Silicon-Level Reading Guide

4,517 words · ~21 min read

VCA-CSA-101 cross-chapter handout. Audience: students who have completed Ch 5 and shipped their first FPGA bitstream on a Tang Primer 25K.

Purpose: to walk you from the FPGA design you just synthesized -- a CPU made of LUT4 cells and flip-flops -- back to the 1970s physical transistors that first implemented the same abstractions in fabricated silicon. You will see a flip-flop in a 1976 MOS chip die photo, trace a memory read through a 1980 Z80 at transistor scale, and understand why the FPGA fabric on your Tang Primer 25K is the same idea at a completely different manufacturing node.

No prior knowledge of chemistry or semiconductor physics is required. Everything here can be read and followed at a browser.

---

§0 Overview and audience

You have just synthesized a working RV32I-Lite CPU. The synthesis tool turned your Verilog into a netlist of LUT4 cells, flip-flops, and block RAM primitives. A bitstream loader configured the Tang Primer 25K's SRAM cells to implement those primitives. Your CPU ran a program.

At no point in Ch 1 through Ch 5 did you need to know what a transistor looks like in physical silicon. But the Petzold thread running through CSA-101 has been building toward this: every abstraction you implemented -- the AND gate in Ch 1, the adder in Ch 2, the flip-flop in Ch 3, the ALU in Ch 5 -- has a physical realization in fabricated silicon that you can see and study in published die-shot photography and interactive simulators.

This handout covers three open-license community resources that let you do exactly that:

- The **righto.com die-shot annotations** by Ken Shirriff, specifically his Z80 register file and Z80 ALU walkthroughs
- The **Visual6502 project**, an open-source interactive simulation of the MOS 6502 at transistor level

- The **Masswerk 6502 emulator**, a cycle-accurate browser sandbox you can use alongside Visual6502

The handout closes with a concrete bridge back to your FPGA work: what your synthesized design's LUT4s and flip-flops correspond to at the physical layer, and why modern FPGAs achieve programmable logic through a completely different physical mechanism than 1970s NMOS.

---

§1 What die shots are and how they're made

A **die shot** is a photograph of the silicon chip inside an IC package, taken after the plastic or ceramic case has been chemically removed. The removal process is called **decapping**. A typical decap uses hot acid (fuming nitric or sulfuric) to dissolve the epoxy mold compound without attacking the silicon die underneath. What is left is the raw chip, typically a few millimeters on a side.

The photograph itself is taken under a high-magnification optical microscope. The colors you see in a die photo are not the natural color of silicon: they come from the different materials in the chip's layer stack reflecting and refracting light differently. Metal interconnect (aluminum or copper) looks bright silver or white. Polysilicon gates look a different shade. Diffusion regions in the substrate have yet another appearance. A trained eye -- or, with practice, a beginner's eye -- can learn to read these color cues.

What you are looking at in a die photo:

A typical 1970s or 1980s NMOS chip like the MOS 6502 or Zilog Z80 has two to three physical layers visible in optical die photography:

1. **Metal layer.** The wiring. Wide horizontal and vertical conductors that connect transistors to each other across long distances. Power and ground rails run here.
2. **Polysilicon layer.** Narrower conductors that often cross the metal layer and form transistor gates where they cross diffusion regions.
3. **Diffusion layer.** The actual doped silicon regions where transistors live. An NMOS transistor is formed wherever a polysilicon stripe crosses a diffusion region: the poly becomes the gate; the diffusion on each side becomes the source and drain.

A single NMOS transistor is about 10 micrometers wide on a 1976-era process. A 1976 MOS 6502 contains approximately 3,510 transistors across a 3.9 mm x 4.3 mm die. Every logic gate, register bit, and ALU carry bit is made of those transistors.

By comparison, the GW5AT-138 FPGA on your Tang Primer 25K uses a 55nm CMOS process with 138,000 LUT4 cells. Each LUT4 cell contains dozens of transistors internally. The die you hold in your hand contains more transistors than the 6502 has on its entire die in every single cell.

Scanning electron microscope (SEM) imaging can resolve features smaller than optical wavelengths. SEM die photos look different -- they are greyscale, taken with an electron beam, and can reveal sub-100nm features on modern chips. The community resources in this handout use optical die photography for 1970s chips, where features are large enough to image clearly in visible light.

---

§2 The MOS 6502: interactive silicon walkthrough

The **Visual6502 project** (https://github.com/trebonian/visual6502) is an open-source CC-BY-SA interactive simulator of the MOS 6502 at transistor level. The project team traced every transistor in the 6502 die photo by hand, extracted a full netlist, and built a JavaScript simulator that runs the chip cycle by cycle. You can click on a transistor in the die image and watch it turn on and off as the CPU runs instructions.

Why the 6502? It was the chip inside the Apple I, Apple II, Atari 2600, Commodore 64, and original NES. Its die has been photographed extensively, its transistors have been counted (3,510), and the community that reverse-engineered it produced one of the best-documented silicon walkthroughs of any CPU. The 6502 is also architecturally simple enough that a student who has just completed CSA-101 Ch 5 can follow the ALU and register paths without a graduate-level background.

Loading the Visual6502 interactive simulator

The interactive simulator lives at **https://visual6502.org/JSSim/** (note: the visual6502.org domain had an expired TLS certificate as of May 2026; you may need to click through a browser warning, or use HTTP instead of HTTPS). An alternative is to clone the GitHub repository and run it locally:

```
git clone https://github.com/trebonian/visual6502
cd visual6502
# open index.html in a browser -- no server required
```

Running locally avoids the cert issue entirely and gives you the same simulator.

Guided walkthrough: finding the ALU

When the simulator loads, you see a colorized top-down view of the 6502 die with a control panel below it. The chip is already running; you can see transistors flickering as the 6502 executes instructions in a small internal RAM.

1. **Pause the simulation** using the "Halt" button in the control panel.
2. **Find the ALU.** On the 6502 die, the ALU is a vertical band running most of the height of the chip. It is toward the left side when you orient the die with the 6502 logo at the top. You can identify it by the repeating cell structure: 8 similar-looking units stacked vertically, one per bit of the 8-bit data path.
3. **Click any transistor** in that region. The simulator highlights that transistor and shows you its node name and connections. Node names like `alu_sums` or `alu_cout` are labeled in the netlist and appear in the sidebar.
4. **Resume and watch** the ALU transistors toggle as the simulated CPU adds numbers. The color shift from dark to light represents the transistor switching from off to on.

Guided walkthrough: identifying the X register

The 6502's programmer-visible registers (A, X, Y, SP, PC) each correspond to a physical bank of flip-flops in the die. The X register is an 8-bit register made of 8 NMOS flip-flop cells.

1. In the simulator, use the "Find" panel (some versions call it "Node search") and search for a node named `x0` through `x7`.
2. The simulator will highlight the transistors for that register bit in the die view.
3. Run an `LDX #42` instruction by setting the program counter in the control panel and stepping through it. Watch the X register cells change state as the value loads.

If the Visual6502 JS interface is unfamiliar, the project's own documentation at https://github.com/trebonian/visual6502/wiki walks through the controls.

Tracing an LDA instruction through silicon

The 6502's `LDA #imm` (Load Accumulator Immediate) is one of the simplest instructions. In silicon:

1. The program counter fetches the opcode byte (`0xA9`) from the memory bus.
2. The instruction decode logic (a PLA -- a programmed logic array of transistors, visible as a grid in the die photo) recognizes the opcode.

3. The sequencer advances to cycle 2, fetching the immediate operand byte.
4. The operand routes through the internal data bus to the accumulator latch cells.
5. The accumulator flip-flops capture the value on the next clock edge.

In the Visual6502 simulator you can step this instruction one clock cycle at a time and watch each of these steps happen as transistors switch. The control panel's "Step" button advances one half-cycle; two half-cycles equal one full clock period.

This is the same sequence your CSA-101 Ch 5 CPU implements in Verilog -- fetch, decode, execute, writeback -- but here you are watching it happen in 3,510 physical transistors rather than in a simulation of an FPGA netlist.

Supplemental sandbox: The Masswerk 6502 emulator at https://www.masswerk.at/6502/ (GPL v2+) provides a cycle-accurate 6502 environment with a readable register/flag display and an assembler built in. It is easier to enter test programs here than in the Visual6502 control panel, then cross-reference the behavior with Visual6502's transistor view.

---

§3 The Z80 register file: silicon annotations by Ken Shirriff

Ken Shirriff is a software engineer and historian who has produced some of the best silicon-level walkthroughs of classic chips available anywhere. His Z80 register file article at **https://www.righto.com/2014/10/how-z80s-registers-are-implemented-down.html** walks through the Z80's register file at transistor scale. Read the article and follow this section together.

What the Z80 register file looks like on the die

The Z80 is an 8-bit CPU from 1976, designed by Federico Faggin and Masatoshi Shima at Zilog after they left Intel. It has more programmer-visible registers than the 6502: AF, BC, DE, HL (plus shadow copies AF', BC', DE', HL'), IX, IY, SP, PC, I, R, and the hidden W/Z internal registers. Fitting all of these on the die required a compact physical layout.

In Shirriff's die photo, the register file is a rectangular block you can identify by its regular grid of transistors. Each bit of storage is a **static latch cell** -- a pair of cross-coupled inverters that hold one bit indefinitely as long as power is applied. This is the same structure as a static RAM cell: two transistors in a bistable loop, with pass transistors on each side that the read and write circuitry switches to load or retrieve a value.

The CSA-101 connection: in Ch 3, you built registers from D flip-flops. A D flip-flop is a clocked version of exactly this cross-coupled latch structure. The 6502 and Z80 registers ARE D flip-flops (or transparent latches, depending on implementation detail) made from NMOS transistors.

The hidden W/Z registers

Here is one of the most instructive things about the Z80 die: the chip contains two registers that never appear in the programmer's manual. They are labeled W and Z internally, and they are physically present and visible in the die photo.

These registers are **temporary holding latches** used by the CPU's microcode to stage operands during multi-byte operations. When you execute a 16-bit load like `LD HL, (nn)` -- which reads a 16-bit value from memory at address `nn` -- the CPU cannot write both bytes to HL simultaneously. It reads the low byte first, holds it in Z, reads the high byte next, holds it in W, then writes WZ to HL in a single atomic step.

The programmer's manual does not document W/Z because no program can directly read or write them. But they exist in the silicon. Shirriff's article shows exactly where they sit in the die photo relative to the named registers.

The pedagogical point: the silicon does not match the programmer's model. The programmer's model is an abstraction layer, deliberately simplified. The hardware underneath has extra state that the CPU uses internally to implement the programmer-visible behavior. Your RV32I-Lite CPU has the same pattern: the microarchitecture (your pipeline registers, your instruction-fetch PC latch, your ALU intermediate register) is not what the RV32I ISA specification describes. The ISA is the contract; the microarchitecture is the implementation.

Bit-cell layout and why registers look like RAM

Shirriff's article points out that the Z80's register cells and RAM cells are physically identical structures. Both are static latch cells. The only difference is address decoding: register cells are selected by hardwired control signals; RAM cells are selected by an address decoder.

This is the same tradeoff visible in your Ch 3 Verilog: a flip-flop array (register file) and a block RAM both hold bits, but their synthesis targets are different FPGA primitives (DFF vs BRAM) because their access patterns differ.

---

§4 The Z80 ALU: 4-bit wide, 8-bit architecture

Shirriff's Z80 ALU article at https://www.righto.com/2013/09/the-z-80-has-4-bit-alu-heres-how-it.html reveals something that surprises almost everyone who reads it: the Z80 is described as an 8-bit CPU, but its ALU is only 4 bits wide.

Why a 4-bit ALU in an 8-bit CPU?

The answer is area. A full 8-bit ALU requires twice the silicon area of a 4-bit ALU. In 1976, silicon area was expensive. Zilog's engineers chose to perform 8-bit arithmetic in two 4-bit passes: the ALU processes the low nibble first, saves the carry, then processes the high nibble using the saved carry. The two-pass operation is invisible to the programmer: from the programmer's perspective, `ADD A, B` adds two 8-bit values atomically in one instruction cycle. The silicon takes two clock phases to do the 4-bit arithmetic, but the programmer never sees that.

This is microarchitectural optimization hidden behind the ISA abstraction -- exactly the concept Ch 5 introduces when it discusses how your RV32I-Lite could implement 32-bit addition either as a single-cycle carry-ripple chain or as a pipelined carry-save adder with completely different area and timing, while presenting identical ISA behavior to the programmer.

What the 4-bit ALU looks like on the Z80 die

In Shirriff's die photo, the ALU is a column of cells visible near the center-left of the chip. Each cell handles one bit of arithmetic. You can count them: there are 4 full cells, not 8. The datapath is literally half the width you might expect.

The article walks through the carry-save structure and identifies the specific transistors that implement addition, subtraction, AND, OR, XOR, and the flag logic. The carry propagation path -- the critical timing path in any ALU -- is visible as a chain of transistors running down the column.

Your CSA-101 Ch 5 ALU implements all of these operations in Verilog. The Z80's silicon ALU implements the same operations in NMOS transistors. The logic is identical; the physical substrate is different.

Flag registers

The Z80 has 8 status flags (S, Z, H, P/V, N, C, and two undocumented flags F3 and F5). These are stored in a dedicated register -- physically identical to the other register cells -- and updated after every ALU operation based on the result.

Your CSA-101 Ch 5 ALU produces carry-out and zero flags. The Z80's silicon flag logic uses additional transistor networks to detect sign (MSB of result), parity (XOR of all 8 result bits), and half-carry (carry out of bit 3). Each detection network is a small transistor circuit you can locate in the die photo.

---

§5 Bridge to your FPGA work

You synthesized a CPU on the Tang Primer 25K. The synthesis tool's output was a bitstream that configures the FPGA's LUT4 cells and flip-flop elements. Here is how those map to what you just read about 1970s silicon.

LUT4 cells vs NMOS gate networks

A 1970s NMOS gate -- an AND gate, for example -- is a network of physical transistors. Two NMOS transistors in series (with pull-up resistors above) implement a NAND gate in about 3-4 transistors. You cannot reprogram it. The transistors are fixed on the die.

An FPGA **LUT4** (4-input lookup table) is a different approach. It is a 16-bit SRAM array with a 4:1 multiplexer tree on top. The 4 input bits select one of 16 SRAM cells; the SRAM cell's value is the output. By programming the 16 SRAM bits, you define any 4-variable boolean function you want. The LUT4 implements a 2-AND just as easily as it implements a 4-XOR; you just program different SRAM contents.

The GW5AT-138 on your Tang Primer 25K contains 138,000 of these LUT4 cells. Each LUT4 cell contains, at the 55nm process level, dozens of CMOS transistors implementing the SRAM array and multiplexer tree. The total transistor count for the full FPGA die is many billions. But none of those transistors are individually programmable: they implement the fixed LUT4 structure, and the SRAM bits inside each LUT4 are what you configure with your bitstream.

The key contrast:

- 6502: 3,510 NMOS transistors, each doing a fixed job determined at fabrication.
- Tang Primer 25K FPGA: billions of CMOS transistors, organized into 138,000 LUT4 + flip-flop cells, each cell configurable by the bitstream you load.

Flip-flops: then and now

The Z80's register bits are static NMOS latch cells. Your Tang Primer 25K's flip-flops are CMOS D flip-flops implemented in dedicated DFF silicon next to each LUT4. They are electrically more complex (they have a clock enable, reset, and set input each) but logically identical: one clock edge captures the input value and holds it until the next edge.

When your Verilog says `always @(posedge clk) reg <= next_val;`, the synthesis tool maps that to a Tang Primer 25K DFF cell. That DFF cell's silicon implementation is a chain of CMOS transmission gates and inverters, not the NMOS cross-coupled latch of the Z80, but both perform the same function: store one bit, update it on a clock edge.

Block RAM vs register file cells

Your Tang Primer 25K has dedicated **block BRAM** cells (hard-macro RAM blocks) separate from the LUT4 fabric. When your Verilog infers a RAM with `reg [31:0] mem [0:N-1]` and a synchronous read port, the synthesis tool places it in BRAM rather than in LUT4 cells. This is the same tradeoff visible in the Z80: register file cells (fast, addressed by fixed control signals) vs a RAM array (slower, addressed by a decoder). On the FPGA, both choices are available; the synthesis tool picks based on inference heuristics.

Process node comparison

Chip	Year	Process	Transistor size	Transistors
MOS 6502	1975	NMOS 8 um	~8 micrometers	3,510
Zilog Z80	1976	NMOS 4 um	~4 micrometers	8,500
GW5AT-138 FPGA	2020s	CMOS 55nm	~55 nanometers	>10 billion

A 55nm transistor is about 145 times smaller in linear dimension than a 4um transistor. The feature density difference is roughly 21,000x. The 6502's entire die, all 3,510 transistors, would fit inside a single modern FPGA LUT4 cell with room to spare.

The abstractions are the same. The scale is not.

---

§6 Architecture Comparison Sidebar

Petzold alignment: CODE Ch 17-19 covers gates, latches, registers, the memory bus. CODE Ch 21 covers the 8080 CPU (the Z80's ancestor). This sidebar extends Petzold's historical arc through four architectures visible at the die level.

MOS 6502 (1975)

- **Die size:** 3.9 mm x 4.3 mm
- **Transistors:** ~3,510 NMOS
- **Registers:** A (accumulator), X, Y (index), SP (stack pointer), PC (program counter), P (status flags). No general-purpose register file in the modern sense; the zero page of RAM functions as extended registers.
- **ALU:** 8-bit, single-pass
- **What you can see in the die:** The 6502 is the most thoroughly documented die in the hobbyist community. Visual6502 has traced every transistor. The PLA instruction decoder is visible as a grid near the top-center of the die.
- **Historical role:** Apple I, Apple II, Atari 2600, Commodore 64, original NES (via the 2A03 derivative). The chip that made the personal computer revolution affordable.

Zilog Z80 (1976)

- **Die size:** approximately 10 mm x 10 mm (larger than 6502)
- **Transistors:** ~8,500 NMOS
- **Registers:** 8-bit AF, BC, DE, HL (plus shadow set), 16-bit IX, IY, SP, PC, plus hidden W/Z internal registers. True general-purpose register file.
- **ALU:** 8-bit architecture, 4-bit physical ALU (two-pass)
- **What you can see in the die:** Register file and 4-bit ALU are both clearly visible and annotated in Ken Shirriff's righto.com walkthroughs (links in §3 and §4 above).
- **Historical role:** TRS-80, Sinclair ZX81/Spectrum, MSX computers, CP/M systems. The Z80 was the dominant 8-bit CPU in non-Apple, non-Commodore personal computers.

Motorola 6800 (1974)

- **Die size:** roughly comparable to 6502
- **Transistors:** approximately 4,100 NMOS
- **Registers:** A, B (two accumulators), IX (index), SP, PC, CC (condition codes). No X/Y separate index registers like the 6502.
- **ALU:** 8-bit
- **Historical role:** Ancestor of the 6809 (used in CoCo, Dragon) and ultimately the 68000 (Macintosh, Amiga, early Sun workstations). Also ancestor of the 6802 and Motorola 68xx embedded family.

- **Die resources:** visual6502.org had an interactive 6800 simulator (same project team); availability uncertain as of 2026 -- check https://github.com/trebonian/visual6502 for current project status.

RV32I-Lite (your design)

- **Die:** not a fixed die -- realized on GW5AT-138 FPGA fabric (55nm CMOS)
- **LUT4 count:** varies by synthesis options; count your DFFs and LUT4s in the Yosys output netlist (see §8 Exercise 3 below)
- **Registers:** x0-x7 (8 general-purpose, x0 hardwired zero), PC. Strict RV32I subset.
- **ALU:** 32-bit combinational; synthesized from LUT4 chains by the tools
- **Historical role:** your design, based on the RISC-V open-specification ISA. RISC-V is the direct descendant of decades of RISC architecture research (MIPS, SPARC, Alpha). A 2020s academic/industry ISA running on 2020s FPGA silicon, connected to 1970s architectural ideas via the same abstractions Petzold traces.

---

§7 Cross-references

Backward-looking (prerequisite material)

- **CSA-101 Ch 1** (*Boolean Logic*): AND, OR, NOT, NAND gates. Each gate in a 1970s NMOS chip is a handful of physical transistors implementing these truth tables.
- **CSA-101 Ch 2** (*Boolean Arithmetic*): adder chains. The Z80's 4-bit ALU in §4 is the physical implementation of the ripple-carry adder chain you built in Ch 2.
- **CSA-101 Ch 3** (*Memory*): flip-flops and latches. The Z80 register cells in §3 are physical D latches; your FPGA flip-flops are the CMOS equivalent.
- **CSA-101 Ch 5** (*Computer Architecture*): the chapter this handout extends. The silicon walkthroughs here fill in the physical picture beneath the Verilog abstraction.
- **Petzold CODE Ch 17-19**: gates, flip-flops, memory, the 8-bit memory bus. The 6502 and Z80 die photos are the physical realization of what Petzold describes.
- **Petzold CODE Ch 21**: Petzold describes the Intel 8080, the Z80's direct ancestor. The Z80 is binary-compatible with the 8080 and shares many register-level design decisions.

Forward-looking (what comes next)

- **CSA-101 Ch 5 §5.8**: synthesis and RAM primitives on the Tang Primer 25K. The FPGA bridge discussion in §5 above extends that section.

- **RE-201** (*Hardware Reverse Engineering*): die-shot analysis and silicon-level reverse engineering are core RE-201 skills. The righto.com articles and Visual6502 project are models of the documentation that professional RE work produces.
- **VCA-ARM-201** (*ARM Architecture*): ARM chips have been die-photographed and partially documented by the community. Silicon RE of ARMv4/ARMv5 cores (used in first-generation iPods and embedded systems) is within reach using the same methods.
- **CON-101** (*Virtus Console*): the NES CPU (MOS 2A03) is a 6502 derivative; Visual2A03 at https://visual6502.org/JSSim/expert-2A03.html provides the same interactive die-shot simulator for the NES CPU. Availability may vary -- check the visual6502 GitHub for current status.
- `cross-chapter-fpga-cell-to-silicon-bridge.md`: the companion handout to this one, focused specifically on LUT4/DFF/BRAM physical structure on modern FPGA silicon.

---

§8 What you can actually do today

These are four concrete exercises you can run right now with free, open-license tools.

Exercise 1: Find the X register in the Visual6502 simulator

Goal: locate a named register in die-shot space.

1. Load Visual6502 at https://visual6502.org/JSSim/ (or clone and open locally from https://github.com/trebonian/visual6502).
2. Halt the simulation.
3. Use the node search to locate `x0`. The simulator highlights transistors in the die photo.
4. Zoom in on the highlighted area. You should be able to see a repeating cell structure: 8 similar units side by side, one per bit of the X register.
5. Resume and enter a sequence of `LDX #n` instructions with different values. Watch the highlighted transistors change state.

What you are seeing: the same 8-bit register that your Verilog models as `reg [7:0] reg_x` -- but made of physical NMOS transistors on 1975 silicon.

Exercise 2: Read both righto.com Z80 articles

Goal: understand what a die-shot annotation looks like as a finished artifact.

1. Read https://www.righto.com/2014/10/how-z80s-registers-are-implemented-down.html (Z80 register file).
2. Read https://www.righto.com/2013/09/the-z-80-has-4-bit-alu-heres-how-it.html (Z80 4-bit ALU).

For each article, answer these questions in your Toolchain Diary:

- Which region of the die does the article highlight?
- How many transistors are used for a single bit of register storage?
- What is one thing you see in the silicon that is NOT visible in the Z80 programmer's manual?

Exercise 3: Count DFFs in your synthesized design

Goal: connect your FPGA synthesis output to the transistor-count numbers in this handout.

Run Yosys on your RV32I-Lite design and produce a synthesis statistics report:

```
yosys -p "synth_gowin -top cpu_top; stat" your_design.v
```

(Adjust the top module name and filename for your project.)

In the `stat` output, find the line that reads `DFF` and the line that reads `LUT4`. Write both numbers in your Toolchain Diary.

Now compare: the MOS 6502 used 3,510 physical transistors to build its entire CPU including ALU, registers, instruction decoder, and bus interface. How many DFF cells did your synthesis produce just for the register file? Each FPGA DFF cell corresponds to one bit of flip-flop storage, realized in roughly 20-30 physical CMOS transistors inside the 55nm FPGA fabric.

Exercise 4: Ratio comparison

Goal: build intuition for how abstraction layers scale.

From Exercise 3, you have your DFF count and your LUT4 count. From this handout, you know:

- MOS 6502 total transistors: ~3,510
- Z80 total transistors: ~8,500
- Typical FPGA DFF cell transistor budget: ~20-30 transistors
- Typical FPGA LUT4 cell transistor budget: ~50-80 transistors (16 SRAM bits + 4:1 MUX tree)

Calculate an approximate transistor budget for your synthesized design. How does it compare to the 6502? To the Z80?

The answer is not meant to be a precise number -- FPGA fabric has additional routing and configuration transistors not counted per-cell. The point is to build intuition for the scale difference between a 1975 chip made of a few thousand transistors and a modern FPGA that configures billions of transistors to emulate the same logic.

Licensing and attribution

All sources in this handout are open-license or freely available for educational citation:

- **righto.com articles** by Ken Shirriff: cited with links and paraphrase; no images reproduced. Ken Shirriff's work does not carry an explicit open license; do not reproduce his die-shot photographs without permission.
- **Visual6502 project** (https://github.com/trebonian/visual6502): CC-BY-SA. The Pulsed Logic Group project.
- **Masswerk 6502 emulator** (https://www.masswerk.at/6502/): GPL v2 or later.
- **Wikimedia Commons** die-shot photographs: CC-BY-SA or CC0 per individual image. Always check the individual image page before using.

Note on Wikichip: the brief for this handout listed Wikichip (en.wikichip.org) as a source. As of May 2026, Wikichip is unreachable (connection refused). It is not cited in this handout. If it comes back online, it is a worthwhile community die-shot archive for many modern CPUs.