Two bits per transistor: high-density ROM in Intel's 8087 floating point chip

The 8087 chip provided fast floating point arithmetic for the original IBM PC and became part of the x86 architecture used today. One unusual feature of the 8087 is it contained a multi-level ROM (Read-Only Memory) that stored two bits per transistor, twice as dense as a normal ROM. Instead of storing binary data, each cell in the 8087's ROM stored one of four different values, which were then decoded into two bits. Because the 8087 required a large ROM for microcode[1] and the chip was pushing the limits of how many transistors could fit on a chip, Intel used this special technique to make the ROM fit. In this article, I explain how Intel implemented this multi-level ROM.

Intel introduced the 8087 chip in 1980 to improve floating-point performance on the 8086 and 8088 processors. Since early microprocessors operated only on integers, arithmetic with floating point numbers was slow and transcendental operations such as trig or logarithms were even worse. Adding the 8087 co-processor chip to a system made floating point operations up to 100 times faster. The 8087's architecture became part of later Intel processors, and the 8087's instructions (although now obsolete) are still a part of today's x86 desktop computers.

I opened up an 8087 chip and took die photos with a microscope yielding the composite photo below. The labels show the main functional blocks, based on my reverse engineering. (Click here for a larger image.) The die of the 8087 is complex, with 40,000 transistors.[2] Internally, the 8087 uses 80-bit floating point numbers with a 64-bit fraction (also called significand or mantissa), a 15-bit exponent and a sign bit. (For a base-10 analogy, in the number 6.02×10^{23}, 6.02 is the fraction and 23 is the exponent.) At the bottom of the die, "fraction processing" indicates the circuitry for the fraction: from left to right, this includes storage of constants, a 64-bit shifter, the 64-bit adder/subtracter, and the register stack. Above this is the circuitry to process the exponent.

Contact

About Ken Shirriff
Mastodon

Popular Posts

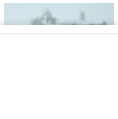

Undocumented 8086 instructions, explained by the microcode

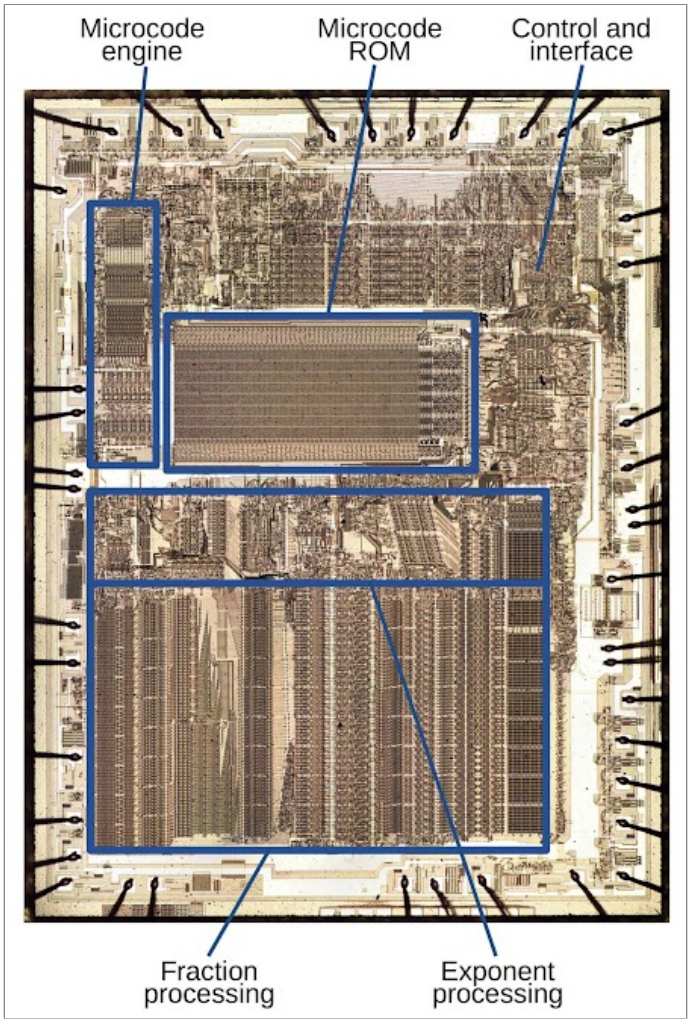
The complex history of the Intel i960 RISC processor


Reverse-engineering the 8086 processor's address and data pin circuits


12-minute Mandelbrot: fractals on a 50 year old IBM 1401 mainframe


A Multi-Protocol Infrared Remote Library for the Arduino


Apple



Die of the Intel 8087 floating point unit chip, with main functional blocks labeled.

An 8087 instruction required multiple steps, over 1000 in some cases. The 8087 used microcode to specify the low-level operations at each step: the shifts, adds, memory fetches, reads of constants, and so forth. You can think of microcode as a simple program, written in micro-instructions, where each micro-instruction generated control signals for the different components of the chip. In the die photo above, you can see the ROM that holds the 8087's microcode program. The ROM takes up a large fraction of the chip, showing why the compact multi-level ROM was necessary. To the left of the ROM is the "engine" that ran the microcode program, essentially a simple CPU.

The 8087 operated as a co-processor with the 8086 processor. When the 8086 encountered a special floating point instruction, the processor ignored it and let the 8087 execute the instruction in parallel.[3] I won't explain in detail how the 8087 works internally, but as an overview, floating point operations were implemented using integer adds/subtracts and shifts. To add or subtract two floating point numbers, the 8087 shifted the numbers until the binary points (i.e. the decimal points but in binary) lined up, and then added or subtracted the fraction. Multiplication, division, and square root were performed through repeated shifts and adds or subtracts. Transcendental operations (tan, arctan, log, power) used CORDIC algorithms, which use shifts and adds of special constants, processing one bit at a time. The 8087 also dealt with many special cases: infinities, overflows, NaN (not a number), denormalized numbers, and several rounding modes. The microcode stored in ROM controlled all these operations.


Teardown and exploration of Apple's Magsafe connector


Understanding silicon circuits: inside the ubiquitous 741 op amp

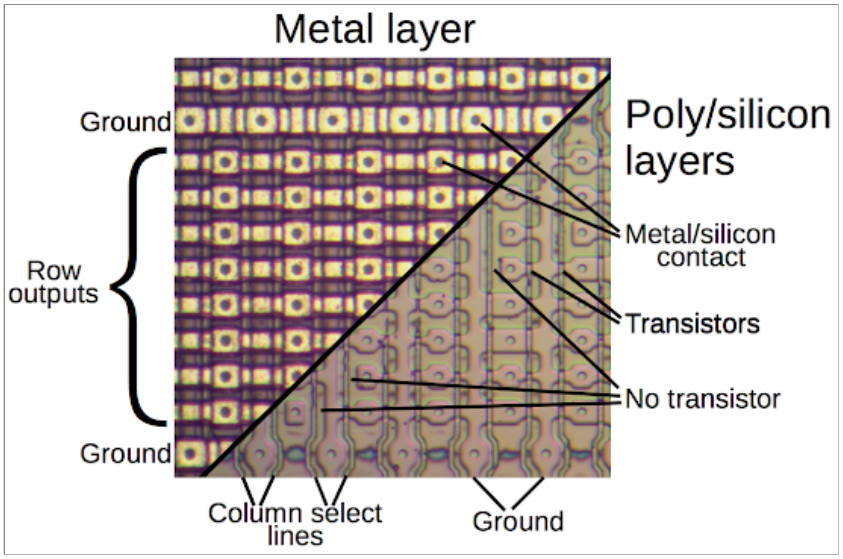
Search This Blog

Search

Labels

6502 8008 8085 8086 8087 alto analog Apollo apple arc arduino arm beaglebone bitcoin c# cadc calculator chips css dx7 electronics f# fpga fractals genome globus haskell html5 ibm ibm1401 intel ipv6

The 8087 chip consists of a tiny silicon die, with regions of the silicon doped with impurities to give them the desired semiconductor properties. On top of the silicon, polysilicon (a special type of silicon) formed wires and transistors. Finally, a metal layer on top wired the circuitry together. In the photo below, the left side shows a small part of the chip as it appears under a microscope, magnifying the yellowish metal wiring. On the right, the metal has been removed with acid, revealing the polysilicon and silicon. When polysilicon crosses silicon, a transistor is formed. The pink regions are doped silicon, and the thin vertical lines are the polysilicon. The small circles are contacts between the silicon and metal layers, connecting them together.



Structure of the ROM in the Intel 8087 FPU. The metal layer is on the left and the polysilicon and silicon layers are on the right.

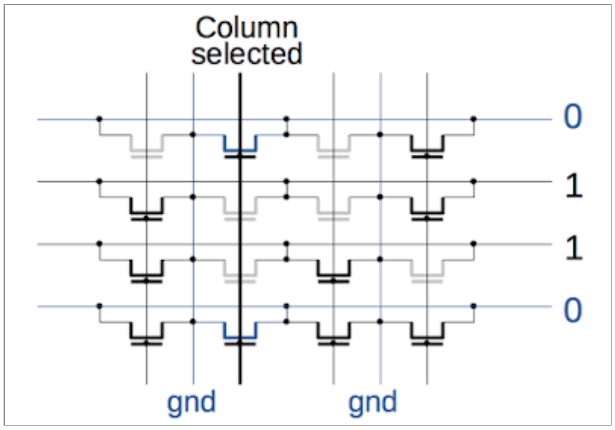
While there are many ways of building a ROM, a typical way is to have a grid of "cells," with each cell holding a bit. Each cell can have a transistor for a 0 bit, or lack a transistor for a 1 bit. In the diagram above, you can see the grid of cells with transistors (where silicon is present under the polysilicon) and missing transistors (where there are gaps in the silicon). To read from the ROM, one column select line is energized (based on the address) to select the bits stored in that column, yielding one output bit from each row. You can see the vertical polysilicon column select lines and the horizontal metal row outputs in the diagram. The vertical doped silicon lines are connected to ground.

The schematic below (corresponding to a 4×4 ROM segment) shows how the ROM functions. Each cell either has a transistor (black) or no transistor (grayed-out). When a polysilicon column select line is energized, the transistors in that column turn on and pull the corresponding metal row outputs to ground. (For our purposes, an NMOS transistor is like a switch that is open if the input (gate) is 0 and closed if the input is 1.) The row lines output the data stored in the selected column.

engineering
sheevaplug snark space spanish synth teardown theory unicode Z-80

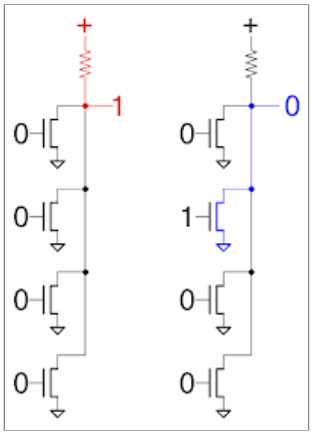
Blog Archive

► 2023 (23)
► 2022 (18)
► 2021 (26)
► 2020 (33)
► 2019 (18)
▼ 2018 (17)
► December (1)
▼ September (4)
Two bits per transistor: high-density ROM in Intel...
Bad relay: Fixing the card reader for a vintage IB...
The printer that wouldn't print: Fixing an IBM 140...
Glowing mercury thyratrons: inside a 1940s Teletyp...
► August (1)
► June (1)
► May (1)
► April (1)
► March (3)
► February (1)
► January (4)
► 2017 (21)
► 2016 (34)
► 2015 (12)
► 2014 (13)
► 2013 (24)
► 2012 (10)
► 2011 (11)
► 2010 (22)
► 2009 (22)
► 2008 (27)



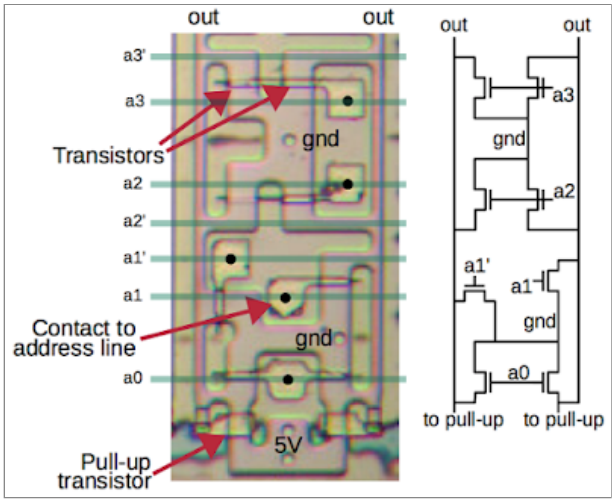
Schematic of a 4×4 segment of a ROM.

The column select signals are generated by a decoder circuit. Since this circuit is built from NOR gates, I'll first explain the construction of a NOR gate. The schematic below shows a four-input NOR gate built from four transistors and a pull-up resistor (actually a special transistor). On the left, all inputs are 0 so all the transistors are off and the pull-up resistor pulls the output high. On the right, an input is 1, turning on a transistor. The transistor is connected to ground, so it pulls the output low. In summary, if any inputs are high, the output is low so this circuit implements a NOR gate.



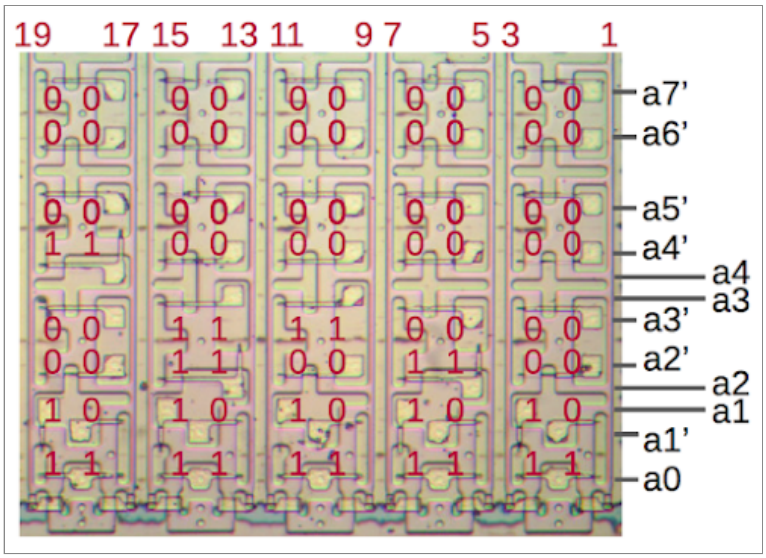
4-input NOR gate constructed from NMOS transistors.

The column select decoder circuit takes the incoming address bits and activates the appropriate select line. The decoder contains an 8-input NOR gate for each column, with one NOR gate selected for the desired address. The photo shows two of the NOR gates generating two of the column select signals. (For simplicity, I only show four of the 8 inputs). Each column uses a different combination of address lines and complemented address lines as inputs, selecting a different address. The address lines are in the metal layer, which was removed for the photo below; the address lines are drawn in green. To determine the address associated with a column, look at the square contacts associated with each transistor and note which address lines are connected. If all the address lines connected to a column's transistors are low, the NOR gate will select the column.



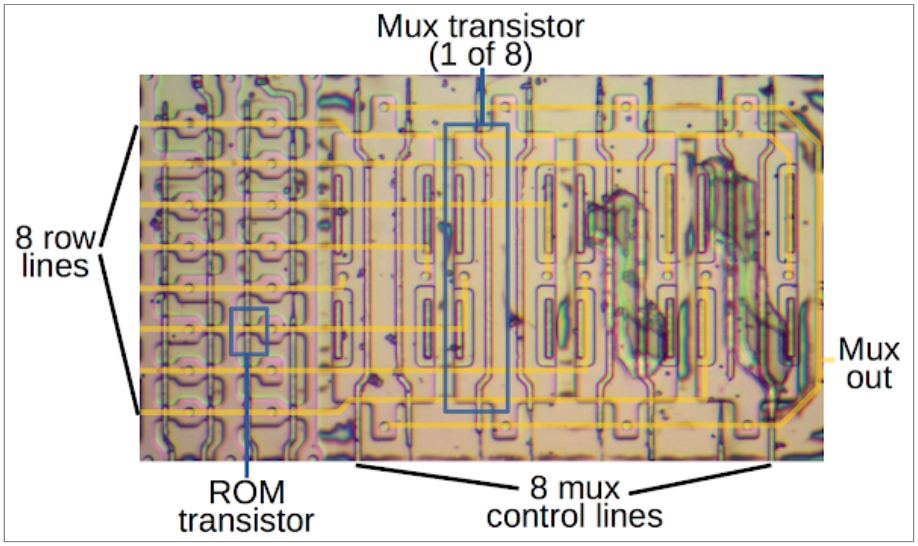
Part of the address decoder. The address decoder selects odd columns in the ROM, counting right to left. The numbers at the top show the address associated with each output.

The photo below shows a small part of the ROM's decoder with all 8 inputs to the NOR gates. You can read out the binary addresses by carefully examining the address line connections. Note the binary pattern: a1 connections alternate every column, a2 connections alternate every two columns, a3 connections every four columns, and so forth. The a0 connection is fixed because this decoder circuit selects the odd columns; a similar circuit above the ROM selects the even addresses. (This split was necessary to make the decoder fit on the chip because each decoder column is twice as wide as a ROM cell.)



Part of the address decoder for the 8087's microcode ROM. The decoder converts an 8-bit address into column select signals.

The last component of the ROM is the set of multiplexers that reduces the 64 output rows down to 8 rows.[4] Each 8-to-1 multiplexer selects one of its 8 inputs, based on the address. The diagram below shows one of these row multiplexers in the 8087, built from eight large pass transistors, each one connected to one of the row lines. All the transistors are connected to the output so when the selected transistor is turned on, it passes its input to the output. The multiplexer transistors are much, much larger than the transistors in the ROM to reduce distortion of the ROM signal. A decoder (similar to the one discussed earlier, but smaller) generates the eight multiplexer control lines from three address lines.

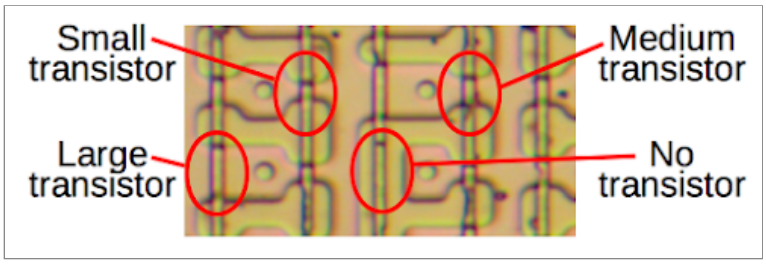


One of eight row multiplexers in the ROM. This shows the poly/silicon layers, with metal wiring drawn in orange.

To summarize, the ROM stores bits in a grid. It uses eight address bits to select a column in the grid. Then three address bits select the desired eight outputs from the row lines.

The multi-level ROM

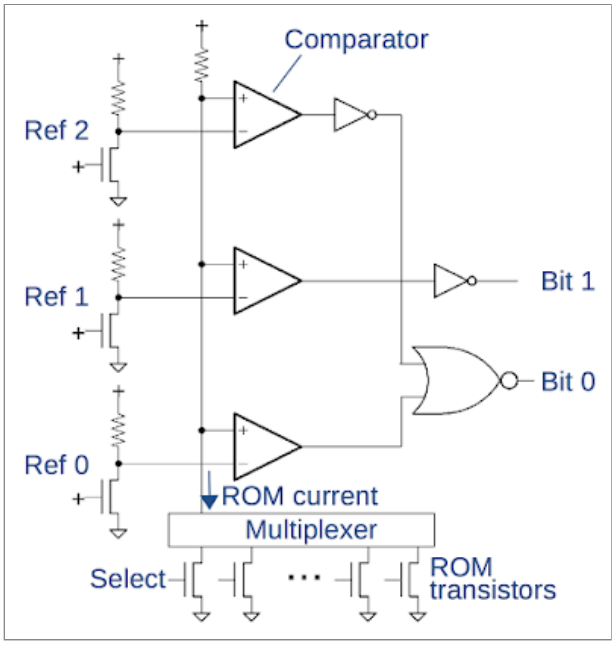
The discussion so far explained of a typical ROM that stores one bit per cell. So how did 8087 store two bits per cell? If you look closely, the 8087's microcode ROM has four different transistor sizes (if you count "no transistor" as a size).[6] With four possibilities for each transistor, a cell can encode two bits, approximately doubling the density.[7] This section explains how the four transistor sizes generate four different currents, and how the chip's analog and digital circuitry converts these currents into two bits.



A closeup of the 8087's microcode ROM shows four different transistor sizes. This allows the ROM to store two bits per cell.

The size of the transistor controls the current through the transistor.[8] The important geometric factor is the varying width of the silicon (pink) where it is crossed by the polysilicon (vertical lines), creating transistors with different gate widths. Since the gate width controls the current through the transistor, the four transistor sizes generate four different currents: the largest transistor passes the most current and no current will flow if there is no transistor at all.

The ROM current is converted to bits in several steps. First, a pull-up resistor converts the current to a voltage. Next, three comparators compare the voltage with reference voltages to generate digital signals indicating if the ROM voltage is lower or higher. Finally, logic gates convert the comparator output signals to the two output bits. This circuitry is repeated eight times, generating 16 output bits in total.



The circuit to read two bits from a ROM cell.

The circuit above performs these conversion steps. At the bottom, one of the ROM transistors is selected by the column select line and the multiplexer (discussed earlier), generating one of four currents. Next, a pull-up resistor[12] converts the transistor's current to a voltage, resulting in a voltage depending on the size of the selected transistor. The comparators compare this voltage to three reference voltages, outputting a 1 if the ROM voltage is higher than the reference voltage. The comparators and reference voltages require careful design because the ROM voltages could differ by as little as 200 mV.

The reference voltages are mid-way between the expected ROM voltages, allowing some fluctuation in the voltages. The lowest ROM voltage is lower than all the reference voltages so all comparators will output 0. The second ROM voltage is higher than Reference 0, so the bottom comparator outputs 1. For the third ROM voltage, the bottom two comparators output 1, and for the highest ROM voltage all comparators output 1. Thus, the three comparators yield four different output patterns depending on the ROM transistor. The logic gates then convert the comparator outputs into the two output bits.[10]

The design of the comparator is interesting because it is the bridge between the analog and digital worlds, producing a 1 or 0 if the ROM voltage is higher or lower than the reference voltage. Each comparator contains a differential amplifier that amplifies the difference between the ROM voltage and the reference voltage. The output from the differential amplifier drives a latch that stabilizes the output and converts it to a logic-level signal. The differential amplifier (below) is a standard analog circuit. A current sink (symbol at the bottom) provides a constant current. If one of the transistors has a higher input voltage than the other, most of the current passes through that transistor. The voltage drop across the resistors will cause the corresponding output to go lower and the other output to go higher.

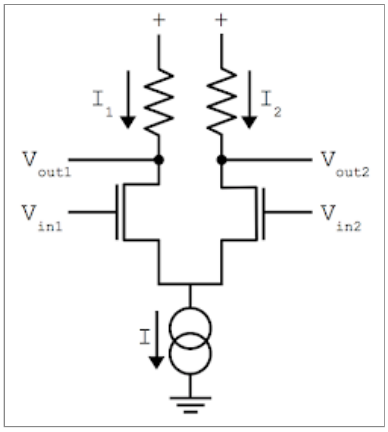
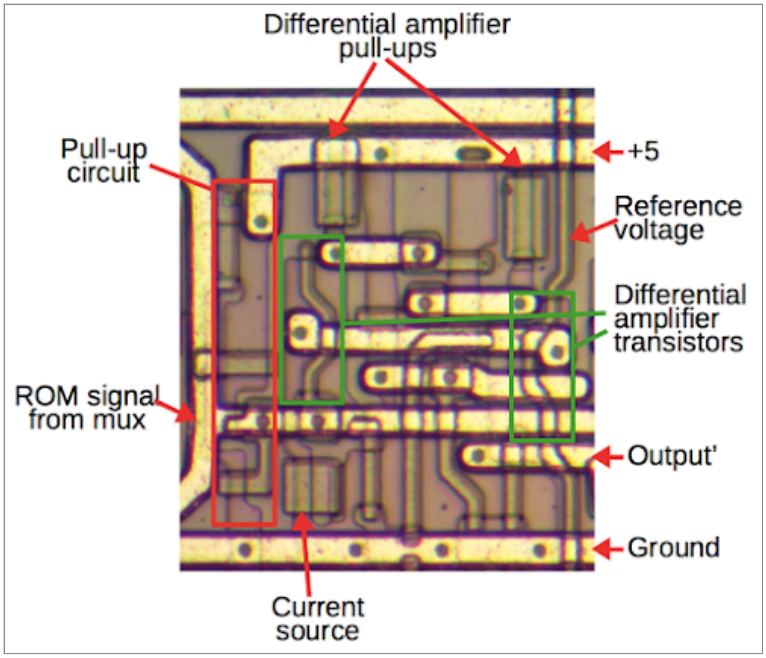


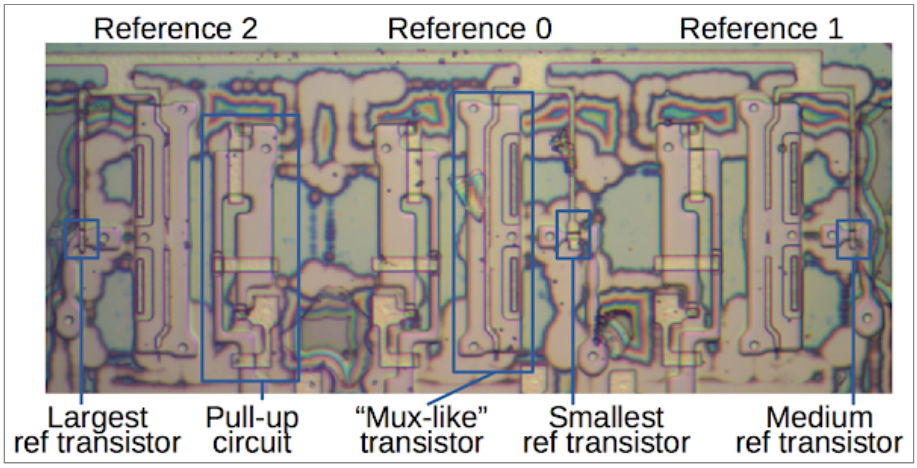
Diagram showing the operation of a differential pair. Most of the current will flow through the transistor with the higher input voltage, pulling the corresponding output lower. The double-circle symbol at the bottom is a current sink, providing a constant current I.

The photo below shows one of the comparators on the chip; the metal layer is on top, with the transistors underneath. I'll just discuss the highlights of this complex circuit; see the footnote [12] for details. The signal from the ROM and multiplexer enters on the left. The pull-up circuit [12] converts the current into a voltage. The two large transistors of the differential amplifier compare the ROM's voltage with the reference voltage (entering at top). The outputs from the differential amplifier go to the latch circuitry (spread across the photo); the latch's output is in the lower right. The differential amplifier's current source and pull-up resistors are implemented with depletion-mode transistors. Each output circuit uses three comparators, yielding 24 comparators in total.



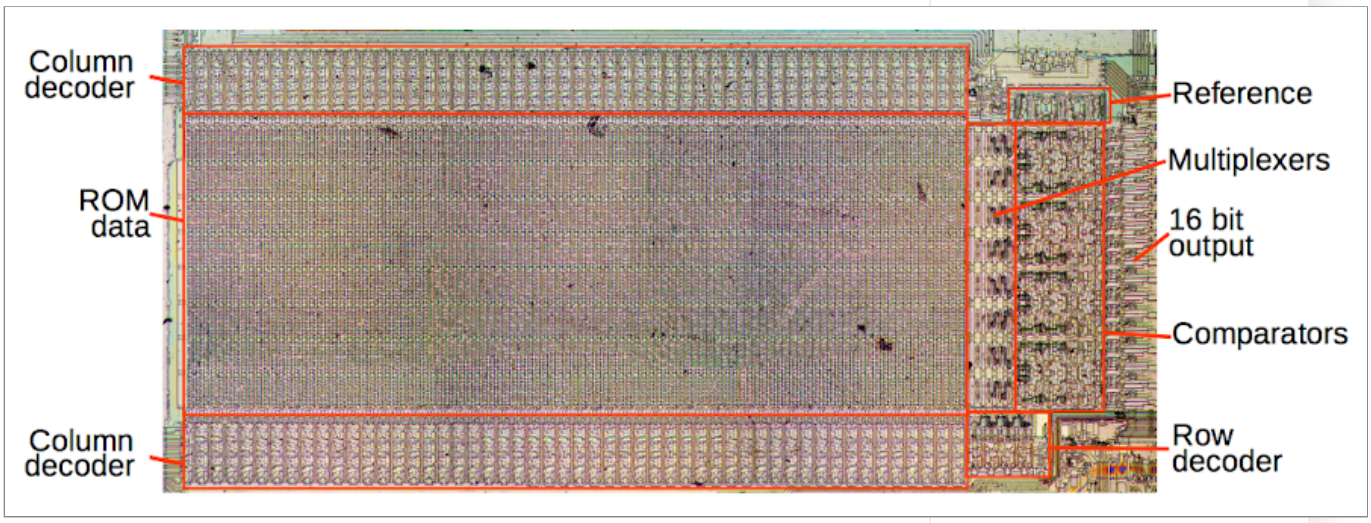
One of the comparators in the 8087. The chip contains 24 comparators to convert the voltage levels from the multi-level ROM into binary data.

Each reference voltage is generated by a carefully-sized transistor and a pull-up circuit. The reference voltage circuit is designed as similar as possible to the ROM's signal circuitry, so any manufacturing variations in the chip will affect both equally. The reference voltage and ROM signal both use the same pull-up circuit. In addition, each reference voltage circuit includes a very large transistor identical to the



Circuit generating the three reference voltages. The reference transistors are sized between the ROM's transistor sizes. The oxide layer wasn't fully removed from this part of the die, causing the color swirls in the photo.

Putting all the pieces together, the photo below shows the layout of the microcode ROM components on the chip.[12] The bulk of the ROM circuitry is the transistors holding the data. The column decoder circuitry is above and below this. (Half the column select decoders are at the top and half are at the bottom so they fit better.) The output circuitry is on the right. The eight multiplexers reduce the 64 row lines down to eight. The eight rows then go into the comparators, generating the 16 output bits from the ROM at the right. The reference circuit above the comparators generates the three reference voltage. At the bottom right, the small row decoder controls the multiplexers.



Microcode ROM from the Intel 8087 FPU with main components labeled.

While you'd hope for the multi-level ROM to be half the size of a regular ROM, it isn't quite that efficient because of the extra circuitry for the comparators and because the transistors were slightly larger to accommodate the multiple sizes. Even so, the multi-level ROM saved about 40% of the space a regular ROM would have taken.

Now that I have determined the structure of the ROM, I could read out the contents of the ROM simply (but tediously) by looking at the size of each transistor under a microscope. But without knowing the microcode instruction set, the ROM contents aren't useful.

doomed iAPX 432 system.[11] As far as I can tell, interest in ROMs with multiple-level cells peaked in the 1980s and then died out, probably because Moore's law made it easier to gain ROM capacity by shrinking a standard ROM cell rather than designing non-standard ROMs requiring special analog circuits built to high tolerances.[14]

Surprisingly, the multi-level concept has recently returned, but this time in flash memory. Many flash memories store two or more bits per cell.[13] Flash has even achieved a remarkable 4 bits per cell (requiring 16 different voltage levels) with "quad-level cell" consumer products announced recently. Thus, an obscure technology from the 1980s can show up again decades later.

I announce my latest blog posts on Twitter, so follow me at @kenshirriff for future 8087 articles. I also have an RSS feed. Thanks to Jeff Epler for suggesting that I investigate the 8087's ROM.

Notes and references

1. The 8087 has 1648 words of microcode (if I counted correctly), with 16 bits in each word, for a total of 26368 bits. The ROM size didn't need to be a power of two since Intel could build it to the exact size required. ↩

2. Sources provide inconsistent values for the number of transistors in the 8087: Intel claims 40,000 transistors while Wikipedia claims 45,000. The discrepancy could be due to different ways of counting transistors. In particular, since the number of transistors in a ROM, PLA or similar structure depends on the data stored in it, sources often count "potential" transistors rather than the number of physical transistors. Other discrepancies can be due to whether or not pull-up transistors are counted and if high-current drivers are counted as multiple transistors in parallel or one large transistor. ↩

3. The interaction between the 8086 processor and the 8087 floating point unit is somewhat tricky; I'll discuss some highlights. The simplified view is that the 8087 watches the 8086's instruction stream, and executes any instructions that are 8087 instructions. The complication is that the 8086 has an instruction prefetch buffer, so the instruction being fetched isn't the one being executed. Thus, the 8087 duplicates the 8086's prefetch buffer (or the 8088's smaller prefetch buffer), so it knows that the 8086 is doing. Another complication is the complex addressing modes used by the 8086, which use registers inside the 8086. The 8087 can't perform these addressing modes since it doesn't have access to the 8086 registers. Instead, when the 8086 sees an 8087 instruction, it does a memory fetch from the addressed location and ignores the result. Meanwhile, the 8087 grabs the address off the bus so it can use the address if it needs it. If there is no 8087 present, you might expect a trap, but that's not what happens. Instead, for a system without an 8087, the linker rewrites the 8087 instructions, replacing them with subroutine calls to the emulation library. ↩

4. The reason ROMs typically use multiplexers on the row outputs is that it is inefficient to make a ROM with many columns and just a few output bits, because the decoder circuitry will be bigger than the ROM's data. The solution is to reshape the ROM, to hold the same bits but with more rows and fewer columns. For instance, the ROM can have 8 times as many rows and 1/8 the columns, making the decoder 1/8 the size.

   In addition, a long, skinny ROM (e.g. 1K×16) is inconvenient to lay out on a chip, since it won't fit as a simple block. However, a serpentine layout could be used. For example, Intel's early memories were shift registers; the 1405 held 512 bits in a single long shift register. To fit this onto a chip, the shift register

computer sensed the holes capacitively (link). Some computers, such as the Xerox Alto, had some microcode in RAM. This allowed programs to modify the microcode, creating a new instruction set for their specific purposes. Many modern processors have writeable microcode so patches can fix bugs in the microcode. ↩

6. I didn't notice the four transistor sizes in the microcode ROM until a comment on Hacker News mentioned that the 8087 used two-bit-per-cell technology. I was skeptical, but after looking at the chip more closely I realized the comment was correct. ↩

7. Several other approaches were used in the 1980s to store multiple bits per cell. One of the most common was used by Mostek and other companies: transistors in the ROM were doped to have different threshold voltages. By using four different threshold voltages, two bits could be stored per cell. Compared to Intel's geometric approach, the threshold approach was denser (since all the transistors could be as small as possible), but required more mask layers and processing steps to produce the multiple implantation levels. This approach used the new (at the time) technology of ion implantation to carefully tune the doping levels of each transistor.

   Ion implantation's biggest impact on integrated circuits was its use to create depletion transistors (transistors with a negative threshold voltage), which worked much better as pull-up resistors in logic gates. Ion implantation was also used in the Z-80 microprocessor to create some transistor "traps", circuits that looked like regular transistors under a microscope but received doping implants that made them non-functional. This served as copy protection since a manufacturer that tried to produce clones on the Z-80 by copying the chip with a microscope would end up with a chip that failed in multiple ways, some of them very subtle. ↩

8. The current through the transistor is proportional to the ratio between the width and length of the gate. (The length is the distance between the source and drain.) The ROM transistors (and all but the smallest reference transistor) keep the length constant and modify the width, so shrinking the width reduces the current flow. For MOSFET equations, see Wikipedia. ↩

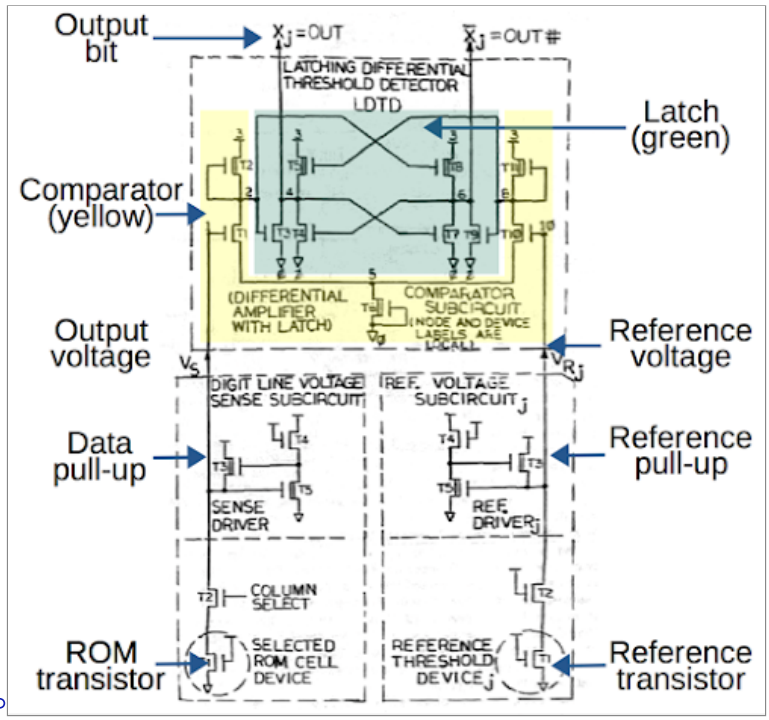
9. The gate of the smallest reference transistor is made longer rather than narrower, due to the properties of MOS transistors. The problem is that the reference transistors need to have sizes between the sizes of the ROM transistors. In particular, Reference 0 needs a transistor smaller than the smallest ROM transistor. But the smallest ROM transistor is already as small as possible using the manufacturing techniques. To solve this, note that the polysilicon crossing the middle reference transistor is much thicker horizontally. Since a MOS transistor's properties are determined by the width to height ratio of its gate, expanding the polysilicon is as good as shrinking the silicon for making the transistor act smaller (i.e. lower current). ↩

10. The ROM logic decodes the transistor size to bits as follows: No transistor = 00, small transistor = 01, medium transistor = 11, large transistor = 10. This bit ordering saves a few gates in the decoding logic; since the mapping from transistor to bits is arbitrary, it doesn't matter that the sequence is not in order. (See "Two Bits Per Cell ROM", Stark for details.) ↩

11. Intel's iAPX 43203 interface processor (1981) used a multiple-level ROM very similar to the one in the 8087 chip. For details, see "The interface processor for the Intel VLSI 432 32 bit computer," J. Bayliss et al., IEEE J. Solid-State Circuits, vol. SC-16, pp. 522-530, Oct. 1981.

processor as a stopgap, releasing it in 1978. While the Intel 8086 was a huge success, leading to the desktop PC and the current x86 architecture, the iAPX 432 project ended up a failure and ended in 1986. ↩

12. The schematic below (from "Multiple-Valued ROM Output Circuits") provides details of the circuitry to read the ROM. Conceptually the ROM uses a pull-up resistor to convert the transistor's current to a voltage. The circuit actually uses a three transistor circuit (T3, T4, T5) as the pull-up. T4 and T5 are essentially an inverter providing negative feedback via T3, making the circuit less sensitive to perturbations (such as manufacturing variations). The comparator consists of a simple differential amplifier (yellow) with T6 acting as the current source. The differential amplifier output is converted into a stable logic-level signal by the latch (green).



↩

Diagram of 8087 ROM output circuit.

13. Flash memories are categorized as SLC (single level cell—one bit per cell), MLC (multi level cell—two bits per cell), TLC (triple level cell—three bits per cell) and QLC (quad level cell—four bits per cell). In general, flash with more bits per cell is cheaper but less reliable, slower, and wears out faster due to the smaller signal margins. ↩

14. The journal *Electronics* published a short article "Four-State Cell Doubles ROM Bit Capacity" (p39, Oct 9, 1980), describing Intel's technique, but the article is vague to the point of being misleading. Intel published a detailed article "Two bits per cell ROM" in COMPCON (pp209-212, Feb 1981). An external group attempted to reverse engineer more detailed specifications of the Intel circuits in "Multiple-valued ROM output circuits" (Proc. 14th Int. Symp. Multivalue Logic, 1984). Two papers describing multiple-value memories are A Survey of Multivalued Memories (IEEE Transactions on Computers, Feb 1986, pp 99-106) and A review of multiple-valued memory technology (IEEE Symposium on Multiple-Valued Logic, 1998). ↩

Peter Ibbotson said...

Microsoft and Borland both used int 34h-3eh followed by the 8087 opcodes, and IIRC if a co-processor was found the int xx instruction was overwritten by a NOP so this was done at runtime rather than link.

I believe that both of the emulators were written by "Tanj Bennett" but I could be wrong.

September 30, 2018 at 1:22 PM

---

gpshead said...

Such a code modification normally be done by the executable loader today. If DOS had that concept, it could. Otherwise the fixer could just be in the executable itself as the first code to be run.

September 30, 2018 at 2:20 PM

KE5FX said...

As I recall, Dave Stafford (now at Amazon) wrote an x87 emulator for Borland. Whether it was used for Turbo Pascal, Turbo C, or both, I'm not sure.

September 30, 2018 at 5:03 PM

Josh O said...

Great write-up. I love these techniques that were used to squeeze the most performance out of limited technology. Coming from early days programming 6502 ASM as a teen, and then later using microcontrollers for some projects at work, I've always had a tendency to not waste resources, which I fear is lost on newer developers who think nothing of creating large buffers or using data-types that are much larger than needed (though matching the native word size of the CPU may be more important depending on whether you are targeting speed or memory use). I think every young developer should be taught a history of computing where they learn about such techniques.

I love the idea of multiplexing multiple bits into a cell. Reminds me of QAM which besides the phase of the wave at a particular moment also uses varying amplitudes/signal-strengths to encode a "bit" more data on each symbol.

September 30, 2018 at 7:37 PM

Anonymous said...

Hi, iAPX432 circuit designer here. The active-mask programmed 2 bits/cell NMOS ROM had two more disadvantages that you didn't mention. First, it was slower. It had 3X as many sense amps as a 1 bit/cell ROM, so each SA was operated at far less current, making them slower. And because the voltage windows around each of the 4 possible bitline voltages were smaller, you need to wait longer (greater N in the expression Twait = N*Rcell*Cbitline) before latching the sensed value, in order to guarantee success in the worst case of imperfect mask alignment and linewidth variation.

Second, it required one metal-to-active contact per two cells. The next generation mask ROM cell layout (at 1 bit/cell) only required one metal-to-active contact per four cells, so it was a lot smaller -- small enough to negate the 2b/cell ROM's supposed area advantages when looking at the entire ROM including decoders & sense amps. And the nextgen 1b/cell ROM was faster too. And it was implant programmed too (rather than active programmed), which is later in the process, which means quicker turnaround for a code change.

Thank you very much! So after I spend another eight months wrapping my brain around this very comprehensive article, I'll be off to Santa Clara to sell my newly acquired skill set to Intel, where I'll likely amaze their staff with my prolific knowledge of their chips.

October 1, 2018 at 8:08 AM

Zom-B said...

Where can we find the die images of the same nmpp resolution as the close-ups, in both pre and post-etching?

October 1, 2018 at 11:46 AM

Unknown said...

This comment has been removed by the author.

October 2, 2018 at 10:57 AM

Unknown said...

Exactly. The 1b ROM may be comparable by area, if drawn carefully, but has much better performance.

October 2, 2018 at 11:00 AM

Cole Johnson said...

NOOOOOO!! My perfect world of 1s and 0s has been ruined! HOW DARE THEY MIX PURE DIGITAL LOGIC WITH ANALOG MIRE!!

Sigh... I guess our world isn't just made of just ONs and OFFs, but of noise, timing, parasitic properties and uncertainty. Its hard to accept for people who've started in programming and digital logic.

I'm wondering how many ROMs were made using this technology, for the people who decode ROMs that can't be obtained any way other than decapping this could make the process more difficult and error prone, but not impossible.

Thanks Anonymous for your knowlege on this subject. I really appreciate the people in the comments section of Ken's blog and CuriousMarc's channel who share their knowlege from old jobs or experiences, much of which is at risk of disappearing as the years go on. Sometimes the discussion is almost as informative as the post.

There's one thing I can add to the discussion, one of the things I've been working on recently is a simulation of the SP0256 speech chip. This varient contains a 2048 cell ROM with normal 1-bit cells. The simulation is here The (7 bit) decoder is above the ROM, the (4 bit) vertical multiplexer is on the right, and the bits leave into a shift register on the left. The full chip does not work correctly yet, due to a(t least one) bug somewhere, but the ROM and the ROM supporting circuitry seems to function as expected

October 2, 2018 at 3:48 PM

CuriousMarc said...

And a comparable 4 level code is now used in the latest high speed I/O outputs of ICs working at 56 Gb/s (and soon at 112 Gb/s) per lane. They use a clock at half the bit rate, but with PAM4 coding which is simply a 4 level coding like in this ROM, to transmit 2 bits per clock cycle. The motivation is to halve the interconnect

recently switched to using the same trick for optical fiber links, to go around the bandwidth limitations of lasers and photodetectors.

October 18, 2018 at 10:26 PM

Richard said...

Impressive!

And better, Conplexas information explained in a simple way.
I was impressed with the area of the processor destined to "Fraction processing " and the "microcode engine "
The ROM microcode was only not higher because of the high density technique.
Impressive analysis, I had never heard of these high density techniques.
Impressive also the implementation of comparators.
Impressive the use of this technique that reduces in 40% of the area destined to microcode.
The microcode contained in the ROM should not follow a specific language. It is a pity to be difficult to try to find out, there is difficulty in checking the transistor sizes by means of a microscope.
Amazing that this high-density microcode technology is back in the new Flash memories. This is very valuable.

Thanks for the article

December 7, 2018 at 3:27 AM

popf said...

Interesting stuff. Thanks!

October 13, 2019 at 4:35 PM

Diomidis Spinellis said...

Both footnote 3 and Peter Ibbotson are partially correct. The way 8087 emulation worked was as follows. The compiler / assembler, when directed to support emulator linking (e.g. with the /FPi ML switch), would generate as code an FWAIT or NOP followed by a 8087 instruction. In addition, it would generate a special linker fixup record. At link time, if an emulation library (e.g. E8087.LIB or EMU87.LIB rather than 8087.LIB or NOEMU87.LIB) was used, the linker would follow the fixup record to replace the FWAIT/NOP instruction with a software interrupt instruction. For example, instruction opcodes 0xD8 to 0xDF would be replaced with interrupts 0x34 to 0x3B. The linker would also replace standalone FWAITs with NOP instructions. This procedure is documented in Chapter 35 of the "PC Interrupts" book by Ralf Brown and Jim Kyle (1991) and page S-61 of Intel's iAPX 86,88 User's Manual (August 1981).

I also seem to remember that on MS-DOS the emulator could be instructed via an environment variable (named 8087 or EMU87) to patch back the original 8087 instructions when it detected the presence of the coprocessor, but I haven't been able to find this documented.

December 4, 2022 at 10:21 AM

Post a Comment

Newer Post Home Older Post