

OTTO

Build Your Own Educational PDP-8 Clone Using an FPGA

by Giovanni Becattini

OTTO is a small FPGA-based quasi-replica of the PDP-8, designed to bring the logic and spirit of DEC's classic minicomputer back to life on inexpensive modern hardware.

This booklet is a practical companion to *PDP-8, the Minicomputer That Anticipated the Microprocessor*. Instead of retelling the full history of the machine, it follows the construction of OTTO: from simulation to FPGA implementation, from the front panel to the serial interface, from loaders and paper-tape emulation to the first programs running again.

It is not only a technical project, but also a hands-on tribute to a deceptively simple computer whose elegance can still be understood, rebuilt, and enjoyed today.



Giovanni (Gianni) Becattini, born in Florence in 1951, is an electronic engineer and pioneering entrepreneur. At the age of sixteen, he began collaborating with *CQ Elettronica*, Italy's leading magazine for amateur electronics.

In 1973, he designed the Child 8 microcomputer and founded General Processor, the first company in Italy to manufacture personal computers.

In 1998, he founded AEP Ticketing Solutions, which became one of the world's leading companies in electronic ticketing systems for public transport and is now part of the Modaxo Group.

The author of numerous publications, he has chosen to devote himself to "technical art books" to celebrate the immense cultural and technological heritage left by great companies such as Apple, Tektronix, Hewlett-Packard, Collins, Siemens, and others, and by their extraordinary products—which he not only documents, but also restores and enthusiastically collects.

Quacktech Editions
Technical Art Books
www.quacktech.it

Giovanni Becattini

otto

GIOVANNI BECATTINI – BUILD YOUR OWN EDUCATIONAL PDP-8 CLONE



Build Your Own
Educational PDP-8
Clone Using an FPGA

Quacktech Editions
Technical Art Books

OTTO

Build Your Own Educational
PDP-8 Clone Using an FPGA

Giovanni Becattini



References

The following references are used in this book:

- [TEO] Tektronix Epic Oscilloscopes – Elektor Books
- [7KS] Tektronix 7000 Series – Mainframes – Elektor Books
- [7KS] Tektronix 7000 Series – Plug-ins - Quacktech
- [TREG] Tektronix Oscilloscopes Restoration Guide – Elektor Books
- [HP1] Hewlett-Packard and its Voltmeters and Oscilloscopes – Quacktech
- [HP2] Hewlett-Packard Signal Sources – Quacktech
- [HP3] Hewlett-Packard Counters and Spectrum Analyzers – Quacktech
- [HP4] Hewlett-Packard Calculators, Computers and HP-IB – Quacktech
- [VRE] Vintage Radio Equipment – Elektor Book
- [MAC] Apple Macintosh – History, Engineering, and Restoration – Elektor Books
- [QT602] The Good Giant – Tektronix DSA 602A Oscilloscope – Quacktech
- [QT11KP] The Last Plug-ins – Tektronix 11000-Series Plug-ins – Quacktech
- [SCLI] Strumentazione Vintage – Edizioni C&C
- [QTCOL] Collins Classics – KWM-2, 30L-1 and 51S-1: History, Engineering, and Restoration – Quacktech Editions
- [QT324] Two Giants and a Shorty – Tektronix 323/324 - Quacktech
- [QTBYP] Become Your Own Publisher – Quacktech
- [QTHPSS] Hewlett-Packard Signal Sources – Quacktech
- [DAB] Digital but Analog – Quacktech
- [TG7] When Machines Learned to Write – Quacktech
- [PDP-8] DEC PDP-8 – The Minicomputer That Anticipated Microprocessors – Quacktech
- [OTTO] Build Your Own Educational PDP-8 Clone Using an FPGA – Quacktech

- All rights reserved. No part of this publication may be reproduced or stored in any form or by any means, electronic or mechanical, without the prior written permission of the Author.
- The Author has made every reasonable effort to ensure the accuracy of the information contained herein but assumes no responsibility for errors or omissions, or for any loss or damage resulting from the use of this material.
- Most photographs were taken by the Author; others derive from commercially licensed or Creative Commons sources, clearly indicated where applicable.
- AI-assisted content disclosure – This book contains AI-assisted content used for language revision and for the creation of original illustrative material. All ideas, technical content, structure, and final editorial decisions remain entirely the Author's own.
- Some illustrations in this book were generated with the assistance of artificial intelligence tools, using original prompts and concepts developed by the Author. These images serve a purely illustrative and metaphorical purpose and are not intended to depict or reference any existing copyrighted characters, brands, or proprietary visual styles.
- Apple, Hewlett-Packard, Tektronix, Teletype, and other trademarks or logos mentioned in this book are acknowledged and remain the property of their respective owners.
- © 2025, 2026 Giovanni Becattini, Florence, Italy.

Table of Content

Introduction	9	Global Definitions	39
The OTTO Project	10	The Simulation Testbench	39
This Booklet	10	The Top-Level Module	40
How OTTO Began	10	The CPU Control Logic	40
Why “OTTO”?	13	The Datapath	40
Why Talk About OTTO?	13	Main Components and 74-Series Models	41
Programmable Logic	14	Memory	41
The Dawn of the Digital Age	14	Running the Simulation	42
The Basic Idea	14	Source Files in the Clean Project	42
GALs	15	A PDP-8 Cross-assembler	44
FPGA Programmable Gate Arrays	18	Compiling PAL on macOS	45
FPGAs	18		
Why an FPGA?	21	Real Implementation	47
A Wonderfully Useless Project	21	FPGA Boards	48
Learning Verilog	21	Hardware for the OTTO Project	48
Simulation	22	FPGA Specifications	48
From Sequential to Parallel	22	Additional Components on the i5 Module	49
From 2013 to Today	24	DDR-SODIMM Connector	50
Latest Updates	25	Muse Lab Development Board	50
Results	26	Price	52
An Inspiring Design	26	Development Tools	54
Final Architecture	28	Preliminary Activities	54
		Development Environment	55
Preliminary Design	31	Installing OSS CAD Suite	55
Developing the Code	33	Verifying the USB Connection	56
A Different Approach	33	USB, JTAG, and Serial Console	56
Original Development Tool	33	Checking the Tool Chain	58
Mac: A Unix Heart	34	CMSIS-DAP Programmer and JTAG Chain	58
Terminal	34	First Test: Loading a Ready-Made Bitstream	58
Today's Development Tools	35	Second Test: A Verilog Project Written by Us	59
Installing Homebrew	36	Third Test: From FPGA to Mac	60
Installing Icarus Verilog	36	Fourth Test: Serial Reception and Echo	61
Generating Waveform Files	37	Modularizing the UART	62
Installing Surfer	37	Toward a PDP-8-Style Console	63
A Logical Laboratory	38	Final State of the Preliminary Tests	64
The OTTO Verilog Project	38	Common Commands	64
		What Is Next	64

Version Control, GIT	66	Building the Front Panel	81
Some Practical Notes	66	Front Panel Structure	81
Basic Commands	67	How-To and Practical Suggestions	81
Conclusions	69	Cutting	82
Flashing FPGA Configuration	70	Drilling	82
Making Configuration Permanent	70	Finishing	84
Misleading Messages	70	Other Advantages	84
Using the FLASH	71	A Peculiar Wiring	86
		A Different Approach	86
		Placing the Boards	86
		Advantages	88
		Interfaces	90
		Power Supply	90
		Improvements	92
		Display	92
		Memory Addresses	92
		Backward EXAMIN	92
		Hardware Breakpoint	93
		RAM Content	93
		Momentary Switches	93
		Pin Assignment	94
		Muse Board Pin Assignment	94
		The Need	97

Construction 73

Lazy Builders	74
Not a Complex Project	74
The Lazy Builder's Notes	75
Building OTTO	76
Basic Decisions	76
Enclosure	76
Choosing Components	78
Switch Choice	78
Display Choice	79
Annunciators	80



Although every effort has been made to ensure accuracy, occasional errors may still be present. If you discover any mistake, however small, I would be grateful if you let me know. Your feedback helps ensure that future readers receive an improved edition.

You can contact us on www.quacktech.it. Thank you for contributing to the growth of this work.

This Book

This booklet is intended as a companion to "PDP-8, the Minicomputer That Anticipated the Microprocessor". While that book tells the broader historical, architectural, and personal story of the PDP-8, the present volume focuses on a more practical subject: the construction of OTTO, a small quasi-replica of the PDP-8 implemented with an FPGA board.

It does not attempt to repeat the complete historical and architectural discussion already presented there. Those topics are recalled only when they are useful for understanding the design choices made here.

The aim of this booklet is more direct: to describe the OTTO project in a compact and practical form, showing how a simplified PDP-8-like computer can be designed, simulated, implemented in programmable logic, and finally brought to life on inexpensive, readily available hardware.

A few words of caution are appropriate. OTTO is not an industrial product, nor is it meant to be a formally verified or cycle-perfect recreation of a DEC computer. It was designed above all for curiosity, learning, and enjoyment. I am not a professional Verilog designer, and several parts of the project certainly reflect this. The implementation is intentionally simple, sometimes pragmatic, and probably improvable in many ways.

Nevertheless, OTTO has proved to be a surprisingly useful educational machine. It can execute real PDP-8 software, including FOCAL and BASIC, and it successfully passes many of the original DEC diagnostic tests. This does not mean that it should be considered a complete or fully compatible PDP-8 implementation. Its performance is modest, some peripherals are simplified or absent, and there may be programs or timing-dependent situations in which it behaves differently from the original hardware.

In that spirit, the reader should regard OTTO as a personal and experimental reconstruction: a working tribute to the PDP-8, not a substitute for it. If it helps someone understand how such a machine worked, or encourages them to build, modify, or improve their own version, then it has achieved its purpose.

OTTO

Build Your Own Educational
PDP-8 Clone Using an FPGA

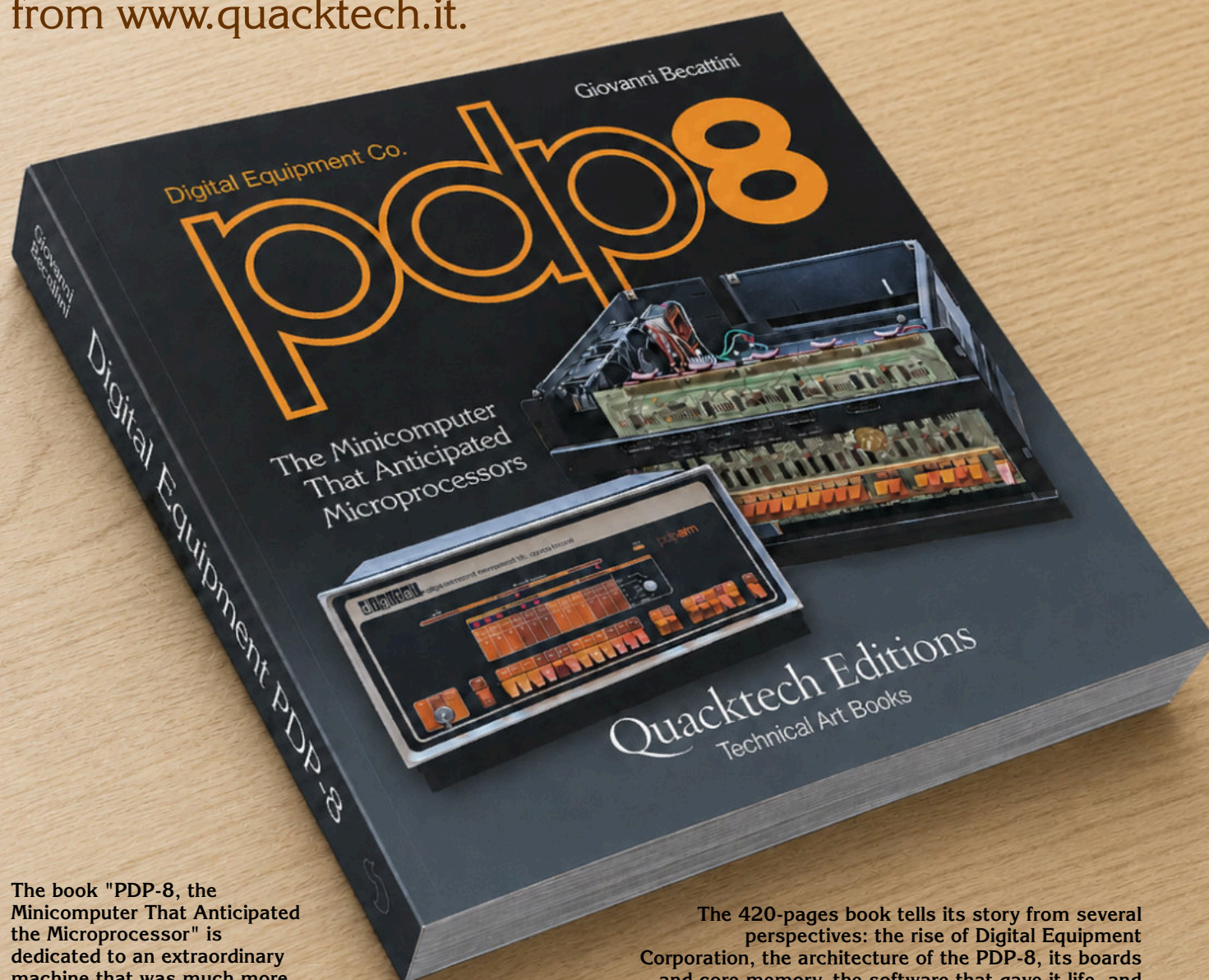
Introduction

Preliminary Design

Real Implementation

Construction

Another Technical Art Book
from www.quacktech.it.



The book "PDP-8, the Minicomputer That Anticipated the Microprocessor" is dedicated to an extraordinary machine that was much more than a small computer. It was one of the systems that helped move computation out of the data center and into laboratories, factories, instruments, and workshops.

The 420-pages book tells its story from several perspectives: the rise of Digital Equipment Corporation, the architecture of the PDP-8, its boards and core memory, the software that gave it life, and the restoration of a real PDP-8/M after decades of silence. It is also a personal journey into the logic of a deceptively simple machine, recreated once more in programmable logic to show that, even today, the elegance of the minicomputer age can still be understood, rebuilt, and enjoyed.

Section

Introduction

Let us pick up the thread of the discussion begun in *PDP-8, the Minicomputer That Anticipated the Microprocessor*, and complete it with the practical construction of OTTO.

The OTTO Project

This Booklet

This booklet is not an independent document, but should be regarded as a complement to my book *Digital Equipment PDP-8* [PDP-8]. It contains material that, for practical reasons, could not be included there, since that book was already rather “oversized” by itself.

Some passages are almost duplicated from the main book, in order to give this booklet a minimum of continuity and readability. The reader who has already read *Digital Equipment PDP-8* will therefore find a few familiar ideas, but here they are used as a starting point for a more detailed technical discussion of the **OTTO project**.

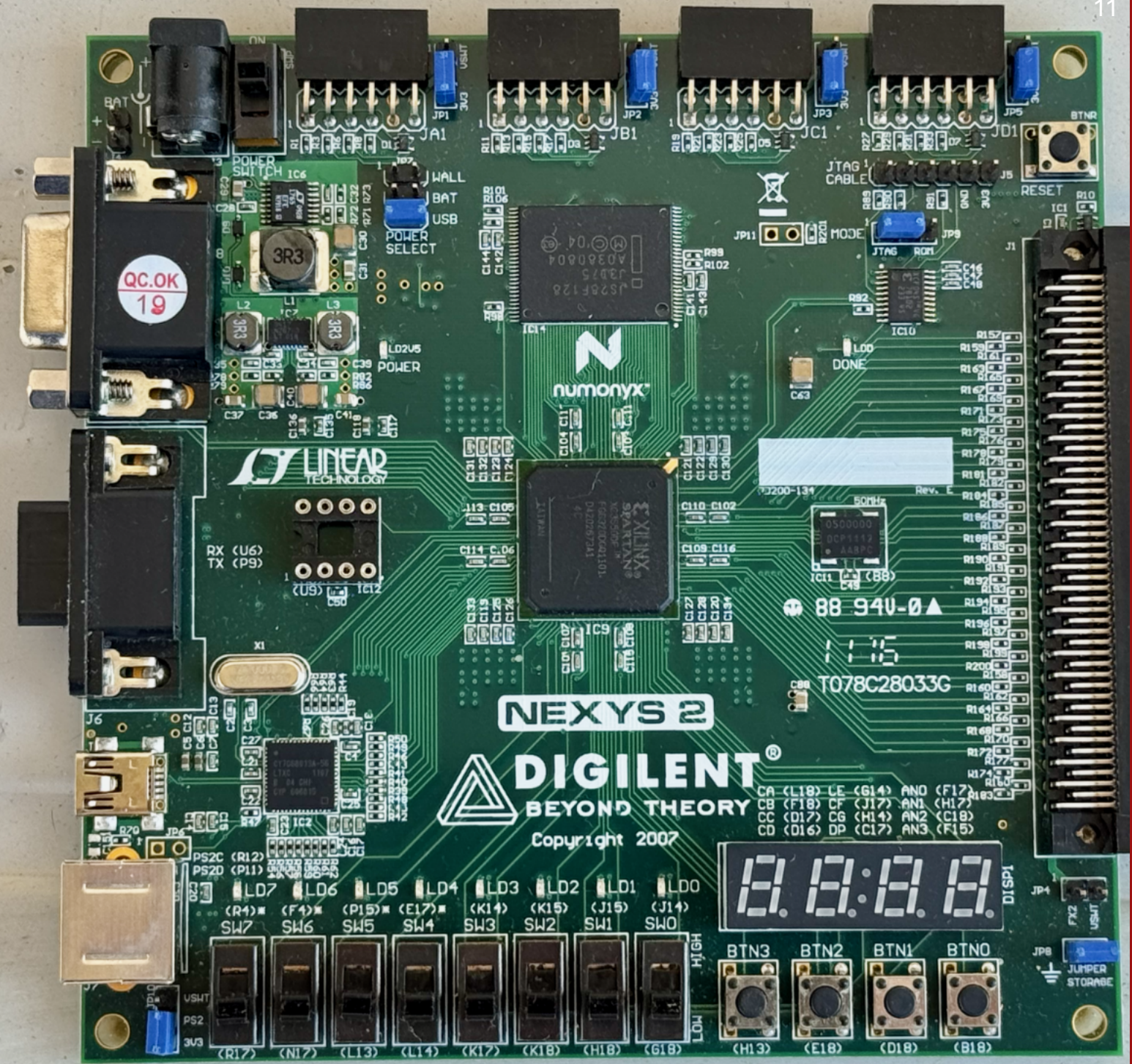
How OTTO Began

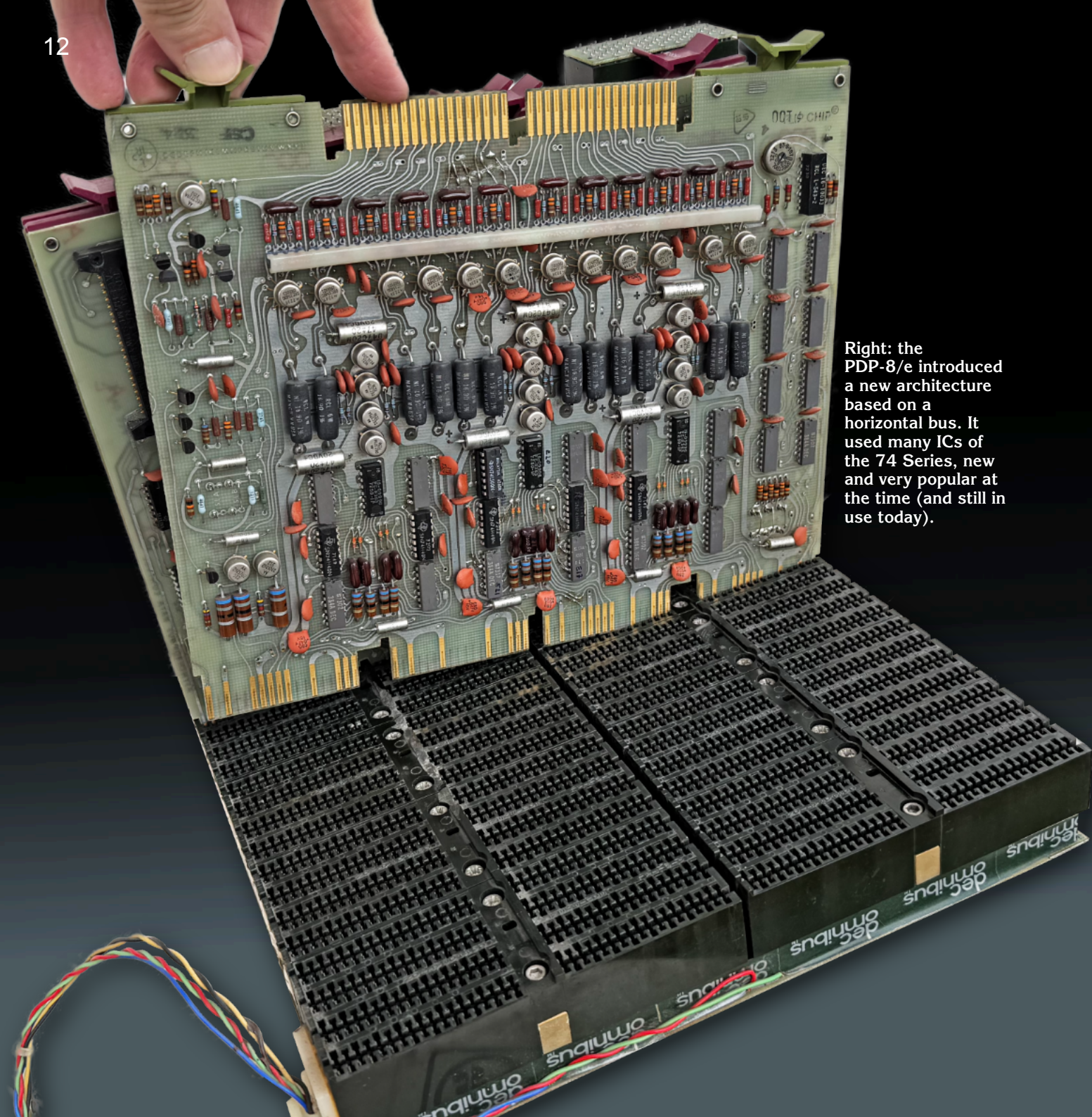
Several years ago, the usual little devil whom I know very well tempted me with the idea of learning a hardware description language.

The usual introductory FPGA examples did not appeal to me very much. Blinking LEDs are useful, of course, but only up to a certain point. I wanted a project difficult enough to force me to learn something real, but still small enough to remain within reach. So I asked myself: why not try to create a PDP-8 clone?

That was the beginning of the OTTO project: my attempt to reproduce a PDP-8-like computer inside an FPGA.

Right: the first board used as a development system for OTTO was the Digilent Nexys 2, equipped with a Spartan-3E 500 FPGA. It proved to be a good choice at the time, also thanks to its on-board switches and displays. Today, however, it feels rather old, and the original Xilinx development environment is too heavy for my taste. I therefore chose to move to open-source tools.





Right: the PDP-8/e introduced a new architecture based on a horizontal bus. It used many ICs of the 74 Series, new and very popular at the time (and still in use today).

Why "OTTO"?

"Otto" is a German personal name, but it also means "**eight**" in Italian, so the name was almost inevitable. But there was another reason too. Otto was also the name of my partner when I founded my company in 1998. Admittedly, he was a dog, but that did not prevent him from being the boss.

In what was then my company, many software tools and internal projects were given names derived from Otto, and some of them still retain that name almost thirty years later. Therefore, when I needed a name for my personal PDP-8 project, OTTO was the obvious choice. It was eight, it was mine, and it was slightly mad. Perfect.

Why Talk About OTTO?

The OTTO project was introduced in *Digital Equipment PDP-8* book [PDP-8] to help answer a number of questions:

- How is a processor like the PDP-8 actually built?
- How do the registers exchange data?
- How are instructions decoded?
- How does the accumulator receive its next value?
- How does the Program Counter advance?
- How does the machine cycle proceed from one state to the next?

The original DEC documentation is extremely valuable, especially for maintenance and repair. However, it was not written as a modern step-by-step explanation of how to design a computer from first principles. It often assumes that the reader already has the mental framework needed to understand the logic diagrams. This is where OTTO becomes useful.

By briefly retracing the design of OTTO, we can gradually build a PDP-8-like architecture from the inside. The intention is not to replace the original DEC documentation, but to approach the same ideas from a more constructive direction.

Instead of starting from hundreds of gates and trying to understand what they do, we start from the logical functions we need and see how a simplified PDP-8-compatible structure can be assembled step by step.

The intention was not to present OTTO as a perfect replica of a DEC machine. Although it passes most DEC instruction tests, It is not. It is a learning instrument: a simplified, understandable, FPGA-based reconstruction inspired by the PDP-8 architecture. This booklet aims to make the complete technical details of OTTO available, so that anyone interested may study it, modify it, or even reproduce it.

Programmable Logic

The Dawn of the Digital Age

The arrival of **digital integrated circuits**, and especially of the **74 Series**, marked the beginning of a tumultuous revolution that gradually relegated analog circuits to the only applications where they were really necessary. Everything changed quickly, and in a surprising way.

My first creation was a Nixie-tube **digital clock**, and I myself used to stare at it for a long time, so wonderful did it seem to me. As I have written many times, in the early 1970s almost no apparatus was able to provide indications in numerical form — and even fewer in textual form — with a few exceptions such as pinball score displays and automobile odometers.

Another novelty was the greater conceptual simplicity compared with analog electronics. With enough patience, one could eventually understand how many digital circuits worked, especially when they were built from simple logic gates. Among them, the **7400** was certainly the best-known star.

Unlike many other components, the 74 Series has survived the barriers of time. Of course, it is no longer used as extensively as it once was, when entire machines — including the PDP-8/m — could be built almost entirely around chips from that family. Yet it is still not unusual to find 74-series chips even in modern circuits.

The Basic Idea

In parallel with the continuing presence of the 74 Series, programmable logic devices began to appear as early as the 1970s.

The basic idea was very simple: a “prefabricated” logic network, like the one shown in the figure above, where you can recognize:

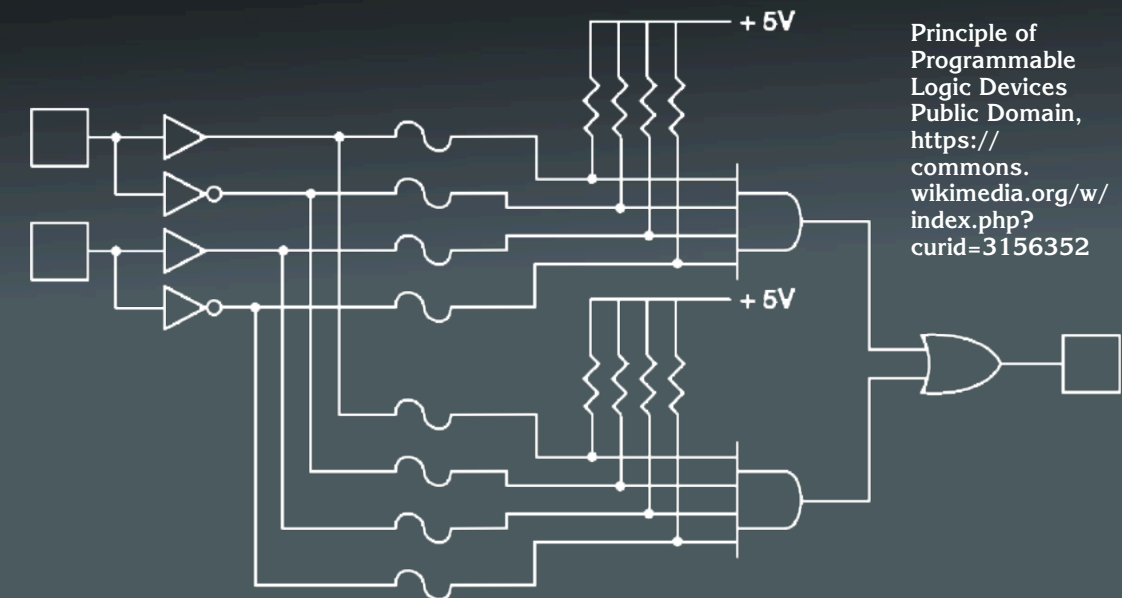
- two input pins;
- for each input, a buffer and an inverter;
- two four-input AND gates;
- an OR gate combining the outputs of the two AND gates;
- one output pin.

The elements drawn with the fuse symbol are indeed fuses. By blowing one or more of them, the desired logic function could be obtained. A special programmer performed the required blowing sequence.

The increasing scale of integration soon made it possible to obtain a larger number of gates and more complex logic, for example by adding a flip-flop on the output pin and providing feedback. This evolution, however, also created the need for more advanced development tools, capable of automatically determining which fuses had to be blown, starting from simple logic equations written by the user in a conventional language.

GALs

Personally, I did not have much sympathy for these devices and never used them. I did not like the idea of irreversible fuse blowing. Things changed, however, in 1985 with the arrival of **Generic Array Logic** devices, or **GALs**. Introduced by Lattice, they combined CMOS technology with EEPROM technology, making them reprogrammable and finally encouraging me to use them extensively, especially the 16V8 and 20V8 types, which could be programmed with tools not too different from



Principle of Programmable Logic Devices Public Domain, <https://commons.wikimedia.org/w/index.php?curid=3156352>

EPROM programmers.

Today, GALs are obviously obsolete, but quite recently I found myself needing to use them again, when I discovered that in my **LeCroy 9361** oscilloscope (de-

16 scribed in [SCLI]) they were used to enable optional functions, such as Fast Fourier Transform, already present in every unit but activated only after payment.

In theory, upgrading a LeCroy oscilloscope to enable all options is simple: the required file can easily be found on the Internet. In practice, however, I had to overcome several obstacles.

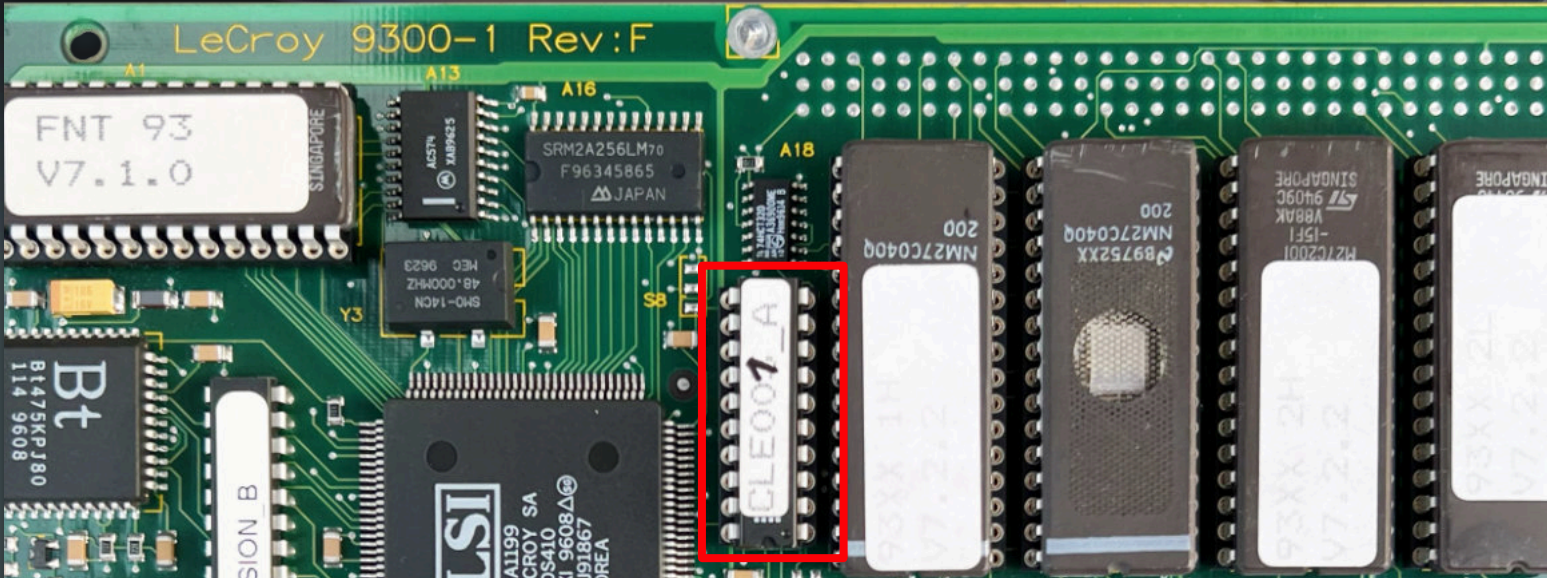
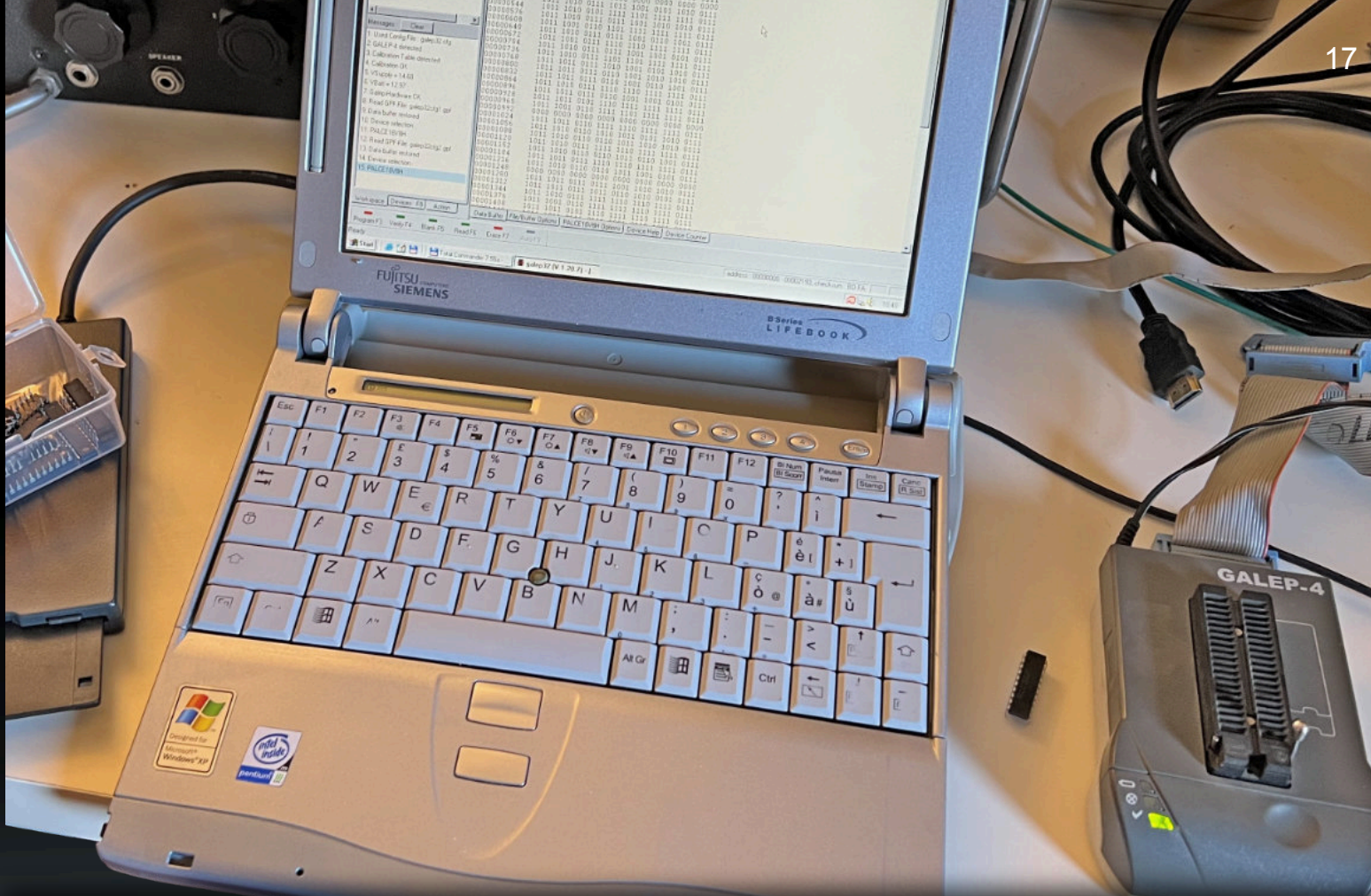
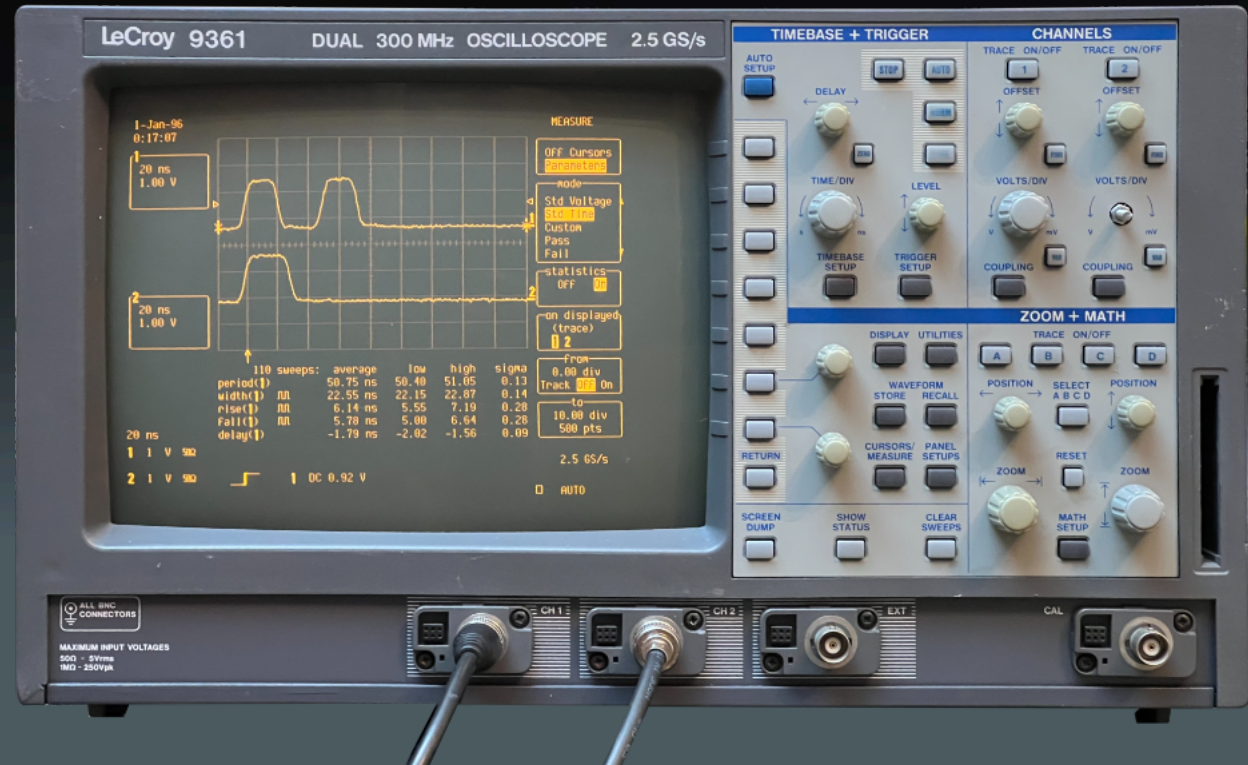
The first was finding a GAL 16V8. I was very lucky, because I found a few of them in a long-forgotten drawer at my company. Nobody remembered anymore what a GAL was...

The second problem was obtaining a programmer, but my friend Gianni Barsi generously gave me his GALEP-4, which can be seen in the photograph.

The third problem, however, was the most difficult: finding a PC with an old Centronics parallel port and a sufficiently dated version of Windows. I had to dig a little, but in the end I found my old Siemens laptop with its docking station.

Days of preparation... and only a few seconds for the actual programming!

Below: The LeCroy 9361 oscilloscope, which I rescued from the scrap bin and completely restored. The whole story is described in [SCLI]. Upper right: To program the GAL, I had to find a suitable programmer and dig an old Siemens PC out of my cellar. Lower right: The LeCroy CPU board, with the GAL outlined in red.



Field Programmable Gate Arrays

FPGAs

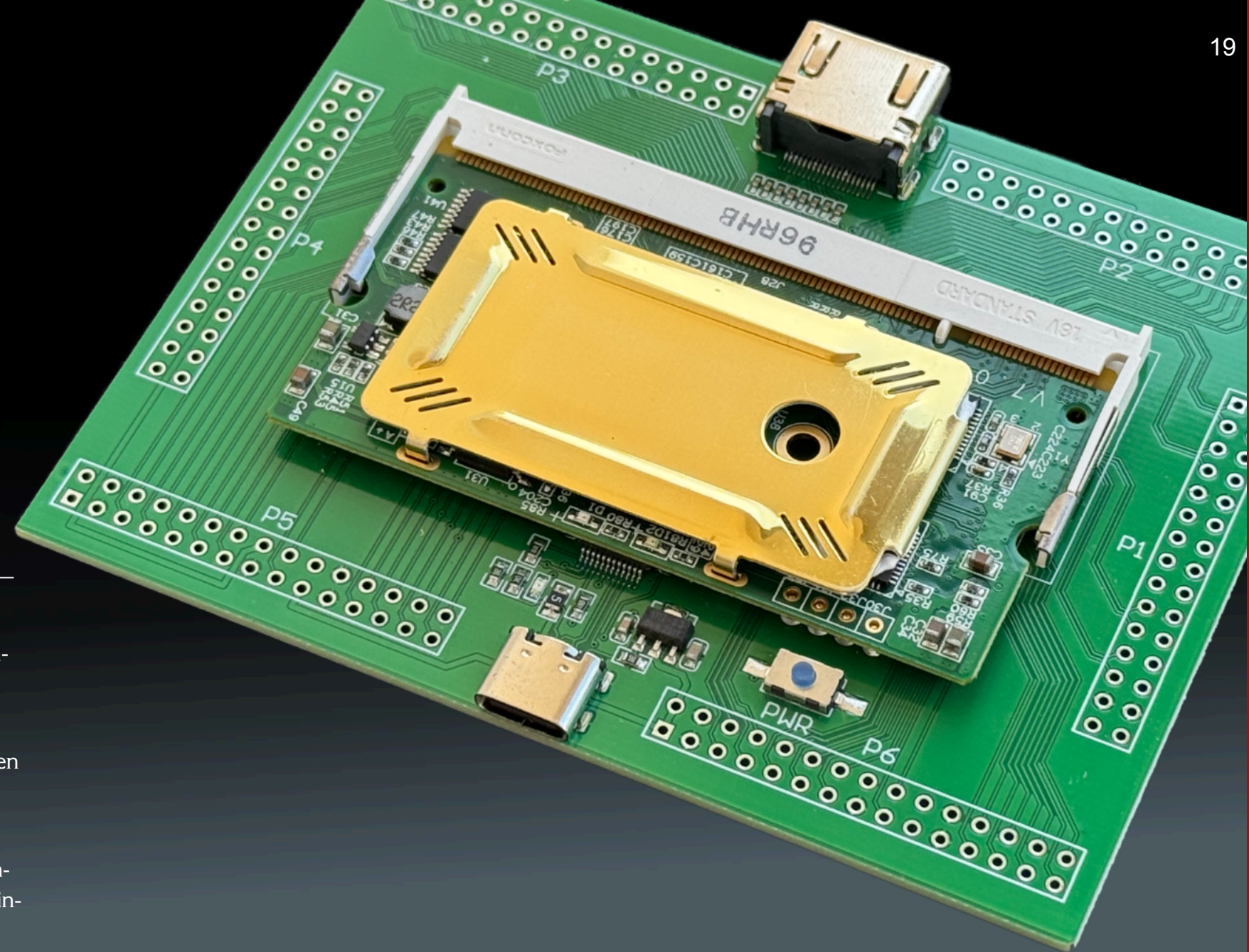
An FPGA, or Field-Programmable Gate Array, is an integrated circuit whose internal logic can be configured by the user after manufacturing. In a way, it may be regarded as a hypertrophic GAL — perhaps even its great-grandson...

In principle, one could obtain the same result by using a large number of 74-series logic ICs, connected together by wires or by a printed circuit board. Flip-flops, multiplexers, decoders, adders, gates, registers, and state machines could all be built in this traditional way.

An FPGA allows us to describe the same kind of logic in a hardware description language and then implement it inside a single programmable device.

This is very different from writing an emulator in software. A software emulator is a program that runs on a processor and imitates the behavior of another processor. An FPGA implementation, instead, is **actual logic**. Registers, buses, adders, and control signals exist as hardware structures inside the FPGA fabric, even if they are defined by a text file rather than by soldered ICs.

Of course, this distinction must not be exaggerated too poetically. An FPGA is still a modern programmable device, and its internal structure is very different from the discrete TTL logic of a real PDP-8/E or PDP-8/M. But conceptually, designing a processor in an FPGA is much closer to designing hardware than to writing a conventional software emulator.



Above: the new version of the OTTO project is based on the Colorlight i5 v7.0 module. Originally designed to drive LED display panels, this module has become popular as a low-cost FPGA development platform. At its heart is a Lattice ECP5 LFE5U-25F, a device with about 24,000 logic elements—far more than enough for a PDP-8-like computer. The module also includes SDRAM, SPI flash memory, clock circuitry, and Ethernet PHYs. Used together with a small base board, it becomes a convenient experimental platform, exposing the necessary power, programming, and I/O connections.



The Art of Digital Design – An Introduction to Top-Down Design by Franklin P. Prosser and David E. Winkel of Indiana University contain a detailed logical/functional description of the PDP-8.

Why an FPGA?

Could I have implemented OTTO in **software**, using a microcontroller or an ordinary computer?

Yes, but for someone like me, who had never used an FPGA before, that would certainly have been the easier path. After all, I still speak C++ more fluently than Italian.

But the attraction of recreating the PDP-8 as a **hardware structure**, rather than merely as a program that imitates it, was too strong. A software emulator would have reproduced the behavior of the machine. An FPGA implementation promised something different: the possibility of rebuilding its internal logic, or at least a simplified version of it.

So, quite naturally, I chose the more difficult route.

A Wonderfully Useless Project

I began developing OTTO several years ago, in 2013, during a business trip to Poland. At the time, I used ISE, the development environment provided by Xilinx, one of the major FPGA manufacturers, and a Digilent Nexys 2 board equipped with a Spartan-3E 500 FPGA.

The project was admittedly entirely **useless**. Naturally, this helped make it one of the **most exciting** and satisfying projects I have ever undertaken.

Learning Verilog

An FPGA can be defined by using a hardware description language (HDL). I chose **Verilog**, a HDL, used to model and describe electronic digital systems.

In practical terms, this means that, by writing Verilog source code, one can define a digital circuit almost as if one were building it from logic gates, flip-flops, registers, and wires, and then implement it inside a device such as an FPGA (Field Programmable Gate Array) simply by loading the appropriate configuration data.

A complete design flow requires much more than the language alone. Complex software tools, like those which we will see below, are used to **translate the Verilog source** files into the configuration bitstream required by the FPGA.

22 Simulation

The same tools, or closely related ones, also make it possible to **simulate** the design before confronting the real hardware. This is of enormous importance, because it allows errors to be identified and corrected at an early stage, when modifications are still easy and painless.

In simulation, the Verilog description is executed in a controlled virtual environment. We can apply clock pulses, reset signals, memory contents, switch settings, or even artificial input characters, and then observe how the internal signals of the machine evolve. Registers, buses, control states, instruction decoding, memory accesses, and I/O operations can all be inspected in a way that would be almost impossible on the real FPGA without adding special diagnostic logic.

This is particularly useful for a processor such as OTTO. A mistake in the Program Counter, in the accumulator path, in the instruction decoder, or in the sequence of fetch, defer, and execute states can be very difficult to understand once the design has been loaded into the FPGA. In simulation, instead, the same error can often be followed cycle by cycle, until its cause becomes evident.

Simulation does not eliminate the need for real hardware tests. Timing constraints, pin assignments, clock generation, serial communication, and the behavior of the actual FPGA board must still be verified on the physical device. Nevertheless, a good simulation phase greatly reduces the number of surprises when the design is finally brought to life in hardware.

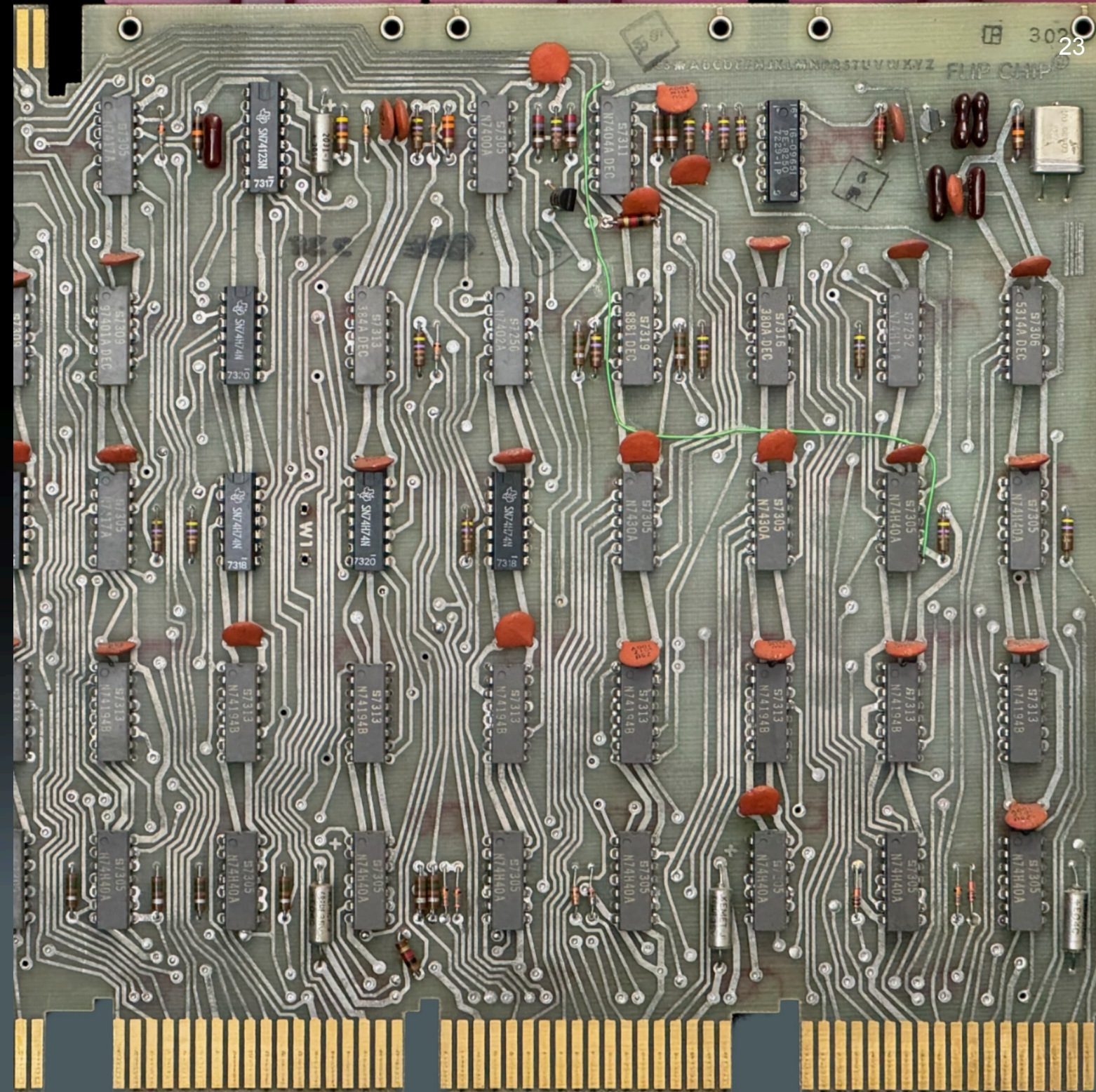
From Sequential to Parallel

For someone accustomed to conventional programming languages, moving to Verilog is not impossible, but it does require a **change of mindset**. One must constantly remind oneself that the two worlds are fundamentally **different**.

Programming languages are essentially **sequential**. They describe a sequence of operations to be performed one after another. Even when the program becomes complex, the basic idea remains that of a flow of instructions executed in time.

Verilog, like other hardware description languages, describes not a sequence of actions in the ordinary software sense, but **a circuit**. A circuit is a structure made of interconnected elements, each of

Right: A PDP-8 board like this, once a complete and tangible piece of computer logic, now represents only a small fraction of what can be implemented inside a modern FPGA. Note the wired correction, almost certainly added during production. In an FPGA, the same correction would require no soldering at all: only a change to the configuration file.



24 which continuously responds to its inputs according to its nature. In this sense, writing Verilog is often much closer to drawing a schematic than to writing a program.

This difference is crucial. A software program generally tells the processor what to do next. A Verilog design, by contrast, defines what hardware exists and how it behaves when signals change. The resulting circuit does not “run” line by line in the way a program does. Rather, it remains there, always present, always ready, and reacts whenever it is properly stimulated.

Of course, Verilog also allows one to describe circuits in a more behavioral way, and this can initially make it look similar to an ordinary programming language. But this is only a matter of description style. In the end, the synthesis tools must still transform that description into actual hardware structures: logic gates, flip-flops, multiplexers, memory blocks, and interconnections. The behavioral form is therefore not a different kind of reality, but simply a more abstract way of expressing the same circuit.

Verilog also includes constructs such as initial blocks, which may seem at first sight to reintroduce a sequential-programming perspective. In reality, their purpose is usually limited and specific. They can be used, for example, to define the initial contents of a memory block, to assign a starting value to a register in simulation, or to establish certain conditions at time zero. They do not change the fundamental nature of the language, which remains that of a hardware description system.

For this reason, the greatest difficulty in learning Verilog is often not syntactic but conceptual. The syntax itself is not particularly forbidding, especially for someone who already knows C-like languages. The real challenge lies in learning to think in terms of parallel structures, signal propagation, clocked registers, combinational paths, and hardware resources. Once this mental shift has taken place, the language becomes far more natural, and one begins to see the design not as a program to be executed, but as a machine to be built.

In that sense, Verilog proved to be not merely a technical tool, but also a way of forcing me to look at the PDP-8 at a deeper level. To describe a machine in Verilog, one must understand not only what it does, but how its internal logic is organized and how its operations arise from the cooperation of simple elements. This makes the exercise particularly fitting for a computer such as the PDP-8, whose elegance lies precisely in the clarity and economy of its internal structure.

From 2013 to Today

Those were years of very intense professional activity for me, and once that first phase of work was over, OTTO simply remained somewhere on a hard disk, dormant and almost forgotten.

But now, I thought, OTTO could be exactly what I needed for my [PDP-8] book. So why not 25 dust off the old project?

At first, the idea seemed almost hopeless. Where on earth could it have ended up? I began frantically searching through my old backups, until I finally found it again. Yes, it was really there.

But bringing it back to life seemed almost as hopeless. Back in those years I had several suppliers who, out of kindness and goodwill, had helped me with software and licensing issues. But now? The thought of facing ISE again, with all its complications, was entirely unappealing.

And yet, at this point, a small miracle occurred. Not only was I able to recover my original OTTO files, but I also received help from ChatGPT, which suggested a set of free, open-source tools suitable for the task. To my astonishment, in just one day I was able to simulate OTTO completely once again.

It felt almost unbelievable. But, as often happens, success immediately encouraged ambition. Once the old project had come back to life, it became difficult to stop there. Appetite, as the saying goes, comes with eating. And so I decided to push the project further.

On these pages, you will find the result of that decision. All the files and information needed to replicate the project, including the less obvious and more troublesome details, are contained in this booklet or available free of charge on www.quacktech.it.

Latest Updates

I don't exactly remember how it went, but I left something incomplete, and this was a good opportunity to finish the work, at least to a certain extent. In particular, the IOT processor instructions had not been implemented and also the console peripheral (teleprinter and keyboard) was not completely compliant with the original specification. For example:

- 6000 SKON Skip if Interrupt System On. The state of the interrupt enable flip-flop is tested. If this flip-flop is set, indicating that the interrupt system is enabled, the PC is incremented to skip the next sequential instruction and the interrupt system is turned off in the same manner as by an IOF instruction.
- 6001 ION Interrupt Turn On.
- 6002 IOF Interrupt Turn Off.
- 6003 SRQ Skip on Interrupt Request. If the interrupt request is asserted, indicating that one or more devices are requesting a program interrupt, the PC is incremented to skip the next sequential instruction.

- 6004 GTF Get Flags.
- 6005 RTF Restore Flags.
- 6006 SGT Skip if Greater Than. If the Greater Than Flag is set, the PC is incremented to skip the next sequential instruction.
- 6007 CAF Clear All Flags. AC and Link are cleared. Clears all I/O device flags and motion control flip-flops, and sets the interrupt enable flip-flop in each peripheral device.

In addition, the complete implementation of the **Reader Enable flag** of the M8650 board has been added, to allow the complete emulation of the serial communication line.

A complete round of testing has been carried out, and many bugs have been fixed, particularly in the memory extension logic, where some functions had not yet been fully implemented.

Results

The resulting OTTO computer is logically similar to a PDP-8/e complete with 16 Kword RAM, KM8-E memory extension management, and KL8-E Console Teleprinter Control (M8650). The resulting machine has been tested with original DEC instruction test programs and is capable of running the FO-CAL programming language, although much more slowly than the original.

An Inspiring Design

To conclude this introduction, we must also say that OTTO is probably only the latest in a long series of attempts to reproduce a PDP-8. Countless designs have been inspired by the PDP-8, or even quite openly derived from it, not only in its own time, when a compatible machine could still have had a practical use and a commercial advantage.

Some of these machines used transistors; others used integrated circuits; others still were based on single-chip PDP-8-compatible processors, such as the Intersil IM6100 or IM6120. Some reproduced the complete machine, while others borrowed only part of the idea. But all of them testify to the same fact: the PDP-8 was an architecture that people could understand, reproduce, adapt, and sometimes even improve.

More recently, many enthusiasts have tried to recreate the glorious DEC computer in new forms.



Some have written software emulators; others have built FPGA implementations, educational reconstructions, or small hardware replicas. This is hardly surprising. A PDP-8 is simple enough to be understood by one person, but rich enough to remain fascinating. That is a rare combination.

The OTTO project is certainly not the best PDP-8 reimplementaion, nor does it pretend to be. Its purpose is mainly educational, with a good dose of personal amusement. It should therefore be considered and evaluated in that spirit: not as an industrial clone, nor as a perfect replica of a DEC machine, but as a way to understand, step by step, how a small computer can be built.

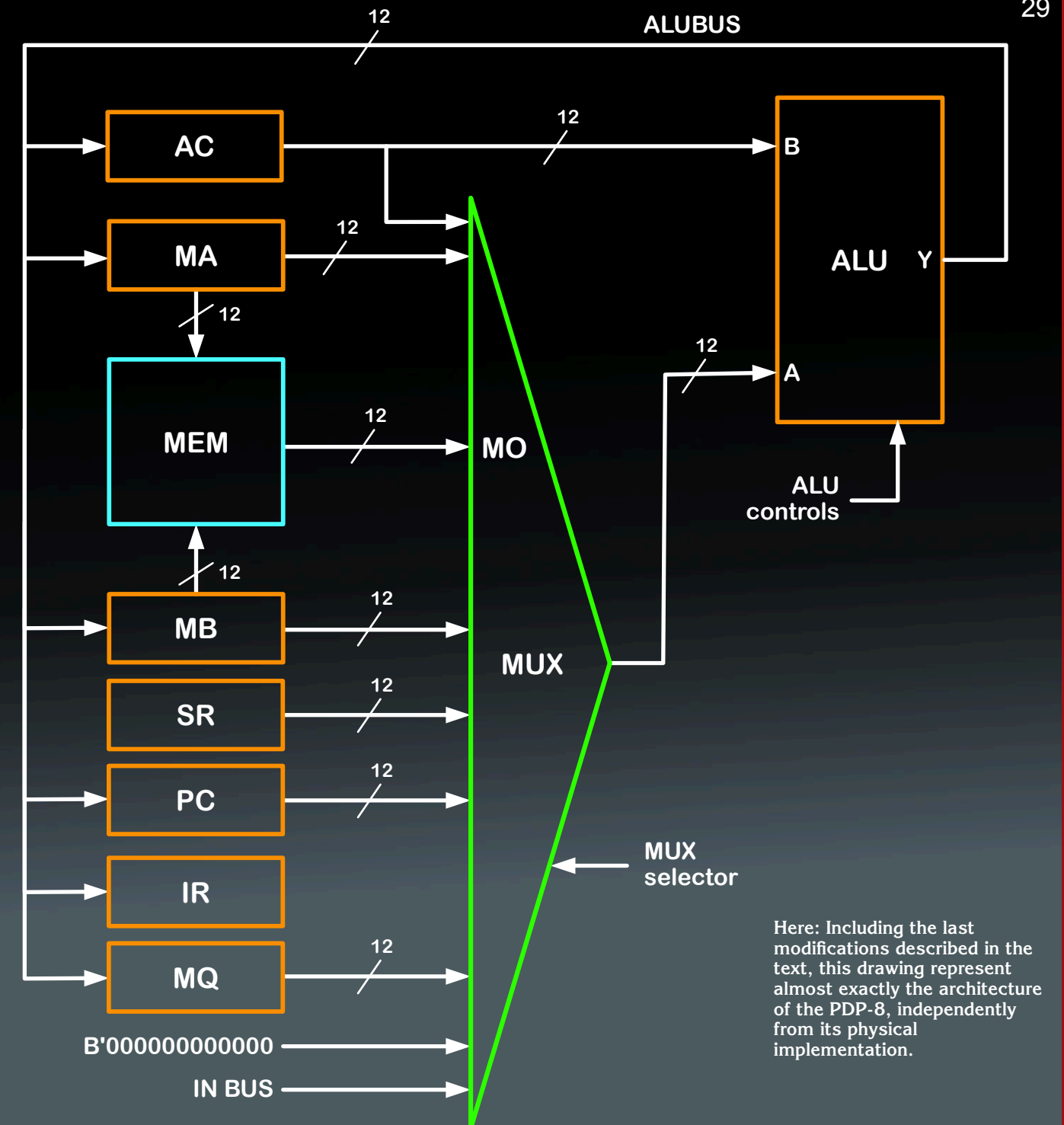
Final Architecture

Eventually, the proposed machine does not require complicated structures. In the final implementation, OTTO uses just 6 registers, a few flip-flops, and a 10-input multiplexer, as shown in figure 5:

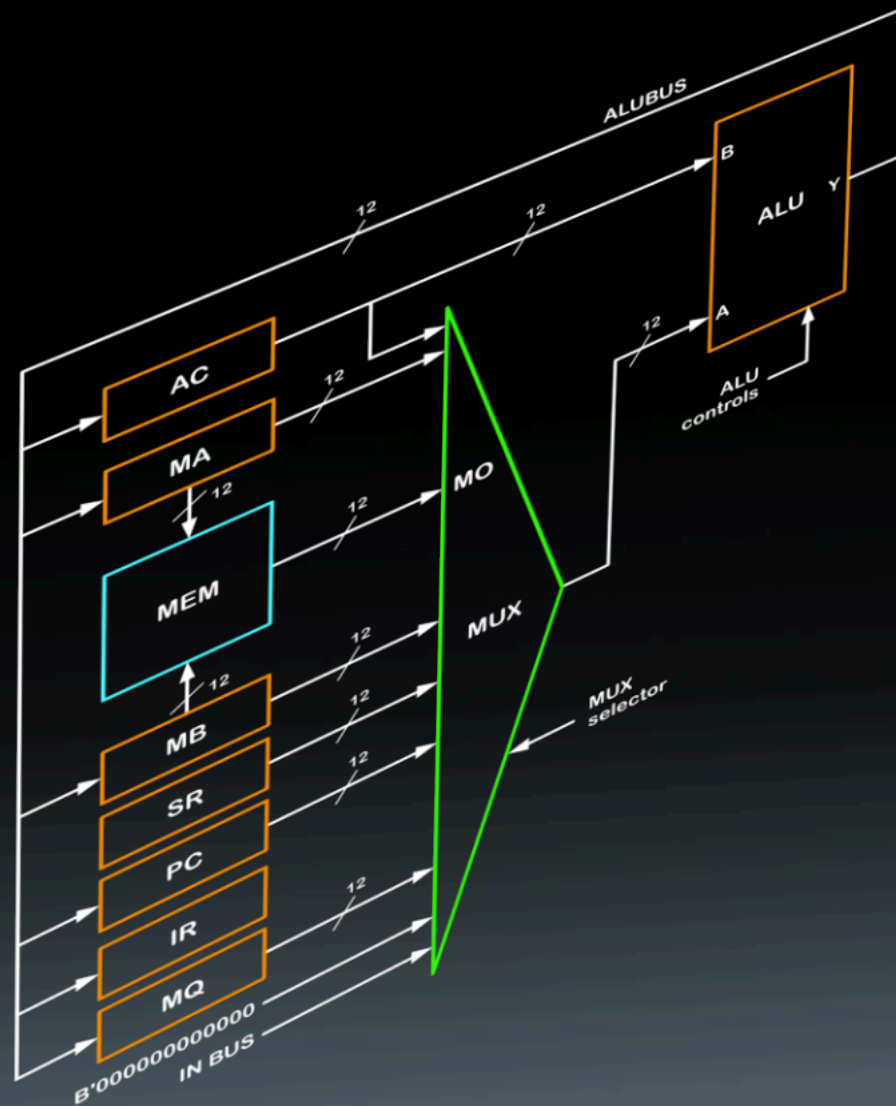
- PC — Program Counter
- MB — Memory Buffer, used for data to and from memory
- MA — Memory Address
- AC — Accumulator
- a fixed 0 value (B'000000000000)
- SWREG — Switch Register, corresponding to the front-panel switches
- MO — Memory Output
- IR — Instruction Register
- IN — used by the TTY peripheral (serial port)
- MQ — Multiply Quotient Register, associated with the optional KE8-E Extended Arithmetic Element, but present also in the basic system, where it is often used by programs as a temporary register

The PDP-8 architecture can perform its basic operations by moving data step by step through a small number of registers, under the guidance of the control logic. Somewhat slow by later standards, but also extremely simple and economical.

| For more information refer to the [PDP-8] book, where the original PDP-8 and OTTO are described in detail.



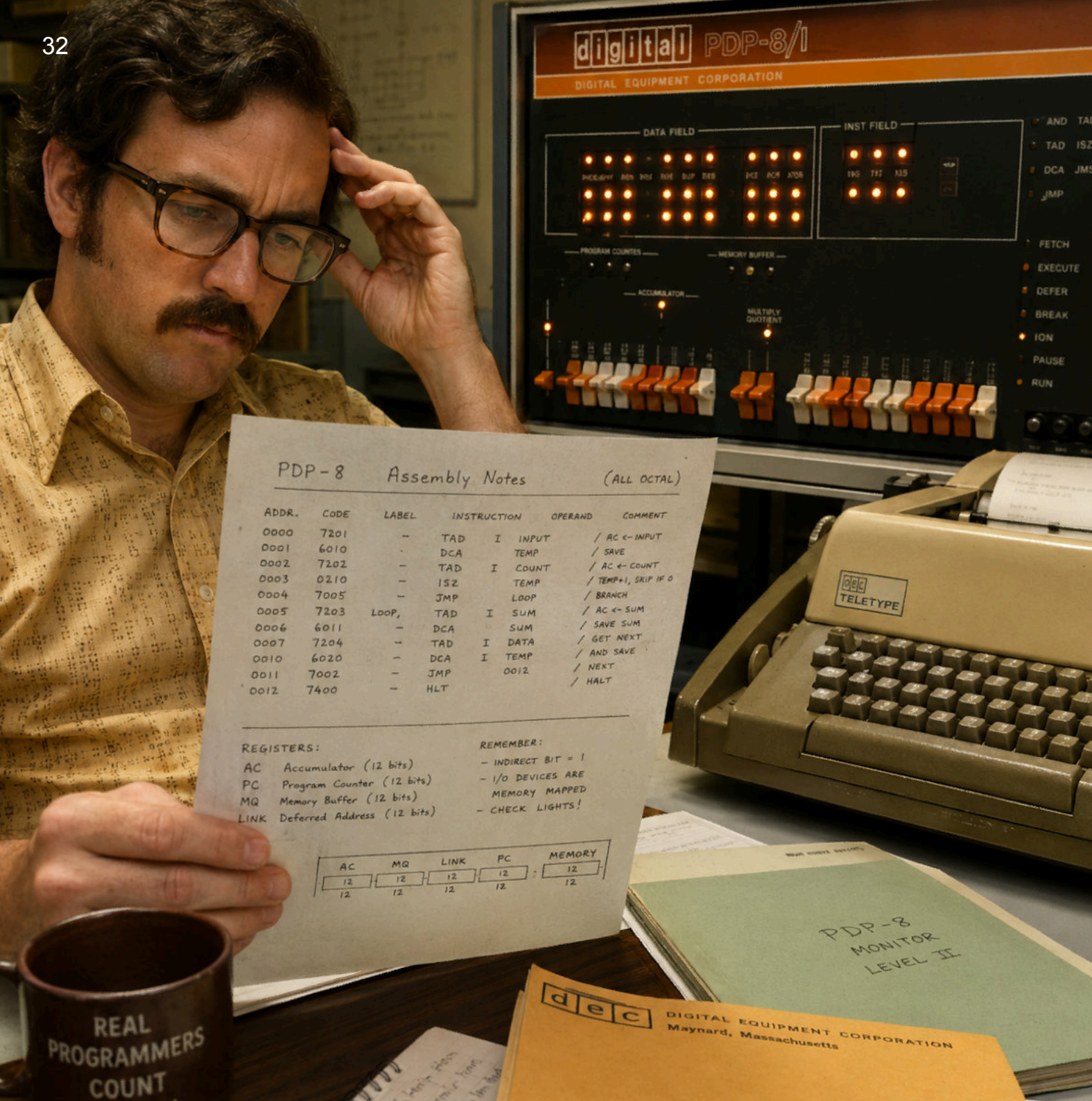
Here: Including the last modifications described in the text, this drawing represent almost exactly the architecture of the PDP-8, independently from its physical implementation.



Section

Preliminary Design

This chapter mainly covers the development and simulation tools used to bring the project to its first working stage, before it was ported to the real hardware.



PDP-8 Assembly Notes (ALL OCTAL)

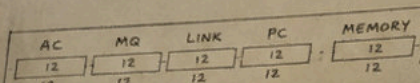
ADDR.	CODE	LABEL	INSTRUCTION	OPERAND	COMMENT
0000	7201	--	TAD I	INPUT	/ AC ← INPUT
0001	6010	--	DCA	TEMP	/ SAVE
0002	7202	--	TAD I	COUNT	/ AC ← COUNT
0003	0210	--	ISZ	TEMP	/ TEMP+1, SKIP IF 0
0004	7005	--	JMP	LOOP	/ BRANCH
0005	7203	LOOP	TAD I	SUM	/ AC ← SUM
0006	6011	--	DCA	SUM	/ SAVE SUM
0007	7204	--	TAD I	DATA	/ GET NEXT
0010	6020	--	DCA I	TEMP	/ AND SAVE
0011	7002	--	JMP	0012	/ NEXT
0012	7400	--	HLT		/ HALT

REGISTERS:

AC Accumulator (12 bits)
 PC Program Counter (12 bits)
 MQ Memory Buffer (12 bits)
 LINK Deferred Address (12 bits)

REMEMBER:

- INDIRECT BIT = 1
 - I/O DEVICES ARE
 MEMORY MAPPED
 - CHECK LIGHTS!



Developing the Code

A Different Approach

Developing a physical board requires a design phase and then you usually wire the prototype or create a PCB, to be tested using several tools like an oscilloscope.

With a FPGA things are quite different: you write your code and then you normally want to simulate it with some software tool and analyze its behavior.

Original Development Tool

As mentioned, my original OTTO project was developed under **Xilinx ISE**. I must admit that I cannot say I was ever particularly fond of it. It was extremely powerful, no doubt, but also very complex, heavy, and demanding in terms of storage space.

Its licensing logic was cumbersome, and one was repeatedly forced to obtain new keys and authorizations. On top of that, changes introduced from time to time often created new complications instead of reducing them. My general impression was that I was using a development environment capable of doing one hundred thousand things, while I needed perhaps ten.

At the time, however, this was the natural route. The first experimental version of OTTO was developed for a Xilinx FPGA board, and ISE was the standard environment for that family of devices. It worked, and it allowed the project to come to life, but it was certainly not a light or especially pleasant tool for a small personal experiment.

One of the reasons I like macOS is that it is based on a true Unix system—not merely inspired by Unix, but derived from it, line by line and process by process.

In 2007, Apple obtained official UNIX® certification from The Open Group, making macOS one of the very few consumer operating systems to carry that designation. Beneath its polished interface, the Mac speaks the same language as servers and mainframes that have been running Unix systems for decades.

This Unix heart also means that many of the practices and suggestions discussed in this book apply equally well to other Unix-like systems, such as Linux. Anyhow, most of what is presented has a corresponding Windows implementation.

Terminal

Terminal is the Unix command-line user interface. I know: at first glance it may feel obsolete and may even evoke memories of old DOS environments. Yet this impression is misleading.

In practice, the Terminal provides several decisive advantages:

- it gives access to capabilities that are otherwise unreachable through graphical interfaces;
- it allows batch execution of commands and fully repeatable workflows, eliminating the risk of forgetting a step or missing a critical checkbox;
- it grants access to powerful tools derived from mature and well-tested Unix and Linux libraries, enabling tasks that would otherwise require very expensive commercial software.

Quite often, commercial applications are nothing more than a thin graphical layer built on top of these same libraries. Using the command line directly ensures finer control, additional options, and access to features that graphical interfaces do not always expose.

When you open Terminal on macOS, the commands you type are interpreted by a shell, a text-based command-line interpreter. For many years, macOS used Bash (the Bourne Again Shell) as its default shell, inherited from the Unix and Linux world, although several alternatives have always been available.

For these reasons, learning a small set of basic commands is extremely useful. Commands such as `ls` (roughly equivalent to the old DOS `dir`) or `cd` (change directory) quickly become second nature. Built-in documentation is always available: typing `man` followed by a command name—for example `man ls`—opens the corresponding manual page.

In the following chapters, we will encounter many practical examples showing when and why the Terminal becomes an indispensable companion.

For those who wish to delve deeper into the topic, I recommend my book *"Become the Publish of Yourself"* available from www.quacktech.it.

Today's Development Tools

If in the time I started the OTTO project you could only use the development tools from the large FPGA manufacturers, today the situation is quite different. With the help of ChatGPT, I was able to set up a much lighter and entirely open-source toolchain, better suited to the spirit of this project.

For the logical design phase, before dealing with the real FPGA board, we mainly need two tools:

- **Icarus Verilog**, usually invoked with the command `iverilog`, to compile and simulate the Verilog design;
- **Surfer**, a waveform viewer used to inspect the signals produced during simulation; something like a virtual oscilloscope.

Strictly speaking, Icarus Verilog is not an FPGA implementation tool. It does not place gates inside a real FPGA, nor does it generate the final bitstream to be loaded into the device. Its role is different, but extremely important: it allows us to verify the logic of the design before confronting the physical hardware.

In other words, Icarus Verilog lets us ask questions such as:

- Does the Program Counter increment correctly?
- Does the accumulator receive the expected value?
- Does the instruction decoder recognize the current opcode?
- Does the machine pass correctly from fetch to decode and execute?
- Are the control signals asserted in the right state?
- Does the simulated memory return the expected data?

Surfer then allows us to observe these answers visually, by displaying the waveforms generated during simulation. This is particularly useful because a processor is not a single operation, but a sequence of events distributed over time. Seeing the clock, state signals, registers, buses, and control lines evolve cycle by cycle is often the simplest way to understand what is really happening.

Installing Homebrew

On macOS, the simplest way to install these tools is through **Homebrew**. If Homebrew is already installed, this can be checked with:

```
brew --version
```

If the command is not found, Homebrew must first be installed following the instructions on its official website. On a fresh Mac, it may also be necessary to install Apple's command line tools:

```
xcode-select --install
```

Once Homebrew is available, the installation of the Verilog simulator and waveform viewer is straightforward.

Installing Icarus Verilog

Icarus Verilog can be installed with:

```
brew install icarus-verilog
```

After installation, the presence of the compiler can be verified with:

```
which iverilog
iverilog -V
```

The first command shows where the executable has been installed; the second prints the version information. The simulator runtime, normally invoked as **vvp**, should also be available:

```
which vvp
vvp -V
```

The usual simulation flow is simple. The Verilog source files and the corresponding testbench are compiled by **iverilog**, producing an intermediate simulation file. This file is then executed by **vvp**.

```
iverilog -o otto_tb.vvp otto_tb.v otto.v
vvp otto_tb.vvp
```

Here **otto_tb.v** is the testbench, while **otto.v** represents the Verilog module, or group of modules, to be tested. In a real project there will usually be many more source files, but the principle remains the same.

Generating Waveform Files

Text output is useful, but it is not enough for debugging a processor. We also want to see how the internal signals evolve over time. For this reason, the testbench can generate a waveform file, typically in **VCD format**. This is usually done inside the testbench with instructions such as:

```
verilog
initial begin
    $dumpfile("otto.vcd");
    $dumpvars(0, otto_tb);
end
```

When the simulation is run, the file ``otto.vcd`` is created. It contains the time evolution of the selected signals: clocks, registers, buses, state variables, control signals, and so on.

This file can then be opened with a waveform viewer.

Installing Surfer

Surfer can be installed with:

```
brew install surfer
```

After installation, it can be checked with:

```
which surfer
surfer --version
```

A waveform file generated by the simulation can then be opened with:

```
surfer otto.vcd
```

Surfer displays the internal signals as timing diagrams. This makes it possible to follow the behavior of the simulated machine clock by clock. For example, one can observe the Program Counter being loaded or incremented, the memory address register selecting a location, the instruction register receiving an opcode, and the accumulator changing its value after an instruction has been executed.

For OTTO, this is invaluable. Many errors that would be very difficult to understand on the real FPGA become almost obvious when seen in a waveform viewer. A missing enable signal, a register loaded one clock too early, an incorrect multiplexer selection, or a wrongly decoded instruction can often be found by simply following the waveform sequence.

A Logical Laboratory

This simulation environment became a kind of logical laboratory. Before moving to the Colorlight i5 board and the real FPGA implementation, the processor could be exercised in a controlled virtual environment.

Of course, simulation does not prove everything. It does not verify the physical pin assignments, the real clock distribution, the behavior of the serial interface on the actual board, or the final timing closure of the FPGA implementation. Those aspects belong to the implementation phase.

Nevertheless, simulation is the right place to make the first mistakes. They are cheaper there, easier to understand, and usually much less frustrating. Only when the logical design behaves correctly in simulation does it make sense to move on to the physical FPGA.

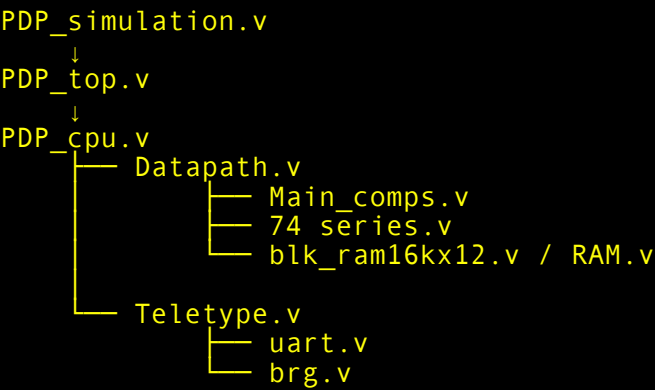
I wish to underline that Icarus Verilog is not part of the final FPGA implementation flow. It is used only for simulation. The same Verilog sources are later processed again by the synthesis and implementation tools, beginning with Yosys, to generate the actual FPGA bitstream.

The OTTO Verilog Project

After the preliminary discussion, we can now look at the actual Verilog project. The version described here is the cleaned-up simulation project, rebuilt in a separate working directory from the original historical files. Its purpose is not yet to implement OTTO on the real FPGA, but to provide a clear logical model that can be compiled and simulated with Icarus Verilog.

The project is composed of a small number of Verilog source files, each corresponding to a reasonably distinct part of the machine. The general structure is hierarchical. At the top there is a simulation testbench, which instantiates the PDP-8 system. Below it, the top-level PDP module instantiates the CPU control logic, the datapath, memory, and the simulated Teletype interface.

In simplified form, the hierarchy is:



This structure reflects the original intention of OTTO: to build a PDP-8-like machine not as a monolithic block, but as a set of recognizable functional units.

Global Definitions

The file `defines.v` contains the global definitions used by the project. It defines simulation options, symbolic constants, multiplexer selections, ALU operations, register destinations, IOT codes, and other values shared by several modules.

Although it is not listed explicitly in the Icarus Verilog command line, it is included by the source files through Verilog ``include`` directives. Therefore, it must always be present in the working directory.

In this simulation version, `defines.v` also enables options such as memory initialization, shortened simulation timing, and an emulated keyboard. These features are useful for testing the processor without having to wait for realistic serial timings.

The Simulation Testbench

`PDP_simulation.v` is the simulation testbench. It is not part of the future FPGA hardware; rather, it provides the virtual environment in which the design is exercised.

It generates the clock, applies reset and control signals, starts the simulated machine, and produces the waveform file used by the waveform viewer. This is also the file that determines how long the simulation runs and which external conditions are applied to the PDP-8 model.

In other words, ``PDP_simulation.v`` plays the role that, in the real world, would be performed by the board, the switches, the operator, and perhaps a terminal connected to the machine.

`PDP_top.v` is the top-level module of the simulated PDP-8 system. It connects the CPU, memory, console interface, switches, LEDs, and other external signals into a single system.

This file still reflects the origins of the project on an older FPGA development board, where switches, pushbuttons, LEDs, seven-segment displays, and a serial line were directly available. For the future Colorlight i5 implementation, it will probably be preferable to create a new top-level module specifically adapted to the new board, while preserving the internal CPU and datapath logic.

The CPU Control Logic

`PDP_cpu.v` contains the main control logic of OTTO. This is where the processor decides what must happen at each step of the machine cycle.

It includes the logic that sequences the fetch, defer, and execute phases; decodes instructions; controls skips and interrupts; handles IOT operations; and generates the control signals required by the datapath.

Conceptually, this file represents the “brain” of the machine. The datapath contains the registers and data routes, but the control unit decides when each register must be loaded, which source must be selected, and which operation must be performed.

The Datapath

`Datapath.v` contains the main data-handling part of the processor. It includes the essential registers of the PDP-8-like architecture, such as the Program Counter, Memory Address register, Memory Buffer, Instruction Register, Accumulator, Link, and MQ register.

It also contains the internal buses, multiplexers, ALU connections, and memory interface. If ``PDP_cpu.v`` decides what should happen, ``Datapath.v`` is where it actually happens.

This separation between control logic and datapath is very useful from an educational point of view. It allows the processor to be understood as two cooperating parts: one that controls the sequence of operations, and one that carries out the data transfers and arithmetic or logical operations.

`Main_comps.v` contains a collection of intermediate-level components used by the datapath and control logic: registers, multiplexers, counters, flip-flops, ALU-related blocks, display helpers, and other small building blocks.

`74_series.v` contains Verilog models inspired by classic 74-series TTL integrated circuits, such as flip-flops, counters, multiplexers, and ALU components. This reflects the original educational spirit of the project, which tried to remain close to the kind of logic that might have been built from real integrated circuits.

Not all of these components are equally important in the final design, and some are inherited from earlier versions of the project. Nevertheless, they help preserve the link between the Verilog description and the traditional hardware logic from which the PDP-8 itself was made.

Memory

The memory subsystem is described by `blk_ram16kx12.v` and `RAM.v`.

`blk_ram16kx12.v` provides the interface used by the PDP-8 datapath. It presents memory as a 12-bit-wide storage system, suitable for a PDP-8-like machine.

`RAM.v` is the actual Verilog memory model used during simulation. It implements the memory array and, in the simulation version, allows the initial memory contents to be loaded from the file `default.mem`.

The file `default.mem` is therefore not Verilog code, but a memory image. It contains the contents that are loaded into the simulated PDP-8 memory before execution begins. This makes it possible to start the machine with a predefined program already present in memory, such as FOCAL or a test routine.

Teletype and Serial Interface

`Teletype.v` implements the simulated console interface. It provides the logic corresponding, in simplified form, to the keyboard and printer side of a PDP-8 serial interface.

It handles IOT operations related to character input and output, manages receiver and transmitter flags, and connects the processor to the lower-level UART logic.

42 The files `uart.v` and `brg.v` support this simulated serial interface. `uart.v` implements the asynchronous serial transmitter and receiver, while `brg.v` provides the baud-rate timing. During simulation, the timing can be shortened so that serial input and output do not make the simulation unnecessarily slow.

This Teletype subsystem is especially useful because it allows the simulated PDP-8 to interact with programs that expect a console terminal.

Running the Simulation

The cleaned-up project can be compiled with Icarus Verilog using a command such as:

```
iverilog -g2012 -o pdp_sim \
  RAM.v \
  "74 series.v" \
  brg.v \
  uart.v \
  Teletype.v \
  blk_ram16kx12.v \
  Main_comps.v \
  Datapath.v \
  PDP_cpu.v \
  PDP_top.v \
  PDP_simulation.v
```

The simulation is then executed with:

```
vvp pdp_sim | tee sim.log
```

The `tee` command is convenient because it shows the simulation output on the terminal while also saving it to the file `sim.log`.

During execution, the testbench also generates a waveform file, normally `pdp8.vcd`, which can be opened with Surfer:

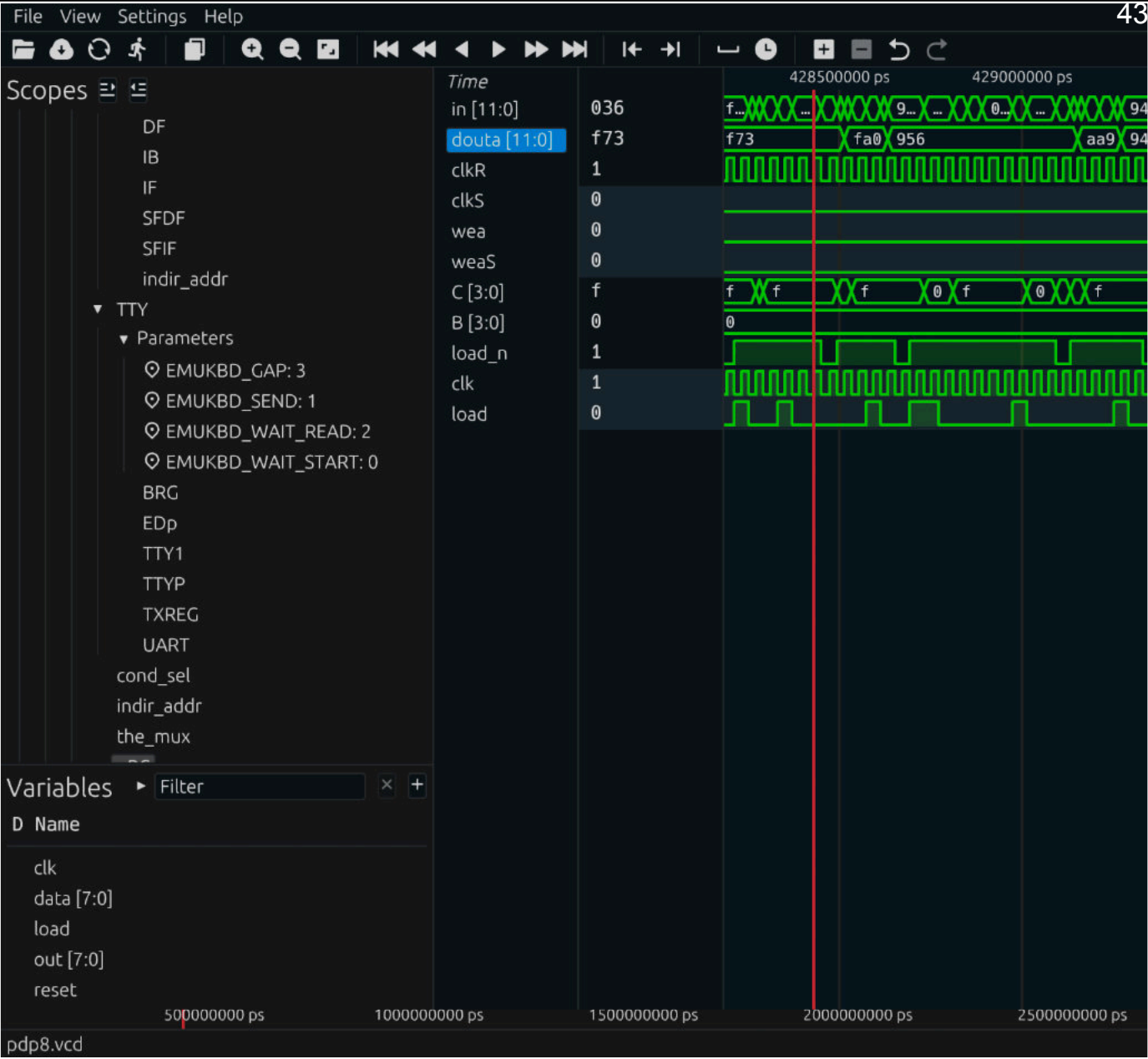
```
surfer pdp8.vcd
```

This allows the internal operation of the simulated machine to be inspected cycle by cycle.

Source Files in the Clean Project

The clean simulation directory therefore contains the following essential files:

```
defines.v
RAM.v
74 series.v
brg.v
```



Above: Surfer is a waveform viewer used to inspect the signals produced during simulation. In practice, it behaves like a virtual oscilloscope, allowing you to follow the evolution of internal signals that would otherwise remain completely invisible. This is extremely useful when debugging the more complex cases, where simply looking at the program output is not enough to understand what is really happening inside the simulated OTTO.

```
uart.v
Teletype.v
blk_ram16kx12.v
Main_comps.v
Datapath.v
PDP_cpu.v
PDP_top.v
PDP_simulation.v
default.mem
run_sim.sh
```

The shell script `run_sim.sh` is not part of the Verilog design, but simply automates the compilation and simulation commands.

This organization is intentionally modest. It is not yet a polished FPGA project, but a working logical laboratory. Its purpose is to make the internal behavior of OTTO visible, understandable, and testable before the design is adapted to the Colorlight i5 board and transformed into a real FPGA bitstream.

A PDP-8 Cross-assembler

Another useful tool, which will be precious for later tests, is a PDP-8 cross-assembler. Its purpose is to assemble short PDP-8 programs on a modern computer, producing binary or RIM-format output that can then be used by the simulator, or eventually loaded into the FPGA implementation.

Most of the test programs used during development are very short. However, even the smallest error in an opcode or address can waste a surprising amount of time. It is therefore much safer to let an assembler generate the correct machine code, rather than entering octal words by hand and hoping that no mistake has slipped in.

For this purpose I chose **PAL**, a two-pass PDP-8 PAL-like assembler written by **Douglas W. Jones**, with contributions by Rich Coon and Bernhard Baehr. It accepts a subset of the PAL-8 assembly language and is sufficiently compatible for the kind of small test programs needed here.

The assembler is not intended to replace the original DEC tools in every detail, but it is extremely useful for practical work. It allows us to write readable PDP-8 assembly source files, with labels and symbolic mnemonics, and then obtain the corresponding machine code automatically.

The program uses the following file-name extensions:

- `.pal` — source code;
- `.lst` — assembly listing;

- `.bin` — assembly output in DEC binary loader format;
- `.rim` — assembly output in DEC RIM loader format.

The command-line options include:

- `-d` — dump the symbol table at the end of assembly;
- `-r` — produce output in RIM format; the default is BIN format.

A typical use is:

```
./pal Testser_01.pal
```

This reads `Testser_01.pal` and produces the corresponding listing and binary output files.

For example, a small test program written in PAL assembly can be assembled on the Mac, inspected in the listing file, and then used as a memory image or as input for further conversion tools. This gives us a much more reliable starting point than manually preparing octal constants.

Compiling PAL on macOS

The original source is old C code and modern compilers are much stricter than the compilers for which it was originally written. On macOS, I compiled it with ``cc``, using the GNU89 dialect and disabling a few warnings that would otherwise stop or obscure the build:

```
cc -std=gnu89 \
    -Wall \
    -Wno-implicit-int \
    -Wno-implicit-function-declaration \
    -Wno-return-type \
    -Wno-missing-braces \
    -Wno-deprecated-non-prototype \
    -Wno-char-subscripts \
    pal_macos.c -o pal
```

After compilation, the executable can simply be kept in the working directory and invoked as:

```
./pal sourcefile.pal
```

or moved to a directory included in the shell search path, so that it can be used directly as the ``pal`` command.



Section

Real Implementation

Once the actual FPGA hardware became available, we switched to the final development tools and began preparing the code that would eventually run on the OTTO computer.

FPGA Boards

Hardware for the OTTO Project

The core of the OTTO project is a **Colorlight i5 Display Receiver Card**, hereinafter simply referred to as the **Colorlight i5**. It was originally designed as a controller module for LED display panels, but it has since become popular as an inexpensive FPGA development platform.

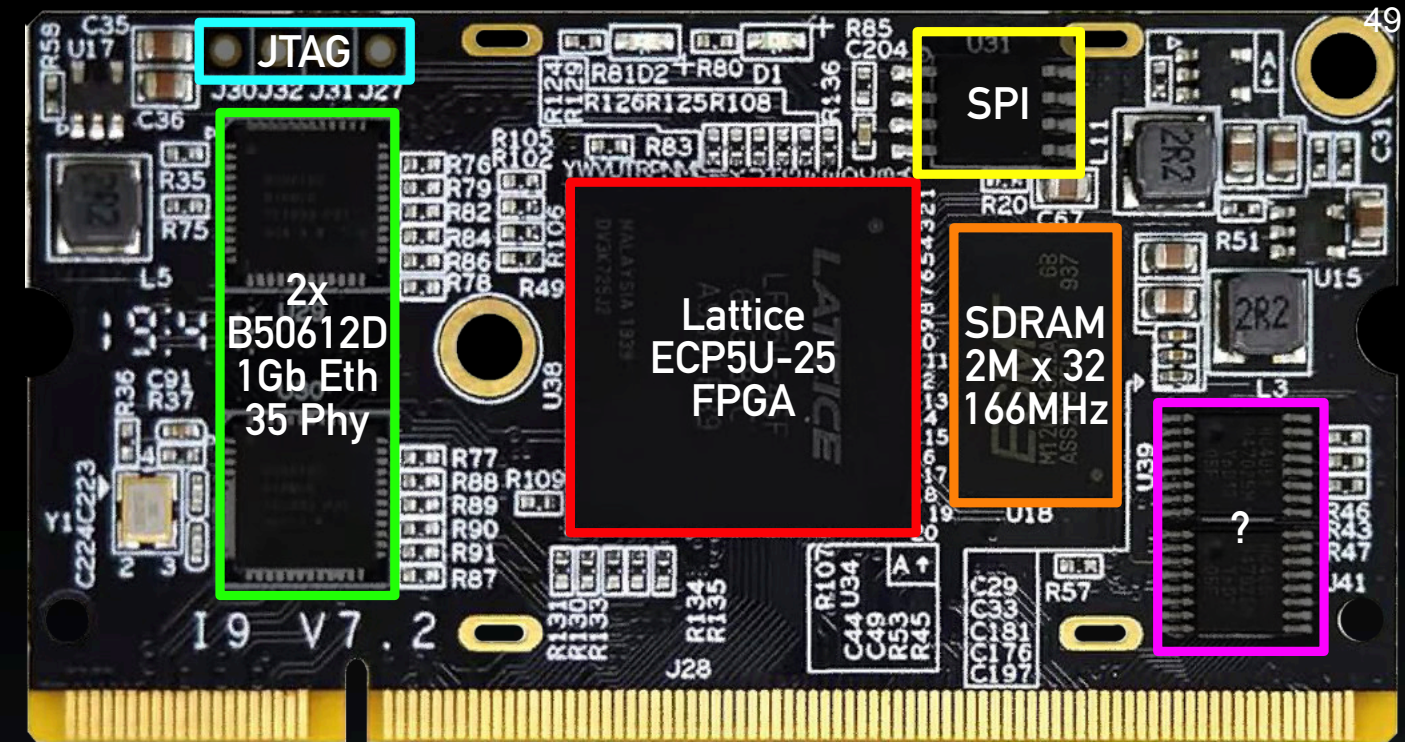
At its heart is a **Lattice ECP5 LFE5U-25 FPGA**, a device more than adequate for a didactical PDP-8 implementation. The module is remarkably compact, measuring only about 68×36 mm, and interfaces with the outside world through a DDR2 SODIMM connector with a proprietary pin assignment.

FPGA Specifications

The -25 version is one of the smaller members of the ECP5 family, but it still offers far more resources than are required for a small educational PDP-8:

- 24K LUTs
- 56 sysMEM block RAMs, 18 Kb each, for a total of about 126 KB of on-chip block RAM
- 194 Kb of distributed RAM
- 28 DSP blocks, each with an 18×18 multiplier and 36-bit accumulator
- 2 PLLs and 2 DLLs
- no SERDES channels
- 381-ball caBGA package

For a project such as OTTO, which aims to recreate a modest 12-bit minicomputer rather than a large modern system, this is already a luxurious amount of logic.



Above: Colorlight i5 Board

Additional Components on the i5 Module

In addition to the FPGA, the Colorlight i5 module includes several other components originally intended for its LED-display controller function:

- an EtronTech EM638325BK-6H SDRAM, organized as $2M \times 32$ bits, for a total of 8 MB;
- two Broadcom B50612D Gigabit Ethernet transceivers; the Ethernet magnetics and RJ45 connectors are not on the module and must be provided externally if Ethernet is to be used;
- a 16 Mbit Quad SPI serial flash device, commonly found as a GD25Q16-type part on these boards. According to the Lattice ECP5 sysCONFIG documentation, the maximum uncompressed bitstream size for this FPGA is about 5.42 Mbit, leaving more than 10 Mbit, or roughly 1.3 MB, potentially available for user data or application storage;
- a 25 MHz oscillator;
- two LEDs;
- the JTAG interface;
- three CD4051B 8-channel analog multiplexer/demultiplexer devices, whose exact original function on the LED receiver card is not especially relevant for our purposes;
- a small heat sink.

The JTAG signals are available on four test points on the module. They are not explicitly marked as a standard connector, but they are accessible and can be soldered if necessary. However, when the module is used with the Muse Lab development board described below, this is not required.

DDR-SODIMM Connector

The Colorlight i5 uses a DDR2 SODIMM-style connector, but not with a memory-module pinout. It is simply used as a compact high-density connector between the i5 module and an external carrier or development board.

The connector provides:

- a power input rail, approximately in the 3.3 V to 6 V range; the module contains its own voltage regulators, but regulated voltages are not brought back out to the connector;
- two groups of four differential pairs for the two Gigabit Ethernet PHYs;
- 106 general-purpose FPGA I/O pins;
- two RCV_BK1 / RCV_BK2 pins, apparently related to receiver backup power in the original LED-display application;
- a number of pins connected to the 8-channel analog multiplexers.

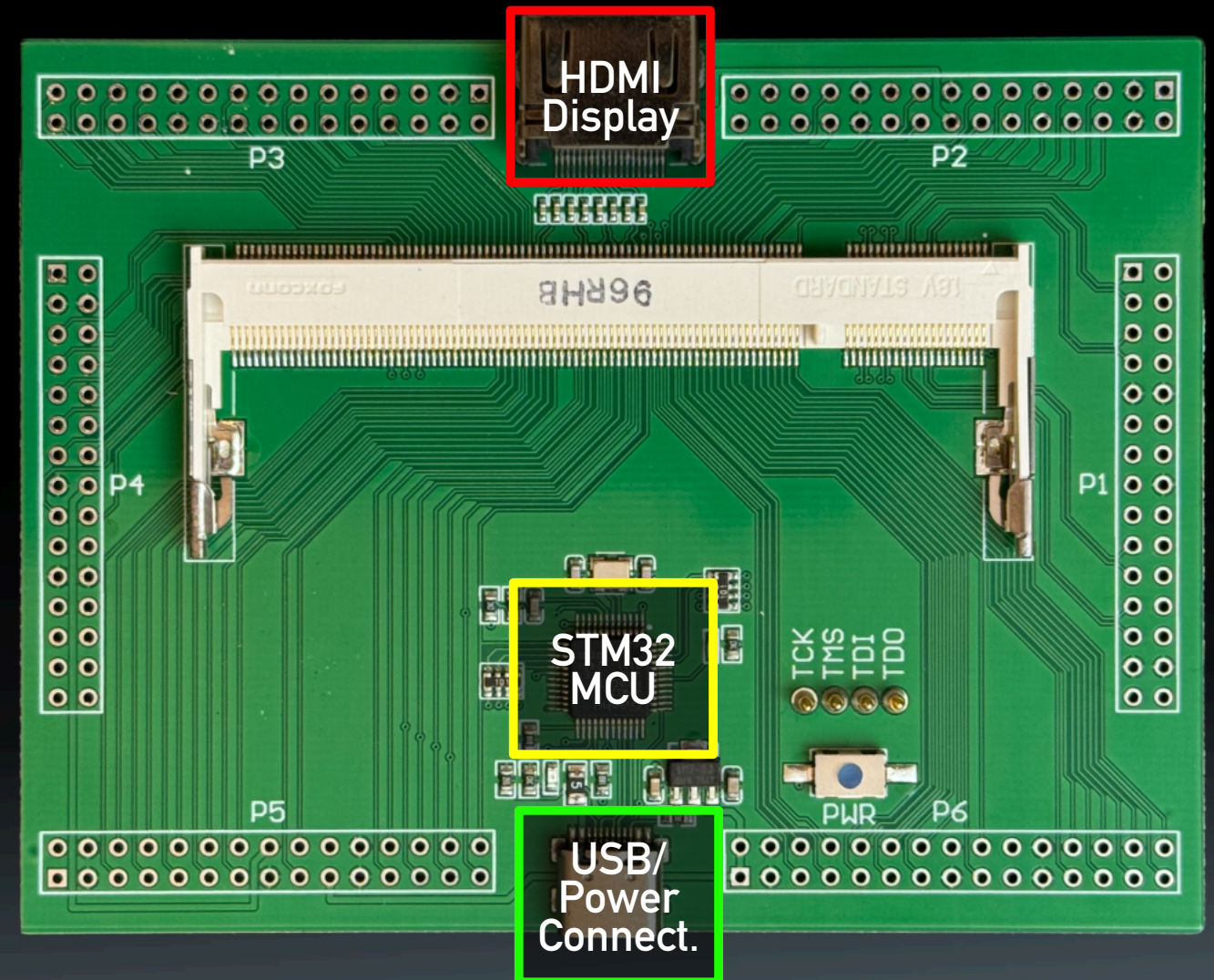
For the OTTO project, the most important feature is simply the large number of FPGA I/O pins made available through this connector.

Muse Lab Development Board

The practical use of the Colorlight i5 is greatly simplified by the Muse Lab development board, into which the i5 module is inserted. This distinction is important: the i5 module contains the FPGA and the original LED-controller circuitry, while the development board provides power, USB access, JTAG programming, convenient connectors, and a USB-to-serial interface.

The development board includes:

- a DDR-SODIMM socket for the i5 module;
- one 30-pin connector for the two Ethernet ports;
- four 30-pin connectors for general-purpose GPIOs. Each connector provides 20 GPIO signals, together with ground, 3.3 V, and 5 V pins;
- one additional 30-pin mixed-use connector for GPIOs and special functions. It has the same gen-



Above: The Muse carrier board makes the Colorlight i5 much easier to use. It provides convenient access to the FPGA pins and includes an STM32 microcontroller, which can be used to control and program the FPGA. It also creates a virtual serial interface for user applications, a feature that proved very useful during the first development and testing phases.

eral arrangement as the other GPIO connectors, but six of its GPIO pins are also assigned to special functions, notably UART and SPI. These signals are also routed to the microcontroller and can therefore be used as a convenient FPGA debug console;

- an HDMI connector, intended for video output only. It is not a complete general-purpose HDMI implementation, but it is sufficient to drive a monitor in simple FPGA video experiments;

- a USB-C port, used to power the development board, program the FPGA, and exchange FPGA UART console traffic with the host computer;
- an STM32 microcontroller running DAPLink/CMSIS-DAP firmware. This device acts as the bridge between the host computer and the FPGA, providing JTAG programming and a USB virtual serial port;
- a USB 5 V to 3.3 V regulator, an AMS1117-3.3, used to generate 3.3 V for the MCU and for the 3.3 V pins on the external connectors. The Colorlight i5 module itself is powered from the USB 5 V rail and uses its own onboard regulators;
- two small LEDs controlled by the microcontroller. These should not be confused with the two brighter LEDs on the i5 module itself. The red LED flickers when there is USB traffic, while the blue LED appears to remain on and is almost hidden underneath the module;
- a power button. When pressed, this interrupts power to the Colorlight i5 module, but not to the microcontroller. To power off the MCU itself, the USB cable must be unplugged;
- four JTAG pogo pins. Although the i5 module has accessible JTAG test points, these signals are not routed through the SODIMM connector. The development board solves this elegantly by contacting the test points with spring-loaded pogo pins, avoiding the need for soldering.

This last feature is especially convenient. It means that the FPGA can be programmed directly through the development board without modifying the i5 module.

Price

I bought the complete kit from Amazon for about 55 euros. It included:

- the Colorlight i5 module;
- the Muse Lab development board;
- a short USB cable for power and communication;
- six 90-degree pin headers, ready to be soldered to the PCB.

Buying directly from China would probably have saved something, but I preferred the additional safety offered by Amazon, especially in case of delivery problems or defective hardware. For this type of experimental work, the small additional cost seemed justified.

I may eventually replace the supplied right-angle headers with straight pins, so that the development board can be mounted more conveniently on a breadboard, or on a custom PCB designed specifically for the OTTO project.

Here: The keys of the original PDP-8 front panel. OTTO faithfully reproduces their functions and arrangement, while also adding a few new features made possible by the FPGA implementation.



Development Tools

Preliminary Activities

Before attempting the OTTO implementation, we need:

- to install the necessary tools;
- to verify that the FPGA board is correctly seen by the computer, can be programmed, and can communicate through a simple serial channel.

This avoids confusing possible problems in the PDP-8 Verilog design with much more elementary issues related to USB, JTAG, pin assignments, macOS permissions, or the development toolchain.

The purpose of the tests described in this section is to establish a safe and reproducible starting point:

- the Mac sees the board through USB;
- `ecpdap` sees the CMSIS-DAP programmer;
- the JTAG chain sees the ECP5 FPGA;
- a ready-made bitstream can be loaded;
- a small Verilog project written by us can be built and loaded;
- the FPGA can transmit characters to the Mac;
- the FPGA can receive characters from the Mac and echo them back.

Only after these steps have been completed does it make sense to connect the PDP-8 CPU, its memory, and the future serial console.

Development Environment

It is important to make clear that we need two different classes of development tools:

- the logical design tools, namely the Verilog compiler and simulator that we used in the preceding chapters;
- the implementation tools, which take the synthesized Verilog design and produce the configuration file to be loaded into the FPGA.

The first class of tools allows us to describe, check, and simulate the logic of the machine. The second class translates that logic into the physical resources of a specific FPGA: lookup tables, flip-flops, block RAMs, routing resources, I/O pins, and clock networks.

In keeping with the choices already made for the logical part of the project, we decided to use open-source tools also for the FPGA implementation. Several components are required, including Yosys, nextpnr, Project Trellis, ecppack, and ecpdap, but the simplest solution is to use OSS CAD Suite, which already includes all the required tools.

For an Intel Mac, the correct package is `oss-cad-suite-darwin-x64-*.tgz`; the `darwin-arm64` package is instead intended for Apple Silicon Macs. Other packages are available for Linux and Windows.

Installing OSS CAD Suite

After downloading the archive, it can be unpacked, for example, in the home directory:

```
cd ~
tar -xzf ~/Downloads/oss-cad-suite-darwin-x64-*.tgz
```

The environment must then be activated in every new Terminal window:

```
source ~/oss-cad-suite/environment
```

At this point the main tools should be visible:

```
which yosys
which nextpnr-ecp5
which ecppack
which ecpdap
```

On macOS, many security warnings may appear the first time the tools are launched, since the executables were downloaded from the Internet. After verifying that the suite is the expected one, the quarantine attribute can be removed with:

```
xattr -dr com.apple.quarantine ~/oss-cad-suite
```

This prevents macOS from asking for confirmation for every single executable in the toolchain.

Verifying the USB Connection

When the Colorlight i5 is connected to the Mac, a virtual USB serial port should appear. The exact name may change from one connection to another, so it is useful to check it each time:

```
ls /dev/cu.usbmodem*
```

In our case, names such as these appeared:

```
/dev/cu.usbmodem14602
```

and, after powering the board off and on again:

```
/dev/cu.usbmodem14302
```

This change in the final number is normal. What matters is that a `/dev/cu.usbmodem...` device appears.

USB, JTAG, and Serial Console

When the development board is connected to the Mac through USB, the host sees the STM32/DAPLink interface. In our tests, a device such as the following appeared:

- `/dev/cu.usbmodem14302`

The final number may change after unplugging or power-cycling the board.

It is important to understand the chain of connections. The Mac does not speak directly to the FPGA over USB. Instead, it communicates with the STM32 microcontroller on the development board. The STM32 then talks to the FPGA through JTAG for programming, and through ordinary 3.3 V UART signals for serial communication.

The chain is therefore:

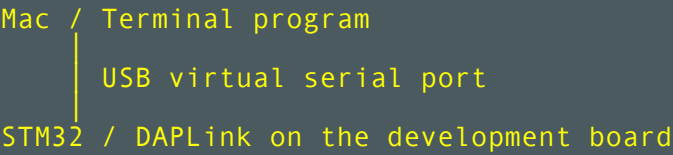


Photo: OTTO compared to the realm PDP-8/m

physical 3.3 V UART signals

Colorlight i5 / ECP5 FPGA

Thus, from the Mac side, the UART appears as a virtual serial port. From the FPGA side, however, it is a real logic-level UART connected to physical FPGA pins. If observed with an oscilloscope, the signals would appear as normal 3.3 V serial waveforms: idle high, start bit low, eight data bits transmitted least significant bit first, and stop bit high.



Checking the Tool Chain

CMSIS-DAP Programmer and JTAG Chain

After activating OSS CAD Suite:

```
source ~/oss-cad-suite/environment
```

the presence of the programmer can be checked with:

```
ecpdap probes
```

The expected result is similar to:

```
Found 1 CMSIS-DAP probe:
  0d28:0204:... DAPLink CMSIS-DAP
```

The JTAG chain can then be scanned:

```
ecpdap scan
```

In our case the result was:

```
Detected JTAG chain, closest to TDO first:
- 0: 0x41111043 (Lattice Semi.) [IR length: 8] [LFE5U-25]
```

This is an important milestone. It proves not only that the Mac sees a USB device, but that `ecpdap` can actually communicate with the Lattice ECP5 FPGA on the board.

First Test: Loading a Ready-Made Bitstream

As a first test, it is better not to compile anything yet. A ready-made demonstration bitstream can be used instead, for example a simple LED blinking design.

One may clone a repository of Colorlight examples:

```
cd ~
git clone https://github.com/wuxx/Colorlight-FPGA-Projects.git
cd Colorlight-FPGA-Projects/demo/i5
```

Then the bitstream can be loaded with:

```
ecpdap program blink.bit -f 5M
```

If everything is working, the green LED on the board should blink.

This programming is volatile: the bitstream is loaded into the FPGA, but not into the SPI flash memory. When the board is powered off, the program disappears. For the first experiments this is exactly what we want, because it allows us to test freely without permanently modifying the board.

Permanent programming of the flash memory should be considered only later, when the design is stable.

Second Test: A Verilog Project Written by Us

After verifying that a ready-made bitstream works, the next step is to compile and load a small project written by us. This verifies the complete flow:

```
Verilog → Yosys → nextpnr-ecp5 → ecppack → ecpdap → FPGA
```

A simple blink project was created, consisting of a counter driven by the 25 MHz clock of the Colorlight board.

Working directory:

```
mkdir -p ~/otto_colorlight/01_blink
cd ~/otto_colorlight/01_blink
```

The essential Verilog file is:

```
module blink (
    input  wire clk_25m,
    output reg  led);

    reg [24:0] counter = 0;
    always @(posedge clk_25m) begin
        counter <= counter + 1;
        led <= counter[23];
    end
endmodule
```

The constraint file, `colorlight_i5.lpf`, assigns the clock and LED pins:

```
LOCATE COMP "clk_25m" SITE "P3";
IOBUF PORT "clk_25m" IO_TYPE=LVCMS33;
FREQUENCY PORT "clk_25m" 25 MHz;
LOCATE COMP "led" SITE "U16";
IOBUF PORT "led" IO_TYPE=LVCMS33;
```

To avoid the usual Makefile tab problems, especially on macOS, a shell script was used:

```
#!/bin/bash
set -e

TOP=blink

rm -f ${TOP}.json ${TOP}.config ${TOP}.bit

yosys -p "synth_ecp5 -top ${TOP} -json ${TOP}.json" ${TOP}.v

nextpnr-ecp5 --25k --package CABGA381 --speed 6 \
  --json ${TOP}.json \
  --lpf colorlight_i5.lpf \
  --textcfg ${TOP}.config \
  --freq 25

ecppack ${TOP}.config ${TOP}.bit
```

After making it executable:

```
chmod +x build.sh
```

the design can be built and loaded:

```
source ~/oss-cad-suite/environment
./build.sh
ecpdap program blink.bit -f 5M
```

The blinking LED confirms that the design has been synthesized, placed, routed, packed, and loaded correctly.

Third Test: From FPGA to Mac

Once the LED test works, the next step is serial communication. For OTTO, a LED is not enough: we need a channel through which the FPGA can communicate with the host computer, initially as a simple console.

On the Colorlight i5, the following pins were used:

```
clk_25m    P3
led        U16
uart_tx    J17
uart_rx    H18
```

The first serial test used only `uart_tx`: the FPGA periodically transmitted the string:

```
HELLO FROM OTTO
```

The terminal was opened with:

```
screen /dev/cu.usbmodem14302 115200
```

Of course, the port name must be replaced with the one actually present on the system.

To exit from screen:

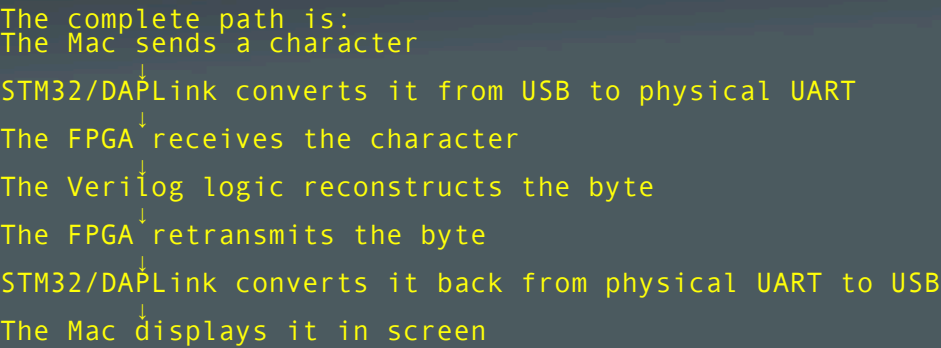
```
CTRL-A
K
Y
```

Transmission was performed at 115200 baud, 8N1 format. With a 25 MHz clock, the divider used was $25,000,000 / 115,200 \approx 217$. Each bit therefore lasts approximately $1 / 115200 \approx 8.68 \mu\text{s}$.

With an oscilloscope connected to the corresponding physical pin, one would see real 3.3 V logic pulses: idle high, low start bit, eight data bits transmitted least significant bit first, and a high stop bit.

Fourth Test: Serial Reception and Echo

The following test added reception. The Mac sends a character through the USB virtual serial port; the STM32/DAPLink converts it into a physical UART signal; the FPGA receives it on its `uart_rx` pin; then the FPGA retransmits the same character on `uart_tx`.



The first version of the UART receiver produced incorrect characters, as if the baud rate were wrong. In reality, the problem was the sampling point. After detecting the start bit, the receiver sampled too early: the first sample still fell inside the start bit, rather than in the middle of data bit 0.

The correct sequence is:

- wait for the RX line, normally high, to go low;
- wait half a bit time;
- verify that the line is still low;
- wait one full bit time;
- sample data bit 0;
- sample the remaining data bits once per baud period;
- ideally verify the stop bit;
- signal that a byte has been received.

After this correction, the echo worked perfectly. This confirmed both serial transmission and serial reception.

Modularizing the UART

At this point the code was cleaned up by separating the UART logic into reusable modules:

- `uart_rx.v`
- `uart_tx.v`
- `uart_echo_top.v`
- `colorlight_i5.lpf`
- `build.sh`

This modularization is important because the future PDP-8 top-level should not contain a monolithic UART implementation. It is better to have two separate blocks:

- `uart_rx` → serial receiver
- `uart_tx` → serial transmitter

The receiver essentially provides:

- `data` = received byte
- `ready` = pulse indicating a valid byte

The transmitter receives:

- `data` = byte to be transmitted

- `start` = transmission request
- `busy` = transmitter busy

These signals are much more suitable for connection to a higher-level peripheral.

Toward a PDP-8-Style Console

As the final preliminary step, another module, `uart_console.v`, was created. This module no longer thinks in terms of serial bits, but in terms of registers and flags. It is the first abstraction of OTTO's future serial console.

The receiving side becomes:

- received character → `kbd_data`
- receive flag → `kbd_flag` = 1
- clear from PDP → `kbd_flag` = 0

The transmitting side becomes:

- `tx_data` = character to transmit
- `tx_load` = transmission request
- `tx_ready` = transmitter free

This structure is close to what will be needed to implement a simplified KL8-E-like peripheral. Later, it can be connected to the PDP-8 IOT instructions:

- KSF → test keyboard flag
- KCC → clear keyboard flag
- KRS → read received character
- KRB → read character and clear flag
- TSF → test transmitter ready flag
- TLS → load and transmit a character
- TCF → possible transmitter flag handling

For the moment, the test top-level still uses this console merely to echo characters. Conceptually, however, the important point is that the physical serial line has now been hidden behind a register-and-flag interface, much closer to the world of the PDP-8.

At the end of this first session, the situation was as follows:

- Colorlight i5 visible from the Mac OK
- `/dev/cu.usbmodem...` port present OK
- CMSIS-DAP recognized by `ecpdap` OK
- JTAG access to LFE5U-25 working OK
- FPGA IDCODE 0x41111043 OK
- ready-made blink bitstream loaded OK
- custom Verilog blink built and loaded OK
- UART TX FPGA → Mac at 115200 baud OK
- UART RX Mac → FPGA OK
- bidirectional echo OK
- `uart_rx` / `uart_tx` separated into modules OK
- first register-and-flag console abstraction OK

Common Commands

Now that the project is finally ready, you normally need only two commands.

The first rebuilds the entire project and programs the FPGA:

```
./build_fpga.sh prog
```

The second is useful when you only need to program the FPGA with an already built bitstream, without running a new build:

```
ecpdap program PDP_colorlight_top.bit -f 5M
```

Later we will see how to make the FPGA configuration permanent by writing it to the SPI flash memory, from which it is automatically loaded at power-up.

What Is Next

This concludes the preliminary operations. The Colorlight i5 is now under control, the toolchain is working, volatile FPGA programming is reliable, and the bidirectional serial channel is ready.

The next step will be to connect this interface to the OTTO logic, starting from a minimal configuration:

- PDP-8 CPU
- internal RAM
- reset
- clock
- serial console

Only after that will it be useful to move to more ambitious features, such as program loading, a more complete console interface, possible SDRAM use, and finally the execution of real PDP-8 software on the FPGA.



Above: The real PDP-8/m described in the [PDP-8] book.

Version Control, GIT

Some Practical Notes

When a project is made only of a few wires and a soldering iron, keeping track of changes is relatively simple: you look at the board, and what you see is what you have done. With an FPGA project, things are different. A working machine may depend on many Verilog files, a constraint file, memory initialization files, build scripts, and a number of generated files that are not really part of the design. After a few evenings of experiments, it becomes surprisingly easy to lose the last known-good version.

For this reason I chose to use Git. Git is a version control system. In simple words, it allows you to take **snapshots** of a project at meaningful moments. These snapshots are called **commits**. Each commit records the state of the files and a short message explaining what was done.

For example, after getting the single-instruction stepping to work, a useful commit message was:

```
Fix HALT single-instruction stepping with synchronized CONT
```

Later, after adding the hardware breakpoint and console controls, another commit could say:

```
Add console controls and hardware breakpoint
```

These messages are not decoration. A good commit message is a note to your future self. When something stops working two days later, you can look at the history and understand where the project changed.

Basic Commands

The basic commands are few.

To see the current state of the project:

```
git status
```

This tells which files have been modified, deleted, or newly created. It is the command I use most often.

To see the actual changes:

```
git diff
```

or, for a shorter summary:

```
git diff --stat
```

When the result is satisfactory and the machine works on the real FPGA, the relevant files are added to the next commit:

```
git add PDP_cpu.v PDP_colorlight_top.v Datapath.v colorlight_i5.lpf
```

Then the snapshot is created:

```
git commit -m "Add console controls and hardware breakpoint"
```

A very important rule is not to add everything blindly. In an FPGA project, many files are generated automatically by the build process. For example, the `.json`, `.config`, and `.bit` files are produced by Yosys, nextpnr, and ecppack. Simulation logs, waveform files, and macOS `.DS_Store` files are also not part of the real source. For this reason, using:

```
git add .
```

is dangerous unless the directory is perfectly clean. It is usually better to add only the files you really want to preserve.

Generated files can be ignored by adding a `.gitignore` file. For this project, a typical one is:

```
gitignore
```

```
.DS_Store
```

```
#FPGA build products
*.bit
*.config
```

```

*.json

# Simulation products
*.vcd
pdp_sim
*.log

# Local backups
backups/
memfiles/

```

This keeps the Git status readable and prevents accidental commits of large or temporary files.

Git stores its data in a hidden directory named `.git`, located at the root of the project. This directory contains the complete history, the commit database, the current branch information, and the staging area. In normal use, it should not be edited manually. One works through Git commands.

One of the most useful features is the ability to return to a known-good state. If a file has been modified experimentally and the experiment fails, it can be restored with:

```
git restore PDP_cpu.v
```

This discards the local changes and restores the file from the last commit. Used carefully, this is extremely reassuring. Before trying a risky modification, make a commit. Then, if the experiment goes wrong, there is always a safe place to return to.

Another useful command is:

```
git log --oneline
```

It shows the project history in compact form. A well-kept history becomes a laboratory notebook. It may look like this:

```

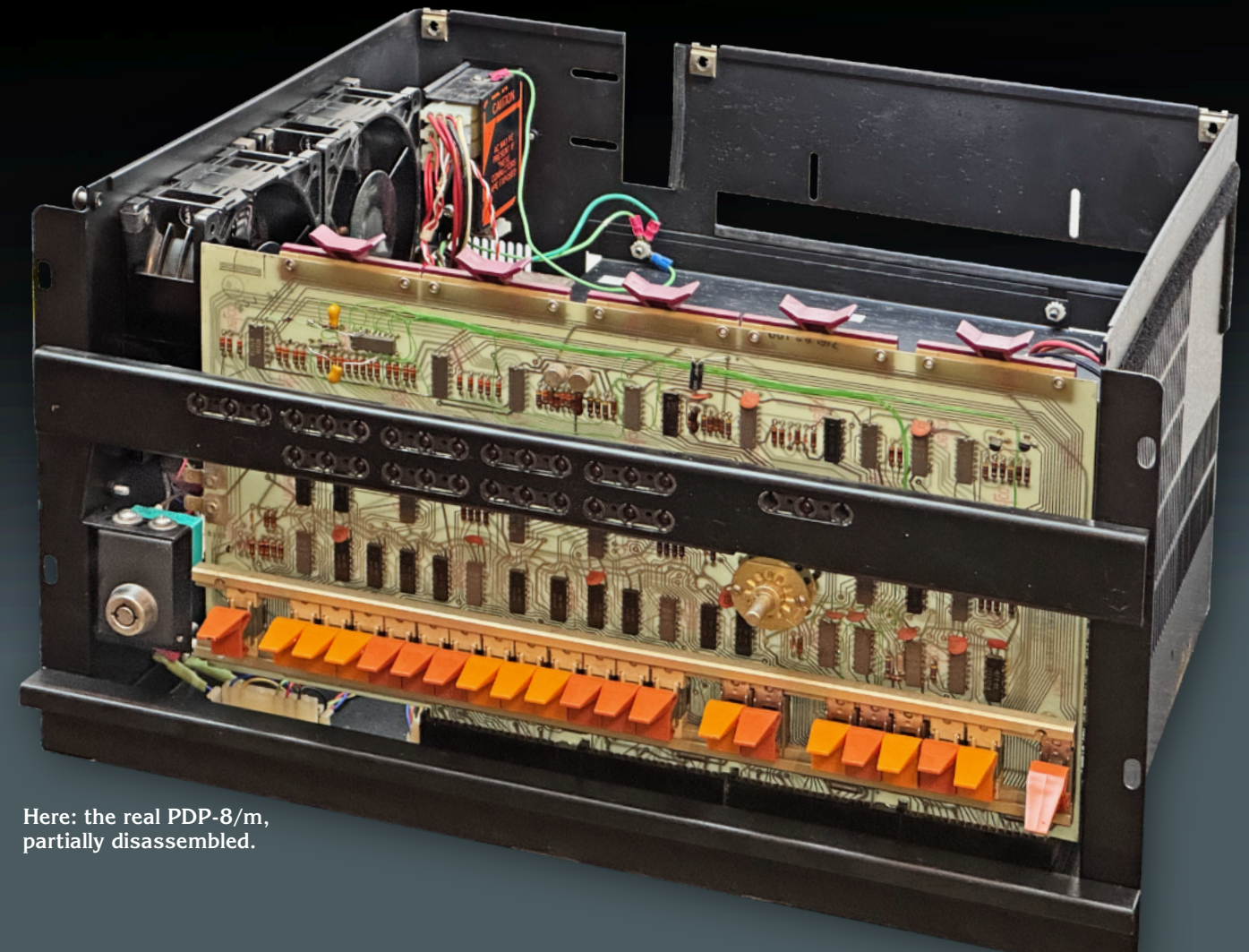
Remove boot trick and reset PC to 0200
Add console controls and hardware breakpoint
Fix HALT single-instruction stepping with synchronized CONT
Add HALT-based single-instruction stepping

```

This is much more useful than a single vague entry such as “various fixes”.

In the OTTO project, Git proved valuable many times. The implementation of the PDP-8 core on the Colorlight i5 went through many small changes: UART fixes, Teletype timing corrections, single-instruction stepping, physical CLEAR and HALT controls, a hardware breakpoint, decimal-point status indicators, and finally programming the SPI flash so that the FPGA could boot by itself. Each successful stage deserved its own commit.

The best practical advice is simple: commit often, but commit only when the project is in a meaningful state. A commit should represent something that worked, or at least something worth remembering. In this way, Git does not merely protect files. It records the reasoning path of the project.



Here: the real PDP-8/m, partially disassembled.

Flashing FPGA Configuration

Making Configuration Permanent

As mentioned several times, an FPGA is a vast expanse of gates, flip-flops, memory blocks, and routing resources, all waiting to be configured into the circuit of our dreams. In practice, this configuration is described by the `.bit` file, which is loaded into the configuration memory inside the FPGA.

There is, however, an important detail. This configuration memory is volatile. When the FPGA is switched off, its contents are lost. At the next power-up, the configuration must be loaded again, just as we did many times during the development described in the previous notes.

The Colorlight i5 board, however, also includes an SPI flash memory. This non-volatile memory can be programmed—or, as one commonly says, “flashed”—with our configuration file. When the board is powered on, the FPGA automatically reads the bitstream from the SPI flash and loads it into its internal configuration memory. In this way, the design becomes permanent, at least until we decide to overwrite it with a new version.

Misleading Messages

Boards such as the Colorlight i5 are interesting little objects. They were not originally designed as general-purpose FPGA development boards, but as controllers for LED display panels. Nevertheless, they are well made, inexpensive, and very attractive for experimental work. The only real problem is often not the hardware, but the information that accompanies it. Some online descriptions look like

copies of copies of copies of poorly translated notes, sometimes mixed with obsolete advice and vendor folklore. As a result, a simple detail may become unnecessarily confusing.

The SPI flash is a good example. Some product descriptions give the impression that, in order to use the board properly, one must replace the flash chip with another device, such as a Winbond W25Q series part. In reality, at least in my case, nothing of the kind was necessary. The flash was already present, correctly connected, large enough for the bitstream, and perfectly usable. It was simply protected against writing by its status register.

Using the FLASH

In this project the flash was accessed using `ecpdap`. The first step was to identify it:

```
ecpdap flash -f 5M id
```

The board reported a GigaDevice flash with a capacity of 16 Mbits, or 2 MiB, more than enough for the approximately 600 kB ECP5 bitstream. A further check was made with:

```
ecpdap flash -f 5M scan
```

This also showed that the block protection bits were set:

```
BP0: true, BP1: true, BP2: true
```

In other words, the flash could be read, but not written. The protection was removed with:

```
ecpdap flash -f 5M unprotect
```

A second scan confirmed that the protection bits had been cleared:

```
BP0: false, BP1: false, BP2: false
```

After that, the bitstream could be written normally:

```
ecpdap flash -f 5M write PDP_colorlight_top.bit
```

Before doing this on an unknown board, it is prudent to save the original contents:

```
ecpdap flash -f 5M read flash_backup.bin
```

Once programmed, the real test is very simple: remove power, wait a few seconds, and power the board again. If everything is correct, the FPGA configures itself from the SPI flash and the OTTO PDP-8 starts directly from its stored configuration. At that point the board is no longer just a target connected to a development system; it has become a small stand-alone computer.



Section

Construction

With the code now developed, we “only” have to build the final OTTO prototype and make the last adjustments.

Lazy Builders?

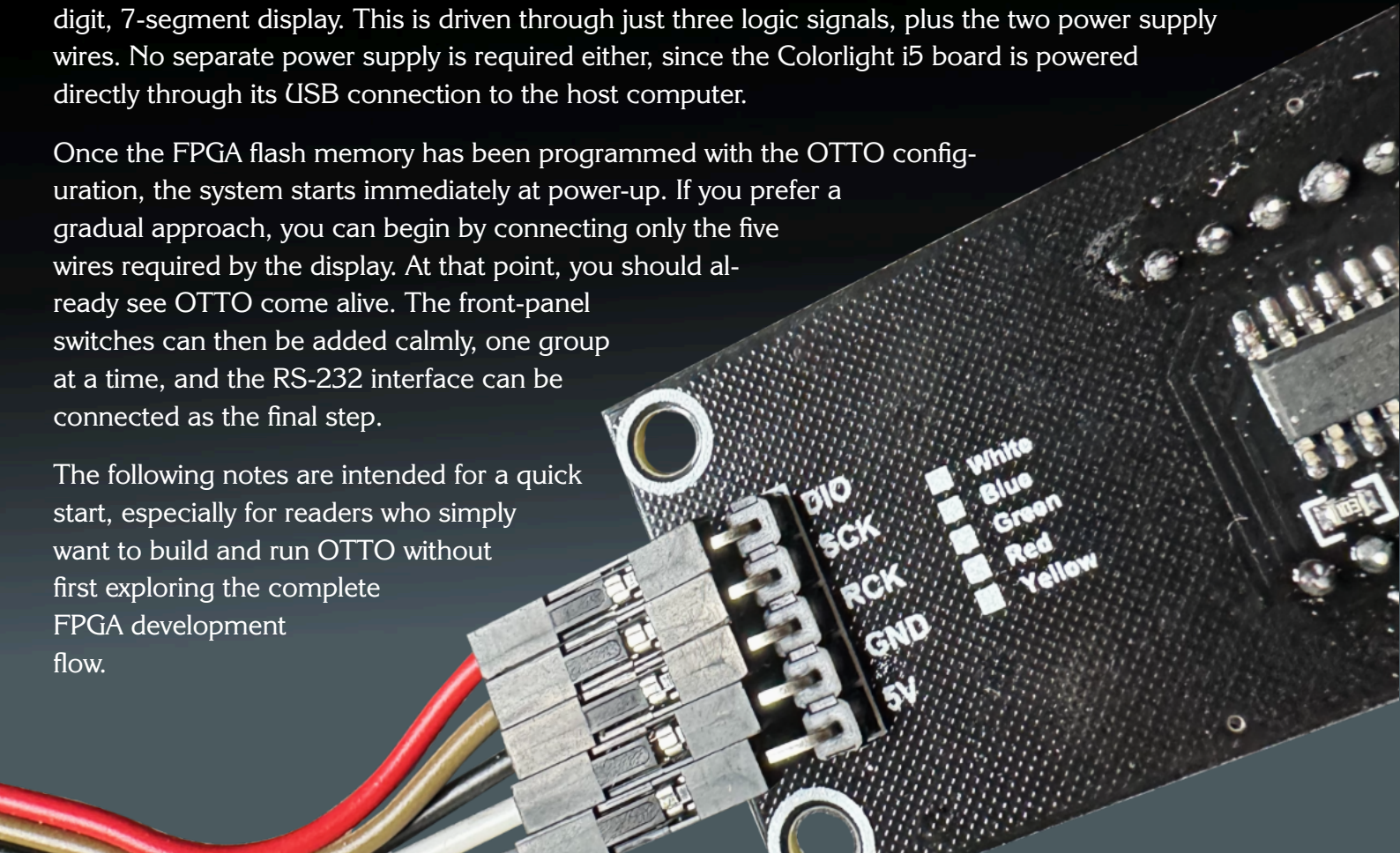
Not a Complex Project

Despite its appearance, OTTO is not a particularly complex project, and building a working replica is not very difficult. Most of the external connections are simply inputs: the switches of the front panel.

If we leave aside the RS-232 interface for the moment, the project has only one real output: the 8-digit, 7-segment display. This is driven through just three logic signals, plus the two power supply wires. No separate power supply is required either, since the Colorlight i5 board is powered directly through its USB connection to the host computer.

Once the FPGA flash memory has been programmed with the OTTO configuration, the system starts immediately at power-up. If you prefer a gradual approach, you can begin by connecting only the five wires required by the display. At that point, you should already see OTTO come alive. The front-panel switches can then be added calmly, one group at a time, and the RS-232 interface can be connected as the final step.

The following notes are intended for a quick start, especially for readers who simply want to build and run OTTO without first exploring the complete FPGA development flow.



The Lazy Builder's Notes

If your only goal is to replicate OTTO, without diving into all the Verilog sources, synthesis tools, place-and-route reports, timing constraints, and other arcane rituals described above, the procedure is much simpler.

In that case, you do not need the complete FPGA development toolchain. You do not need `Yosys`, `nextpnr`, `ecppack`, or any other tool required to rebuild the project from source. You only need `ecpdap`, the precompiled OTTO bitstream, and, of course, a correctly wired Colorlight i5 board.

The practical steps are the following:

- build the OTTO hardware, using the photographs and mechanical notes in this chapter as a guide;
- study the `colorlight_i5.lpf` file, which describes the required connections between the Colorlight i5, the Muse carrier board, the front panel, the display, and the RS-232 interface;
- wire the front panel and the serial interface accordingly;
- copy the supplied `PDP_colorlight_top.bit` file into a convenient folder;
- connect the Colorlight i5 carrier board to a USB port of your computer;
- open a terminal in the folder containing the bitstream.

Before programming the board, it is useful to verify that `ecpdap` can see the CMSIS-DAP interface and the FPGA:

```
ecpdap probes
ecpdap scan
```

To load the OTTO configuration temporarily into the FPGA SRAM, use:

```
ecpdap program PDP_colorlight_top.bit -f5M
```

This is useful for a first test, but the configuration will be lost when the board is powered off.

To make OTTO start automatically at power-up, the bitstream must be written into the SPI flash memory:

```
ecpdap flash write PDP_colorlight_top.bit
```

On some Colorlight i5 boards, the SPI flash may initially be protected. If the write operation fails, unlock the flash first with:

```
ecpdap flash -f5M unprotect
```

and then repeat the flash write command.

After the bitstream has been written to the flash memory, power-cycle the board. If everything has been wired correctly, OTTO should start by itself, without requiring the development tools or any further programming operation.

Building OTTO

Basic Decisions

I like building things with my own hands, and I consider this phase of a project both rewarding and relaxing. In other times, I asked myself if I was developing electronic projects just to have the opportunity to create a nice box for them.

OTTO, however, had not been intended for industrial production. In that case, I would have pursued a much more carefully engineered and easily reproducible structure. But since I only had to build one example for my own amusement, I allowed myself a different kind of freedom.

The result was not necessarily the most rational construction possible, but I tried to make the front panel attractive and pleasant to use, even if this meant accepting more internal wiring than I would have tolerated in a professional piece of equipment. In a one-off project, appearance, convenience, and pleasure count as much as strict producibility, and, in any case, I think I find a building technique very well suited to the aim.

Enclosure

For the enclosure I simply reused the **Ganzerli** box of an old power supply that I had built many years ago. Ganzerli was an Italian brand that produced good-quality enclosures for electronic equipment; as far as I know, it is probably no longer active today. Not very many people still enjoy this kind of activities.



The OTTO computer, fully assembled

Choosing Components

Front panels are the business card of equipment, so it's best to try to do them as best as possible, within the boundaries set by the scope of this device.

Obviously, I wanted to give OTTO a look and feel resembling the true PDP-8, and obtained it with a proper colored graphics, maintaining however with a certain freedom from the original, and even adding some small improvements.

Switch Choice

Switches play a primary role in any PDP-8 replica. The original machines owe much of their charm to their front panel, and it would be a pity to build a PDP-8-like machine without giving the operator some real switches to operate.

I would have liked to find some nice paddle switches, closer in spirit to the originals, but I could not find any suitable type readily available. They would also have required more front-panel space, probably a printed circuit board, and a rather more complex mechanical arrangement than I wanted for this version.

In the end, I settled for normal miniature lever switches. They are inexpensive, easy to find, and can be mounted directly on the panel with a simple 6 mm hole. They are not as spectacular as the original DEC switches, but they are practical, reliable, and quite adequate for a small-size personal replica.

Just one note: their quality is acceptable but not exceptional, and their plastic tend to mold when soldering the terminals. Be quick and set the switch in the opposite position of the terminal you want to solder, so that the internal contacts help dissipating heat.

Display Choice

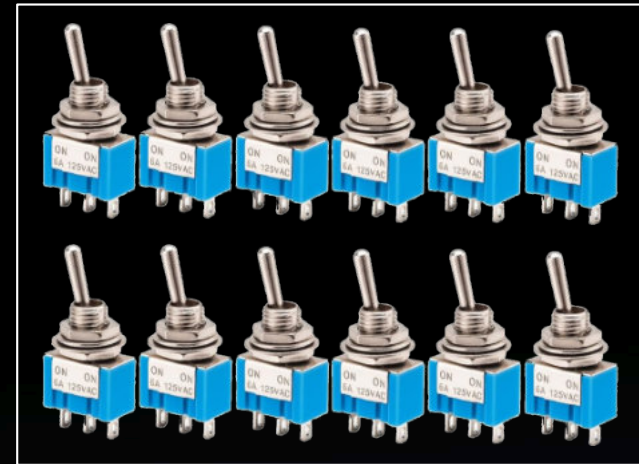
For the display I decided to treat myself to a small luxury. Instead of using rows of individual LEDs, I used an 8-digit, 7-segment display module that I had bought some months earlier on Amazon.

These displays are inexpensive and readily available, but they also look good and make the front panel more attractive. They are also much easier to read than a row of binary LEDs. Anyone who has spent some time reading binary lamps knows how easy it is to make a mistake, especially when tired or when trying to follow a program quickly.

There is also a very practical advantage: these modules greatly reduce the number of wires. This matters, because the switch register alone already requires quite a few connections. The display module uses the popular 74HC595 shift-register integrated circuits and needs only three control lines from the FPGA. With just data, clock, and latch signals, the FPGA can drive all eight digits, including the decimal points that I later used as useful status indicators.

Actually, they should be powered at 5 V. I used 3.3 V to avoid level translators from the FPGA outputs. Even this way, the brightness is more than acceptable.

Below: The 8-digit, 7-segment display is easy to interface and requires far fewer wires than individual LEDs. At the time of writing, three of these modules can be bought on Amazon for about €11. To avoid using 3.3 V to 5 V level shifters, I powered them directly at 3.3 V; their visibility remains good.



Above: I used the simplest type of toggle switch, bought on Amazon for €8.99 for a pack of twelve. The momentary version, with a central rest position, is slightly more expensive but allows each switch to control two functions. In my case, I used this possibility only for setting the breakpoint address.



With one of these displays, two 12-bit registers can be shown at the same time in octal form. A rotary switch selects among four different display combinations, allowing the operator to choose which internal registers are currently shown.

Annunciators

Some annunciator LEDs are still necessary, of course—for example RUN, HALT, BREAK ENABLE, and BREAK HIT. To keep the wiring simple, I reused the decimal points of the 7-segment digits for this purpose. With a suitable graphic on the front panel, these decimal points become small but perfectly readable status indicators, without requiring additional LEDs or extra FPGA outputs.



Building the Front Panel

Front Panel Structure

For practical, aesthetic, and sentimental reasons, I devoted quite a lot of time to the design of the front panel, trying to reproduce, at least in part, the general look and feel of the original PDP-8.

I used a technique I had already adopted for the **Baubox**, described in my [TG-7] book. The concept is simple: to create a **sandwich** composed of the **original metal front panel** of the enclosure you have chosen, a **front-panel artwork** carefully printed on paper, and a transparent **Lexan plate**. The metal panel provides the mechanical strength, the paper artwork gives the panel its appearance, and the Lexan sheet protects it.

I tried to use the original PDP-8 colors in the artwork and, as you can see in the photos, the result is not bad at all.

OTTO, as you know, does not pretend to be a perfect replica, but it captures enough of the original spirit to make the machine pleasant to look at and, more importantly, pleasant to use.

How-To and Practical Suggestions

To build a panel in this way you need the metal panel, of course, and a Lexan sheet about 1 to 1.5 mm thick. Lexan, that is, polycarbonate, is preferable to acrylic sheet such as Plexiglas, because it is less prone to cracking.

I was not used to this type of tool and was surprised by its ease of use and by the lower risk of injury compared with other types of cutters. With a minimum care the result is ensured.

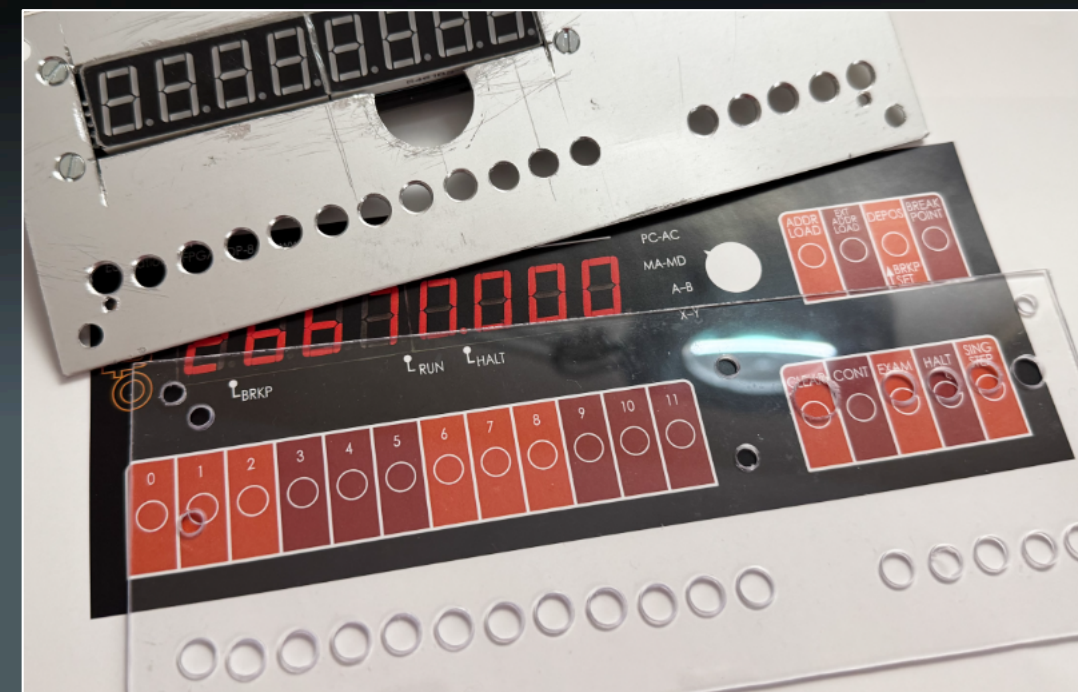
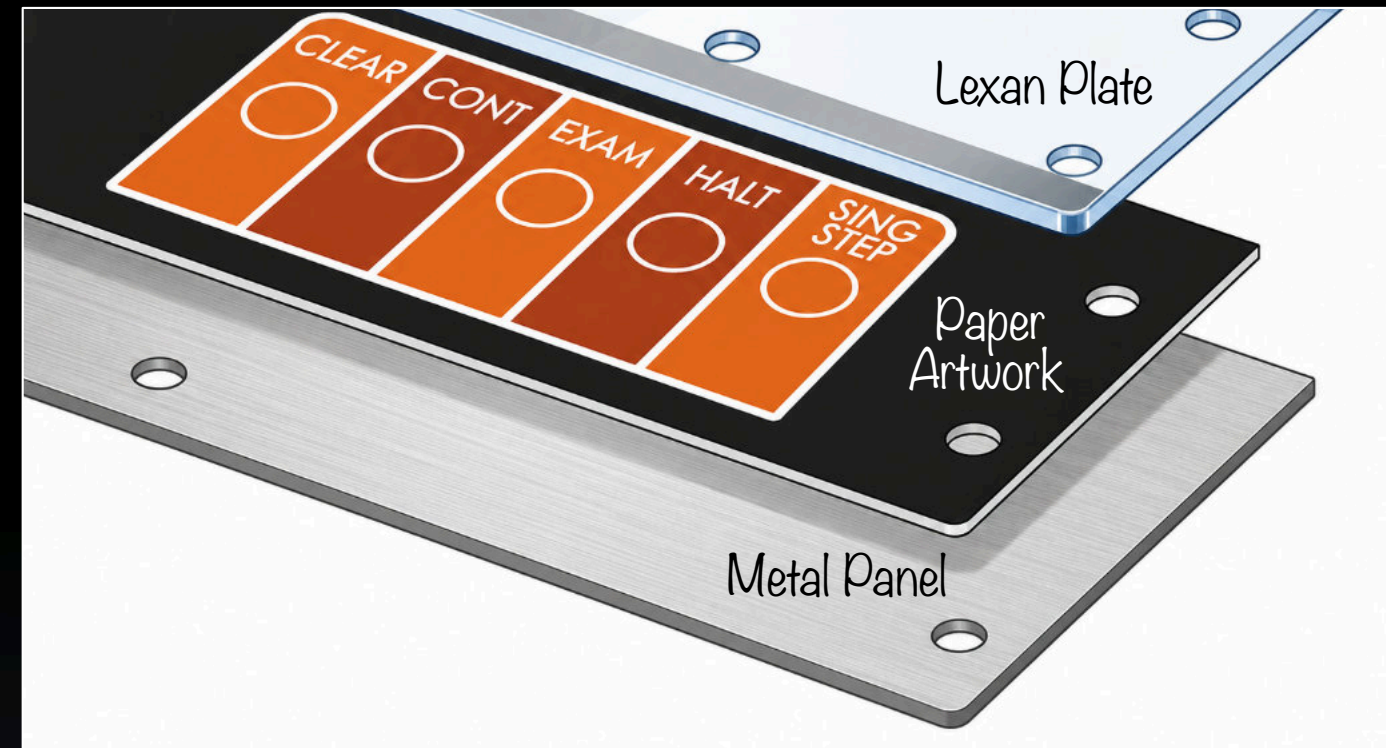
Drilling

I fixed this template to the metal panel with adhesive tape and used it to mark all drilling points. Before drilling, I marked each hole center with a hammer and a center punch. This is important, because it prevents the drill bit from wandering when it first touches the metal surface.

Once the metal panel was finished, I used it as a template for the Lexan plate. After drilling the first two holes, I inserted screws and nuts through them, so that the metal panel and the Lexan sheet remained tightly aligned while drilling the remaining holes.

Do not remove the protective films from the Lexan sheet until all mechanical work has been completed. They protect the surface from scratches and make the whole operation much less stressful.

The paper artwork is more difficult to drill or cut cleanly. In theory, one can use a hollow punch set to cut the holes in the paper artwork. I bought the set shown in the photo from Amazon, but it proved not to be very suitable for paper, so in the end I had to improvise. In any case, this operation requires patience, because paper tends to tear or deform much more easily than metal or plastic.



Above is a drawing illustrating the “sandwich” concept described in the text.

Right are the actual components used to build the OTTO front panel.

Facing page: An oscillating multi-tool. Many different models are available; the one shown here is made by Bosch, while I use a Makita, which has proved to work very well.

Finishing

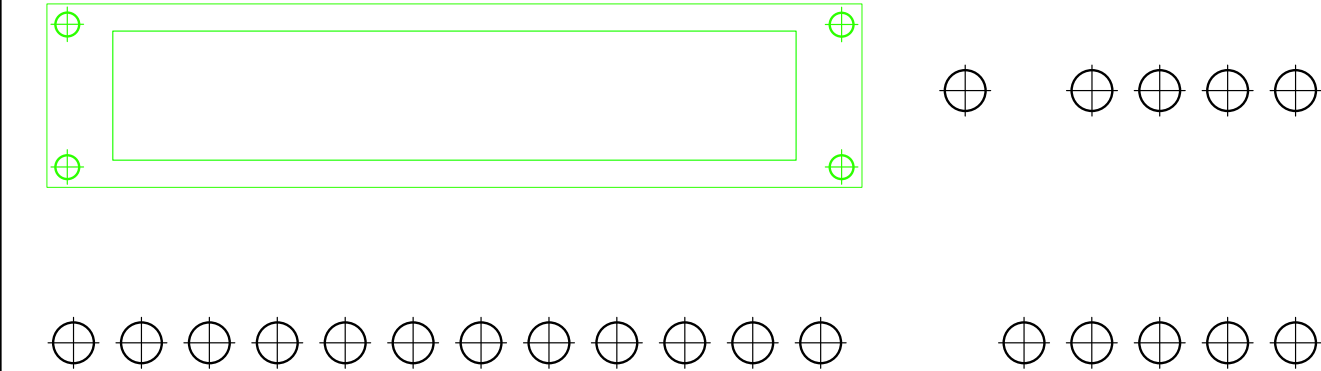
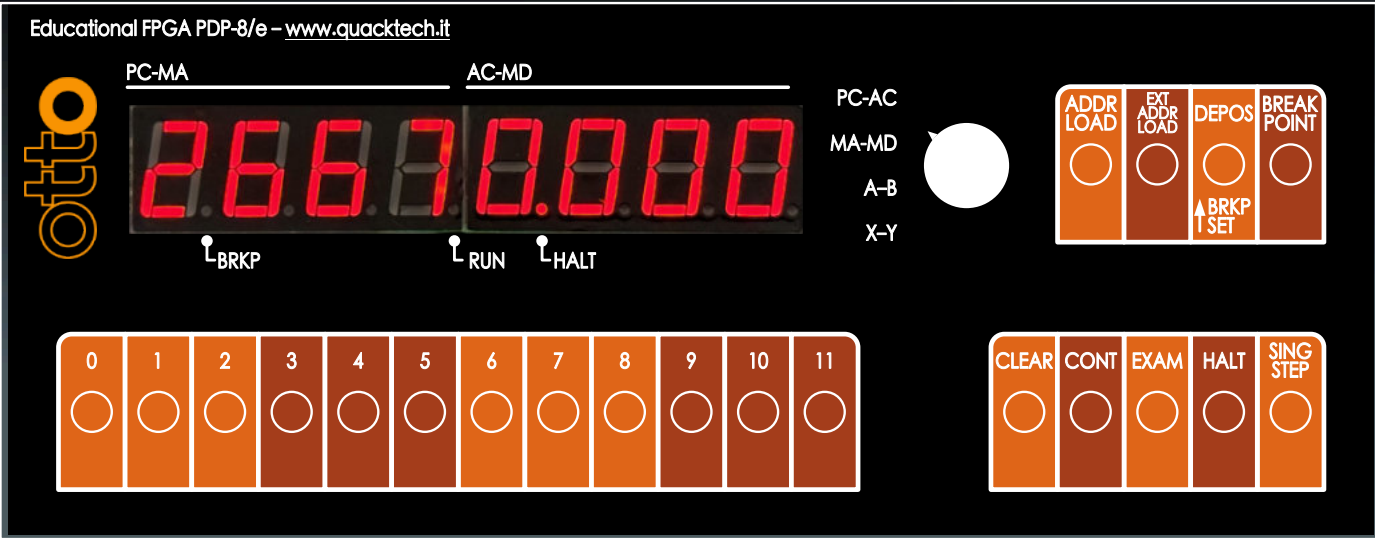
Plastic materials are rather tricky to machine. Even after careful drilling, some burrs often remain, and they are not always easy to remove with a file or sandpaper. Nevertheless, the edges must be smoothed, both for appearance and to avoid interference when the panel is assembled.

For the holes, the best solution is to use a countersink. If you do not have one, a large drill bit can also be used very carefully by hand to remove the sharp edge. For the outer edges, a simple deburring tool works surprisingly well. Again, mine came from Amazon, and I found it much more useful than I expected.

Other Advantages

This construction method has another important advantage. Not only are the machining marks on the metal panel hidden, but, as in this case, you can also reuse a panel that already has some unwanted extra holes. Once the printed artwork and the Lexan plate are mounted in front of it, those holes simply disappear into the background.

Below: The front panel artwork, created with Affinity. Although Affinity is not a CAD program, it still allows accurate control of sizes and distances. A separate layer, hidden in this view, contains the hole centers and cutting lines, as shown on the facing page. Most modern color printers are good enough for this kind of work. The most critical aspect is, of course, obtaining a deep and uniform black background. If your own printer is not up to the task, most stationery stores can provide inexpensive, high-quality prints.



Above: Cutting and drilling require some care. I used a different version of the artwork layer, showing the drilling centers and the display cut-out lines, and fixed it to the panel with adhesive tape to use it as a template.

Below left: The hollow punch set used to cut the holes in the paper artwork. I bought the set shown in the photo, but it proved not to be very suitable for paper.

Below right: The deburring tool works surprisingly well and gives the edges a much more professional finish.



A Peculiar Wiring

A Different Approach

With this unit, I experimented with a different wiring method, at least for me, which proved to be very practical.

Instead of soldering every single wire, I found it very convenient to use inexpensive short lengths of preassembled colored ribbon cable, fitted with single-pin female connectors. These are easily found on Amazon and can be separated or cut as needed.

This solution is especially useful in a prototype, where connections often have to be changed, moved, or rearranged during development.

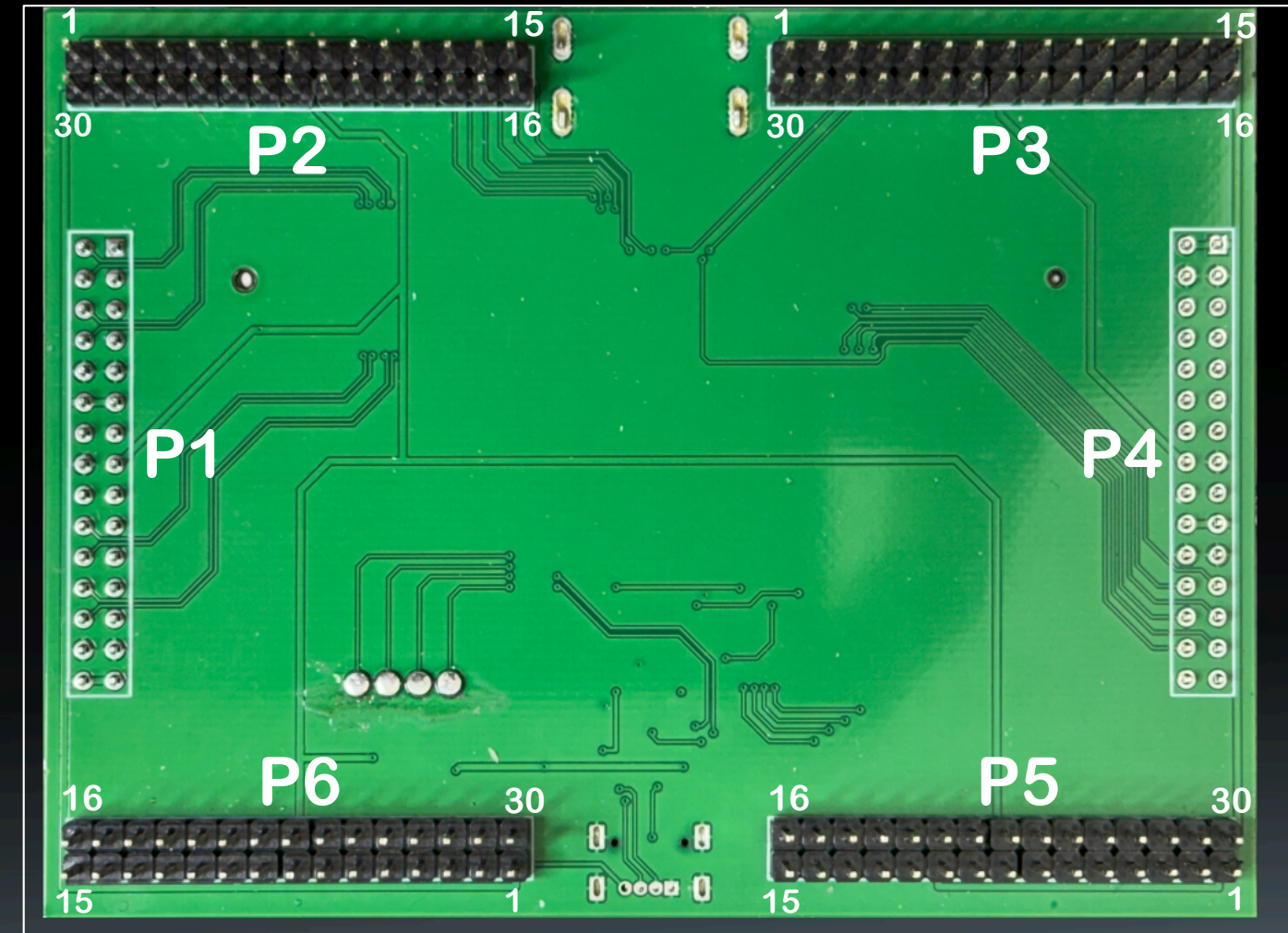
Placing the Boards

To take full advantage of these wires, I adopted a somewhat unusual mounting arrangement for the Muse carrier board and, consequently, for the Colorlight i5 module.

The Muse board provides many more connector positions than are actually required in this application, and this gives a certain freedom in choosing how to use them. I fitted it with straight male pin headers instead of the 90-degree female connectors supplied in the kit, but not all on the same side:

- P2, P3, P5, and P6 are mounted on the bottom side of the board;
- P1 and P4 are mounted on the top side.

In this way, since the Muse board has no mounting holes of its own, P1 and P4 serve simply as mechanical supports. As shown in the photo on the next page, the Muse board is therefore mounted upside down and plugged into a small piece of perfboard screwed to the bottom of the enclosure.



Above: The Muse board connectors, seen from below. P1 and P4 were mounted on the opposite side of the board and are used only as mechanical supports for the assembly. This was necessary because the Muse board has no dedicated mounting holes.

This arrangement leaves the P2, P3, P5, and P6 connectors easily accessible from above. The terminated jumper wires can then be inserted, removed, or rearranged in a moment. This is extremely useful in a prototype assembly, where modifications are often required.

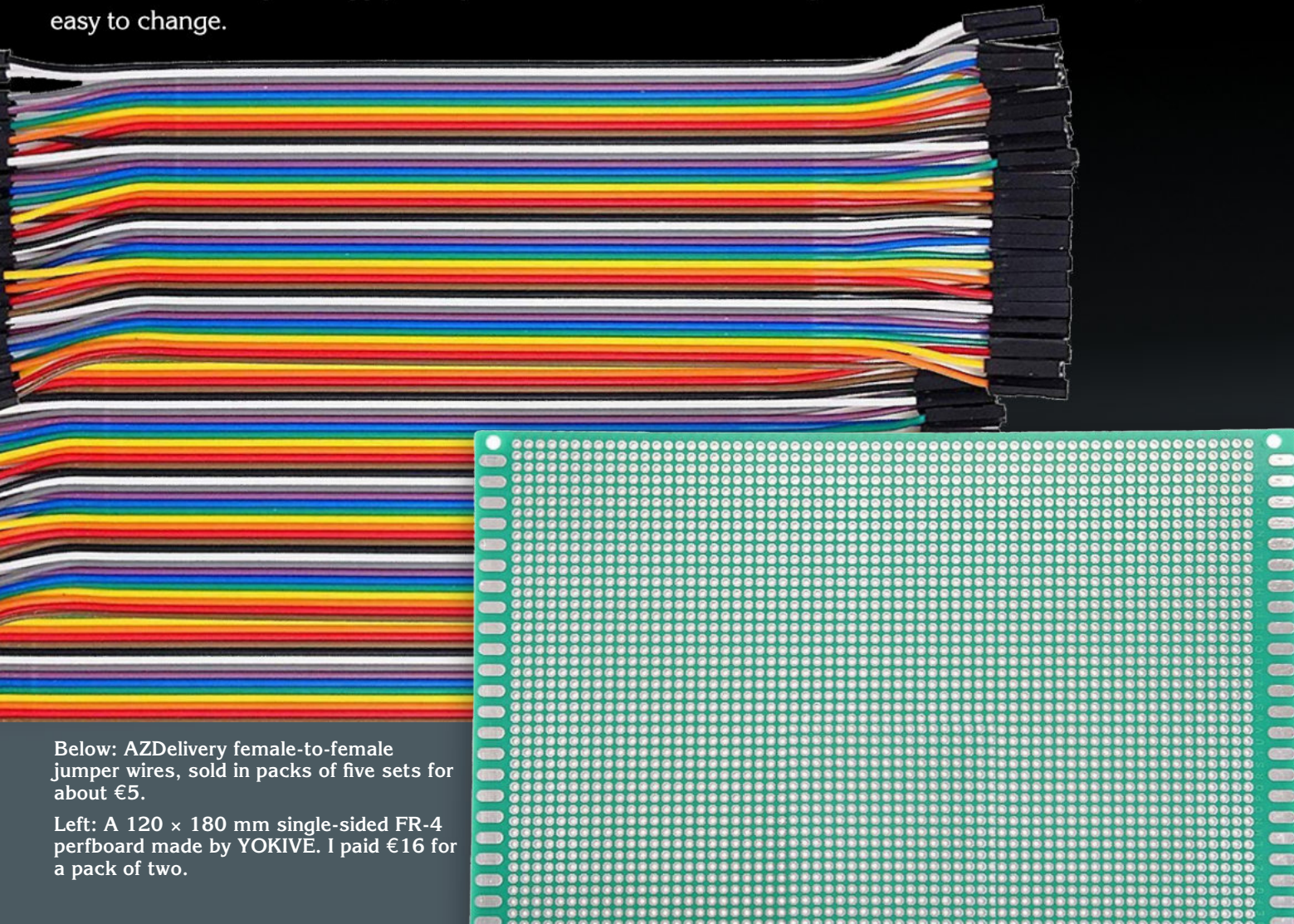
I also considered using the same approach on the switch side, but in the end I preferred to solder the wires directly to the switches. This allowed me to cut each wire to the most suitable length and to obtain a cleaner result on the front-panel side.

Advantages

The solution proved to be very convenient, not only during the many adjustments required in development, but also because, at first, I had mounted the board rotated by 180 degrees.

Thanks to the removable connections, I was able to correct the problem in a few minutes. Had the wires been soldered directly to the board, the same mistake would probably have required hours of work to fix.

The same advantage will apply to any future modification: the wiring remains accessible, flexible, and easy to change.



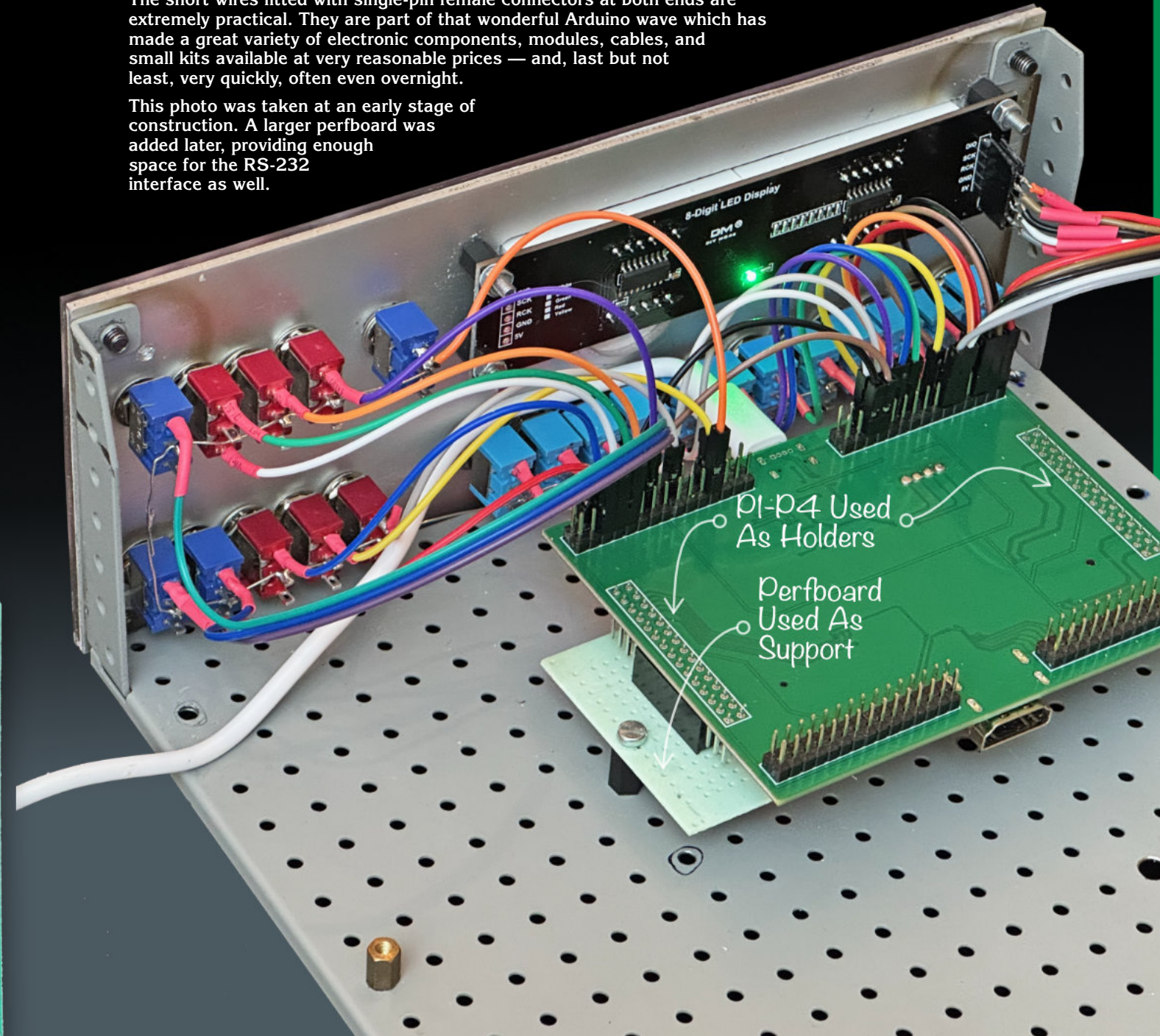
Below: AZDelivery female-to-female jumper wires, sold in packs of five sets for about €5.

Left: A 120 × 180 mm single-sided FR-4 perfboard made by YOKIVE. I paid €16 for a pack of two.

This photo shows the details of the solution adopted to support the Muse board and its precious cargo, the Colorlight i5, while at the same time creating a flexible, solderless wiring system. The connections can be changed in seconds, and the whole board can be replaced in a few minutes.

The short wires fitted with single-pin female connectors at both ends are extremely practical. They are part of that wonderful Arduino wave which has made a great variety of electronic components, modules, cables, and small kits available at very reasonable prices — and, last but not least, very quickly, often even overnight.

This photo was taken at an early stage of construction. A larger perfboard was added later, providing enough space for the RS-232 interface as well.



90 Interfaces

OTTO has only two external communication interfaces:

- USB
- RS-232

The first is mainly a service interface. It is used to load the FPGA configuration, or to store it permanently in the SPI flash memory. However, as we have seen, it can also be used as a virtual serial channel for OTTO: from the computer side it appears as a USB serial port, and can therefore be used with an ordinary terminal program.

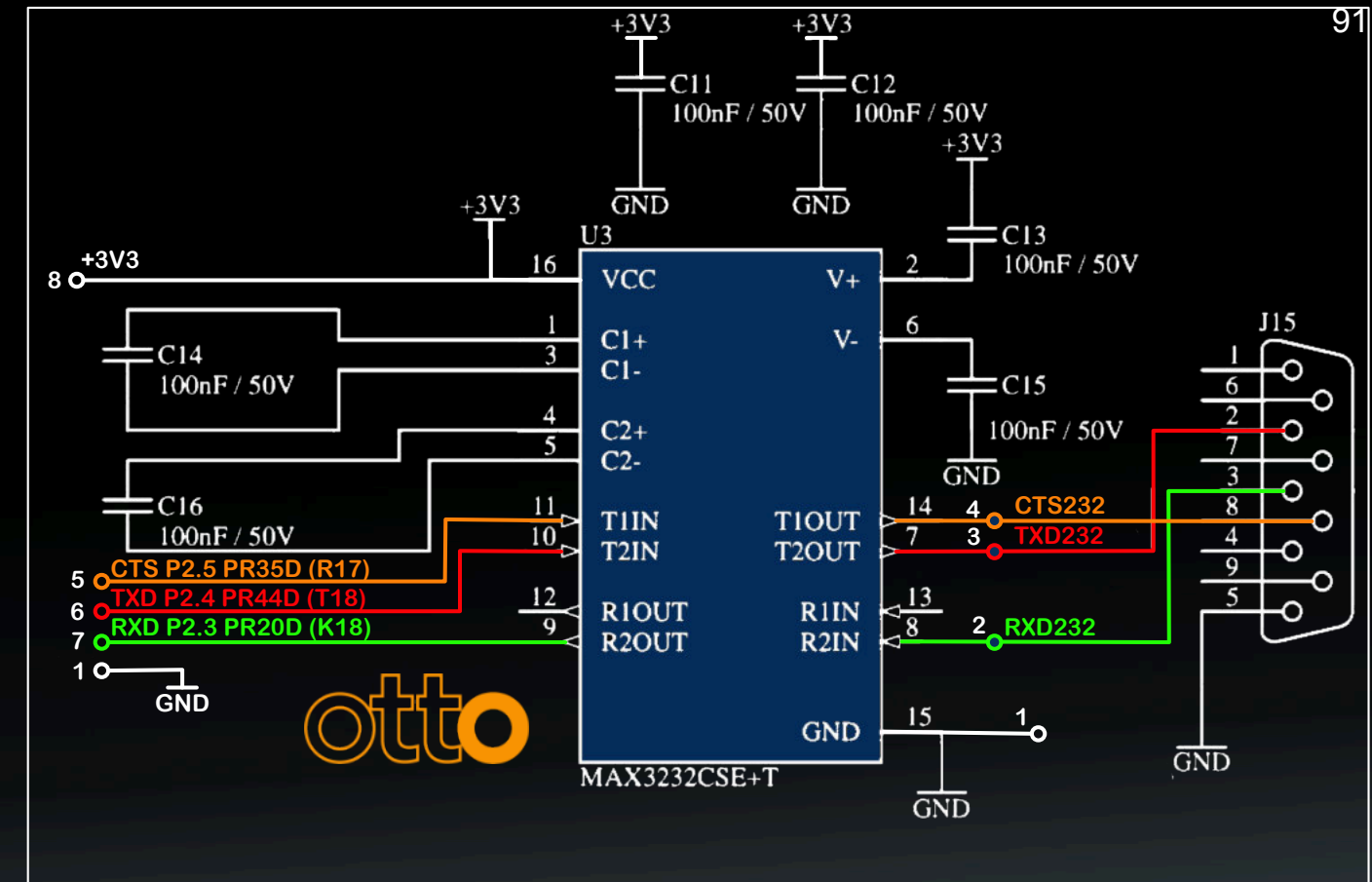
This USB path has one **limitation**, however. It carries only the TXD and RXD signals. This is sufficient for normal terminal operation, but it does not provide the additional control line needed to emulate the **paper-tape reader** properly. Like the original PDP-8 used with an ASR-33 Teletype, OTTO benefits from a Reader Enable signal, used as a hardware handshake to regulate the flow of incoming data according to the machine's ability to receive it. Without this signal, the USB virtual serial port can still be used, but mainly for interactive operation, or for file transfers that do not require accurate reader emulation.

For this reason, I also added a **separate RS-232 interface**. In this interface the CTS line is used to carry OTTO's Reader Enable signal. The level converter must be compatible with the 3.3 V logic used by the FPGA, so a device such as the MAX3232 is appropriate. I had hoped to use one of the inexpensive modules available from Amazon, many of which already include both a MAX3232 and a 9-pin D-type connector. Rather surprisingly, however, most of these modules use only the TXD and RXD channels and ignore the remaining drivers and receivers, thus discarding a useful feature that could have been implemented at almost no additional cost. I therefore had to wire a small interface circuit on perfboard.

Power Supply

The USB interface has another important function: it powers the whole OTTO system. I found the convenient connector shown in the photo on the right, which allows the USB connection to be brought inside the box without soldering. It has a USB connector on the outside and internal USB connectors on the inside. It is rather expensive—about 11 euros—but very practical.

When OTTO is not connected to a computer, any suitable smartphone charger can be used as the power supply.



Above: Electrical circuit of the RS-232 interface, used to connect OTTO to a device emulating the original Teletype ASR-33, typically a computer.

Left: This practical connector allows a USB signal to pass through a metal enclosure without stripping cables or soldering. It has two USB connectors on one side and two on the other; the two channels are completely separate.

A 22 mm hole is required for mounting. It is available on Amazon and costs about 11 euros—not exactly cheap, but very convenient.

I could not drill a hole of exactly the required size, only a slightly larger one, so I 3D-printed two washers to compensate.

Improvements

Far be it from me to belatedly improve the PDP-8. After all, it is much easier to redesign a small part of a historic machine sixty years later, with an FPGA, than it was to design the original machine under real cost, component, and production constraints. However, in a few places the temptation was too strong.

Display

The first change has already been mentioned: the display. A real PDP-8 front panel has its own beautiful and unmistakable logic, but my construction method, based on a compact FPGA board and a small number of external components, almost forced me to adopt a different solution. The 7-segment display is less authentic, but very practical, especially when showing two 12-bit registers in octal form.

Memory Addresses

A second small improvement concerns memory examination. On the original machine, as on many computers of that period, the operator must always keep very clearly in mind whether the address displayed is the one just examined, the next one to be examined, or the previous one. This is exactly the kind of small mental ``+1/-1`` trap in which I am perfectly capable of falling. In OTTO, I therefore chose to make the memory display follow the address actually being examined or deposited, so that the value shown always corresponds to the address shown.

Backward EXAMIN

For the same reason, I also added the possibility of examining memory backwards. This is not a revolutionary feature, of course, but it is surprisingly convenient when checking a recently entered program or correcting a few words after a mistake. Instead of reloading the starting address and moving forward again, one can simply step back.

Hardware Breakpoint

The most useful addition, however, is probably the hardware breakpoint. The breakpoint address can be loaded directly from the switch register, using a dedicated front-panel control, and then enabled or disabled as required. When the processor reaches that address, OTTO stops before executing the corresponding instruction. The address remains visible on the display, so that the instruction about to be executed can be examined, modified, or single-stepped. This is obviously a luxury feature compared with the original minimalist philosophy of the PDP-8, but it is extremely helpful when running diagnostic programs, experimenting with FOCAL, or simply trying to understand what a program is doing.

RAM Content

When generating the FPGA configuration file, you can also initialize the RAM with useful code, so that the required programs are already available at startup. I created several RAM images, but eventually decided to include both the RIM Loader and the Binary Loader, allowing any other software to be loaded quickly using the GBT terminal emulator.

In addition, I added a small front-panel function that I called A7777. It is activated by lifting, rather than pressing, the ADDR LOAD switch, and performs two operations in sequence: CLEAR, followed by ADDR LOAD with the switch register forced to 7777. As a result, starting the Binary Loader becomes very simple. Press A7777, then press CONT, and the loader starts immediately, ready to receive a file from GBT.

Momentary Switches

One of the aims of the OTTO project was to create a small-sized PDP-8 quasi-clone, although in the end I used a rather deep enclosure simply because I already had it. The front panel, however, is still small, and there was no room to waste.

For this reason, for the momentary controls I used center-off toggle switches, which in practice provide two switches in one. For example, the EXAM switch must be pressed downward, as on the original PDP-8, but when pressed upward it performs the backward EXAM function described above.

A similar solution was adopted for the SET BREAKPOINT function, which shares the same physical switch with DEPOSIT. In this case, however, I preferred to reverse the direction of DEPOSIT. On the original PDP-8, DEPOSIT is operated upward, but it is a frequently used command, and these inexpensive switches are not of the highest quality and are rather stiff. Moving DEPOSIT downward therefore makes the panel more comfortable to use.

Pin Assignment

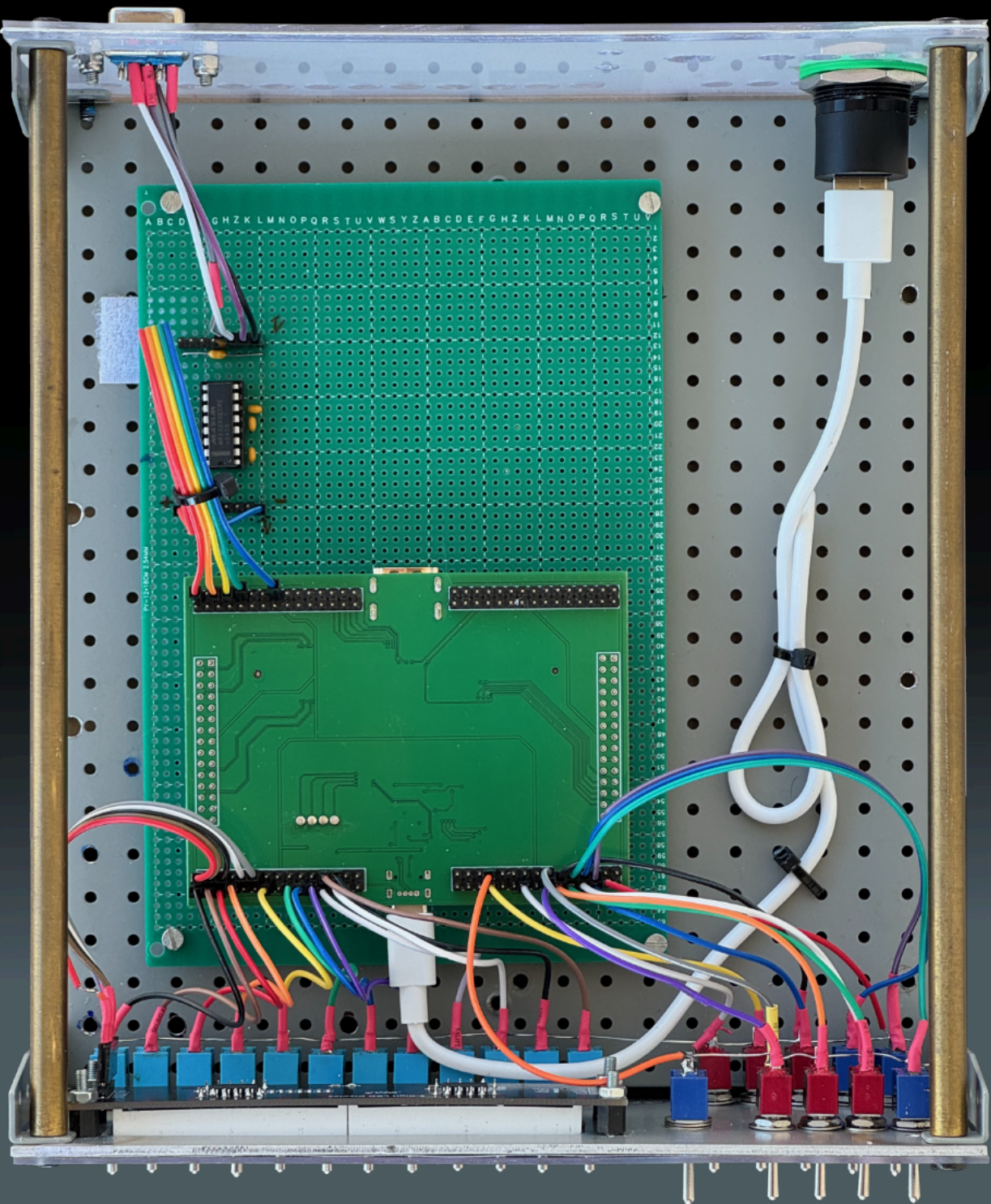
Muse Board Pin Assignment

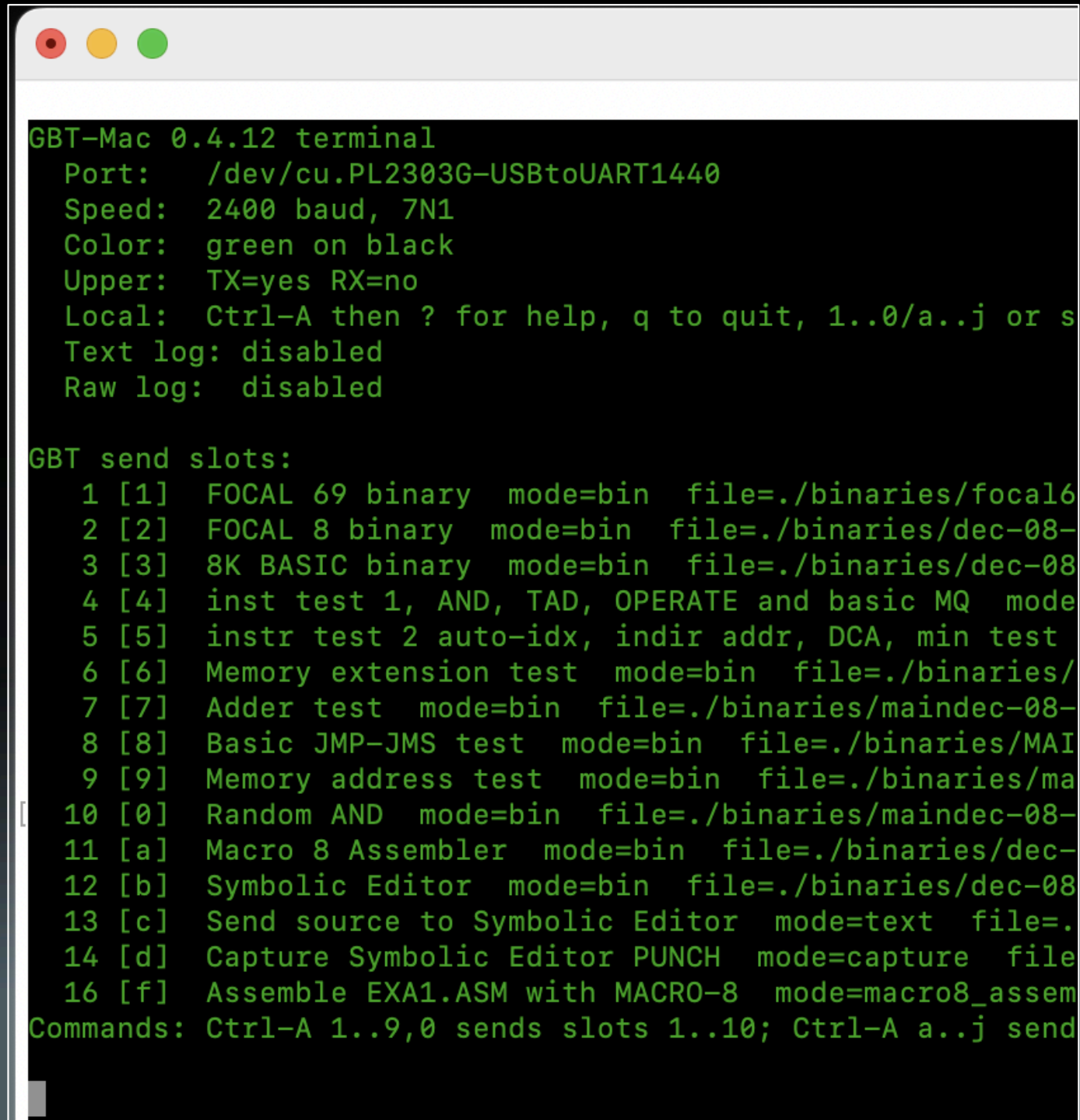
DIO	- red	- P6	pin 16	- PB4A	-
SCK	- brown	- P6	pin 17	- PB6A	-
RCK	- black	- P6	pin 18	- PB9A	-
SW00	- black	- P6	pin 15	- PB4B	-
SW01	- brown	- P6	pin 14	- PB9B	-
SW02	- red	- P6	pin 13	- PB6B	-
SW03	- orange	- P6	pin 12	- PL44A	-
SW04	- yellow	- P6	pin 9	- PB13A	-
SW05	- green	- P6	pin 7	- PB18A	-
SW06	- blue	- P6	pin 6	- PL38C	-
SW07	- violet	- P6	pin 5	- PL35C	-
SW08	- grey	- P6	pin 4	- PR35B	-
SW09	- white	- P6	pin 3	- PR20C	-
SW10	- black	- P6	pin 28	- PR11D	-
SW11	- brown	- P6	pin 27	- PL35D	-
EXAM	- blue	- P5	pin 3	- PL11A	-
EXAM BK	- black	- P5	pin 25	- PR5A	- D18
BRKSET	- green	- P5	pin 4	- PT6A	-
DEPOSIT	- white	- P5	pin 5	- PT58B	-
EXTADDR	- blue	- P5	pin 6	- PR2B	-
ADDRLD	- violet	- P5	pin 7	- PR5B	-
SSTEP	- violet	- P5	pin 28	- PL11C	-
HALT	- blue	- P5	pin 27	- PL11D	-
BRKON	- green	- P5	pin 26	- PL20B	-
A7777	- grey	- P5	pin 24	- PR2C	(E16)
MA-MD	- orange	- P5	pin 19	- PR8C	-
CLEAR	- grey	- P5	pin 9	- PR2D	-
CONT	- yellow	- P5	pin 12	- PR5C	-
RXD	- orange	- P2	pin 3	- PR20D	- K18 (RS-232 7)
TXD	- yellow	- P2	pin 4	- PR44D	- T18 (RS-232 6)
CTS	- orange	- P2	pin 5	- PR35D	- R17 (RS-232 5)

Consider this listing as a reference only. The actual pin assignments are defined in the colorlight_i5.lpf file. You may modify them as needed without changing the rest of the project.

Take care not to use pins reserved for dedicated functions, such as the serial port, the SPI interface for the flash memory, or other onboard peripherals.

Above: A top view of OTTO's internals. As you can see, the machine could be built in a much smaller enclosure, although the front panel itself would not be so easy to reduce. In any case, making the package too small would reduce the visual presence and perceived importance of the unit. After all, this is almost a PDP-8!





```

GBT-Mac 0.4.12 terminal
Port: /dev/cu.PL2303G-USBtoUART1440
Speed: 2400 baud, 7N1
Color: green on black
Upper: TX=yes RX=no
Local: Ctrl-A then ? for help, q to quit, 1..0/a..j or s
Text log: disabled
Raw log: disabled

GBT send slots:
1 [1] FOCAL 69 binary mode=bin file=./binaries/focal6
2 [2] FOCAL 8 binary mode=bin file=./binaries/dec-08-
3 [3] 8K BASIC binary mode=bin file=./binaries/dec-08
4 [4] inst test 1, AND, TAD, OPERATE and basic MQ mode
5 [5] instr test 2 auto-idx, indir addr, DCA, min test
6 [6] Memory extension test mode=bin file=./binaries/
7 [7] Adder test mode=bin file=./binaries/maindec-08-
8 [8] Basic JMP-JMS test mode=bin file=./binaries/MAI
9 [9] Memory address test mode=bin file=./binaries/ma
10 [0] Random AND mode=bin file=./binaries/maindec-08-
11 [a] Macro 8 Assembler mode=bin file=./binaries/dec-
12 [b] Symbolic Editor mode=bin file=./binaries/dec-08
13 [c] Send source to Symbolic Editor mode=text file=.
14 [d] Capture Symbolic Editor PUNCH mode=capture file
16 [f] Assemble EXA1.ASM with MACRO-8 mode=macro8_assem
Commands: Ctrl-A 1..9,0 sends slots 1..10; Ctrl-A a..j send

```

Teletype Emulator

The Need

One might think that almost any terminal emulator would be suitable for emulating a Teletype ASR-33, but that is not the case. The first problem is that most modern hardware cannot operate at 110 baud. Even when the PDP-8 or OTTO is set to 2400 baud, many essential functions are still missing, such as emulation of the paper tape reader and punch.

This led to GBT, an application created specifically for this purpose, which eventually became a project within the project.

GBT Overview

GBT is a small terminal and paper-tape utility written to support the use of a PDP-8, or a PDP-8-compatible machine such as OTTO, from a modern computer. It is not intended to be a general-purpose terminal emulator. Its purpose is more specific: to reproduce, as faithfully as possible, the practical functions that were once performed by a Teletype, a paper-tape reader, and a paper-tape punch.

In normal terminal mode GBT behaves like a simple serial terminal. Characters typed on the Mac keyboard are sent to the PDP-8, and characters received from the PDP-8 are displayed on the Mac screen. This is the mode normally used with FOCAL, BASIC, ODT, the Symbolic Editor, or any program expecting an interactive terminal. The terminal path usually operates in 7-bit ASCII, with no parity and one stop bit.

The program also includes several controlled file-transfer modes. A binary paper-tape image, such as a RIM, BIN, MAINDEC, FOCAL, BASIC, or MACRO-8 tape, can be sent as an 8-bit stream. When the PDP-8 interface provides a Reader Enable signal, GBT can use it as a pacing signal, sending each byte only when the PDP-8 is ready to receive it. In the OTTO setup described here, this signal is carried on CTS. This makes file loading much more reliable than a simple timed upload.

Left: The GBT screen. The program provides convenient access to a small library of original DEC software and diagnostic tapes, which can be uploaded and run on OTTO or on a real PDP-8. The collection includes hardware test routines, useful for checking OTTO's compatibility with the original machine, together with classic PDP-8 software such as FOCAL, BASIC, MACRO-8, and the Symbolic Editor.

Text transfer is handled differently. When sending a source file to the Symbolic Editor, the file is transmitted as ordinary 7-bit text, with small delays between characters and a longer delay after each carriage return. This imitates the behavior of a human operator or a slow terminal closely enough for the editor to keep up. Text files created on modern systems may contain LF or CR/LF line endings; GBT can translate these into the carriage returns expected by the PDP-8.

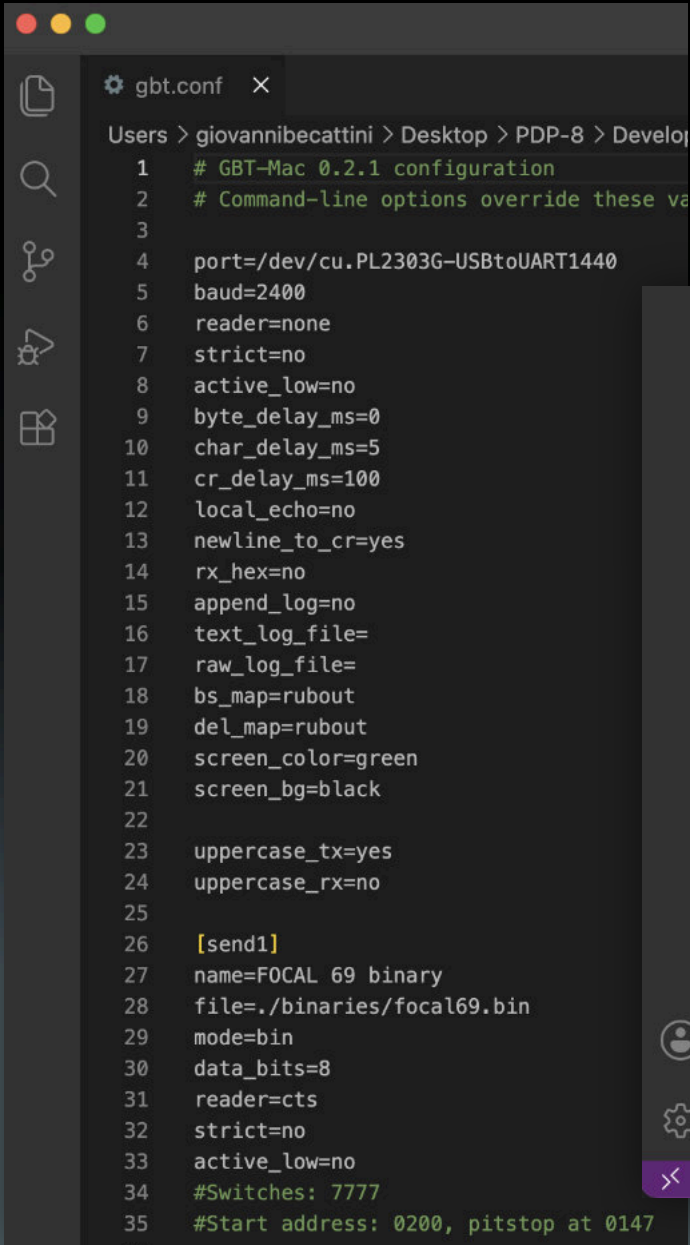
A special mode is provided for MACRO-8. This assembler has rather peculiar paper-tape input requirements. A simple text upload is not sufficient: MACRO-8 expects its source as a paper-tape stream with the high bit set. It also performs its assembly in several passes, and can produce either a binary output tape or a listing. GBT therefore includes a guided MACRO-8 workflow. It sends the source for the first pass, waits for the operator when the PDP-8 stops, sends the source again while capturing the generated binary tape, and can optionally perform the third pass to capture the listing. The captured binary file can be given a synthetic leader, while the listing is cleaned so that it becomes an ordinary text file usable with modern editors.

GBT can also capture output produced by the PDP-8. This is useful, for example, when the Symbolic Editor executes its `PUNCH` command. The operator starts the punch operation on the PDP-8, selects a capture slot in GBT, and then continues the PDP-8 program. The received characters are written to a file until the line becomes silent for a configurable time, or until the user stops the capture manually. In this way a punched paper tape can be replaced by a file on the Mac.

The configuration file, `gbt.conf`, is deliberately simple. It contains global serial and terminal settings, followed by up to twenty operation slots. Each slot describes one useful action: load FOCAL, load BASIC, send a source file to the Symbolic Editor, assemble a file with MACRO-8, capture a punched source, and so on. Slots may also include notes such as the required switch-register value, the start address, or any operational warning. These notes are shown only when the slot is selected, keeping the slot list compact.

In practice, GBT becomes a small software front panel for paper-tape work. It does not try to hide the PDP-8 operating procedure: the user still loads RIM, starts the Binary Loader, sets the switch register, presses `ADDR LOAD`, `CLEAR`, `CONT`, or raises the required switches when the original software expects it. What GBT does is remove the mechanical part of the process. The paper tapes are files, the reader is paced by CTS, and the punch can be captured directly into a modern text or binary file.

For the OTTO/PDP-8 configuration used here, the usual serial speed is 2400 baud. Interactive terminal and Symbolic Editor operations use 7N1. Binary paper tapes and MACRO-8 assembly use 8N1. Reader Enable is normally connected to CTS, with loose pacing rather than strict ON/OFF handshaking. These settings proved reliable for loading DEC software, running MACRO-8, capturing listings, and exchanging source files with the Symbolic Editor.



Two fragments from the GBT configuration file. The program behavior is controlled by several parameters, and the same file can also define a number of ready-to-use “slots”. Each slot corresponds to a specific operation, such as loading FOCAL or BASIC, sending a source file to the Symbolic Editor, running MACRO-8, or capturing a punched output file. This makes repetitive PDP-8 software operations much faster and less error-prone (see the GBT readme.txt for details.)

Library of Vintage Electronics

Tektronix

Tektronix Epic Oscilloscopes

Tektronix and its most classical oscilloscopes

Hard cover, 600 pages

www.elektor.com

Tektronix Oscilloscopes Restoration Guide

An illustrated handbook dedicated to vintage scopes repair and preservation.

Hard cover, 380 pages

www.elektor.com

Tektronix 7000-Series Mainframes

Models, technologies and restoration techniques.

Hard cover, 441 pages

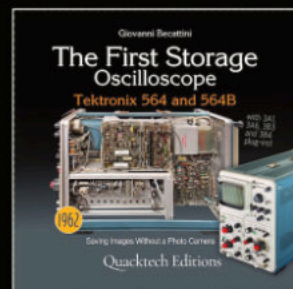
www.elektor.com

Tektronix 7000-Series Plug-ins

Models, technologies and restoration techniques.

Hard cover, 408 pages

www.quacktech.it

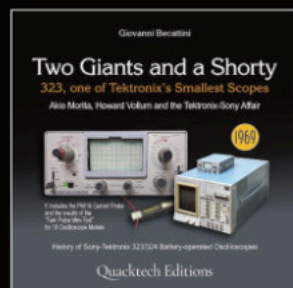


The First Storage Oscilloscope

Tektronix 564 and 564B oscilloscopes

Paperback, 152 pages

www.quacktech.it

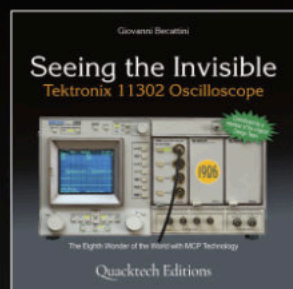


Two Giants and a Shorty

Sony Tektronix joint venture and 323/324 oscilloscopes

Paperback, 104 pages

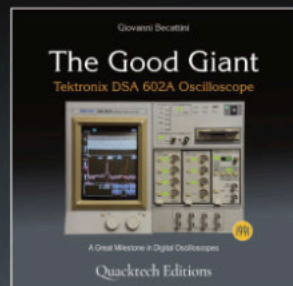
www.quacktech.it



Seeing the Invisible

Tektronix 11320 oscilloscope
Paperback – Full color – 106 pages

www.quacktech.it

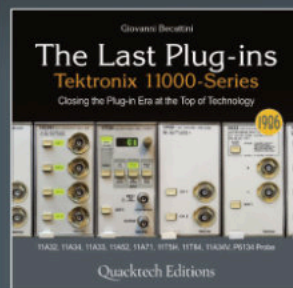


The Good Giant

Tektronix DSA 602A oscilloscope

Paperback, 106 pages

www.quacktech.it

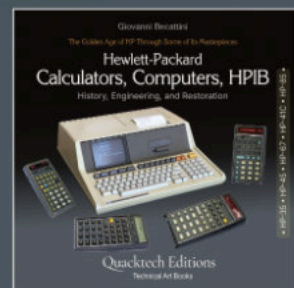
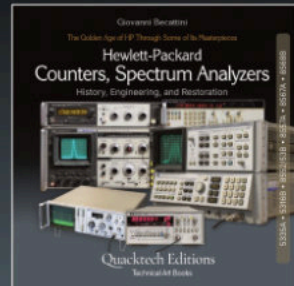
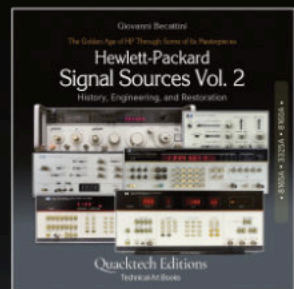
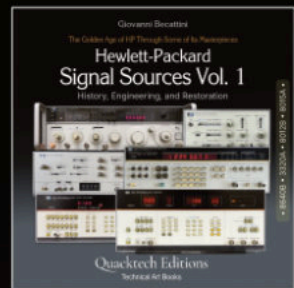
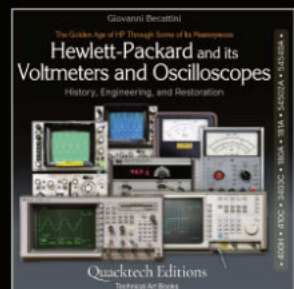


The Last Plug-ins

Tektronix 11000-Series Plug-ins

Paperback, 120 pages

www.quacktech.it



Hewlett-Packard

HP Voltmeters and Oscilloscopes

400H, 410C, 3403C, 180A, 181A, 54502A, 54540A

www.quacktech.it

HP Signal Sources Vol.1

History, Engineering, and Restoration

8640B, 3320A, 8012B, 8015A

www.quacktech.it

HP Signal Sources Vol.2

History, Engineering, and Restoration

8165A, 3325A, 8160A

www.quacktech.it

HP Counters, Spectrum Analyzers

History, Engineering, and Restoration

5335A, 5316B, 8552/53B, 8557A, 8568B

www.quacktech.it

HP Calculators, Computers, and HPIB

History, Engineering, and Restoration

HP-35, HP-45, HP-67, HP-41C, HP-85

www.quacktech.it

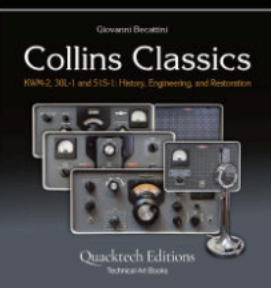


Radio & Comm

Vintage Radio Equipment

Hardcover 590 pages about radios equipment from WW-II to 70s

www.elektor.com



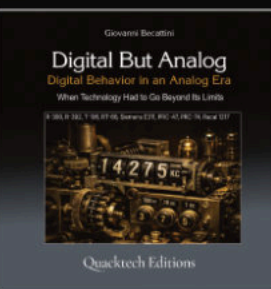
Collins Classics

KWM-2, 30L-1 and 51S-1

History, Engineering, and Restoration

Hardcover or paperback 350 pages

www.quacktech.it

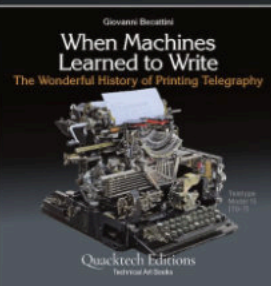


Digital But Analog

When Technology Had to Go Beyond Its Limits

R-390, R392, T-195, Siemens E311, PRC-47, PRC-74, Racal 1217

www.quacktech.it



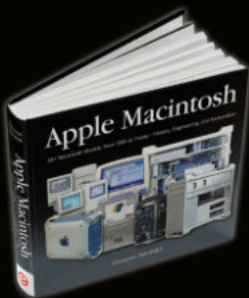
When Machines Learned to Write

The wonderful history of Printing Telegraph

Teletype TG-7, Model 15 etc.

www.quacktech.it

Quacktech Editions
Technical Art Books



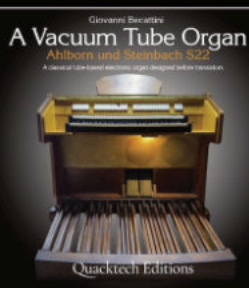
Apple

Apple Macintosh

20+ Macintosh Models, from 1984 to Today - History, Engineering, and Restoration

Hard cover, 600 pages

www.elektor.com



Other

A Vacuum Tube Organ

Ahlborn & Steinbach S22

The story and the restoration of a classical vacuum tube organ

Paperback, 70 pages

www.quacktech.it

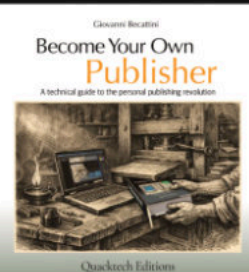


Strumentazione Vintage

Storia, Tecnica e Restauro

TS-505, TS-375A, TS-585, TV-7D, ZM-3, ZM-11, BC-221, TS-497B Tektronix 143, Philips PM 5501, 3350, 3217, Millen 90651, DuMont OS-34/USM-32, LeCroy 9361, Advantest R3132

www.edizionicec.it



Become Your Own Publisher

A technical guide to the personal publishing revolution

Paperback, 202 pages

www.elektor.com



Diventate editori di voi stessi

Una guida tecnica alla rivoluzione del personal publishing

Paperback, 202 pages

www.quacktech.it

Quacktech Editions

Technical Art Books

Quacktech Editions is an independent publisher dedicated to celebrating the remarkable heritage of electronic engineering.

We believe that the great masterpieces of technology whose stories are told in our pages—and the visionaries who created them—have long deserved to be celebrated in real art books, made with true luxury paper and carefully designed layout, imagery, structure, and typography, to offer the best possible physical reading experience. An experience that can never be conveyed by a mere, cold electronic document.

This commitment to quality, combined with the convenience of print-on-demand production, allows you to select and purchase only the titles that truly interest you, without being forced into large and costly volumes.

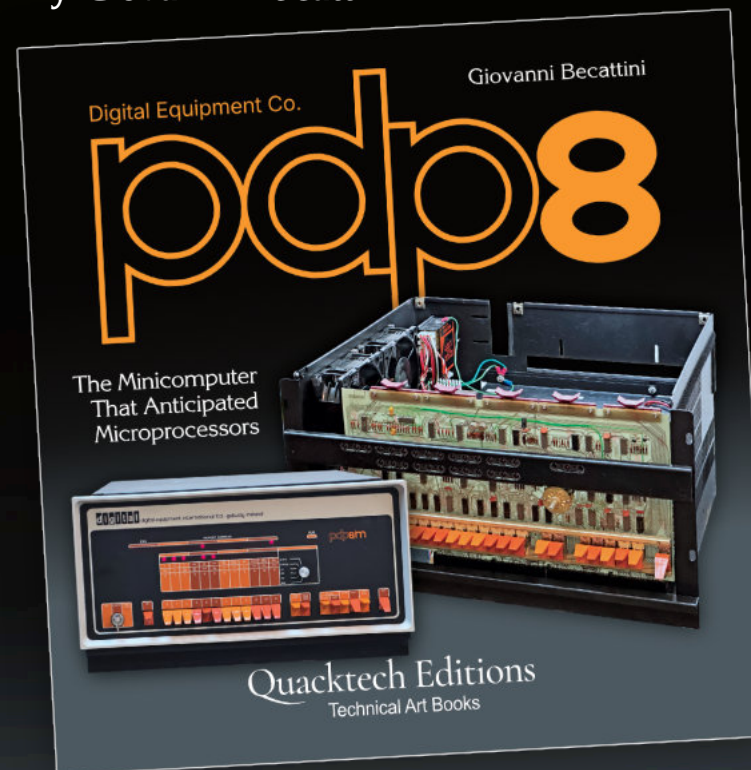
Each book in our collection is the result of extensive research and hundreds of hours of meticulous work. We strive for accuracy, clarity, and visual excellence, and—why not—a touch of humor, always paired with rigorous technical content, to honor the beauty of these extraordinary achievements.

Although every effort has been made to ensure accuracy, occasional errors may still be present. If you discover any mistake, however small, I would be grateful if you let me know. Your feedback helps ensure that future readers receive an improved edition. You can contact us on www.quacktech.it. Thank you for contributing to the growth of this work.

Digital Equipment PDP-8

The Minicomputer That Anticipated Microprocessors

by Giovanni Becattini



The PDP-8 was more than a small computer. It was one of the machines that helped move computation out of the data center and into laboratories, factories, instruments, and workshops.

This book tells its story from several perspectives: the rise of Digital Equipment Corporation, the architecture of the PDP-8, its boards and core memory, the software that gave it life, and the restoration of a real PDP-8/M after decades of silence.

It is also a personal journey into the logic of a deceptively simple machine, recreated once more in programmable logic to show that, even today, the elegance of the minicomputer age can still be understood, rebuilt, and enjoyed.

424 pages, from www.quacktech.it

Freely downloadable resources

Freely downloadable resources:

- this eBook
- the Verilog source code of OTTO;
- the information required to implement OTTO on the Colorlight i5 module;
- the GBT terminal emulator, including its source code and Mac executable;
- a software package containing the diagnostic tests and application programs that I have tested and that can be uploaded to OTTO using GBT.

From www.quacktech.it

- Saved Jul 4, 2026, 17:52
- Released Jul 4, 2026, 18:01

