A Triptychon of Digital Circuits

Niklaus Wirth, August 2005

Here we describe an experiment and tutorial about various ways to implement digital circuits. Our example is a binary counter and a shifter, implemented once by ordinary TTL MSI chips, once by a PLD, and once by a microcontroller. They represent the technologies that emerged between 1975 and 1995.

The three versions were built on printed circuit boards with counter and shifter driving light emitting diodes (LED). Put side by side in a frame, they look like a triptychon (see Fig. 1).

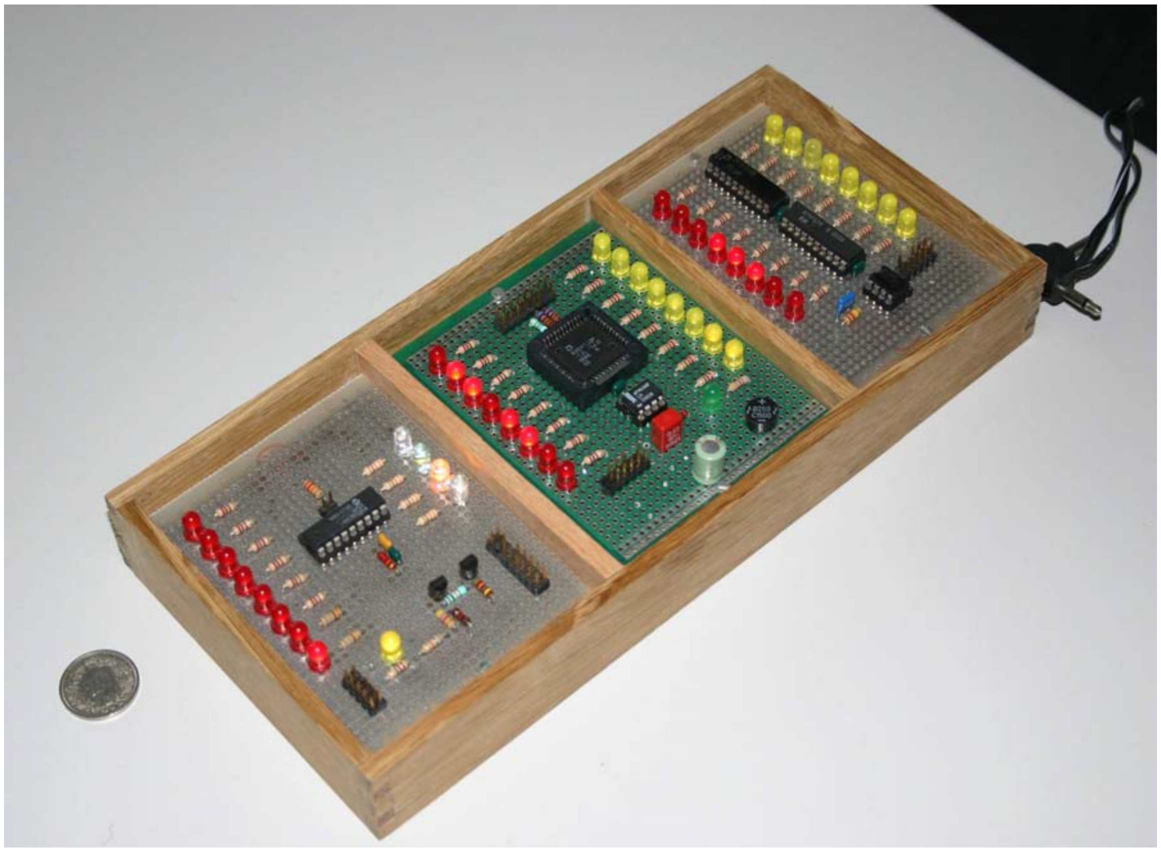

Fig.1. The triptychon of circuits

1. The Circuit

The circuit chosen as a sample to illustrate the three technologies consists of an 8-bit synchronous binary counter and an 8-bit shifter, each driving 8 LEDs. Let sequences of 8 ones followed by 8 zeroes be rippling through the shifter.

An n-bit counter can be described by the following diagram for each cell. All 16 registers are driven by the same clock. All three versions of the circuit require a single 5V power supply.

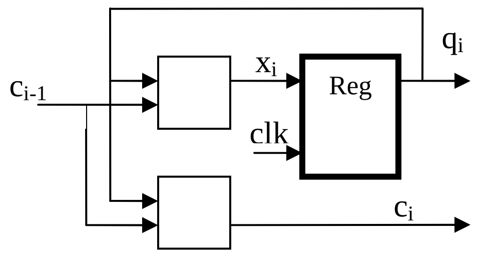

Fig.2. Counter cell

The combinational functions for i = 1 … n are

$x_i = q_i$ xor c_{i-1}
$c_i = q_i$ and c_{i-1} $\quad c_0 = 1$

The shifter consists of n registers with $x_i = q_{i-1}$.

2. The TTL version

Our TTL version consists of 2 MSI (medium scale integration) chips, namely an F579 8-bit counter and an LS299 8-bit shifter We use a simple RC-oscillator based on a N555 timer chip, and a frequency of about 8 Hz. The circuit is shown in Fig. 3. The requirement of shifting sequences of 8 ones followed by 8 zeroes is met by using q3 of the counter as serial input to the shifter.

The circuit consumes about 200mA current. This is high, because of the use of a fast chip of the F class (Signetics).

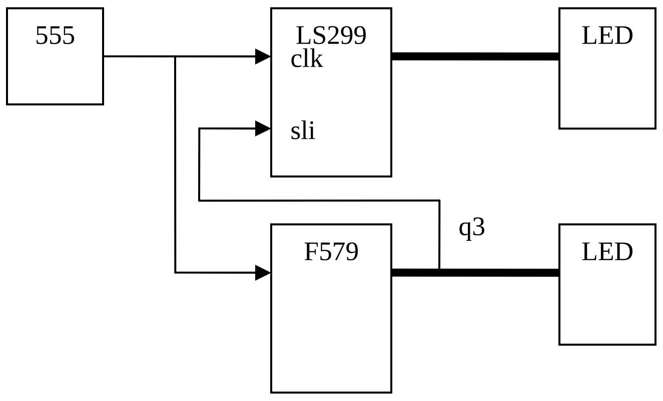

Fig. 3. Shifter and counter with 3 TTL chips

3. The PLD version

It is possible to construct a counter and a shifter together on a single, programmable chip, on a programmable logic device. These devices were first produced around 1975. They typically contain a matrix of and-gates followed by a matrix of or-gates, which in principle allow the design of an arbitrary logic function of a number of inputs. These were called *programmable logic arrays* (PLA). The addition of a row of registers to the outputs of the or-matrix led to the possibility to construct arbitrary *finite state machines* (FSM), and the combination of gate matrices and a register row was called *programmable logic device* (PLD).

Since both counters and shifters are intrinsically finite state machines, they can obviously be implemented using a PLD by mapping their logic functions onto the two matrices. The one technical question is, whether the chosen chip contains gate matrices that are large enough to represent both circuits. For our purpose we chose AMD's MACH 211SP chip. Even by 1996, and much more so 10 years later, PLDs contain very large gate arrays, and they are much more than large enough for our modest task. The circuit now consists of a single PLD plus the clock generator, as shown in Fig. 4. The circuit draws 50mA at 5V.

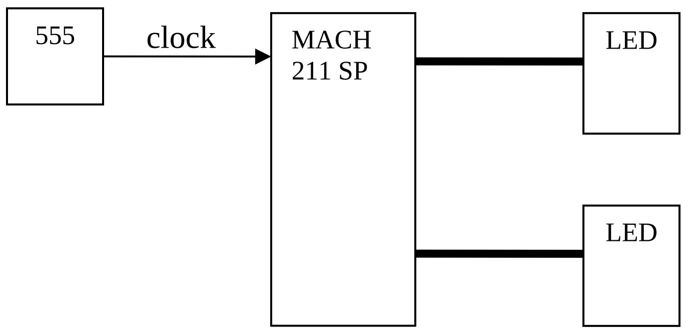

Fig. 4. Shifter and counter with PLD

What determined the choice of the MACH 211SP was the fact that its configuration, i.e. the data that render the gate arrays into the desired circuit, is held in an *erasable* memory. In early PLDs, this configuration data were burnt into "fuses", thus confining the chip to the one chosen purpose once and forever. Here, however, the information is held in an EEPROM, an electrically erasable programmed read-only memory. This makes this chip particularly suitable for experimentation. The configuration data are computed by a program and loaded via the chip's configuration pins. The input of said program is typically a set of logic equations written in a certain hardware description language (HDL).

We specify the desired circuit in the simple hardware description language Lola as follows. The variables q0, q1, etc. are represented by the mentioned registers of the PLD.

```
module ShiftCount;
  out q0, q1, q2, q3, q4, q5, q6, q7, s0, s1, s2, s3, s4, s5, s6, s7: bit;
begin
  s0 := reg(q3); q0 := reg(~q0);
  s1 := reg(s0); q1 := reg(q1 - q0);
  s2 := reg(s1); q2 := reg(q2 - q1*q0);
  s3 := reg(s2); q3 := reg(q3 - q2*q1*q0);
  s4 := reg(s3); q4 := reg(q4 - q3*q2*q1*q0);
  s5 := reg(s4); q5 := reg(q5 - q4 *q3*q2*q1*q0);
```

```
    s6 := reg(s5); q6 := reg(q6 - q5*q4*q3*q2*q1*q0);
    s7 := reg(s6); q7 := reg(q7 - q6*q5*q4*q3*q2*q1*q0)
  end ShiftCount.
```

4. The micro-controller version

The third and last of our implementation of the shifter-counter pair is based on a micro-controller. For this purpose, we chose Microchip's PIC16F84. This represents a complete, programmable computer with 13 input/output ports (pins). It even contains its own oscillator circuit (with external R/C), and thus reduces our implementation to one single chip. The circuit is shown in Fig. 5.

The following program contains variables *sh* and *cnt* representing the values to be shifted and counted. In each step, which is artificially prolonged by a delay loop, the values are output to the ports *A* and *B* driving LEDs. (*S* is the PIC's status register).

```
module ShiftAndCount;
    int sh, cnt, x;
begin !S.5; A := $F0; B := 0; !~S.5;  (*configure ports to output*)
    sh := 0; cnt := 0;
    repeat A := cnt; B := sh; x := 255;
        repeat x := x-1 until x = 0;   (*delay*)
        cnt := cnt - 1; sh := sh + sh;   (*shift left sh*)
        if cnt.3 then !sh.0 end
    end
end ShiftAndCount.
```

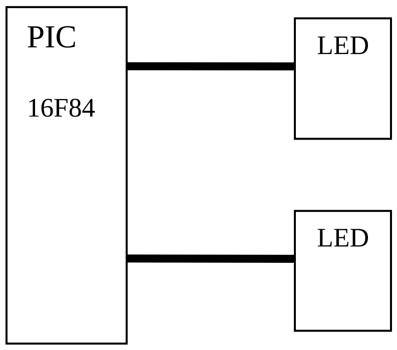

Fig. 5. Shifter and counter using micro-controller

5. Analysis and Discussion

Well, what is the point in these three circuits? After all, they show exactly the same pattern of changing lights representing shifts and counts. Would not one circuit be quite enough?

One possible answer is that the circuits demonstrate in a nutshell that the same effect can be obtained in many different ways, even with vastly different means. A further answer is that they show in an exemplary way the reduction of the number of components (chips) in the past decades due to technological progress. Whereas in 1950 hundreds of vacuum tubes were requested for this task, and in 1960 many hundreds of transistors, in 1970 only 3 chips fulfilled the same task; in 1980 there were 2, and in 1990 a single one would do.

However, our essay may also show that the popular measure of chip count does no longer serve as a measure of complexity, and in fact is misleading. Our solution based on the PLD actually features a much more complex circuit with many more transistors (many of which remain unused). The PLD has a very regular structure, allowing for high density of circuit elements, whereas the solution with the micro-controller is even more complex and less regular. Semiconductor technology has made tremendous advances, with the effect of obscuring relationships in logical complexity. While miniaturization has made many things technically feasible which were impossible earlier, it has also encouraged waste of things having become abundant. Thereby it has often led to sloppy design, and to unnecessary features and facilities, just to make some use of otherwise unused resources.

The most fundamental difference, however, is not only a matter of technology, but of principle. The third version uses what we call *software*, whereas the first is a pure *hardware* solution. It is a digital circuit, and nothing else. The micro-processor, on the other hand, is executing a program. So far, so good. But what about the PLD-solution? There is no processor running a program, but the circuit is said to be programmed or, if you wish, *configured*. Is this also software? And if so, what *is* software?

One is tempted to offer the easy answer that hardware is what you can see and touch, that has to be fabricated, while software is what is stored in memory, and therefore is volatile and can (too) easily be changed. But there are two arguments which debunk this answer as being too simplicistic. What about the software being stored in a read-only memory, or perhaps a Flash memory? Here storing, writing, is called *programming*. In early ROMs this implied the burning of “fuses”, thus establishing the actual circuit. Second, PLDs store their configuration also in a memory, but still what is loaded is not called a program. There have been attempts to design dynamically reconfigurable PLDs (or FPGAs), letting the configuration, i.e. the circuit change while in use. Software changing the circuit, or the circuit changing software? A confusing and risky business!

Perhaps the one truly concrete answer is that hardware requires materials and a fabrication, whereas software is just “written”. However, modern methods of design also obscure this distinction. Both software and hardware are specified by texts in a formal notation, called *language*. They undergo a translation or process yielding machine code or a layout obeying certain design rules. Attempts have been made to translate programs into circuits directly, without involving their execution on a computer.

Perhaps it is the design process and the dominant consideration of the designer that may help us to find a genuine difference between hardware and software, perhaps causing us to look at them from a different perspective.

The computer as we know it today, has its origin in the idea of breaking up a complex task into smaller tasks. Subtasks are further broken up until they are reduced to a computer’s instructions. The gain is that with a relatively small set of instruction one may compose solutions to very complex computations. This was mandatory, because circuits were very expensive, and because a very large number would have reduced reliability and mean time between failures. The price was time, because the instructions are performed purely sequentially, always using the same circuit elements. Even the simplest computers are able to perform any task, given enough time. The key idea is *sequentiality*.

With hardware becoming cheaper, computers were designed with more complex instructions in the hope of reducing the break up process. This proved to be unsuccessful. Modern computers use very simple, elementary instructions. The computer designers' cleverness is directed purely at making them execute faster and faster.

Designers of circuits also have an execution process in mind. Its steps are called *clock cycles*. Committing a slight simplification, we may say that all elements of their circuit coexist and perform concurrently. The designer concentrates on having as many elements as possible actively contributing to progress in as many cycles as possible. The key idea is *concurrency*.

At this point we must be aware that concurrency in software, without being backed up by concurrent hardware, is always an illusion. Systems featuring so-called *threads* (coroutines) do not perform concurrent processes, unless they are based on hardware containing multiple processors. At best, they perform quasi-concurrent processes, they give the illusion of concurrency, switching the processor from one thread to another, sequentially.

Does then, perhaps, concurrency vs. sequentiality mark the key difference between hardware and software? This idea gains plausibility, if we realize that both are essentially specified by sets of logical expressions and assignments. Even a static circuit description looks remarkably similar to a program.

This suggests that the traditional dividing line perhaps ought to be drawn at a different level: Hardware has substance, hence obeys the laws of physics, the laws of electricity. Hardware is not (only) described by logical expressions, but (also) by physical devices. Intrinsically involved are current, voltage, charge, capacitance, conductivity, and even the laws of quantum physics. We should be fully aware that digital circuits are an abstraction. In reality they are always "analog". The circumstance that transistors are typically either shut off or driven into saturation does not contradict their "analog" nature. And dynamic memory cell are anything but digital! Modern ones even store not only one, but several bits, represented by distinct ranges of voltage levels.

It is time to rethink our traditional, hazy distinction between hardware and software. But perhaps this is quite irrelevant.