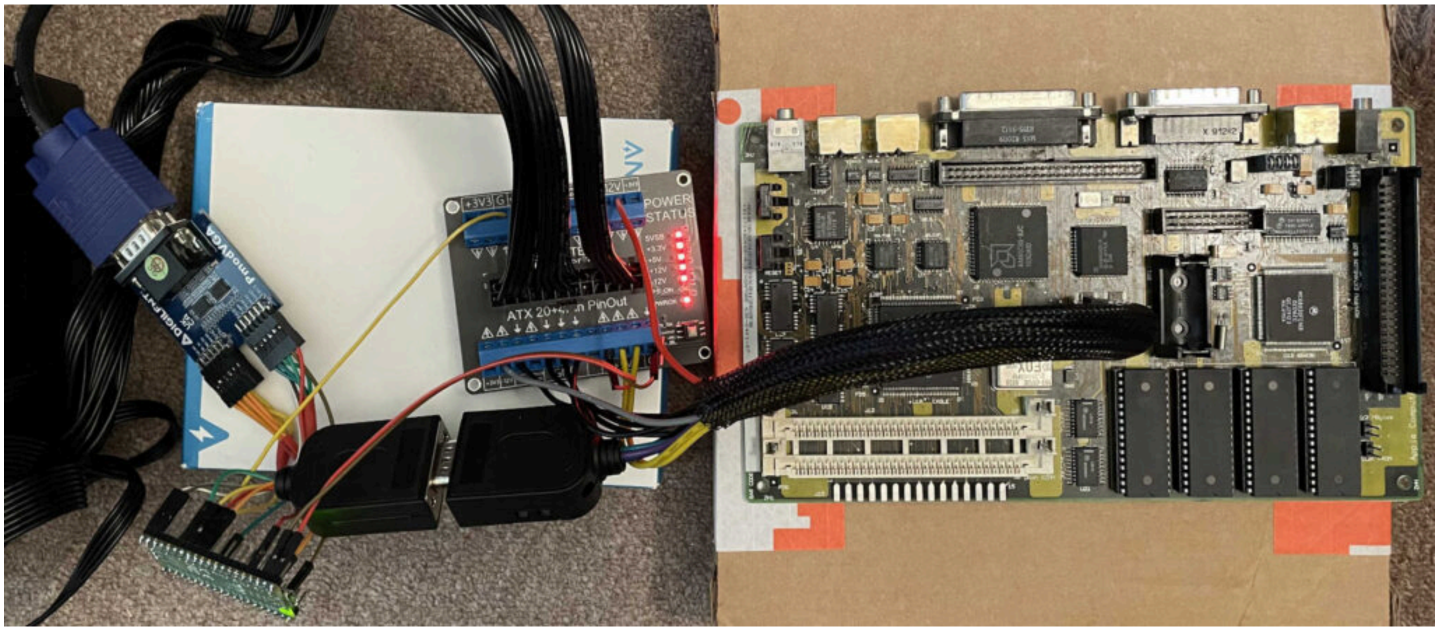


I got impatient while waiting for my RGBtoHDMI to arrive for converting the Classic II's video signal into HDMI, and eventually discovered another solution created by GitHub user guruthree involving a Raspberry Pi Pico that converts the signal to VGA. I made a few cables, tweaked the code until the timings and colors were correct, and ended up with this concoction to adapt between a CGA DE-9 video connector and VGA:

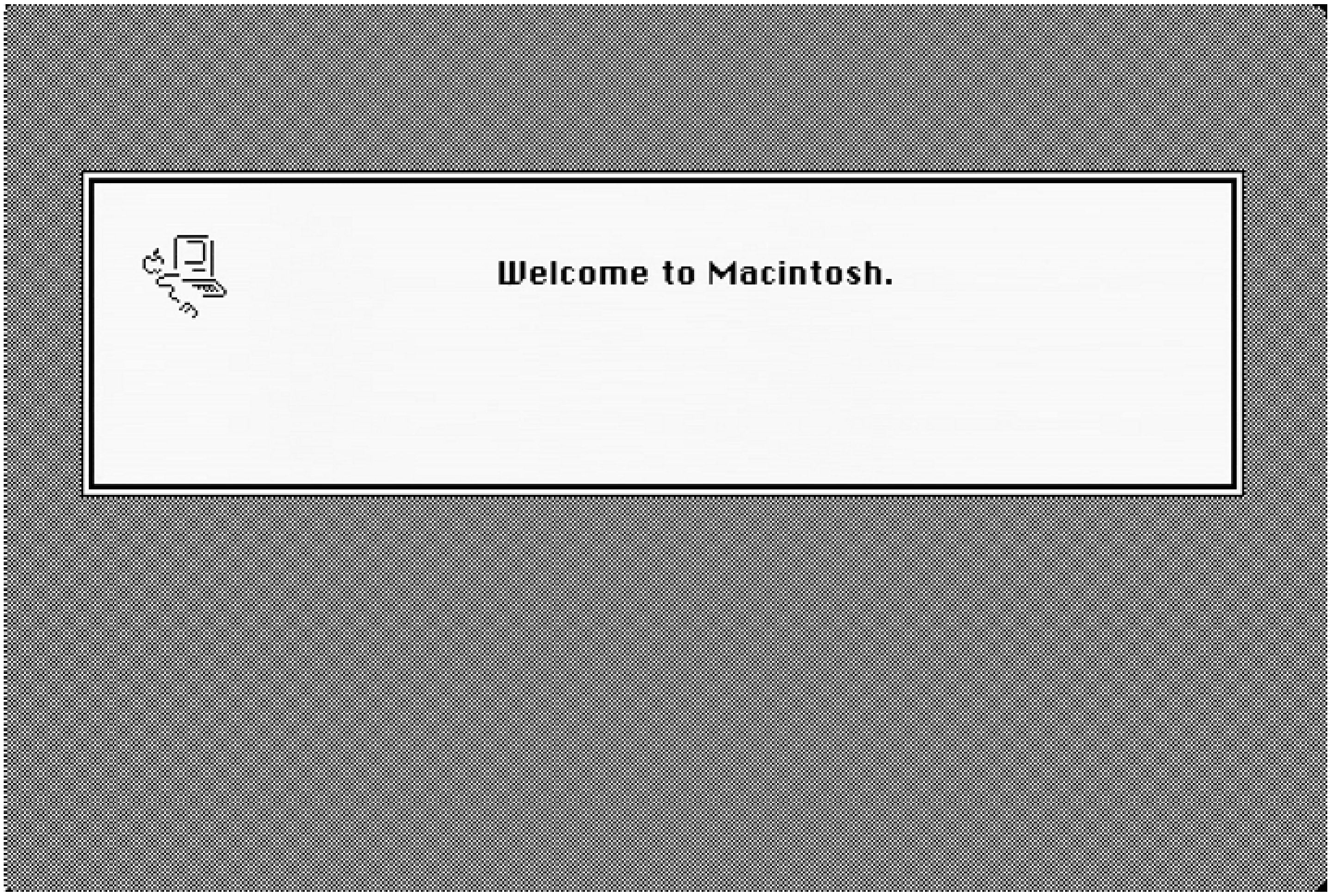


Although I would never do anything like this in a real production environment where reliability matters, I didn't even use level shifters for the 5V signals coming from the Mac. Technically, [the RP2040 is "sort of" 5V tolerant], even though it's not documented in the datasheet. That, combined with the fact that I had a few Picos laying around that I didn't care about destroying, gave me enough confidence to try it out.

Here's the whole mess, all wired up together, along with an ATX power supply. I also swapped the original ROM chips out with some programmable SST29EE010 EEPROMs I had on hand. Interestingly, the factory chips from Apple were actually UV-erasable EPROMs, so I could have even used the original chips if I had a UV chip eraser.



I successfully booted from a SCSI drive using this setup, and could capture the video signal with one of my many video capture devices:



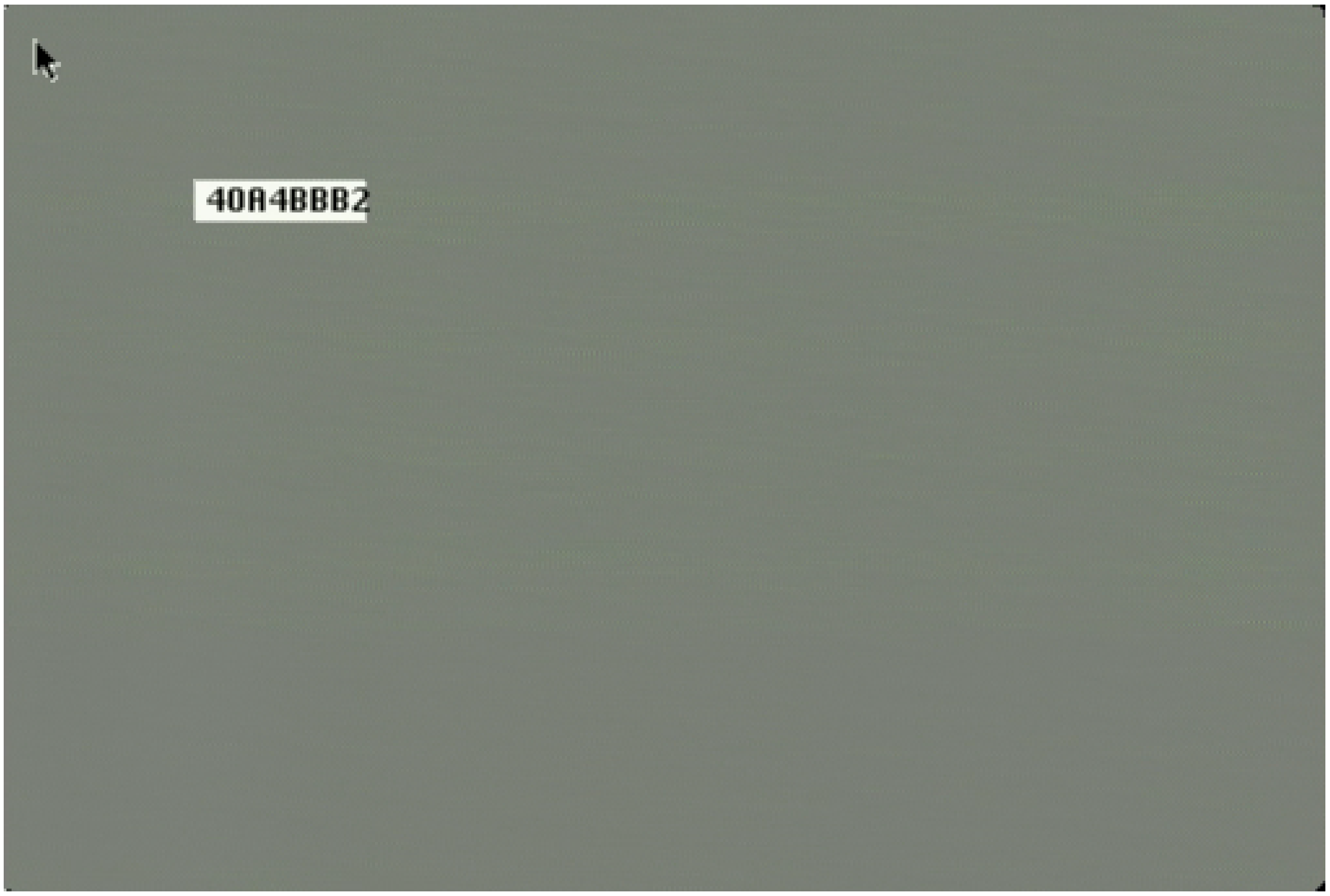
Welcome to Macintosh.

By the way, I think there's something wrong with at least one column of pixels on the left side. I think it has something to do with the tweak I made to the code running on the RP2040 in order to make it sync up correctly with the Classic II's video signal. I'm sure I could fix it if I played around more. The RGBtoHDMI, which arrived a few days later, does not have this problem.

Anyway, I knew I was in business. I threw together some 68030 assembly code to display the value of A1 on the screen, found an unused place in the ROM to put it, and then came up with three custom ROMs to try on the Classic II:

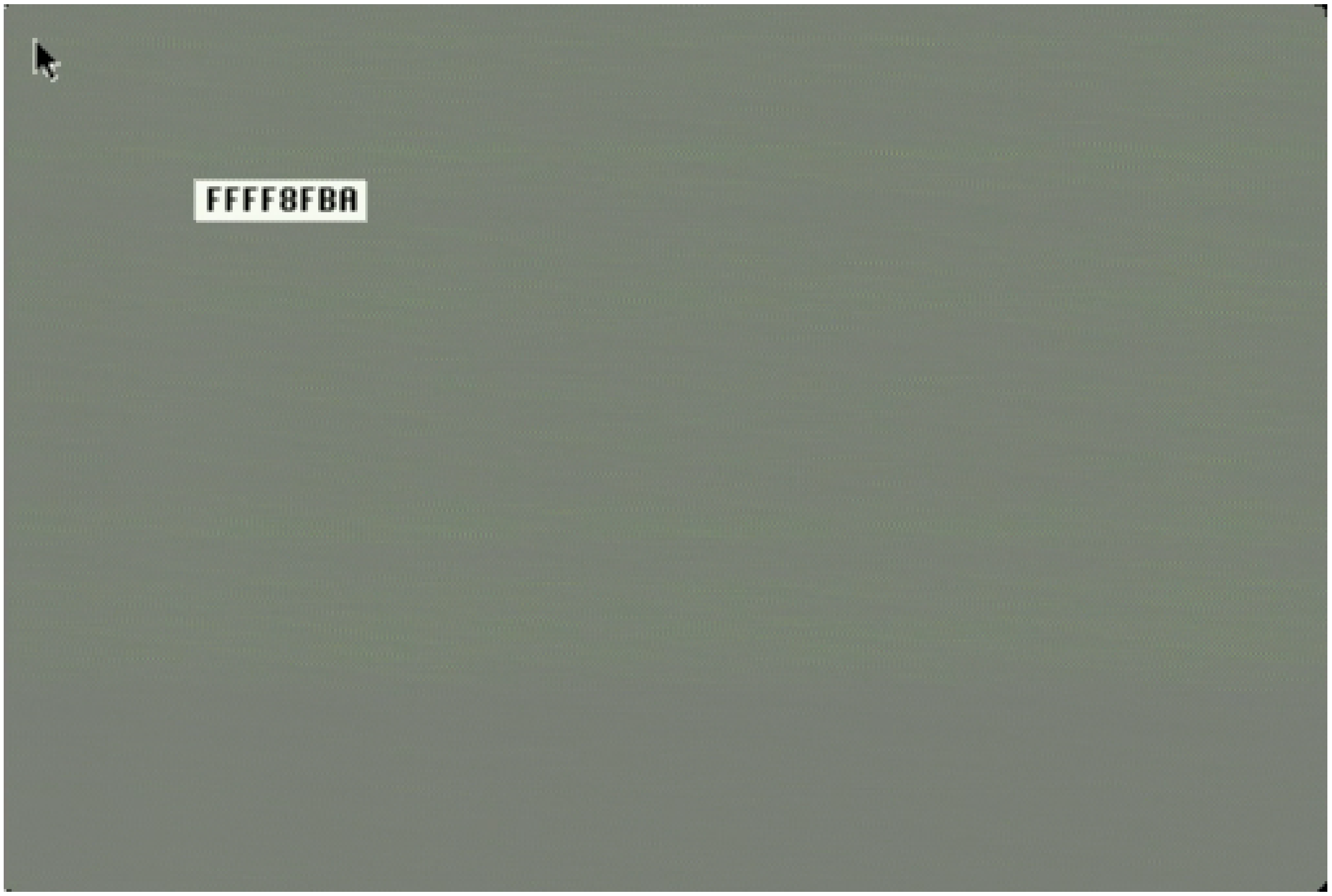
- **Custom ROM 1:** Replace the instruction at 0x40A43B9C that results in a Sad Mac (the MOVE.B instruction) with a jump to my special code that draws A1 to the screen, so we can see what A1 is on hardware at that point. Also, this would verify whether this code even runs at all on hardware.
- **Custom ROM 2:** Replace the instruction at 0x40A43B94 (the CAS instruction) with the same jump to my special code. This would verify whether the out-of-bounds jump was really happening, and what the value of A1 was leading up to it.
- **Custom ROM 3:** Replace the instruction at 0x40A43B94 (the CAS instruction) with NOPs. This would ideally replicate exactly what I was seeing in MAME, proving that the bad CAS instruction was vital to the Classic II's ability to boot.

Let's look at the results one by one. Here's the result that was displayed when I ran Custom ROM 1:


```
40A4BBB2
```

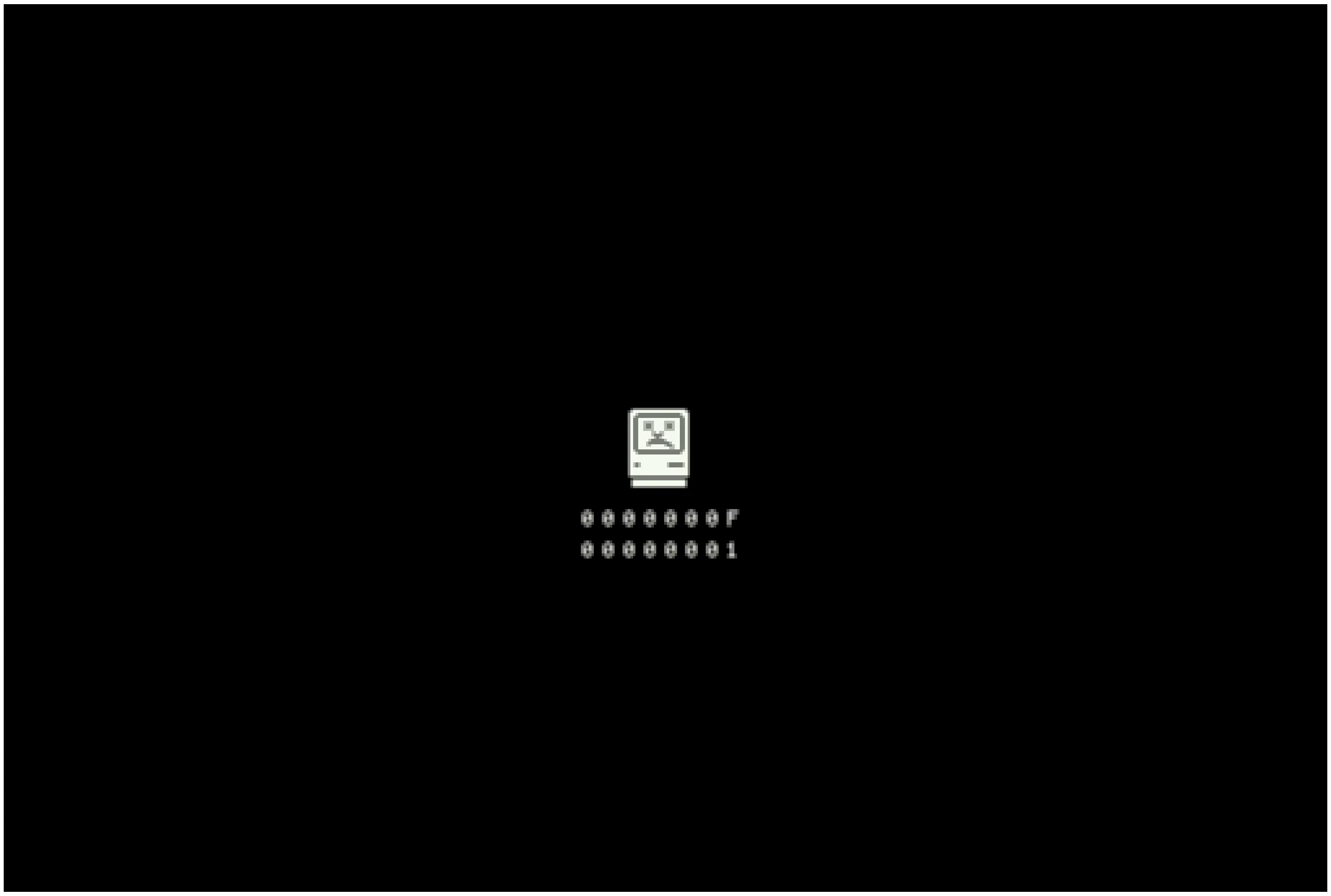
This verified that the section of code I was looking at definitely ran on hardware. It also showed that A1 was set to a very interesting value when it reached the instruction that crashed on MAME. 0x40A4BBB2 is not really an address you would write to because it’s a ROM address, but it doesn’t cause a bus error if you attempt it.

Here is what I got when I ran Custom ROM 2:


FFFF8FBA

The same A1 value that we saw in MAME! This proved two things. One, the out-of-bounds table jump was definitely happening — if it wasn't, my custom A1 drawing code wouldn't have ran at all, since the JMP to it was stored out of sync of the normal intended code flow just like the accidental CAS instruction. Second, it also proved, along with the first test, that the CAS instruction was indeed fixing A1 on hardware, just like I theorized.

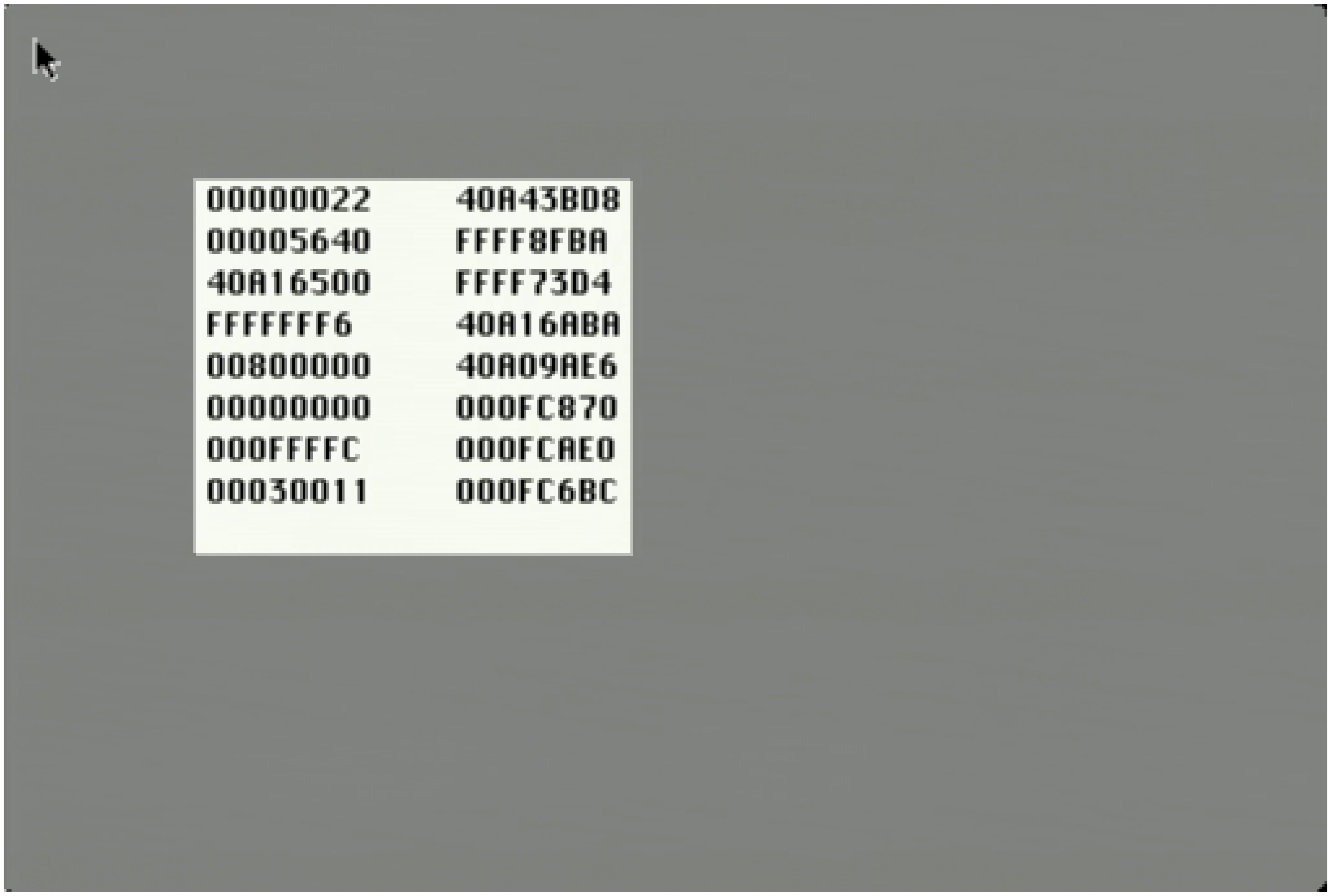
Lastly, my test run of Custom ROM 3, which eliminated the CAS instruction from the situation altogether, gave me the final proof I needed:



```
0000000F
00000001
```

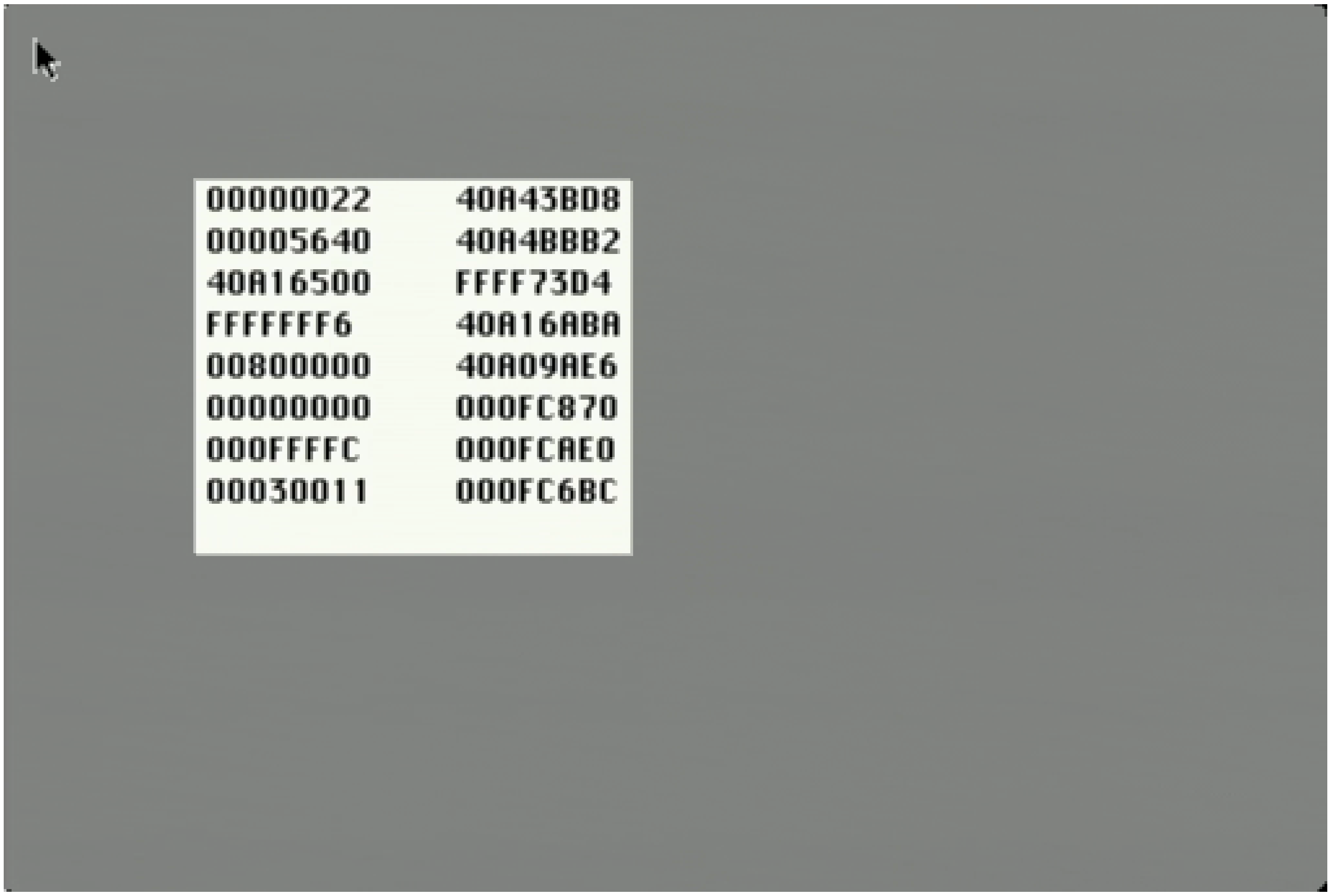
A Sad Mac, just like I saw with MAME in 32-bit mode. I also discovered during this test that on hardware, the same Sad Mac happens in 24-bit mode too. So MAME is actually more tolerant than hardware of that invalid write in 24-bit mode.

These results motivated me to make a couple more hacked ROM images to run on hardware in order to glean the values of all of the CPU’s data and address registers immediately before and after the CAS instruction. The data register values are shown in the left column, address registers in the right column. Before:



```
00000022     40A43BD8
00005640     FFFF8FBA
40A16500     FFFF73D4
FFFFFFF6     40A16ABA
00800000     40A09AE6
00000000     000FC870
000FFFFC     000FCAE0
00030011     000FC6BC
```

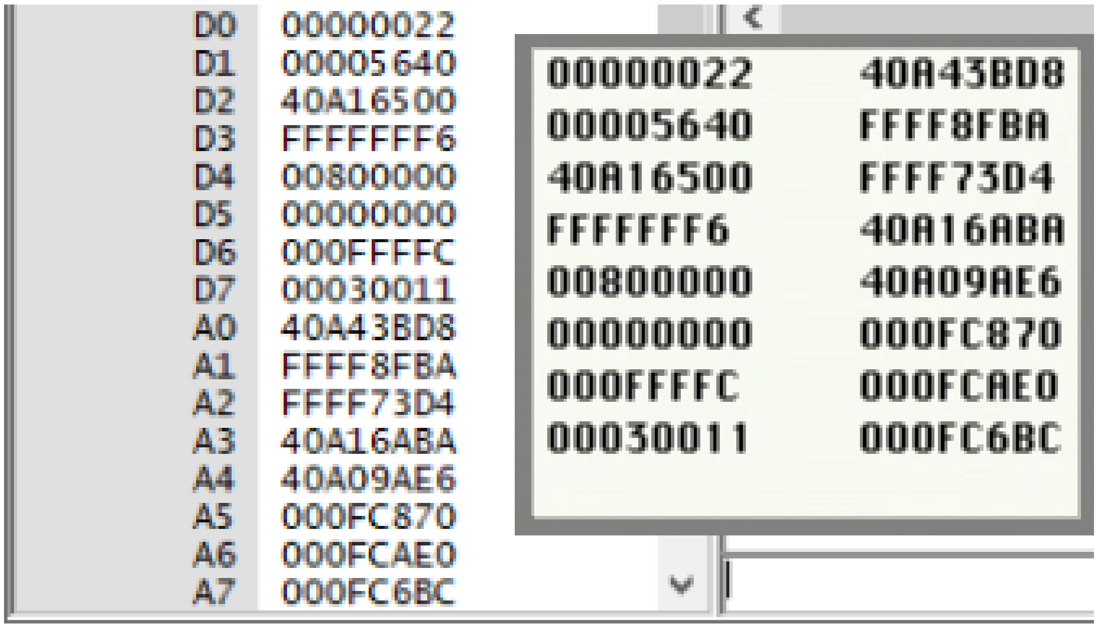
And after:



```
00000022    40A43BD8
00005640    40A4BBB2
40A16500    FFFF73D4
FFFFFFF6    40A16ABA
00800000    40A09AE6
00000000    000FC870
000FFFFC    000FCAE0
00030011    000FC6BC
```

Yep! Everything is the same except for A1, which has magically been transformed from FFFF8FBA to 40A4BBB2. The mystery instruction is definitely what was responsible for that.

One fun part about this test was being able to successfully verify that everything on hardware was exactly identical to what MAME did up to the bad instruction. The entire register state shown in the “before” picture is a perfect match to what MAME shows when booting in 32-bit mode prior to the bad CAS instruction. See for yourself:



```
D0  00000022
D1  00005640
D2  40A16500
D3  FFFFFFF6
D4  00800000
D5  00000000
D6  000FFFFC
D7  00030011
A0  40A43BD8
A1  FFFF8FBA
A2  FFFF73D4
A3  40A16ABA
A4  40A09AE6
A5  000FC870
A6  000FCAE0
A7  000FC6BC
```

```
00000022    40A43BD8
00005640    FFFF8FBA
40A16500    FFFF73D4
FFFFFFF6    40A16ABA
00800000    40A09AE6
00000000    000FC870
000FFFFC    000FCAE0
00030011    000FC6BC
```

If your brain is fried after reading all this, first of all I don't blame you at all, and second, let me bring everything together to explain what this all means:

I've discovered an undocumented MC68030 instruction that performs a read-modify-write bus cycle and also changes the value of the A1 register.

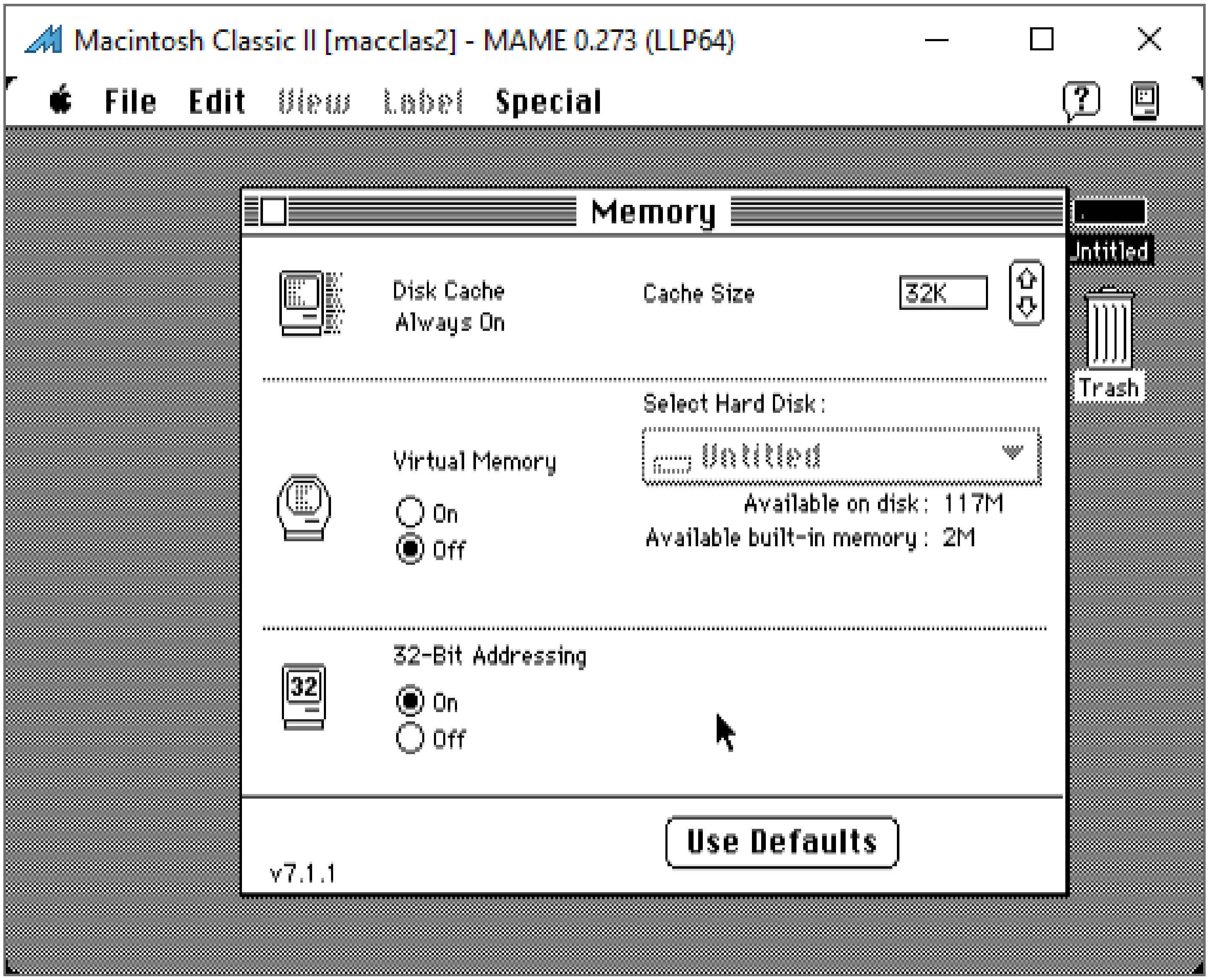
This newly-discovered instruction turns out to be the glue that's accidentally holding the Classic II together. Without this instruction modifying A1, the Classic II can't boot. I'm confident that it was a mistake and not something intentional. A totally understandable mistake, at that. If the pesky 68030 hadn't been hiding the bug from Apple's ROM developers, there is no doubt they would have caught it before the Classic II shipped.

I searched deeper and found the same chunk of code in the newer Macintosh IIvx ROM, and in that ROM they finally increased the size of the jump table. I confirmed that the case for the Classic II in that code does nothing at all. It just jumps directly to an RTS instruction. I wonder if the Apple ROM developer working on that chunk of code in the IIvx ROM scratched their head in confusion when they added new entries for a bunch of new models, including the Classic II, after the Classic II ROM had already been finalized and shipped. Who knows? I'm not sure how Apple handled all the different ROM variants back then.

Because of this new discovery, I think it's very likely that there is not a 100% perfect Motorola MC68030 emulator or replica in existence right now. This might be the only case in existence where it matters though. What this means is I could write a small chunk of code that determines whether it's running on a physical 68030 or an emulator, by simply using that instruction and looking at the resulting value of A1.

What can MAME do in order to work around this problem and allow the Classic II to boot? We don't really know the exact details of what this instruction does. With some limited testing, I believe I've observed that the resulting value of A1 depends on the original A1 value, the value of A7, and the program counter. But I'm not sure. Maybe someone can make a program that tries out a bunch of different register values and memory contents, and attempt to deduce what exactly the instruction does so that it can be emulated accurately. Until someone decides that it's worth trying to figure out, MAME is patching this bug out of the ROM in order to allow the Classic II to boot. As Arbee pointed out, we're a little late to get Motorola/Freescale/NXP to issue an errata. Unless someone who worked on the 68030 happens to see this post and might have a clue about what's going on here…

Here's a screenshot of MAME with Arbee's patch applied, now able to successfully emulate a Classic II with 32-bit addressing enabled. Yay!



Memory

Disk Cache Always On — Cache Size: 32K

Virtual Memory: On [✓ Off]

Select Hard Disk: Untitled

Available on disk: 117M

Available built-in memory: 2M

32-Bit Addressing: [✓ On] Off

v7.1.1

After all that, what's the lesson we can learn from this story? I guess it's that emulators can teach us new things about hardware that we never would have thought to look into! I bet this bug in the ROM would have gone undiscovered for all eternity if not for MAME providing emulation of the Classic II, which isn't a particularly notable machine compared to more popular compact Macs like the SE/30 and Color Classic.

It also goes to show you how bugs can be lurking in the background in places where you might think everything is totally polished. I think it's also a good example of how some bugs just aren't that big of a deal. This bug fits that category pretty well. The machine worked fine and nobody noticed.

Oh, and as for the original reason I somehow managed to pull myself into this investigation in the first place: the command+power key combination does not work in MAME. Now that I have a real Classic II, I have been able to confirm that the keystroke does indeed work on hardware. It only works with MacsBug installed, which is likely due to what I said earlier about the Egret disabling it by default. Either way, it really should work in MAME when MacsBug is installed. I suppose that's another MAME fix for me to work on!
Address: https://www.downtowndougbrown.com/2025/01/the-invalid-68030-instruction-that-accidentally-allowed-the-mac-classic-ii-to-successfully-boot-up/
« Easy repair of a defective NZXT Signal 4K30 capture card
The gooey rubber that's slowly ruining old hard drives »
Trackback

29 comments

1. Dan Allen @ 2025-01-25 13:58

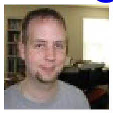
   Thanks for an interesting writeup. I followed it all with interest as I was in the developer of MacsBug at Apple from 1985 until 1988.

   I mainly worked on the first 3 generations of Macintosh, like the Mac II, Mac SE, Mac Plus, etc.

   I miss using MacsBug!

   Dan Allen
   ex Apple (1985-1994)

2. Doug Brown @ 2025-01-25 14:18

   Wow, thanks for your comment, Dan! I’m honored that you were able to read my writeup. I am too young to have used MacsBug for development during the time when it was heavily used, but I’ve gotten a lot of great use out of it recently while experimenting with these Macs from the era. Thank you for all the work you did on MacsBug! It truly is an excellent piece of software.

3. Dan Allen @ 2025-01-25 16:55

   Thank you Doug for your kind words.

   Dan

4. tim lindner @ 2025-01-25 20:36

   I had something like this recently happen the Mame’s Color Computer driver. A game loader was using an undocumented instruction intentionally.
   Fortunately we have someone in our community to really dig into undocumented 6809 instructions.
   https://github.com/hoglet67/6809Decoder/wiki/Undocumented-6809-Behaviours

5. Christian Zietz @ 2025-01-26 00:06

Really nice debugging and write-up!

I encountered similar weird behavior while debugging differences between a real Atari TT and an emulator. (The Atari TT has a 68030 CPU, too.) What I learned from that: Setting reserved bits in the extension word of the instruction does NOT cause an illegal instruction exception. It is even documented that it won't. Instead, all sorts of undocumented/undefined behavior can be triggered with "illegal" extension words.

6. Bob Felts @ 2025-01-26 10:26

   I miss the 68K, Macsbug, MacNosy, and MPW.

   Are you sure the CAS isn't triggering an illegal instruction exception and the handler isn't doing something screwy with A1?

   CAS was a wonderful instruction, but atomicity was lost when the address was that of a NuBus peripheral.

7. Doug Brown @ 2025-01-26 11:02

   Very cool link Tim! I wonder if this will lead to a similar analysis of the 68000 series.

   Thanks, Christian! Yeah, that seems to be the case — if the first word is valid, it won't be treated as an illegal instruction.

Funny that you mention the Atari TT. I recently bought a hard drive that ended up being from somebody's TT or Falcon (not sure which). Sadly I couldn't boot the image in an emulator because it crashes during boot.

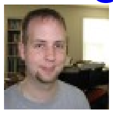
Bob, yes I am absolutely positive that it's not being treated as an illegal instruction. If I change it to a real illegal instruction like 0xFFFF and step into it, MacsBug notices and says "Unimplemented Instruction at xxxxxxxx". That doesn't happen with this instruction. It would be interesting for some of the Amiga and Atari folks to see if they can reproduce my results on an 030 too.

8. Christian Zietz @ 2025-01-26 11:07

   I tried the instruction sequence from the article ("0x0CEC 0x08A9 0x0004") in an Atari upgraded with a 68020. It does similar weird things as you observed on the 68030, writing to (A4+4) and altering A1 in the process. Like you, I couldn't deduce the actual "formula" for A1.

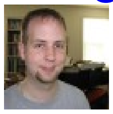
   (My Atari TT and Falcon are in storage right now, which is why I couldn't test it on a 68030. But it'll most likely be the same.)

9. Doug Brown @ 2025-01-26 11:13

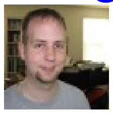
   

   Thanks Christian! That's great to see you reproduce it on a machine other than a Mac. Also interesting that it happens on the 68020 as well.

An interesting comment on the English Amiga Board by Toni Wilen points out that one of the bits that is 1 (probably bit 3?) may be choosing whether the D1 is really an A1, but it returns weird results. I'm guessing by changing bits 0-2 it will change which address register it messes with.

10. Doug Brown @ 2025-01-26 11:40


    Yeah, confirmed. If I change the low byte of the second word so that it's 0xA8 through 0xAF, it changes A0 through A7 respectively. If I use 0xA0 through 0xA7 instead (turning off bit 3), it changes D0 through D7 respectively (and seems to act more like a real CAS instruction, but I didn't test extensively).

11. Doug Brown @ 2025-01-26 19:20


    A couple more things to mention:

    MAME's debugger now has a fix for how it showed a wrong register in CAS instructions (even valid ones). Just to be clear, this is a minor unrelated thing I touched on in the post, not a fix for the execution of the undocumented instruction.

    I realized that there was a confusing point toward the end of the article. Without the CAS instruction fixing the address, a real Classic II crashes regardless of whether you're in 24-bit or 32-bit mode. The fact that it didn't crash in 24-bit mode during my initial testing in the emulator is a MAME-specific thing. MAME doesn't signal a bus error on the bad access in 24-bit mode, even though hardware does. I shouldn't have made the comment about how Apple's developers obviously didn't develop only in 24-bit mode, because it was completely irrelevant. It would have crashed either way if the 68030's CAS instruction wasn't fixing A1. I removed that sentence from the article, hopefully it's less confusing now.

12. Jim Murphy @ 2025-01-26 20:16

    Doug,

    This was some very good detective work, and could have been a great KON & BAL's Puzzle Page back in the old Develop magazine.

    I was the lead maintainer of MacsBug from 1993 until it was retired in the early 2000s.

    Jim
    still @Apple

13. The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully boot up – OSnews @ 2025-01-28 06:23

    […] ↫ Doug Brown […]

14. Zafer Akçalı @ 2025-01-28 10:02

    If somebody knows the instruction is, Jim Drew from utilites unlimited. if you can reach him now, you can ask.

15. Zafer Akçalı @ [2025-01-28 10:14]()

    Jim Drew may be the admin in that forum, and you can ask him questions: https://www.cbmstuff.com/forum/forumdisplay.php?fid=49

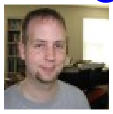
16. James Burgess @ [2025-01-28 11:34]()

    This is fascinating! I had built hardware and worked on compilers for m68k at college before ever seeing a Macintosh in 1993. My first reaction was, wait a minute, my code can crash this machine, what the heck…!? A year in I was loving MacsBug on my IIfx. Dan Allen thank you for such a delightful tool! I really miss programming that way, was mostly using ThinkC (instant incremental linking!) with the occasional chunks of assembler so debugging with MacsBug. Not much improvement since then in developer tools. Intellisense is about all I can think of.

17. SkYhAwK @ [2025-01-29 00:45]()

    Interesting would be to know how the misconstructed CAS instruction ended up like that in ROM. C compilers don't emit CAS instructions, so it could be only a buggy assembler, manual hex entry by the coder, or simply random/trashed byte sequence that happens to resemble a misconstructed CAS instruction.

18. Doug Brown @ 2025-01-29 07:09



Thank you all for your comments!

SkYhAwK, I covered that in the article. The assembler didn't assemble a CAS instruction. What happened is a bug in the ROM is jumping into the middle of an intended instruction, which just so happens to start with the same pattern that a CAS instruction would start with.

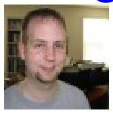
19. Peter Jerde @ 2025-01-29 09:32

What a great story, and a wonderful writeup! Thank you!

I wonder when it will be possible to emulate the *hardware* of a processor like the 68030 at the gate level. I know the 6502 has been done — I think you can actually watch an animation of one operating in "real time" somewhere on the web.

Being able to emulate the VLSI chips from machines of that era will be great, too, assuming there will ever be a way to reconstruct them virtually. I wish more companies would consider publishing the technical documents from their vaults for stuff from more than twenty years ago, before such details truly get lost to the sands of time. Though it's amazing what work has been done decapping and photographing the silicon to reverse engineer these things. I recently ran across a 74 MB jpeg image of the SWIM disk controller chip's gates, for example. Amazing.

20. Doug Brown @ 2025-01-29 11:59



Jim, I'm sorry that it took so long for your comment to show up. It ended up being incorrectly marked as spam by Akismet. Thank you for your comment, and I'm glad you were able to read my writeup! Thank you for all your hard work on MacsBug as well!

Thanks Peter! I agree with what you're saying. I think I saw the 6502 gate level emulation you are talking about. I too would love to see some of the old technical details published publicly so that we can preserve everything.

21. David Shayer @ 2025-01-29 14:52

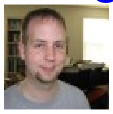
That's an awesome tale! What a fun bug to track down. Jim Murphy is right, that could have been in Kon & Bal's Puzzle Page in the old Apple Develop magazine.

I loved working on the classic Mac. The system was small enough that an engineer could understand most of it, and disassemble it all with tools like Macsbug. Modern OSes are so large and complex that there's no way for even a very skilled engineer to understand more than a small part in depth.

I taught classes on debugging with Macsbug in Apple engineering, and I co-wrote the Macsbug Reference and Debugging Guide.

https://www.amazon.com/Macsbug-Reference-Debugging-Guide-Technical/dp/0201567679

If you're doing this kind of work, you must read How to Write Macintosh Software by Scott Knaster. It's all about how the internals of classic Mac OS really works.

22. Doug Brown @ 2025-01-29 19:52


Hi David,

Thanks! You're absolutely right, it was a lot of fun. And an excuse to buy another Classic Mac 🙂 I know what you mean about how everything was small enough to understand so much about it back then. Plus, I am a big fan of the 68000-series assembly. I find it so much easier to read than some of the other architectures, although ARM is pretty nice too.

That is really cool that you were one of the authors of Apple's MacsBug book! It sounds just like what I need to look at. It has been really fun hearing from several Apple folks from the time who were heavily involved with MacsBug. I'm so glad you all have been commenting! I think it would be cool if one of the ROM developers ends up seeing this. I wonder if any of them knew that this sneaky little "bug" existed. I put it in quotes since it didn't actually cause any problems.

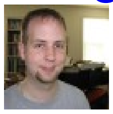
Thank you for the book recommendation. I'll check it out! Sounds like everyone should check out some of the old Develop magazines too, since both you and Jim mentioned it!

23. Mark Lentczner @ 2025-01-31 11:25

Fantastic work!

What memories… I was on the Mac team from 85 though 89, and this machine was after me. But I spent tons of time in macsbug and coding 680×0 by hand. I was the co-developer of the Apple Sound Chip, so it was extra fun to see the tie-in, here.

24. Doug Brown @ 2025-01-31 21:12



Thanks Mark! Wow, that's really cool! I spent a lot of time a while ago learning how to interact directly with the ASC in order to customize my IIci by giving it a special sampled startup sound instead of the original one played with the wave table synthesizer. It was a neat little chip.

25. The Mac Classic II Shouldn't Have Worked — 512 Pixels @ 2025-02-07 07:30

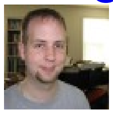
[…] The Mac Classic II Shouldn't Have Worked → […]

26. Ed Rupp @ 2025-02-07 17:59

Nice sleuthing… I agree that bit3 is what is modifying D1 to A1 and suspect that bit9 would also affect Du. Illegal opcode decoding is probably not smart enough to look at bits beyond the first 2 (or MAYBE 4) bytes. I worked at Motorola Austin from 78-89 and wrote the microcode assembler for the 68020. Up to that point, the microcode was done on 3×5 cards…

Unrelated, but maybe interesting: At some point I was asked (maybe by an Apple programmer?) what instruction caused the most potential page faults. I think the answer was CAS2 because it could touch two unrelated memory addresses and the instruction could straddle 2 pages. So 6 potential page faults for this instruction.

27. Doug Brown @ 2025-02-07 19:33

    Thanks, Ed! That makes sense about bit 9. Very cool that you wrote the microcode assembler for the 68020. It sounds like the 68020 has very similar behavior to what I observed on the 030 based on comments from others. Wow…3×5 cards for the microcode…can only imagine what that would have been like!

    6 page faults in a single instruction, that would be a lot!

28. Alex Rosenberg @ 2025-02-11 10:03

    Ed, we asked that question about page faults at one of the Apple WWDC Stump the Experts nights. IIRC the answer at that time was a MOVE16 from the ‘040 and it was a lot more than six faults.

29. Ed Rupp @ 2025-02-11 13:30

    Nick Tredennick wrote the 68000 microcode and used 3×5 cards. The 68010 was a close derivative and I think still used the cards. Doug MacGregor wrote the 68020 microcode. Design verification was done by building the whole processor with 74xx TTL on giant wire-wrap boards.

The 6 faults was for the ‘030. I never investigated the maximum for the ‘040, but MOVE16 sounds right. The last thing I did on the ‘040 was to clean up the floating point emulation layer. I think the code eventually got released to the Linux kernel but that was after I left. At the time, the ‘040 held the record for most mask layer revisions before production, I think…

Add your comment now

Name (required)

Email (Will NOT be published) (required)

URL

Submit

- **Subscribe**

- ## Recent Posts
    - The gooey rubber that’s slowly ruining old hard drives
    - The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully boot up
    - Easy repair of a defective NZXT Signal 4K30 capture card
    - How webcams with focus control work (Razer Kiyo Pro repair)
    - The capacitor that Apple soldered incorrectly at the factory
    - Hardware repair of an Elgato HD60 S that only worked on Mac
    - Are wireless gamepads terrible? Mario Maker TAS playback with an RP2040
    - Fixing an Elgato HD60 S HDMI capture device with the help of Ghidra
- ## Categories
    - Chumby 8 kernel (13)
    - Classic Mac (11)
    - Computer repair (10)
    - Electronics repair (8)
    - iOS (3)
    - Linux (43)
    - Mac ROM hacking (11)
    - Microcontroller lessons (11)
    - Microcontrollers (4)
    - Product reviews (5)
    - Python (1)
    - Qt (5)

    - Reverse engineering (3)
    - Uncategorized (20)
    - Windows (7)

- ## Archives

    - March 2025 (1)
    - January 2025 (2)
    - December 2024 (1)
    - November 2024 (1)
    - October 2024 (2)
    - September 2024 (1)
    - August 2024 (1)
    - July 2024 (3)
    - June 2024 (4)
    - May 2024 (1)
    - April 2024 (2)
    - December 2023 (1)
    - November 2023 (2)
    - September 2023 (3)
    - August 2023 (3)
    - June 2023 (1)
    - May 2023 (1)
    - April 2023 (1)
    - March 2023 (2)
    - January 2023 (1)
    - December 2022 (3)
    - August 2022 (1)
    - May 2022 (2)
    - March 2022 (1)
    - December 2021 (1)
    - June 2021 (1)

- April 2021 (1)
- January 2021 (1)
- September 2020 (1)
- August 2020 (1)
- July 2020 (1)
- May 2020 (1)
- June 2019 (1)
- April 2019 (1)
- December 2018 (1)
- August 2018 (1)
- May 2018 (1)
- April 2018 (3)
- February 2018 (1)
- October 2017 (1)
- July 2017 (1)
- May 2017 (3)
- March 2017 (1)
- October 2016 (1)
- June 2015 (1)
- March 2015 (1)
- November 2014 (1)
- August 2014 (3)
- July 2014 (1)
- April 2014 (1)
- March 2014 (1)
- February 2014 (1)
- November 2013 (1)
- August 2013 (1)
- June 2013 (3)
- April 2013 (1)
- March 2013 (1)
- January 2013 (2)
- December 2012 (2)

- August 2012 (1)
- July 2012 (2)
- June 2012 (1)
- May 2012 (1)
- February 2012 (3)
- January 2012 (1)
- November 2011 (1)
- October 2011 (2)
- August 2011 (3)
- May 2011 (1)
- April 2011 (1)
- March 2011 (2)
- November 2010 (2)
- October 2010 (3)
- July 2010 (5)

Recent Comments

- Doug Brown on The gooey rubber that’s slowly ruining old hard drives
- Steve on The gooey rubber that’s slowly ruining old hard drives
- Chris on The gooey rubber that’s slowly ruining old hard drives
- Ed Rupp on The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully boot up
- Alex Rosenberg on The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully boot up
- Doug Brown on The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully

boot up
    - Ed Rupp on The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully boot up
    - The Mac Classic II Shouldn't Have Worked — 512 Pixels on The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully boot up
    - Doug Brown on The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully boot up
    - Mark Lentczner on The invalid 68030 instruction that accidentally allowed the Mac Classic II to successfully boot up

- ## Spam Blocked

251,018 spam

blocked by Akismet