Three Counters

Niklaus Wirth, October 2015

This short paper shows, how a simple counter is implemented on various levels of a system. The System is based on an FPGA, and it also refers to the software system Oberon. The paper is intended as a tutorial, explaining the structure and basic functioning of a whole system. It is almost of a philosophical nature :-).

1. The Counter as a Circuit

The underlying device is not a computer, but a field-programmable gate array (FPGA). The circuit which it represents can be programmed, that is, changed according to needs. The FPGA consists of gates, registers, memory blocks, and a *configuration memory.* This memory is loaded with a configuration from a host computer, on which the circuit is specified, typically in a hardware-description language (HDL), and then compiled (synthesized).

For this purpose we use the language *Lola*. A circuit description (program) is then translated from Lola to the language Verilog, from where it can be synthesized and downloaded.

https://www.inf.ethz.ch/personal/wirth/Lola/index.html

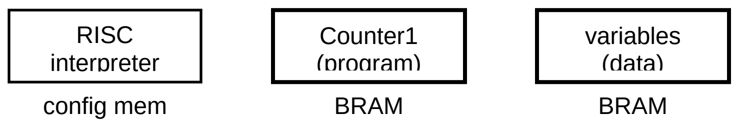
```
MODULE Counter0 (IN CLK50M, rstIn: BIT;
    IN swi: BYTE; OUT leds: BYTE);
REG (CLK50M) rst: BIT;
    cnt0: [16] BIT;  (*milliseconds*)
    cnt1: [10] BIT;  (*half seconds*)
    cnt2: [8] BIT;
VAR tick0, tick1: BIT;
BEGIN leds := swi.7 -> swi :  cnt2;
    tick0 := (cnt0 = 49999);
    tick1 := tick0 & (cnt1 = 499);
    rst := ~rstIn;
    cnt0 := ~rst -> 0 : tick0 -> 0 : cnt0 + 1;
    cnt1 := ~rst -> 0 : tick1 -> 0 : cnt1 + tick0;
    cnt2 := ~rst -> 0 : cnt2 + tick1
END Counter0.
```

In fact, this is more than a counter. It describes 3 counters. Assuming that the FPGA is clocked by a 50 MHz oscillator, the first counter *cnt0* generates a pulse *tick0* ,every millisecond. The second counter *cnt1* generates the pulse *tick1* every half second. This pulse is used to drive the third counter *cnt2* that is displayed on 8 LEDs. All counters are synchronous, driven by the same clock, and they can be reset by the signal *rst.*

2. The Counter as the Program

Here we consider the counter as a program. In order to interpret (execute, run) the program, we need a computer. Fortunately, the FPGA is flexible enough to represent an entire computer, a processor. We chose the processor RISC https://www.inf.ethz.ch/personal/wirth/ProjectOberon/index.html. Now the configuration memory is used to contain the circuit representing the RISC processor in place of the counters. The program for the counters is now described in the programming language *Oberon* https://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf. It is compiled on a host computer and downloaded. The compiled code now resides in a memory (a BRAM) in the FPGA, and so do the variables of the program.

```
MODULE Counter1;
   VAR x, y: INTEGER;
BEGIN y := 0;
   REPEAT LED(y); x := 1000000;
   REPEAT x := x-1 UNTIL x = 0;
   y := y+1
  UNTIL FALSE
END Counter1.
```



Whereas in the first scenario the configuration memory is reloaded, whenever a new application is to be launched, in the second scenario *the configuration memory remains untouched*, and only the new program is loaded into BRAM. The RISC interpreter is typically loaded on system reset from a small flash ROM.

As an aside, we here point out the apparent similarity of the hardware program in the first, and the software program in the second scenario. Indeed, the languages (formalisms) Lola and Oberon have a similar appearance. This hides the inherent difference between hardware and software. In the first case all elements work concurrently, all assignments take place concurrently. In a software process they occur sequentially, strictly one after the other.

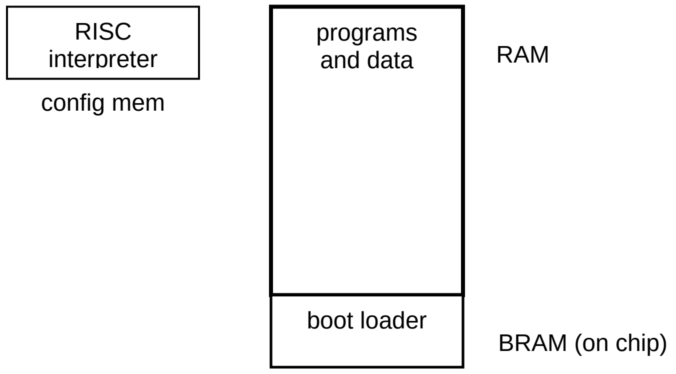
What makes the hardware version "run" lies in the difference between variables (signals) and registers.. The latter delay a signal by one clock cycle And we consider synchronous circuits only, in which all registers are ticked by the same clock signal. Only registers thus make time enter the picture.

3. The Counter as a Program within an Operating Environment

In this third scenario, the FPGA is not only to host an interpreter, but to run under an entire operating system capable of editing, compiling, and loading programs. As before, the configuration memory contains the RISC interpreter. The program code and data require a larger memory. It is typically a RAM located physically external to the FPGA, but rather on the same board as the FPGA.

Now this memory is supposed to always contain a program loader (boot loader). Upon system reset, not only is the configuration memory loaded with the RISC interpreter,also the memory is loaded. But after startup, unlike in the previous scenario, the BRAM containing the boot loader remains untouched.

If memory is located external to the FPGA, there is no built-in mechanism to load that memory. We circumvent this problem by retaining one FPGA-internal BRAM (and map it onto the ordinary address space). Now the start-up process loads the boot loader into this BRAM, and control is transferred to the loaded program after the loading is completed.



All this boils down to the fact that system reset is far from a trivial process. Still, on a Spartan-3 board, loading the entire Oberon OS takes less than 2 seconds.

An Afterthought

Most of the millions of owners of computers work in scenario 2. They always use a single program, a combination of text editor, slide editor, e-mail handler, and an Internet browser. They do not program. Evidently, they exploit the flexibility and capability of their computer only to a small fraction.