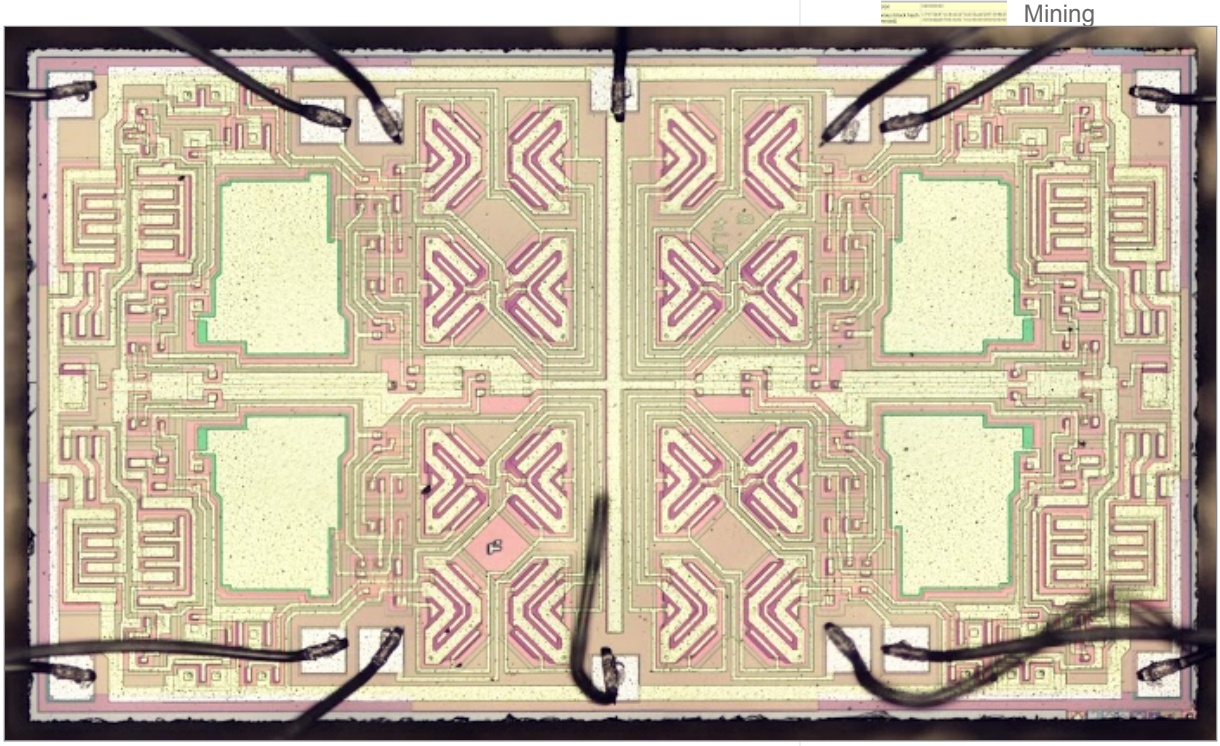
Ken Shirriff's blog

Xerox Alto restoration, IC reverse engineering, chargers, and whatever

Silicon die analysis: inside an op amp with interesting "butterfly" transistors

Some integrated circuits have very interesting dies under a microscope, like the chip below with designs that look kind of like butterflies. These patterns are special JFET input transistors that improved the chip's performance. This chip is a Texas Instruments TL084 quad op amp and the symmetry of the four op amps is visible in the photo. (You can also see four big irregular rectangular regions; these are capacitors to stabilize the op amps.) In this article, I describe these components and the other circuitry in the chip and explain how it works. This article also includes an interactive chip explorer that shows each schematic component on the die and explains what it does.



Die photo of the TL084 quad op amp. The bond wires got a bit bent while cleaning the chip.

An integrated circuit consists of a tiny piece of silicon. To make an integrated circuit, regions are treated with various atoms to change the properties of the silicon, giving them different colors under a microscope. On top of the silicon, a thin layer of metal connects different parts of the chip. This metal is clearly visible in the photo as yellowish traces and regions. Under the metal, a thin, glassy silicon dioxide layer provides insulation between the metal and the silicon, except where contact holes in the silicon dioxide allow the metal to connect to the silicon.

Contact

About Ken Shirriff

Popular Posts

Silicon die analysis: inside an op amp with interesting "butterfly" transistors

Mining

Bitcoins the hard way: Using the raw Bitcoin protocol

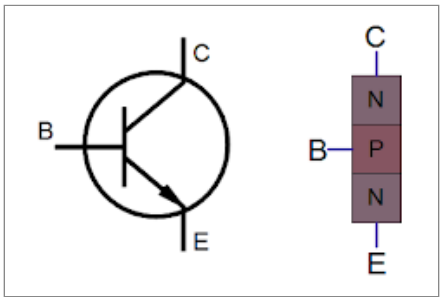
Apple iPhone charger teardown: quality in a tiny expensive package

Xerox Alto zero-day: cracking disk password

Around the edge of the chip, thin bond wires connect the metal pads to the chip's external pins.

NPN transistors inside the IC

Transistors are the key components in a chip. This op amp chip uses several types of transistors: NPN and PNP bipolar transistors as well as JFETs. (Many newer op amps use low-power CMOS transistors instead.) If you've studied electronics, you've probably seen a diagram of an NPN transistor like the one below, showing the collector (C), base (B), and emitter (E) of the transistor, The transistor is illustrated as a sandwich of P silicon in between two layers of N silicon; the N-P-N layers make an NPN transistor. It turns out that transistors on a chip look nothing like this, and the base often isn't even in the middle!



Symbol and simplified structure of an NPN transistor.

The photo below shows one of the transistors in the TL084 as it appears on the chip. The different brown, purple and green colors are regions of silicon that has been doped differently, forming areas called N and P regions (negative with an excess of electrons, and positive lacking electrons). The yellow areas are the metal layer of the chip on top of the silicon—these form the wires connected to the collector, emitter, and base. Underneath the photo is a cross-section drawing showing approximately how the transistor is constructed. There's a lot more going on than just the N-P-N sandwich you see in books, but if you look carefully at the vertical cross section below the 'E', you can find the N-P-N that forms the transistor. The emitter (E) wire is connected to N+ silicon. Below that is a P layer connected to the P+ base contact (B). And below that is an N layer connected (indirectly) to the collector (C).[1]

protection on a 45 year old system

Labels

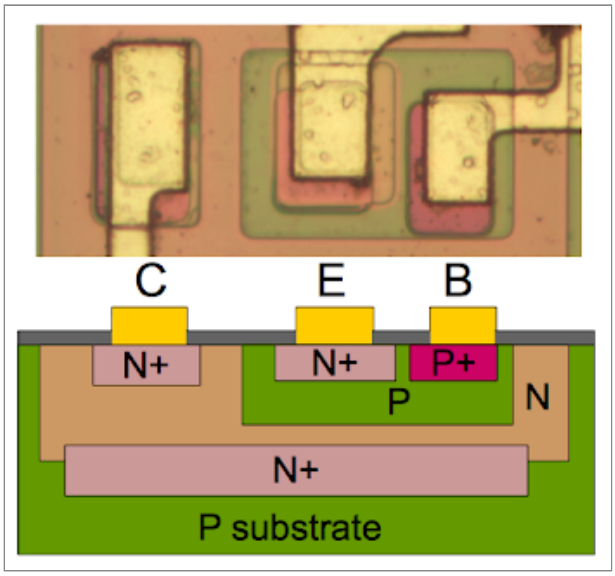
6502 8008 8085 alto apple arc arduino arm beaglebone bitcoin c# calculator css electronics f# fpga fractals genome haskell html5 ibm1401 intel ipv6 ir java javascript math oscilloscope photo power supply random reverse-engineering sheevaplug snark spanish teardown theory unicode Z-80

Email:
kens@arcfn.com

Blog Archive

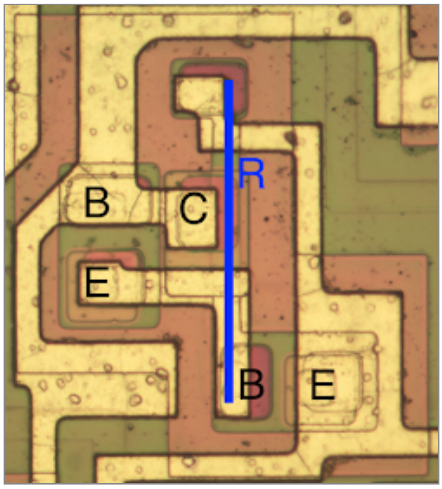
▼ 2018 (11)

▼ June (1)



Structure of an NPN transistor in the TL084 op amp

While most of the transistors follow the above pattern, some of the transistors in the TL084 chip are optimized in confusing ways, such as the part of the die below. In this circuit, two transistors share one collector (C), while a resistor (blue line) runs between them. (This took me a while to figure out, even with the schematic.)



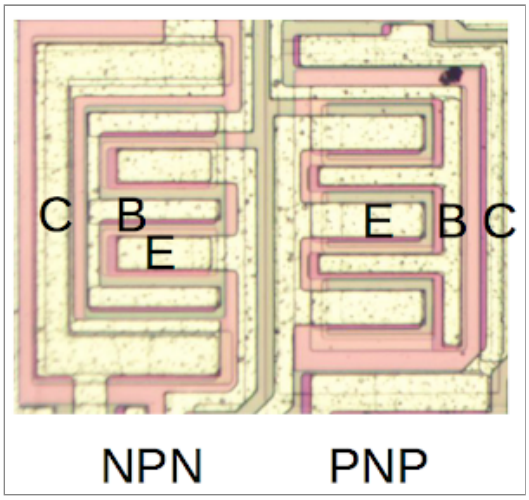
A complex part of the TL084, where two transistors share a collector while a resistor runs through them.

The output transistors (below) in the TL084 are larger than the other transistors and have a different structure in order to produce the chip's high-current output. The output transistors must provide milliamps of current, compared to microamps for the internal transistors. Note the interlocking "fingers" of the emitter (E) and base (B), surrounded by the large collector (C). Although the NPN and PNP transistors look similar, the dark purple P silicon is visible on the base of the NPN transistor and the emitter and collector of the PNP transistor, showing their opposite construction.

Silicon die analysis: inside an op amp with intere...

► 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)



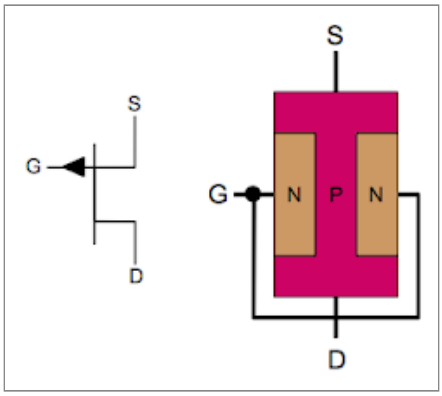
High-current NPN and PNP transistors drive the output of the TL084 op amp

How capacitors are implemented in silicon

The TL084 contains four capacitors to provide stability for the op amps. You can see the four capacitors in the die photo; they are the largest structures on the chip. A capacitor in the chip is essentially a large metal plate separated from the silicon by an insulating layer. The main drawback of capacitors on ICs is they are physically very large. The TL084's capacitors have a very small capacitance value (a few picofarads) but take up a large fraction of the chip's area.[2]

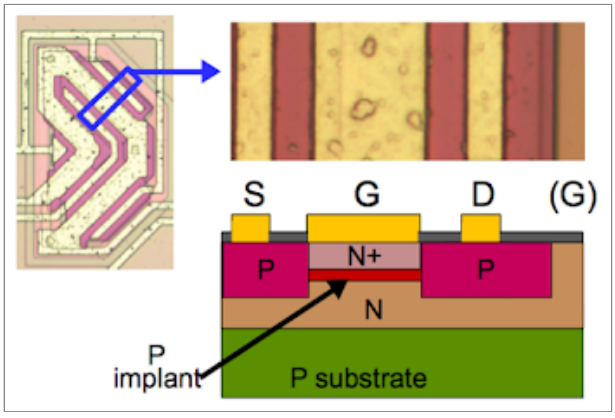
JFET transistors[3]

A special type of transistor called a JFET is the key to the high performance of the TL084 chip. The JFET transistor is related to the more common MOSFET transistor: they both controls current between the *source* and the *drain*, under control of the *gate*. But while the MOSFET has has an insulating oxide layer between the gate and the body of the device, the JFET lacks this layer and has a silicon P-N junction instead (and thus is called a Junction FET). The chip used P-channel JFETS, where current flows through a channel of P silicon; the schematic symbol and basic structure is shown below. Normally, current flows between the source (S) and drain (D) through the channel. As the voltage on the gate increases, it "pinches" the channel closed, reducing and then stopping the current flow. An important feature of a JFET is that very, very little current flows through the gate; the gate resistance is an amazingly large $10^{12}\Omega$. (This is because the gate junction acts as a reverse-biased diode, blocking current flow.) This high input impedance is an important feature for an op amp.



Symbol and simplified structure of a JFET transistor (P-channel).

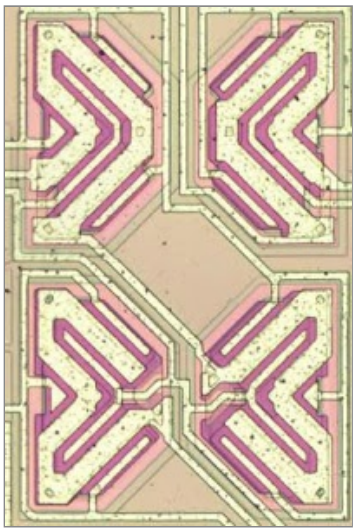
On the chip, the JFETs are constructed like the diagram above but rotated horizontally. The diagram below shows a JFET as it appears on the die (left), along with a close-up slice. (The JFET channel is wide and snakes around in order to pass more current. It also has drains on both sides of the source.) The cross section below shows the internal structure of the JFET. The P region connects the source and the drain, and it is surrounded above and below by the gate's N region. (The connection to the lower N region is outside the region shown.) The JFETs in this chip are built with ion implantation, which shoots accelerated ions into the chip to produce the P and N regions. Ion implantation provides accurate control of the doping and dimensions of the P channel between the source and drain, allowing the input JFETs to be built for high performance.



Cross section of an input JFET transistor, showing the construction of the JFET.

Manufacturing JFET op amp ICs was difficult when they were first sold decades ago. Hybrid (two separate dies in one package) JFET op amps were introduced in 1970. These were followed shortly afterwards by monolithic (i.e. a single die) op amps, but difficulties in manufacturing consistent JFETs caused these op amps to have poor characteristics. In 1974, National Semiconductor engineers developed the ion implantation technique for fabricating consistent, high quality JFETs and used this "BIFET" technique to build better JFET op amps. Two years later Texas Instruments introduced their JFET op amps, including the TL084 which was the first four-in-one op amp using the BIFET process.[4]

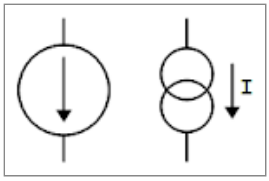
You might have noticed that each op amp has four input JFETs on the die (forming the butterfly pattern below), even though the op amp only has two inputs. The explanation for this is that for good performance the input transistors in an op amps should have identical electrical characteristics. But unfortunately chips can have thermal gradients (i.e. hotter on one side than the other) that affect the transistor characteristics and unbalance the inputs. A standard solution used in the TL084 is that each input uses two cross-coupled transistors, diagonally opposite from each other. If one side of the chip is hotter than the other, both inputs will have an affected transistor, canceling out the effect of the temperature gradient.



To insure the input transistors are matched, each input transistor is actually two connected transistors, diagonally opposite. Wiring connects each transistor pair.

IC component: The current mirror

There are some subcircuits that are very common in analog ICs, but may seem mysterious at first. Before explaining how the TL084 works, I'll first give a brief overview of the current mirror and differential pair circuits.

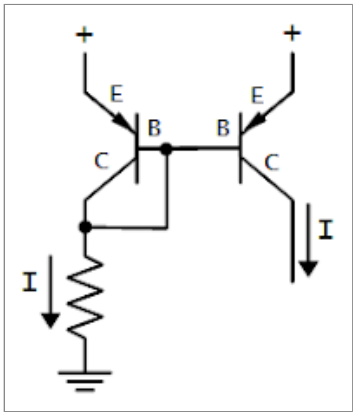


Schematic symbols for a current source.

If you've looked at analog IC block diagrams, you may have seen the above symbols for a current source and wondered what a current source is and why you'd use one. The idea of a current source is you start with one known current and then you can "clone" multiple copies of the current with a simple transistor circuit.

The following circuit shows how a current mirror is implemented.[5] A reference current passes through the transistor on the left. (In this case, the current is set by the resistor.) Since both transistors have the same emitter voltage

and base voltage, they source the same current, so the current on the right matches the reference current on the left.

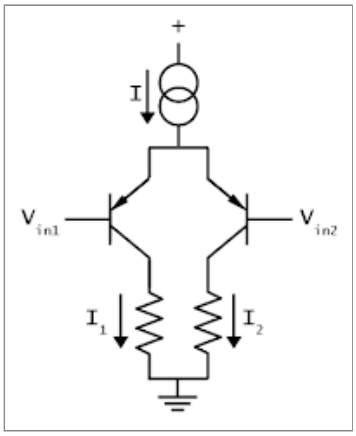


Current mirror circuit. The current on the right copies the current on the left.

A common use of a current mirror is to replace pull-up resistors. Resistors inside ICs are both inconveniently large and inaccurate. It saves space to use a current mirror instead of a resistor whenever possible. Also, the currents produced by a current mirror are nearly identical, unlike the currents produced by two resistors.

IC component: The differential pair

The second important circuit to understand is the differential pair, the most common two-transistor subcircuit used in analog ICs.[6] You may have wondered how the op amp subtracts two voltages since it's not obvious how to make a subtraction circuit. This is the job of the differential pair.



Schematic of a simple differential pair circuit. The current source sends a fixed current I through the differential pair. If the two inputs are equal, the current is split equally.

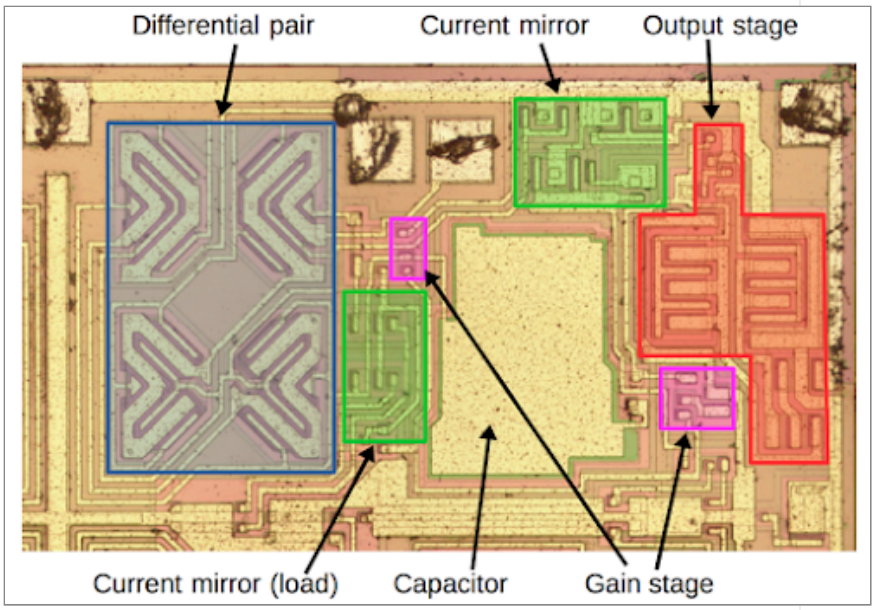
The schematic above shows a simple differential pair. The key is the current source at the top provides a fixed current *I*, which is split between the two input transistors. If the input voltages are equal, the current will be split equally into the two branches (*I1* and *I2*). If one of the input voltages is a bit lower than the other, the corresponding transistor will conduct more current, so one branch gets more current and the other branch gets less. As one input continues to increase, more current gets pulled into that branch. Thus, the differential pair is a surprisingly simple circuit that routes current based on the

difference in input voltages. The TL084 uses JFETS instead of bipolar transistors in the differential pair, but the principle is the same.

The internal blocks of the op amp

The internal circuitry of the TL084 op amp is similar to the 741 op amp, which has been explained in many places[7], so I'll just give a brief description of the main blocks. The interactive chip viewer below provides more explanation.

The two input pins are connected to the differential amplifier, which is based on the differential pair described above. The output from the differential amplifier goes to the second (gain) stage, which provides additional amplification of the signal. Finally, the output stage has large transistors to generate the high-current output, which is fed to the output pin. The capacitor stabilizes the op amp to avoid oscillation. The current mirror at the top provides currents to other parts of the chip. The current mirror at the bottom functions as an active load increasing the gain of the differential pair.

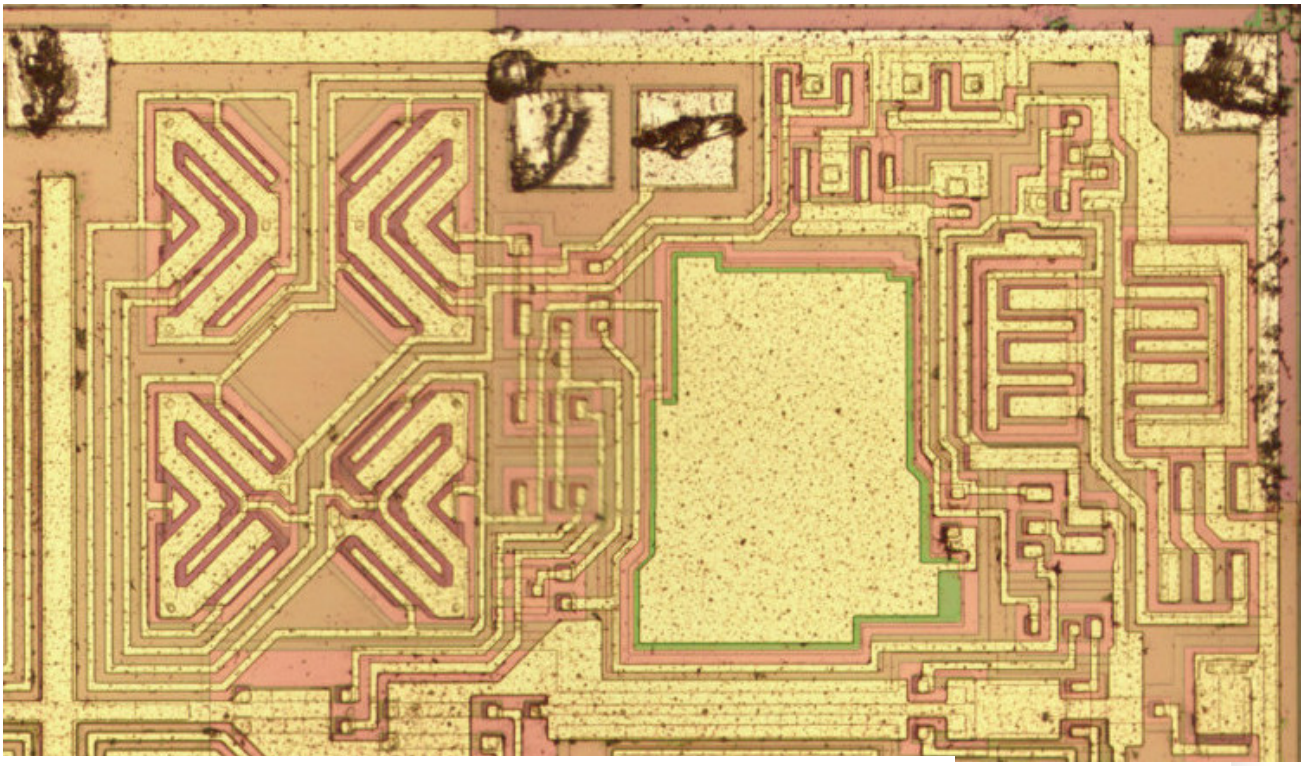


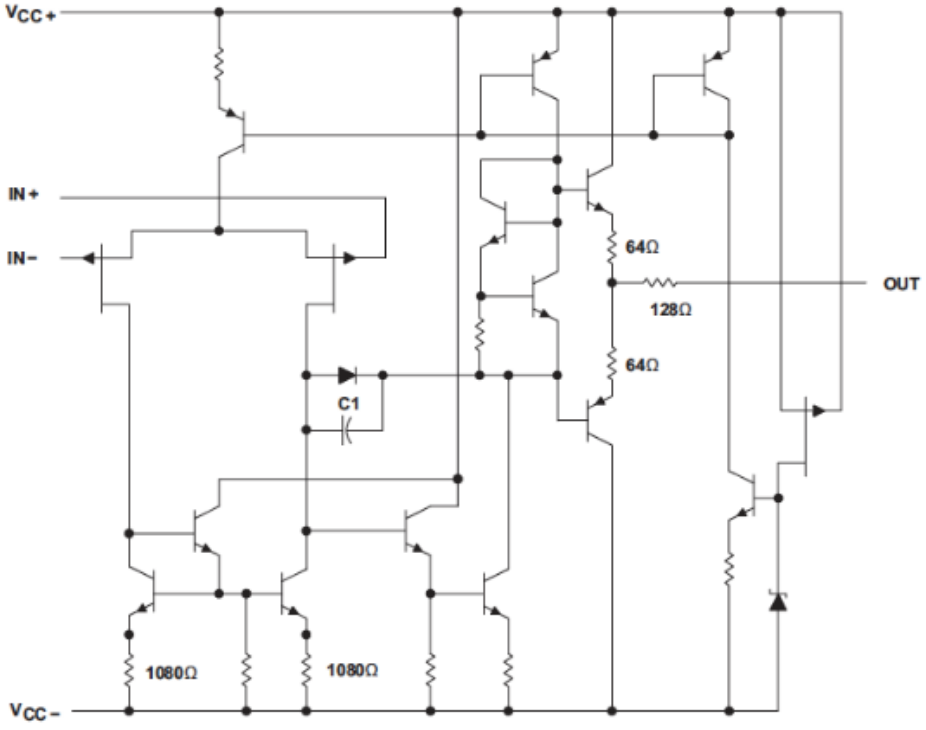
Die for the TL084 op amp, showing the main functional units.

Interactive chip viewer

The image and schematic[8] below are an interactive exploration of the TL084. Click a component to see its location on the die and in the schematic highlighted. The box below will give an explanation of the component. The die image below shows one of the four op amps on the chip; the others are essentially identical.

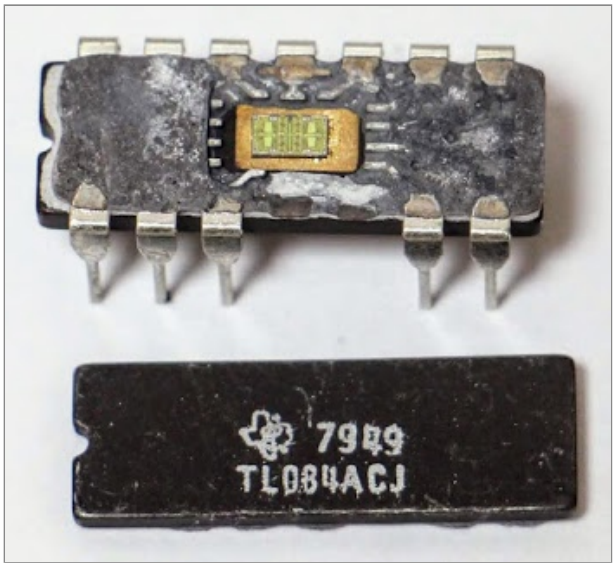
Click components in the image below for more information.





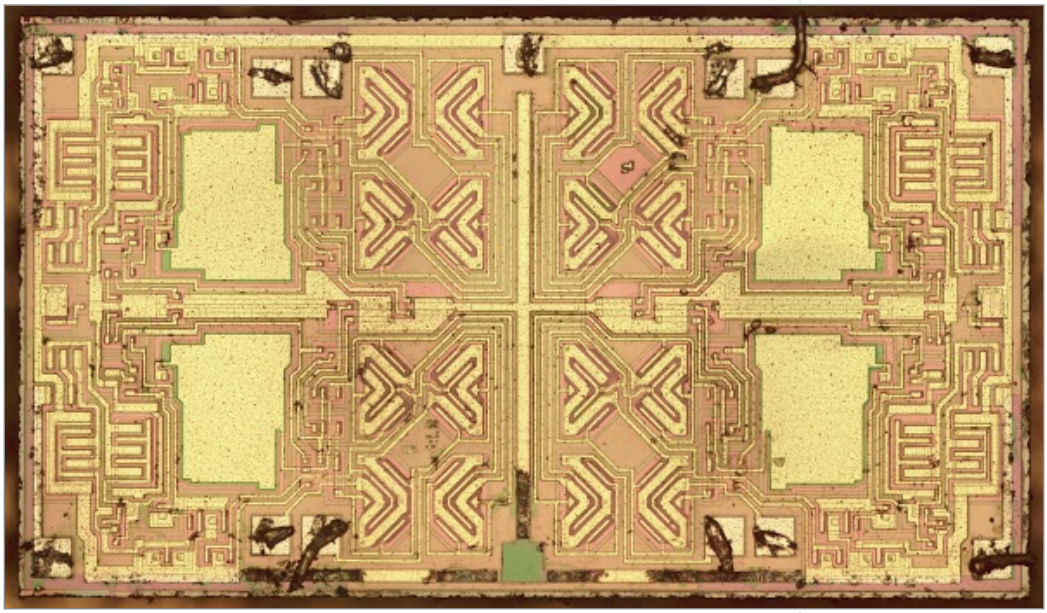
How I photographed the die

Getting to the die of an integrated circuit can be difficult since integrated circuit usually come in a black epoxy package. I'd rather avoid using dangerous concentrated acid to dissolve the epoxy package and see the die. Fortunately some ICs, such as the TL084, are available in ceramic packages that can be easily opened with a chisel. The photo below shows the chip package after removing the lid. The four large capacitors are visible on the die even without a microscope.



The TL084 op amp. The ceramic package has been opened, exposing the die inside. A couple pins fell off when the package was opened.

To obtain the die photos, I used a metallurgical microscope, which shines light from above through the lens, unlike a normal microscope which shines light from below. A metallurgical microscope is the secret to getting clear photos at higher magnification, since the die is brightly illuminated. I used Hugin to stitch multiple images together into high-resolution pictures. Below is a second die photo; the bond wires are removed in this one.



Die photo of the TL084 op amp with the bond wires removed.

Conclusions

Texas Instruments introduced the TL084 in 1976 as one of the first high-performance quad op amps. I was motivated to study this chip by the pretty butterfly-like patterns on the die, but found some interesting circuitry inside the chip. The butterfly-like structures turned out to be JFET transistors that improved the chip's performance by providing high impedance for the op amp inputs. If you enjoyed this look inside an analog silicon chip, you may also like my analysis of the 741 op amp and 555

timer. Follow me on Twitter at @kenshirriff for my latest blog posts, or use my RSS feed. The chip was provided by Eric Schlaepfer (@TubeTimeUS).

Notes and references

1. You might have wondered why there is a distinction between the collector and emitter of a transistor, when the simple picture of a transistor is totally symmetrical. Both connect to an N layer, so why does it matter? As you can see from the die photo, the collector and emitter are very different in a real transistor. In addition to the very large size difference, the silicon doping is different. The result is a transistor will have poor gain if the collector and emitter are swapped. ↩

2. The capacitor in the op amp is located at a special point in the circuit where the effect of the capacitance is amplified due to something called the Miller effect. This allows the capacitor to be much smaller than it would be otherwise. Given how much of the die is used for the capacitor already, taking advantage of the Miller effect is very important. ↩

3. Yes, I realize that "JFET transistor" is a redundant acronym. Since some readers may not be familiar with JFETs, I want to remind them that JFETs are transistors. ↩

4. For an extremely detailed history of op amps, including the development of JFET op amps in the 1970s, see Op Amp History by Walt Jung. My section on JFET op amp history is based on this source. ↩

5. For more information about current mirrors, you can check Wikipedia or chapter 3 of Designing Analog Chips. If you're interested in how analog chips work, I strongly recommend you take a look at Designing Analog Chips. ↩

6. Differential pairs are also called long-tailed pairs. According to Analysis and Design of Analog Integrated Circuits differential pairs are "perhaps the most widely used two-transistor subcircuits in monolithic analog circuits." (p214) For more information about differential pairs, see Wikipedia, any analog IC book, or chapter 4 of Designing Analog Chips. ↩

7. The TL084 op amp's design is similar to the 741 op amp, which is described in Wikipedia, Operational Amplifiers, IC Op-Amps Through the Ages and UNCC class notes. See any of those sources for more details on how op amps are constructed. ↩

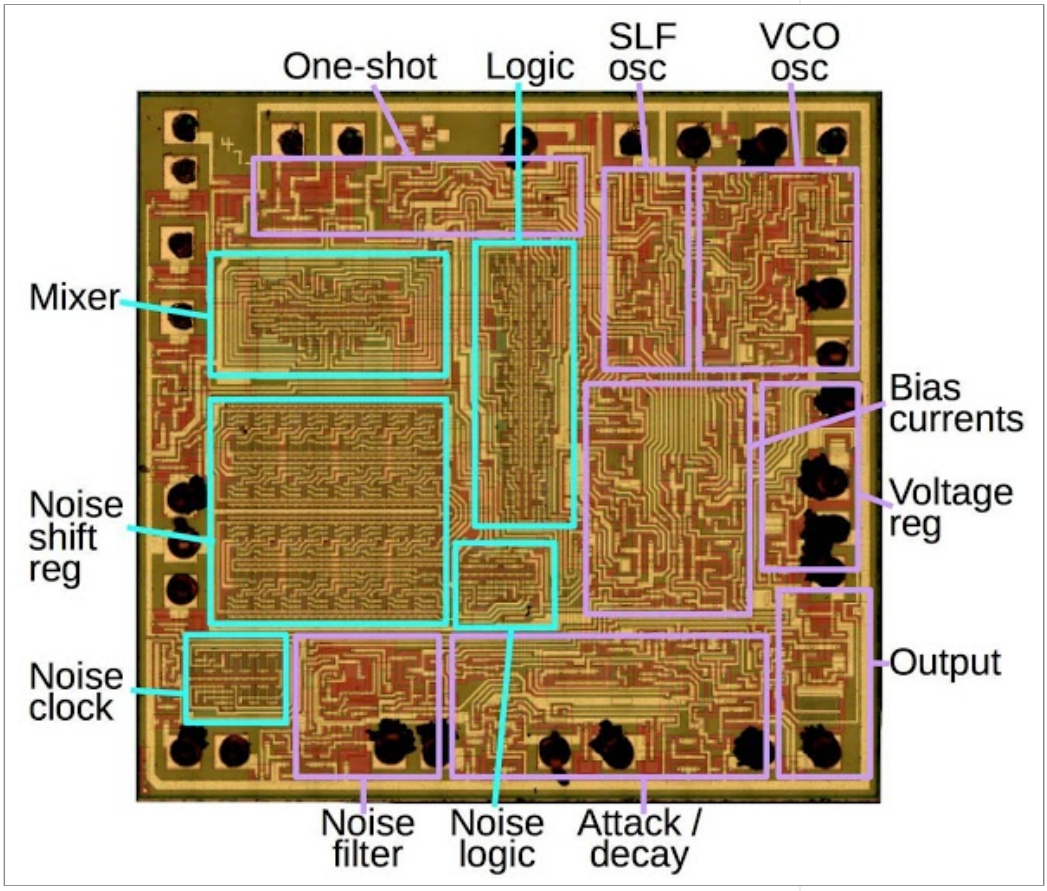
8. The schematic is from the TL084 datasheet. ↩

12 comments:

Labels: electronics, reverse-engineering

Inside the 76477 Space Invaders sound effect chip: digital logic implemented with I2L

The 76477 Complex Sound Generation chip (1978) provided sound effects for Space Invaders[1] and many other video games. It was also a popular hobbyist chip, easy to experiment with and available at Radio Shack. I reverse-engineered the chip from die photos and found some interesting digital circuitry inside. Perhaps the most interesting is a shift register based white noise generator, useful for drums, gunshots, explosions and other similar sound effects. The chip also uses a digital mixer to combine the chip's different sound generators. An unusual feature of the chip is that it uses Integrated Injection Logic (I^2L), a type of digital logic developed in the 1970s with the goal of high-density, high-speed chips. (I wrote about the chip's analog circuitry last year in this article.)

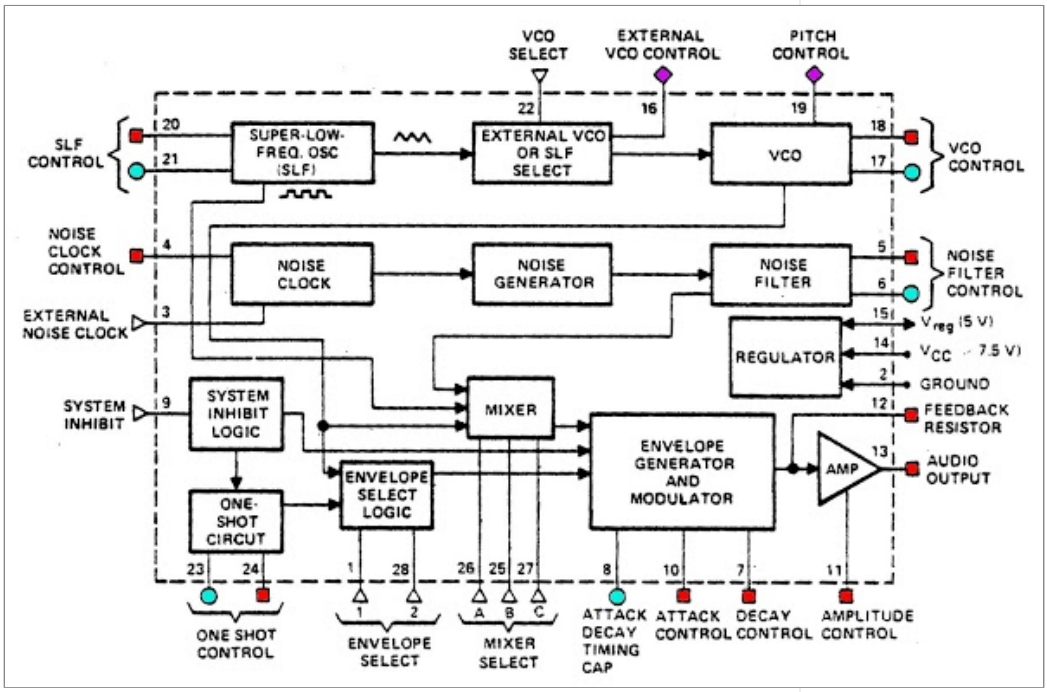


Functionality blocks inside the 76477 sound chip, indicated on the die. Die photo courtesy of Sean Riddle.

Looking under a microscope, you can see the circuitry that makes up the chip. The yellowish lines above are the metal traces that connect the circuits of the die. The reddish and greenish regions are the silicon of the chip, forming transistors and resistors. The black blobs around the edges show where tiny bond wires connected the die to the integrated circuit pins. I've outlined the analog circuits outlined in purple, while digital circuits are in cyan. The 76477 is primarily analog—most control signals are analog, the chip doesn't have digital control

registers, and most sounds are generated from analog circuits —but about a third of the chip's area is digital logic.

The block diagram below shows the 76477 chip's functional elements and can be compared to the die photo above. The voltage-controlled oscillator (VCO) produced a tone whose frequency depends on the control voltage. The "super low frequency" SLF oscillator generated a triangle wave. Feeding this into the VCO generated a varying pitch, useful for bird chirps, sirens, or the warbling sound of the UFO in Space Invaders. The "one-shot" produced a pulse of a fixed length to control the length of the sound. The envelope generator made the sound more realistic by ramping its volume up at the start (attack) and down at the end (decay). The digital white noise generator was used for drums, gunshots, explosions and other similar sound effects. Finally the digital mixer combined these signals and fed them to the output amplifier.

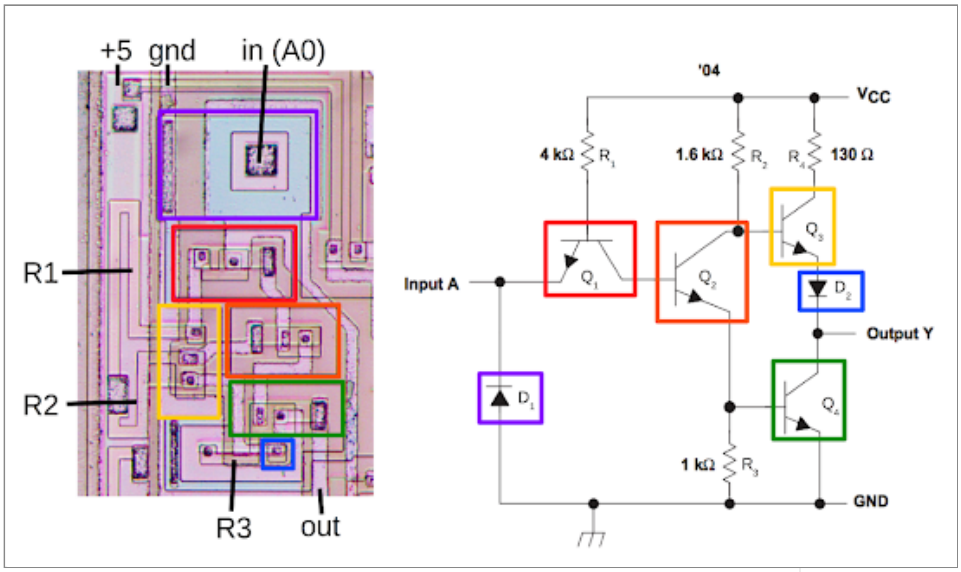


Block diagram of the 76477 sound chip, from the datasheet. Digital inputs: triangles, resistor inputs: red, capacitor inputs: cyan, voltage inputs: violet.

The remainder of this article will dive into how the digital circuitry of the 76477 chip was implemented. First I'll explain Integrated Injection Logic (I^2L), a short-lived logic family that was supposed to revolutionize computer chips. Next, the noise generator, control logic and the digital mixer are reverse engineered and explained.

Integrated Injection Logic

You may be familiar with TTL integrated circuits, such as the popular 7400 family. These chips are built from bipolar transistors—NPN and PNP transistors—and were fast. (Minicomputers were built from boards full of TTL chips, taking advantage of its speed.) Unfortunately, TTL gates required multiple transistors and bulky resistors, so it was difficult to fit a lot of TTL circuitry into an integrated circuit. The die photo below illustrates the complexity of a single TTL inverter. (For details on how it works, see my earlier post.)

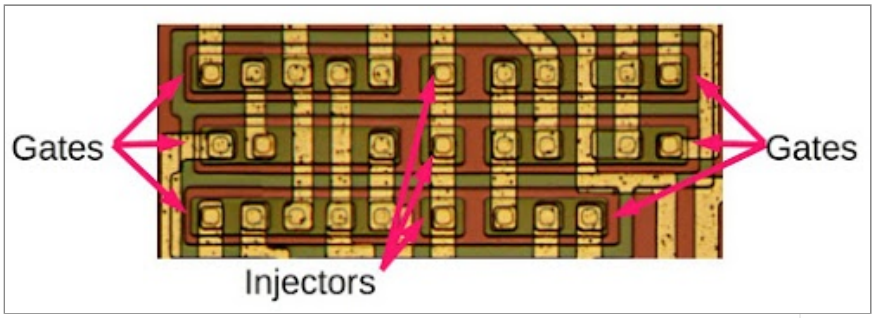


Die photo of a TTL inverter. The inverter uses four transistors, four resistors and two diodes.

A competitor to TTL was MOS; in the early 1970s, integrated circuits based on MOS transistors led to the rise of the microprocessor. Unlike TTL, MOS transistors could be densely crammed onto VLSI integrated circuits, but unfortunately, MOS was much slower than TTL. This posed a dilemma in the 1970s: TTL couldn't implement dense circuitry and MOS was too slow. The integrated circuit industry needed a technology that combined the speed of TTL with the density of MOS, and it looked like I^2L was the solution. (Spoiler: I^2L wasn't the wave of the future; CMOS was.)

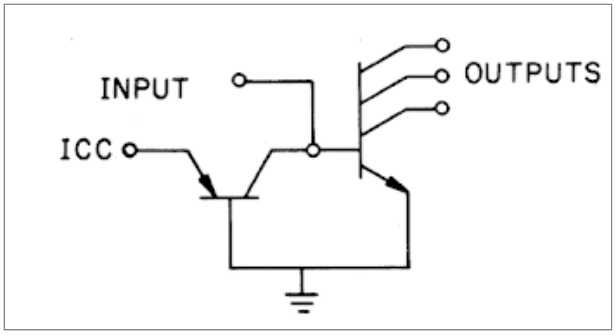
I^2L solved the problems of TTL and MOS with a clever new design. Each I^2L gate was built from a single multi-collector bipolar transistor instead of the multiple transistors of TTL. Even better, the transistors could be arranged in a high-density grid. Finally, the bulky resistors of TTL were replaced by an "injector", a tiny transistor that injected the necessary current. For a final bonus, I^2L was compatible with existing silicon fabrication techniques and could be built on the same chip with bipolar analog circuitry. (This was important for the 76477 sound chip, which combined analog and digital circuits on one chip.) In the late 1970s and early 1980s, I^2L looked like the dream technology that would take over the integrated circuit industry. A variety of I^2L chips were built, including digital watch chips and some microprocessors, as well as the 76477 sound chip.

The diagram below shows six logic gates on the 76477, implemented with I^2L logic. There are two columns of gates, with injectors down the middle. Each gate consists of one transistor (inner green rectangles). Note the high density of the circuitry, without wasted space. Each I^2L gate is implemented with a single transistor, compared to multiple transistors and bulky resistors for a single TTL gate (compare with the TTL photo above). Unlike TTL, which requires considerable wiring inside a gate, I^2L only uses wiring between gates. (Gates are connected by a layer of metal, which appears as thick yellow lines in the die photos.)



Six I^2L gates in the 76477 chip. Each gate is implemented with a single, compact transistor.

I^2L is a bit tricky to understand; it's like being in a crazy backwards world compared to TTL. A normal logic gate (such as a NAND gate) has a few inputs and one output. But an I^2L gate has one input and multiple outputs! How can that work? The schematic below shows an I^2L gate, with one input and three outputs. Normally the current from the injector (ICC) turns on the output transistor, pulling the output low. But if the input is low, the output transistor turns off and the output will be high.[2] Thus, the gate inverts the input. (You can think of the injector as a pull-up resistor on the input.)



Implementation of a I^2L gate. Note that it has a single input and multiple outputs. Icc is the injected current. From "Integrated Injection Logic: A Bipolar LSI Technique".

Since the circuit above has a single input, it may seem to be just an inverter. But by wiring several signals together at the input, you get an AND gate "for free": if any signal is low, it will pull the wire low, and otherwise the signal is high. This is called "wired-AND". The wired-AND input to the I^2L inverter results in a NAND gate.

One problem arises with wired-AND: if you connect an output to more than one wired-AND, everything gets shorted together. The solution is to have multiple outputs from the inverter. Thus, each I^2L NAND gate has a single input and multiple identical outputs. The outputs from various gates (A and B below) are connected together and fed to the input of a I^2L gate, creating a NAND gate.

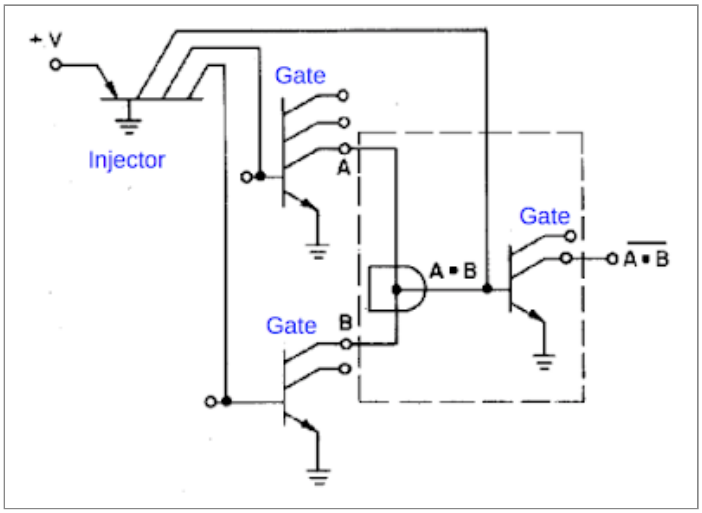
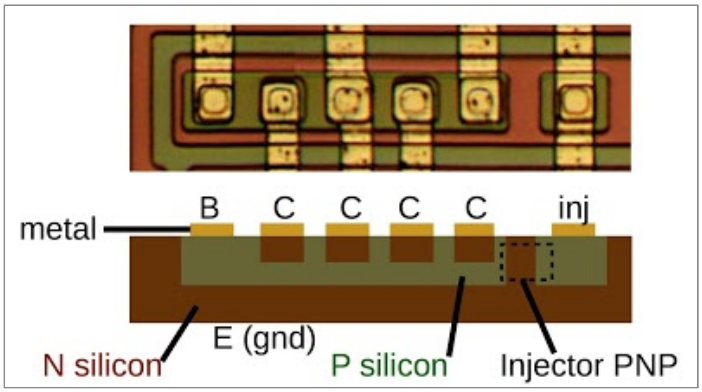


Diagram of a NAND gate implemented in Integrated Injection Logic (I^2L). From "Integrated Injection Logic: A Bipolar LSI Technique".

Compared to TTL, I^2L is also constructed "backwards". The transistors in I^2L have multiple collectors, while the transistors in TTL have multiple emitters.[3] It may seem strange to think of transistors with multiple collectors, but the diagram below shows how they are constructed. Each collector has an N region (brown) with a P region (green) below for the base, and another N region at the bottom, forming an NPN transistor. The multiple collector is built by creating multiple N regions. Note that the transistor's emitter is the grounded substrate. Also note that the injector PNP transistor is just a P region, reusing the emitter and base's N and P regions; this make the injector more compact than a "full" transistor.



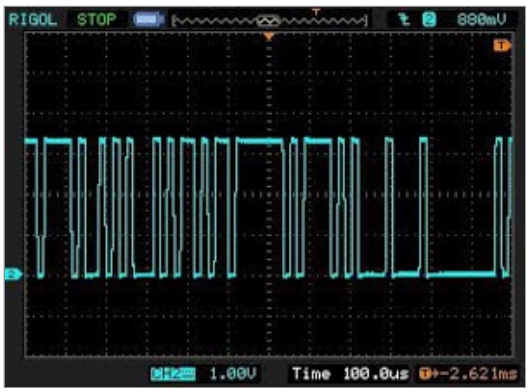
Die photo and cross section diagram of an I^2L gate in the 76477 sound effects chip. The transistor base, collectors, and emitters are labeled along with the current injection.

Noise generator

The 76477 generates sounds such as a gunshot, explosion, steam train, or drum by producing a burst of white noise, a hissing staticky sound. Although white noise is relatively difficult to generate with an analog circuit, it can be simulated by a digital shift register circuit.[5] Linear feedback shift registers are a well-known technique, generating each new bit by combining some of the existing bits with an exclusive-or operation. With a careful design, a LFSR with an n-bit shift register can output 2^n-1 pseudorandom bits before repeating. The 76477, however, uses a nonlinear feedback shift register,

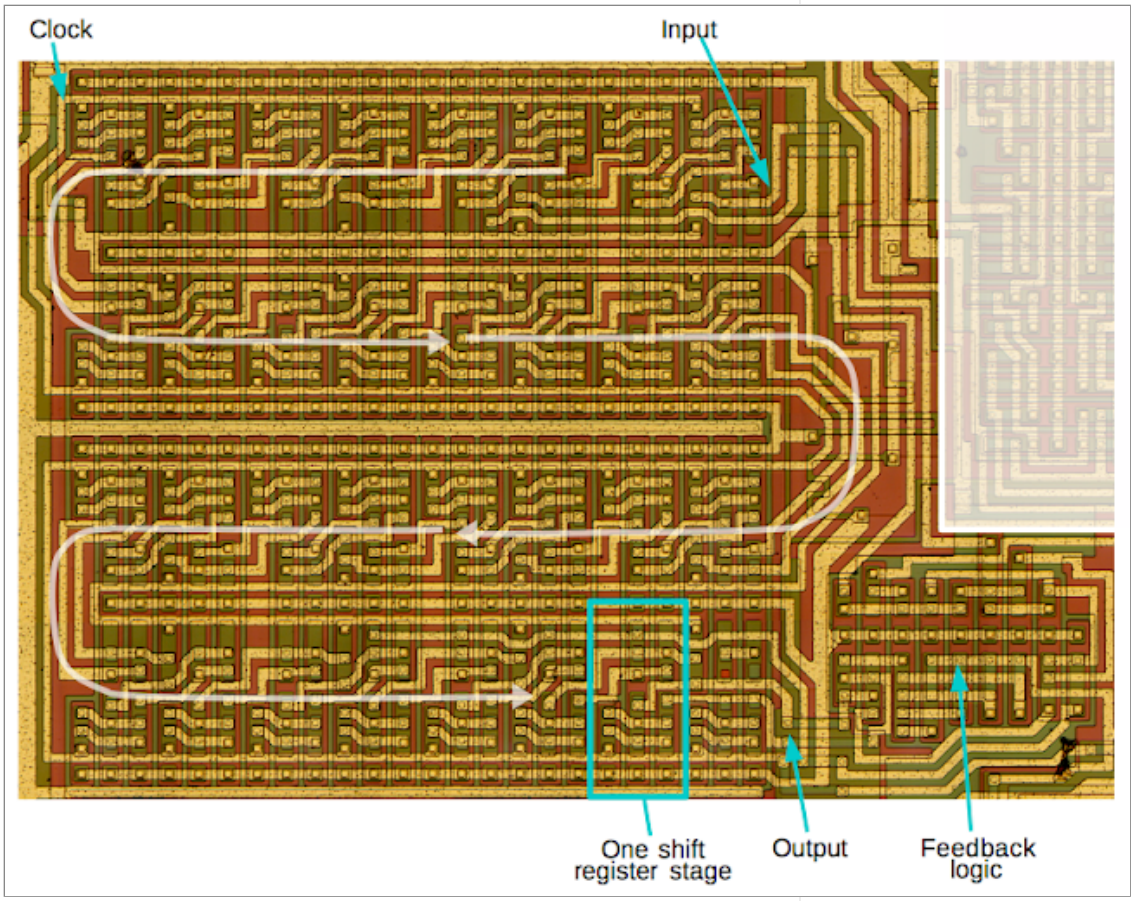
a less-known technique that uses a different function to generate new bits.

The output from the noise shift register is a pseudorandom sequence of 0's and 1's, output at the shift register's clock frequency. The oscilloscope trace below shows the output sequence.



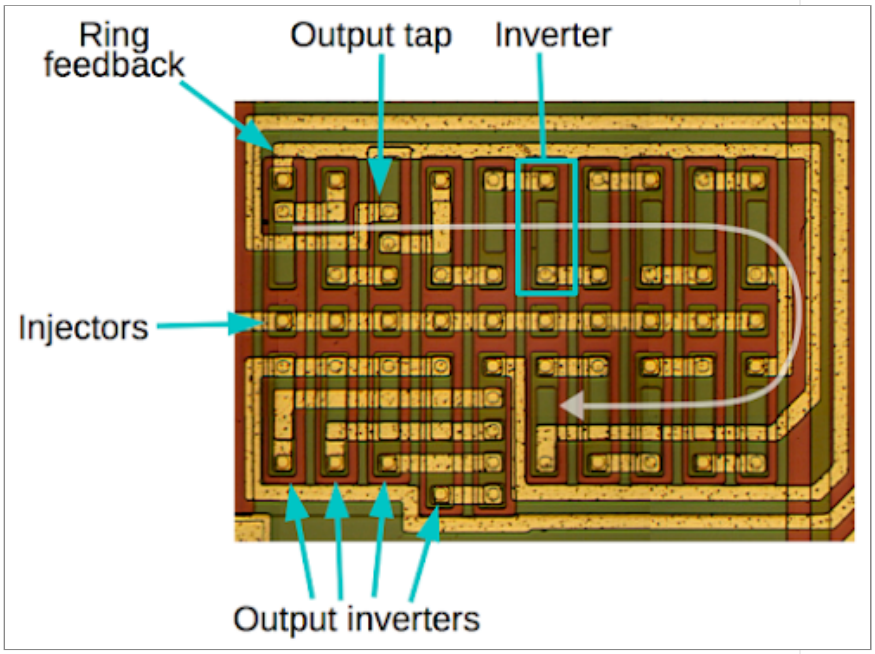
Random noise output form 76477 sound chip.

The shift register consists of 32 stages, each built from a two-latch flip flop.[8] Looking at the die, we can see the shift register and determine the function that generates new bits. Bits move through the shift register as indicated by the white arrow. The feedback logic generates each new input bit from the current bits.[7] The output is a pseudo-random bit that repeats after 56883 cycles.[6]



Die photo of the shift register white noise generator in the 76477 sound effects chip.

The noise circuitry is driven by a clock signal, either external or internal. The internal clock is produced by a ring oscillator consisting of 15 inverters wired in a loop. Because the number of inverters is odd, the input signal will be inverted after it goes through the loop and reaches the input. Thus, the circuit will continuously oscillate.

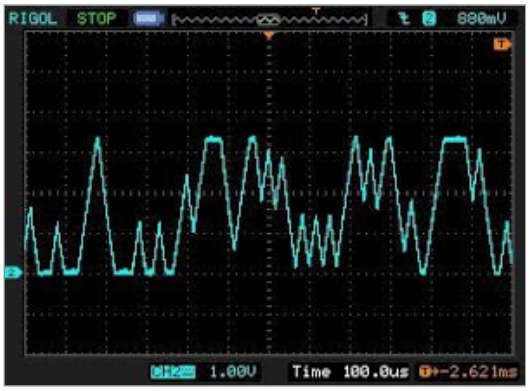


The noise shift register is driven by a clock generated from fifteen inverters forming a ring oscillator.

The die photo above shows the structure of the ring oscillator. Each inverter is connected to the next, as indicated by the white arrow. The last inverter is connected to the first through the "ring feedback" wire. The output from the ring oscillator is a bit unusual, due to the single-input multi-output structure of I^2L. The output is tapped from the third inverter, which has a second output. It is connected to a 4-output inverter, which drives the four output inverters in parallel. This produces an output with four times the regular current, sufficient to drive the shift register.

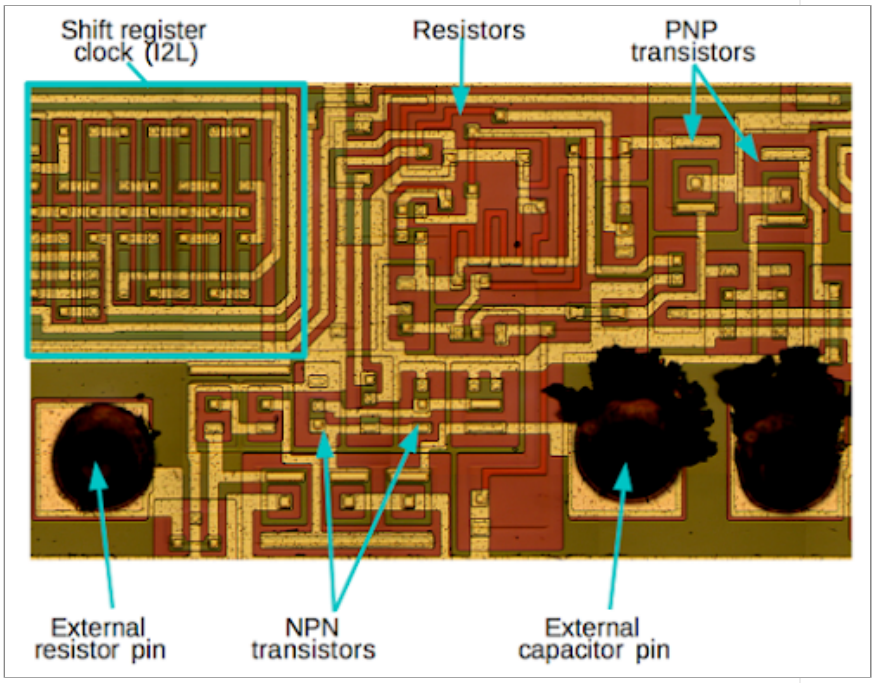
The noise filter

The sound of the noise can be tuned for different effects such as a high-pitched gunshot or a lower-pitched steam engine. This filtering is done by the noise filter, an adjustable low-pass filter that processes the pseudo-random white noise produced by the shift register. While this circuit is analog, I'll explain it here since it's part of the noise circuitry. The filter is basically an integrator, controlled by an external resistor and capacitor. In other words, the filter converts the sharp pulses from the shift register into ramps up or down, for inputs of 1 or 0 respectively, yielding the waveform below. (A slower ramp up and down yields an output signal that changes more slowly and is biased towards low frequencies, filtering out more high frequencies.) This signal is then converted back into a digital signal, which is used as the noise signal by the rest of the chip.



Random noise output form 76477 sound chip after filtering and before conversion back to digital pulses.

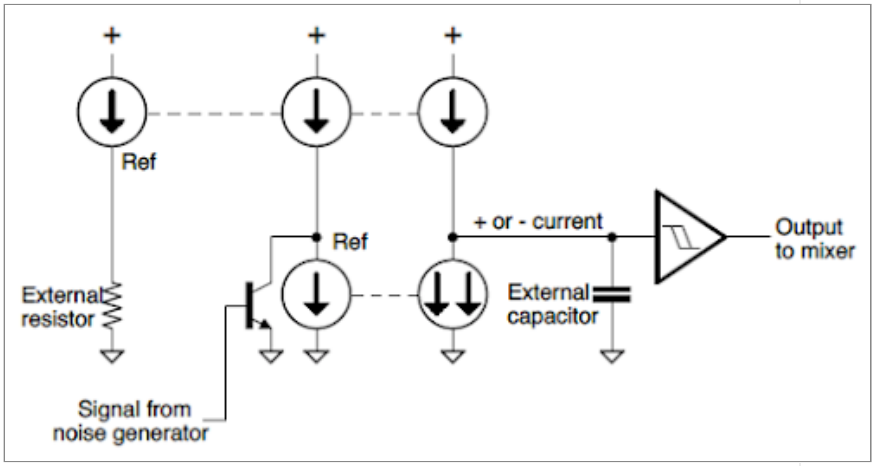
On the die, the noise filter is made of NPN and PNP transistors, along with some resistors (long red squiggles). You can see that analog circuits are not as dense as the I^2L transistors. The black blobs are where bond wires connect the die to the IC pins.



The noise filter circuitry is next to the shift register on the die. It is an analog circuit, built from NPN and PNP transistors.

While the noise filter is controlled by an external resistor and capacitor, it is built very differently from a typical R-C filter. The schematic (below) shows that the noise filter is built largely from current mirrors. (A current mirror is a controllable current sources built from a few transistors, shown as an arrow in a circle on the schematic.) The external resistor sets the current through the current mirror reference. Under the control of the shift register output, the double current mirror (two arrows) will either sink twice the resistor current or nothing. Summing the two currents yields either a charging current or a discharging current for the capacitor, equal to the resistor current. The result is the external capacitor will be charged or discharged with a steady current, with the rate controlled by the external resistor. Finally, the capacitor voltage is converted to a clean digital output by a Schmitt trigger, and fed to the mixer. The

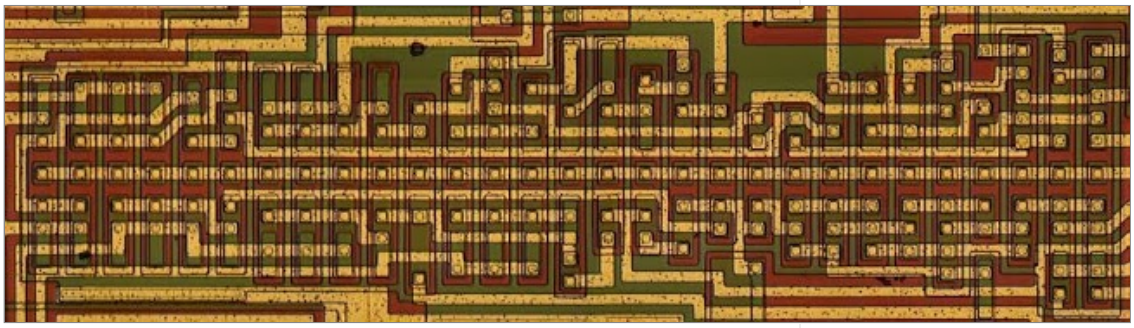
chip uses this current mirror "trick" in multiple places and I've written about it here if the description here is too brief.



Schematic of the noise filter. This low-pass filter integrates the input signal, using multiple current mirrors (arrow symbol). The frequency response is controlled by the external resistor and capacitor.

Miscellaneous logic

There's a block of I^2L circuitry that implements miscellaneous digital logic to control the chip. It implements the envelope select logic that generates the attack and decay signals that shape the output sound.[11] This logic block also processes the inhibit and reset inputs, used to produce bursts of sound. I won't go into the details of this logic; it's basically NAND gates, inverter buffers, and a T flip flop built from NAND gates. The die photo below shows the efficient, dense packing of I2L logic; each column contains two gates.



Miscellaneous logic circuits in the 76477 sound effects chip.

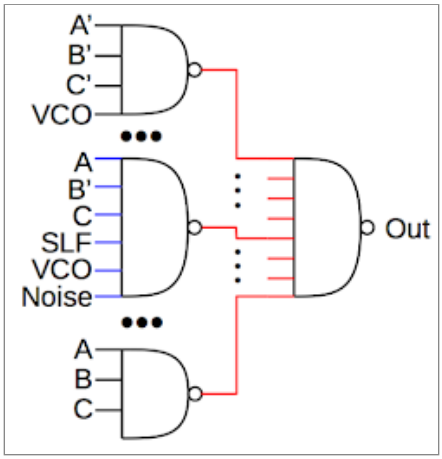
The mixer

The 76477 includes a mixer that allows any combination of the three sound sources (Voltage-Controlled Oscillator, Super-Low Frequency oscillator, and noise) to be combined to form the output. The mixer isn't an analog mixer, but just ANDs together the digital inputs. Unfortunately this makes the mixer less useful since inputs aren't combined as audio sounds, but essentially gate each other.

MIXER SELECT INPUTS			MIXER OUTPUT
C (PIN 27)	B (PIN 25)	A (PIN 26)	
L	L	L	VCO
L	L	H	SLF
L	H	L	NOISE
L	H	H	VCO/NOISE
H	L	L	SLF/NOISE
H	L	H	SLF/VCO/NOISE
H	H	L	SLF/VCO
H	H	H	INHIBIT

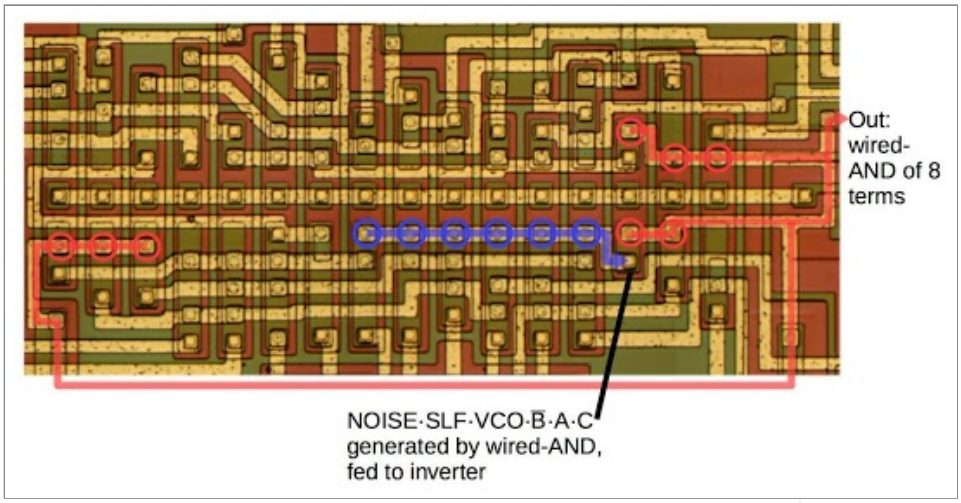
The three mixer input pins select which sound sources are combined to form the output.

The mixer is implemented by an 8-input multiplexer, consisting of eight NAND gates feeding an output NAND gate (below). [13] Each gate is enabled by one of the eight mixer select combinations (above), and passes the corresponding sound signals. Finally the 8-input NAND gate merges the eight branches to produce the output. [12] For instance if A, B and C are low (selecting VCO), the top NAND gate is active, passing the VCO signal to the output. If A is high, B is low and C is high, the SLF, VCO and Noise signals are ANDed and passed to the output. If A, B and C are all high (Inhibit), the bottom gate pulls the output high. The logic is essentially a direct implementation of the table above.



Schematic of the mixer, showing three of the eight multiplexer gates.

The die photo below shows the mixer. The black line indicates the middle NAND gate, which on the die has a single input and a single output. The key point is that with I^2L, a 6-input NAND gate is implemented by wiring together the 6 desired signals to a single gate input. This wire is indicated in blue with the 6 desired input signals marked with blue circles. Likewise, the eight NAND gate outputs (red circles) are wired together (red line) for the output. Interpreting an I^2L circuit is confusing because of the conceptual reversal of inputs and outputs.



Die photo showing the digital mixer in the 76477 sound effects chip.

Conclusions

In the late 1970s, I^2L was heralded as the technology of the future, combining the speed of TTL integrated circuits with the density of MOS.[9] I^2L reached its peak with the production of 16-bit microprocessors by Fairchild and Texas Instruments in the 1970s and 1980s.[14] Fairchild's I^2L Microflame processors lived up to their name, running so hot that some chips were packaged on beryllium oxide, a toxic ceramic that conducts heat better than most metals. Unfortunately for I^2L, CMOS turned out to be the winning technology. CMOS's extremely low power consumption and scalability allowed CMOS chips to hold exponentially more circuitry (as described by Moore's Law) making modern microprocessors possible. Thus, the processor you're using now is built from CMOS and I^2L is a historical footnote.

The 76477 sound chip is an unusual combination of analog and digital circuitry, using I^2L for the digital logic. Since the sound chip was primarily controlled by resistors, capacitors and voltages, it was difficult to control with a microprocessor. As a result, the 76477 sound chip was soon overshadowed by digital sound chips, such as the AY-3-8910 and the 76489 that could easily be interfaced to a microprocessor.[15] Nevertheless, the 76477 chip was popular with hobbyists as it was easy to experiment with and readily available at Radio Shack. Although long obsolete, the 76477 still lives on in the occasional retro project and (inexplicably) an iPhone app.

The 76477 sound chip was popular with hobbyists and sold at Radio Shack. Photo courtesy of Bill Lewis.

I announce my latest blog posts on Twitter, so follow me at kenshirriff. I also have an RSS feed. Thanks to Sean Riddle for the die photos.

Notes and references

1. The Space Invaders schematics show that the video game used seven different circuits to create its different sounds. The 76477 generated the "UFO" sound, while other sounds (saucer hit, explosion, missile, invader hit, etc.) were mostly generated by collections of op amps. ↩

2. In the I^2L schematic as shown, the output will be floating when not pulled to ground. But in use, the output will be connected to another gate, and its injector current will pull the output high. ↩

3. If you're familiar with TTL circuitry, you may know that it uses transistors with multiple emitters. The multiple-collector transistors used by I^2L are constructed in essentially the same way on die, except the role of the emitter and the collector are swapped. ↩

4. When looking at the 76477 die, a collector and a base are almost identical, which I didn't expect. Fortunately, there is a subtle difference that lets you distinguish them: both have a circle inside a square, but for some reason the base's circle touches the square while the collector's circle is centered. ↩

5. Because the output of the shift register is pulses at a fixed clock frequency, the noise isn't "genuine" white noise, which has a flat frequency spectrum and is more random. However, the shift register output is a reasonable approximation below the clock frequency. Some discussion is here. ↩

6. The pseudo-random noise output from the shift register repeats after 56883 cycles. This is much worse than you could get from a 32-bit LFSR (a cycle of length 2^32-1), so they could have used a much smaller shift register. Perhaps the nonlinear terms help initialize the shift register; a linear-feedback shift register can get stuck in the all-zeros state. ↩

7. The nonlinear feedback function is `NOT ((r2 XOR r30) OR (r2 AND r26 AND r27 AND r28 AND r29 AND r30))`. I verified that the chip's output matches this formula. Interestingly, the Mame emulator's code for the 76477 uses a similar, but not identical noise algorithm. ↩

8. One puzzling feature of the shift register is that there is no wiring between the stages! How do bits get from one stage to the next? Did the chip have another layer of wiring that wasn't in the photos? Was there some sort of hidden connection? Eventually I noticed that there wasn't an isolation ring between the stages—a silicon barrier that separated most I^2L circuits. Without this isolation ring, an "invisible" PNP transistor exists between the stages, apparently allowing one stage to flip the next stage to the right value. Each shift-register stage is constructed from two NAND-gate latches. When the clock is low, the first latch is forced into an indeterminate state. When the clock goes high, the latch ends up as 0 or 1 based on the bias it receives from the previous stage through the "invisible" PNP transistors. Thus, the latch becomes edge-sensitive since it will change right on the clock's rising edge. I found a paper ("Injection-Coupled Synchronous Logic", 1978) that describes a similar technique for an I^2L shift register[9], so I think this is the right explanation, even though the circuit seems a bit sketchy. ↩

9. See the 1976 article "Integrated Injection Logic: A Bipolar LSI Technique"for an overview of I^2L (pdf). The most thorough reference I've found on I^2L is the book Integrated Injection Logic (1980), a collection of dozens of articles on I^2L. ↩

10. The die has a multi-transistor circuit connecting the env1 pin to the shift register. It looks like pulling the env1 input above 5 volts should reset the shift register. Perhaps this is an undocumented feature for testing. However, I couldn't make this work on an actual chip, so it's a bit mysterious. ↩

11. Strangely, the envelope control logic has separate logic to generate the attack and decay signals, even though they are simply complements of each other. I wonder if the designers originally planned a more complex envelope, perhaps attack, sustain, and decay. ↩

12. The mixer implementation can be viewed as AND gates feeding an OR gate (rather than NAND gates feeding a NAND gate). This follows from De Morgan's Law. ↩

13. The mixer implementation is more complex than it needed to be for no apparent reason. An easier approach would be to have each mixer select input control one of the three signals with a simple NAND gate. Instead the mixer options are ordered apparently randomly, requiring the complex multiplexer. ↩

14. I'll give a brief summary of I^2L microprocessors. Fairchild's Microflame architecture series started with the 16-bit F9440 processor in 1977. (The Microflame processors used Fairchild's "Isoplanar Integrated Injection Logic" technology (I3L); presumably this was one better than regular I^2L.) The Fairchild F9445 microprocessor (1979) implemented the architecture of the Data General Nova minicomputer. In 1980 the US Air Force came up with a standard 16-bit processor architecture called MIL-STD-1750A and multiple companies built microprocessors meeting this standard. Fairchild's I^2L entry in this market was the F9450. Meanwhile, Texas Instruments produced the 4-bit SBP0400 bit-slice processor (1975) with 1660 gates. This was followed by I^2L processors in its 16-bit 9900 family: the SBP9900 with 6034 gates (1979) and improved SBP9989 with 5000 gates (1981). I^2L processors were used in the niche market of radiation-resistant processors for space applications. For example, the TI SBR9000 (1985), a successor to the 9900. One radiation-resistant I^2L microprocessor was the TI SBR9000 ↩

15. The 76477 has a few design decisions that don't make sense to me. One is the complex, suboptimal shift register configuration for the noise generator. Another is the digital mixer configuration that makes the mixer circuit unnecessarily complex. Finally, the envelope control logic seems like it was designed for more complex envelopes than than the chip actually implemented. I don't like to criticize chip designs, figuring the designers must have known what they were doing, but I have to wonder about the 76477. ↩

4 comments:

Labels: electronics, reverse-engineering

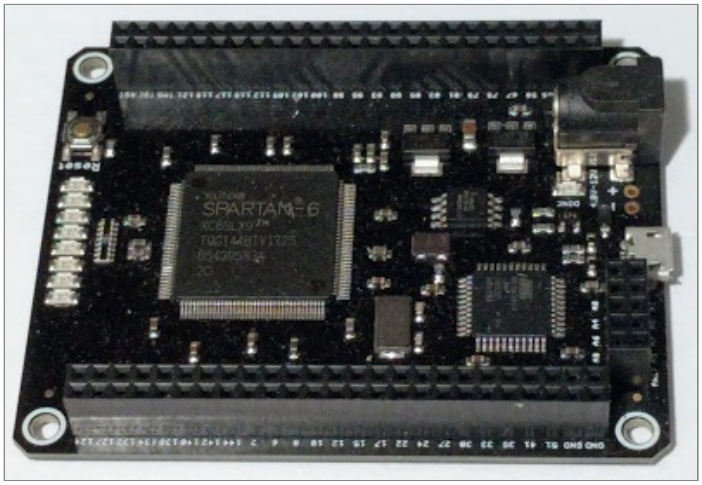
Using an FPGA to generate raw VGA video:FizzBuzz with animation

This blog post shows how you can generate a video signal with an FPGA, using the FizzBuzz problem as an example. Creating video with an FPGA was easier than I expected, simpler than my previous serial-line FizzBuzz on an FPGA. I got a bit carried away with the project and added animation, rainbow text and giant bouncing words to the display.

FizzBuzz from an FPGA board. The board generates raw VGA video output with the results animated, along with the words "Fizz" and "Buzz" that bounce around the screen.

If you're not familiar with the "FizzBuzz test", the problem is to write a program that prints the numbers from 1 to 100, except multiples of 3 are replaced with the word "Fizz", multiples of 5 with "Buzz" and multiples of both with "FizzBuzz". Since FizzBuzz can be implemented in a few lines of code, it can be used as an interview question to weed out people who can't program at all. But it's much more of a challenge on an FPGA.

An FPGA (Field-Programmable Gate Array) is an interesting chip that you can program to implement arbitrary digital logic. This lets you build a complex digital circuit without wiring up individual gates and flip flops. It's like having a custom chip that can be anything from a logic analyzer to a microprocessor to a video generator. For this project, I used the Mojo FPGA board (below).



The Mojo FPGA board. The Spartan-6 FPGA chip dominates the board.

Generating the VGA signals

There's a learning curve to an FPGA, since you're designing circuits, not writing software that runs on a processor. But if you can blink five LEDs with an FPGA, you're most of the way to creating a VGA video signal. The VGA video format is a lot simpler than I expected: just three signals for the pixels (red,

green and blue), and two signals for horizontal sync and vertical sync.

The basic idea is to use two counters: one to count pixels horizontally and one to count lines vertically. At each spot on the screen, the desired pixel color is generated from these coordinates. In addition, the horizontal and vertical sync signals are produced when the counters are at the right positions. I used the basic 640x480 VGA screen resolution[2] which requires counting to 800 and 525.[1 11] Horizontally, there are 800 pixels for each line: 640 visible image pixels, followed by 16 blank pixels, 96 pixels of horizontal sync and 48 more blank pixels. (There are historical reasons for these strange numbers.) Meanwhile, the vertical counter must count out 525 lines: 480 image lines, 10 blank lines, 2 lines of vertical sync and 33 more blank lines.

Putting this all together, I created a `vga` module (source) to generate the VGA signals. This code is in Verilog (a standard language for FPGAs); I won't explain Verilog thoroughly, but hopefully enough to show how it works. The code below implements the `x` and `y` counters. The first line indicates action is taken on the positive edge of each (50 MHz) clock signal. The next line toggles `clk25` each clock, creating the 25 MHz signal we'll use for the pixel clock. (One confusing thing is that `<=` indicates assignment, not comparison.) The code increments the `x` counter from 0 to 799. At the end of each line, `y` is incremented, running from 0 to 524. Thus, this code generates the necessary pixel and line counters.

```
always @(posedge clk) begin
  clk25 <= ~clk25;
  if (clk25 == 1) begin
    if (x < 799) begin
      x <= x + 1;
    end else begin
      x <= 0;
      if (y < 524) begin
    y <= y + 1;
      end else begin
    y <= 0;
      end
    end
  end
end
```

While Verilog code looks like a standard programming language, its effects are very different. This code doesn't generate instructions that are executed sequentially by a processor, but instead causes circuitry to be instantiated in the FPGA chip. It creates registers from flip flops for `clk25`, `x` and `y`. Binary adders are generated to increment `x` and `y`. The `if` statements turn into logic gate comparators controlling the registers. All this circuitry runs in parallel, triggered by the 50 MHz clock. To understand FPGAs, you need to get out of the sequential program mindset and think of the underlying circuits.

Getting back to the vga module, the horizontal and vertical sync signals are generated from the x and y counters by the code below. In addition, a `valid` flag indicates the 640x480 region where a video signal should be generated; the screen must be blank outside this region. As before, these Verilog statements are generating logic gates to test the conditions, not creating code.

```
assign hsync = x < (640 + 16) || x >= (640 + 16 + 96);
assign vsync = y < (480 + 10) || y >= (480 + 10 + 2);
assign valid = (x < 640) && (y < 480);
```

The "useful" part of the VGA signal is the red, green, and blue pixel signals that control what appears on the screen. To test everything out, I wrote a few lines to turn `r`, `g` and `b` on for various regions of the screen, blanking them all outside the visible (`valid`) area.[3] (The question mark is the ternary conditional operator, as in Java)

```
if (valid) begin
  rval = (x < 120 || x > 320) ? 1 : 0;
  gval = (y < 240 || y > 360) ? 1 : 0;
  bval = (x > 500 && (y < 120 || y > 300)) ? 1 : 0;
end else begin
  rval = 0;
  gval = 0;
  bval = 0;
end
```

I ran the code on my FPGA board and nothing happened—the monitor stayed black and I wondered what went wrong. Fortunately, after a couple seconds[4] the monitor completed its normal startup cycle and displayed the output. I was pleasantly surprised that VGA output worked the first time, even if the output was just arbitrary blocks of color.[5]

My first VGA program produced random color blocks on the screen. Not very meaningful, but it showed that everything worked.

Putting characters on the screen

The next step was to display text characters on the screen. I implemented a character generation module to provides the

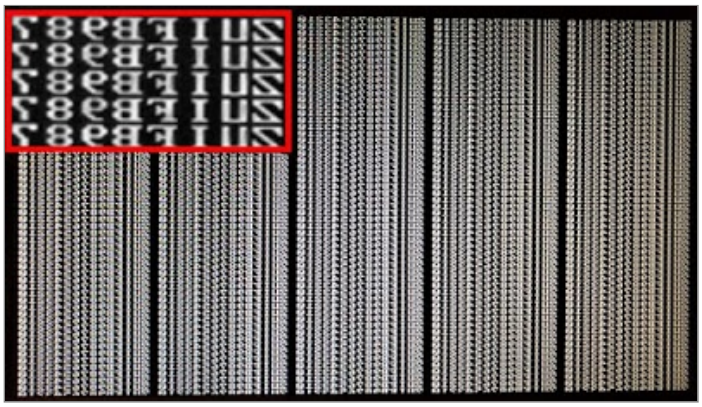
pixels for an 8x8 character. Rather than include the full ASCII character set, I only used the characters necessary for FizzBuzz: 0 through 9, "B", "F", "i", "u", "z" and blank. Conveniently, this worked out to 16 character values, fitting into a 4-bit input.

Thus, the module takes a 4-bit character code and a 3-bit row number (for the 8 lines of each character) and outputs 8 pixels for that row of the character. The code (excerpt below, full code here) is simply a big case statement to output the appropriate bits.[6] This code essentially compiles into a ROM, which is implemented in the FPGA by lookup tables.

```
case ({char, rownum})
  7'b0000000: pixels = 8'b01111100; //  XXXXX
  7'b0000001: pixels = 8'b11000110; // XX   XX
  7'b0000010: pixels = 8'b11001110; // XX  XXX
  7'b0000011: pixels = 8'b11011110; // XX XXXX
  7'b0000100: pixels = 8'b11110110; // XXXX XX
  7'b0000101: pixels = 8'b11100110; // XXX  XX
  7'b0000110: pixels = 8'b01111100; //  XXXXX
  7'b0000111: pixels = 8'b00000000; //

  7'b0001000: pixels = 8'b00110000; //   XX
  7'b0001001: pixels = 8'b01110000; //  XXX
  7'b0001010: pixels = 8'b00110000; //   XX
  7'b0001011: pixels = 8'b00110000; //   XX
  7'b0001100: pixels = 8'b00110000; //   XX
  7'b0001101: pixels = 8'b00110000; //   XX
  7'b0001110: pixels = 8'b11111100; // XXXXXX
...
```

I updated the top-level program to use the low bits of the X pixel position for the character and pixel index, and the low bits of the Y pixel position for the row index. The results can be seen below; I've magnified the text in the red box so you can see the characters.



My first implementation of text. (Zoomed region in red.) Due to a bug, all the characters are backwards.

Oops, I implemented the character generator with bit 7 on the left, while the pixel index values have bit 7 on the right, so the characters were displayed backwards. But a quick fix got the characters to display correctly.

Generating a line of FizzBuzz

Once I could display characters, I needed to provide the right characters for the FizzBuzz output. The algorithm is the same as my previous FizzBuzz program, so I'll just give the highlights here.

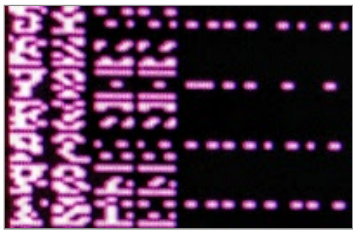
Converting the numbers from 1 to 100 into characters is trivial on a microprocessor, but more difficult with digital logic since there's no built-in divide operation; dividing by 10 and 100 requires many logic gates. My solution was to use a binary-coded decimal (BCD) counter, using a separate 4-bit counter for each digit.

The next challenge was testing if the number was divisible by 3 or 5. As with division, the modulo operation is easy on a microprocessor, but hard with digital logic. Instead of computing modulo values, I used counters for the value modulo 3 and the value modulo 5. The value modulo 3, for instance, simply counts 0, 1, 2, 0, 1, 2, ... (See the footnote for other approaches.[7])

A FizzBuzz output line can be up to 8 characters long. The `fizzbuzz` module (source) outputs the appropriate eight 4-bit characters as a 32-bit variable `line`. (The normal way to generate video would be to store all the screen's characters or pixels into video RAM, but I decided to generate everything dynamically.) An `if` statement (excerpt below) updates the bits of `line` to return "Fizz", "Buzz", "FizzBuzz" or the number as appropriate.

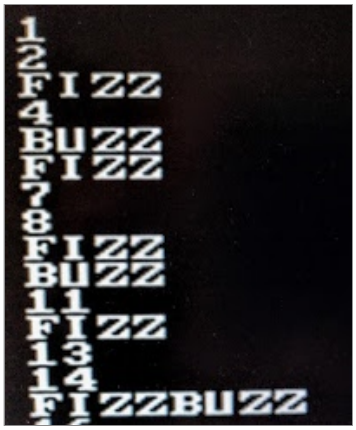
```
if (mod3 == 0 && mod5 != 0) begin
  // Fizz
  line[3:0] <= CHAR_F;
  line[7:4] <= CHAR_I;
  line[11:8] <= CHAR_Z;
  line[15:12] <= CHAR_Z;
end else if (mod3 != 0 && mod5 == 0) begin
  // Buzz
  line[3:0] <= CHAR_B;
  line[7:4] <= CHAR_U;
  line[11:8] <= CHAR_Z;
  line[15:12] <= CHAR_Z;
...
```

The FizzBuzz module needed a signal to increment the counts to the next number so I modified the `vga` module to indicate the start of a new line and used this to move to the next number. When I tried my code, I got very strange output with alien symbols; the photo below shows a detail.



Changing the character every row yields mysterious symbols, not the desired characters.

The bug was that I was moving to the next FizzBuzz value after every line of pixels instead of waiting 8 lines to draw a full character. Thus, each displayed character consisted of slices from 8 different characters. Incrementing the FizzBuzz counter every 8 lines instead of every line fixed this problem, as you can see below. (Debugging VGA code is much easier than other FPGA things, since problems are visible on the screen. You don't need to poke around with an oscilloscope trying to figure out what went wrong.)



The FizzBuzz output, as displayed on a VGA monitor.

At this point I had the FizzBuzz output appearing on the screen, but the static display was kind of boring. Changing the foreground and background color of each row was easy—just using some bits of the Y value for the red, green and blue colors. This yielded colored text with a 1980s PC aesthetic.

With the foreground and background colors based on the line, the text is more interesting.

Next, I added some basic animation. The first step was to add an output from the `vga` module to indicate when the screen was redrawn, a new field every 60th of a second. I used this to change the colors dynamically. (Tip: don't flash the screen color every 60th of a second unless you want a headache; use a counter and change it less often.)

Trying out different graphical effects is fun and addictive, since you get immediate feedback. I decided to have the "Fizz" and "Buzz" lines slide across the screen with a rainbow color trail (inspired by Nyan Cat). To do this, I changed the character's start position based on a counter. For the rainbow color effect, I selected the color based on the row number (so each row of

the character could have a different color) and added the rainbow trail.

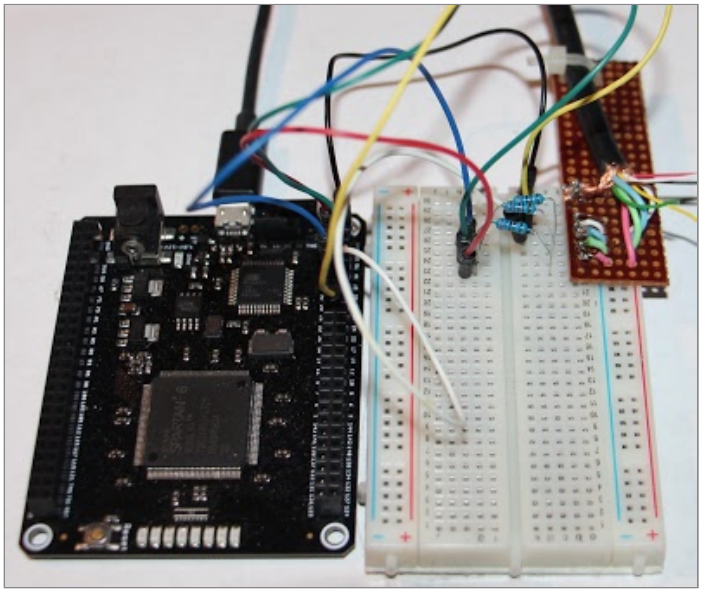
A closeup of the final output.

The final effect I added was giant words "Fizz" and "Buzz" that bounced around the screen. The bouncing effect was based on a bouncing invisible box (inspired by FPGA Pong) that held the word.[8] A variable `dx` keeps track of the X direction and `dy` keeps track of the Y direction. On each new screen (i.e. 60 times a second), the box's X and Y coordinates are incremented or decremented based on the direction variables. If the box reaches the right or left edges, `dx` is toggled. Similarly, `dy` is toggled if the box reaches the top or bottom. The big text is then drawn inside the box using another instantiation of the character generator described earlier. The word is enlarged by a factor of 8 by dropping the three low-order bits from the coordinate. You're probably tired of Verilog code by now so I won't show the code here, but it's on github. The final result is shown in the video clip below.

Hardware details

For this project, I used the Mojo V3 FPGA board development board (below), which was designed to be an easy-to-use starter board. It uses an FPGA chip from Xilinx's Spartan 6 family. Although the Mojo uses one of the smallest Spartan 6 chips, the chip still contains over 9000 logic cells and 11,000

flip flops, so it can do a lot. (For other options, see this list of cheap FPGA boards.)



Interfacing the FPGA board to VGA is almost trivial: just three 270Ω resistors. The perf board is just to attach the cable to a header.

If you want to use VGA, it's easier to use a development board that includes a VGA connector. But if your board lacks a connector (like the Mojo), adding VGA is straightforward. Simply put 270Ω resistors between the FPGA's red, green and blue output pins and the VGA connection. The horizontal and vertical sync can be wired directly from the FPGA.[9]

I used Xilinx's development environment (called ISE) to write and synthesize the Verilog code. (For details on writing code and getting it onto the FPGA board, see my previous FPGA article.) To specify which physical FPGA pins to use, I added lines to the `mojo.ucf` configuration file. This maps the red output `pin_r` to pin 50 on the board, and so forth.

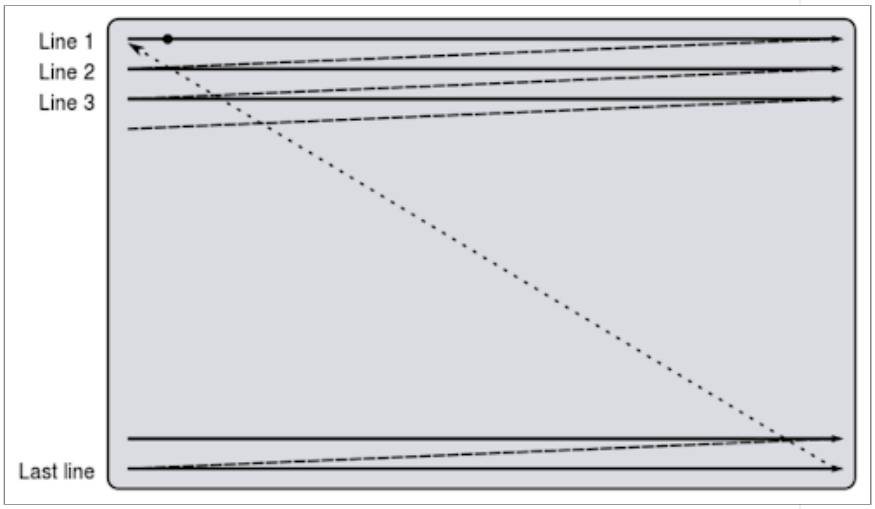
```
NET "pin_r" LOC = P50 | IOSTANDARD = LVTTL;
NET "pin_g" LOC = P40 | IOSTANDARD = LVTTL;
NET "pin_b" LOC = P34 | IOSTANDARD = LVTTL;
NET "pin_hsync" LOC = P32 | IOSTANDARD = LVTTL;
NET "pin_vsync" LOC = P29 | IOSTANDARD = LVTTL;
```

Driving VGA from an FPGA is common, so you can find lots of similar projects on the web such as FPGA Pong, 24-bit color using a DAC chip, the Basys 3 board and displaying an image. If you want a schematic, see this page. The book Programming FPGAs has a whole chapter on VGA.

Understanding the VGA format

The VGA video format may seem a bit strange, but looking at the history of television and the CRT (cathode ray tube) provides context. In a CRT, a beam of electrons scans across the screen, lighting up the phosphor coating to produce an image. Scanning happens in a raster pattern: the beam scans across the screen left-to-right, and then a horizontal sync pulse causes the beam to rapidly return to the left during the

horizontal retrace. The process repeats line-by-line until the bottom of the screen, when a vertical sync pulse triggers *vertical retrace* and the beam returns to the top. During the horizontal and vertical retrace, the beam is blanked so retrace doesn't draw lines on the screen. The diagram below illustrates the raster scan pattern.

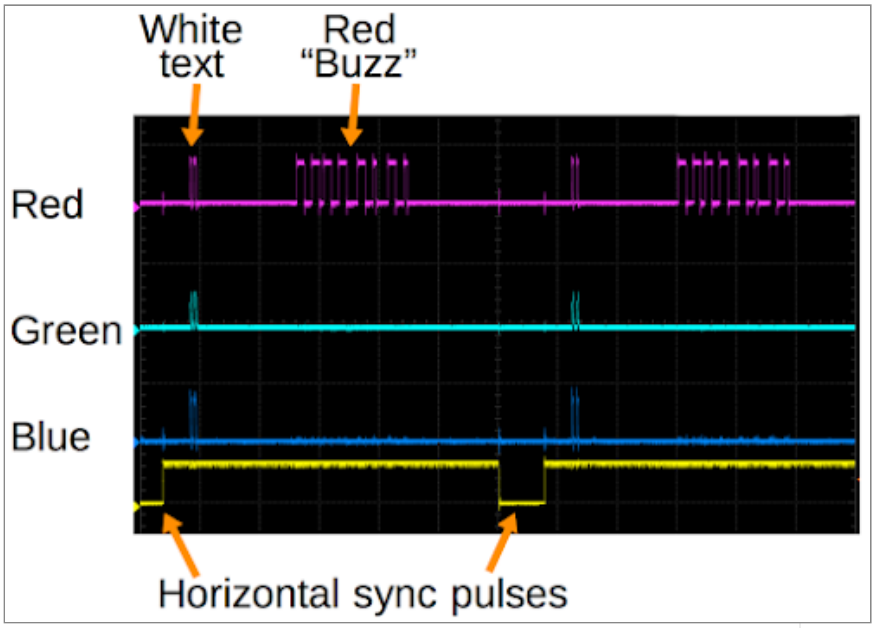


Raster scan pattern for a CRT. (Wikipedia)

These characteristics carried over to VGA, resulting in the horizontal and vertical sync pulses, and the long intervals during which the video must be blanked.

In 1953, the NTSC standard for color television was introduced.[10] VGA, however, is much simpler than NTSC since it uses five wires for the sync and color signals, instead of cramming everything into a single signal with a complex encoding. One strange feature of NTSC that carries over to VGA is the screen refresh rate isn't the claimed 60 Hertz, but actually 59.94 Hertz.[11]

The oscilloscope trace below shows the VGA video signals across two lines. The horizontal sync pulses (yellow) indicate the start of each line. The brief red, green and blue pulses near the start of the line are white pixels from a FizzBuzz number. The red signal near the middle of the line is the floating red word "Buzz".



Oscilloscope trace of VGA signals, showing red, green, blue and horizontal sync.

Conclusion

Driving a VGA monitor from an FPGA was much easier than I expected, but I certainly wouldn't get hired if I encountered FizzBuzz as an FPGA interview question! If you're interested in FPGAs, I highly recommend playing around with video output. It's not much harder than blinking an LED and much more rewarding. Creating video output is also much more fun than debugging with an oscilloscope—you get immediate visual feedback and even if things go wrong, they are often entertaining.

Follow me on Twitter or RSS to find out about my latest blog posts. My FPGA code is on github.

Notes and references

1. The VGA signal is supposed to have a 25.175 MHz pixel clock. Instead, I divided the Mojo board's 50 MHz clock by 2 to get a 25 MHz clock. The VGA standard says that the pixel clock must be 25.175 MHz ± 0.5%. A clock rate of 25 MHz is off by 0.7% so technically is a bit off from the spec. However, monitors are designed to handle signals generated by cheap graphics cards, so they will usually manage to display whatever you throw at them. ↩

2. The detailed timings for dozens of standard video resolutions can be found here. Page 17 has the 640x480 60 Hz resolution I used. ↩

3. When I forgot to blank the pixels outside the valid screen area, the monitor still managed to display an image, but it was very dim because the monitor got confused about what voltage represented "dark". Just a tip in case you find your display mysteriously darkened. ↩

4. It's kind of amazing if you think about what an LCD monitor needs to do in order to display a VGA image designed for a CRT. The monitor has to examine the incoming signals to "guess" what video resolution a computer is sending to it. Then it needs to resample the video signal to match the resolution of the LCD panel. Finally, it sends the new pixel data to the LCD. ↩

5. For some reason, the output was shifted down about 25 pixels. It worked fine on a different monitor, so I'm not sure if was just something with my monitor's alignment or if I had a bug. I could adjust this by making the vertical sync pulse 25 rows later. ↩

6. It would be tedious to type all the code for character generation, so I wrote a Python program to create the Verilog code from a font file. The original font is here. ↩

7. After my last FPGA FizzBuzz, people suggested some other approaches so I tried them out on the VGA project. With modulo counters (my original approach), the entire project used 276 LUTs (lookup tables) and 277 flip flops. One suggestion was to use ring counters (i.e. one shifted bit) instead of binary counters for the modulo counters. This used 277 LUTs and 281 flip flops, so slightly worse. Another suggestion was to add the digits and take the modulo 3 value. The logic to add and perform modulo brought the count to 305 LUTs, much worse. Adding the modulo 3 values (instead of digits) brought it down to 289 LUTs, still worse. ↩

8. The big floating words are essentially sprites—bitmaps that can be arbitrarily positioned on the screen. Sprites were popular with video games and computers in the 1970s and early 1980s such as Pacman, Atari and Commodore 64. Sprites let slow processors perform animation; instead of moving all the pixels around in memory, the processor would just update the sprite coordinates. The top-level code (source) ties together all the pieces and combines everything onto the screen: the text, the rainbow trail, the giant "Fizz" (in a rainbow pattern) and the giant "Buzz" (in red). ↩

9. To connect to the VGA monitor, I cut a VGA cable in half and connected its wires to the FPGA board, reusing the cable from my project to read monitor data using I2C. Don't take this approach—the tiny wires inside are difficult to solder and break easily. Instead, just use a VGA connector. The interface to VGA is simply three 270Ω resistors, to get the right voltages for the red, green and blue signals. You can use more resistors to get more colors, essentially building 2-bit or more A/D converters. ↩

10. NTSC was a remarkably clever way to introduce color, while remaining compatible with black-and-white televisions. However, it is very hard to understand. I was scared off from video by the talk of encoding color with phase and quadrature and a color subcarrier synced to a

color burst signal. Because NTSC combines the color and sync signals into a single broadcast signal, the encoding is much more complex than VGA. ↩

11. While monitors say they have a 60 Hz refresh rate, the actual refresh rate is 59.94 Hz, a strange frequency dating back to the origins of color television. A screen refresh frequency of 60 Hz was desirable to cancel out power line interference. However, to prevent the color carrier frequency from interfering with the audio frequency, various frequencies had to be adjusted, resulting in the screen refresh frequency of 59.94 Hertz. It almost seems like numerology, but the frequency choices are explained in detail in this video and in Wikipedia. ↩

2 comments:

Labels: electronics, fpga

Home Older Posts