# Development of libraries for radiofrequency instrumentation using FPGAs

N. Chrétien[1,2], M. Lamothe[1,2], G. Goavec-Mérou[3], J.-M Friedt [2]

[1] Institut FEMTO-ST, Besançon

[2] Association Projet Aurore, Besançon

[3] Armadeus Project

Slides available at `http://jmfriedt.free.fr`

July 4, 2010

# Introduction

Facts:

- a general-purpose processing unit is unable to generate signals with frequencies above a few MHz (a few kHz when an OS is running)
- most signal processing tasks are complex to implement in low-level languages (assembly language, VHDL)
- using an operating system removes the need to waste time developing some basic tasks (scheduler, communication over a network, data storage, memory management ...)

Objective: complement a processor running an operating system with a software-reconfigurable electronc gate matrix array (FPGA), and hence get the best of both architectures.

application to reconfigurable **radiofrequency instruments**

# FEMTO time & frequency department

Provide instruments as support for the development of radiofrequency sensors

Flexibility of the software approach ⇒ developments on platforms combining a general purpose CPU and an FPGA:

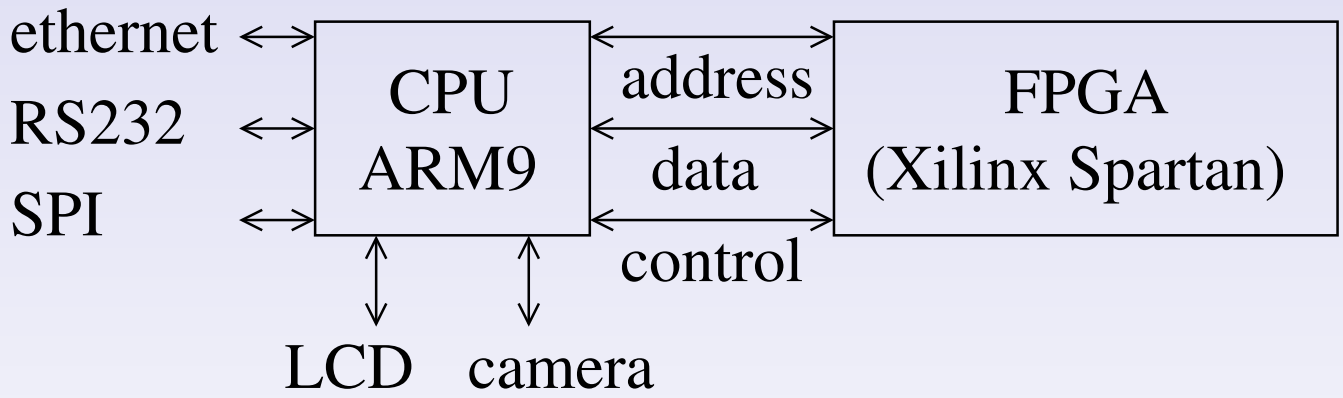
- T. Rétornaz (frequency counter + camera), Armadeus APF9328 [1]
- T. Rétornaz (software defined radio), Ettus USRP [2]
- G. Goavec-Mérou[1] (xenomai and latency measurement using the FPGA), Armadeus APF9328 & APF27 [2]
- G. Goavec-Mérou (microsystem control), Armadeus APF27 [3]
- N. Chrétien (fast sampling/RADAR application), APF9328
- M. Lamothe (high resolution frequency counter), APF9328

[1] `http://free-electrons.com/pub/video/2008/rmll`
[2] `http://free-electrons.com/pub/video/2009/rmll`
[3] dMEMS 2010, Besançon, France (2010), `http://trabucayre.com/conf/presentationDMEMS.pdf`

[1] funded by Armadeus Systems

# Platform selection

- Opensource tools (or at least free of charge: Xilinx ISE)
- Processor running GNU/Linux
- User communauty, wiki & sample progam database
- Common busses (data/address) between the CPU and FPGA for efficient data sharing




1. presentation of IP management (VHDL) in the FPGA (POD)
2. efficient communication between the CPU-FPGA (hardware aspects + software/ Linux kernel module)
3. practical applications

[1] `http://free-electrons.com/pub/video/2009/rmll`

# The Wishbone bus

The Wishbone interface is a bus optimized for reconfigurable hardware (similar to the Avalon bus used by Altera) based on freely available specifications.

Advantage of using the Wishbone bus:

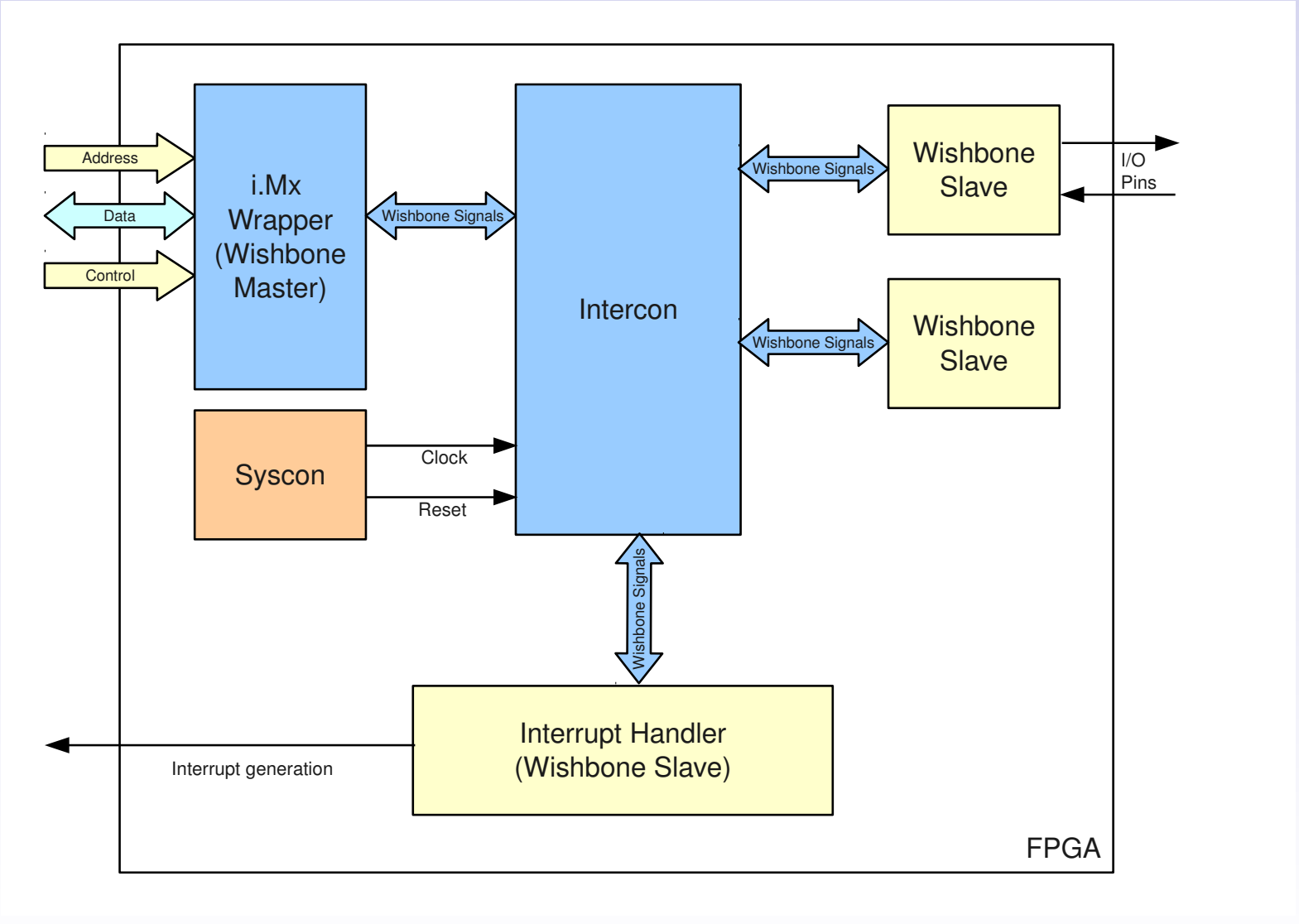
- Rational communications FPGA-CPU communication
- Ease the addition of new IPs to an existing environment
- Opensource communication bus dedicated to FPGAs

Components (VHDL) used for a functional bus:

- i.MX Wrapper: the microprocessor/FPGA interface,
- Syscon: CLK signal generation (directly provided by the i.MX CPU) and RESET (synchronous),
- Intercon: links all the components connected to the Wishbone system,
- Interrupt manager: interrupts sent to the CPU,
- Wishbone slaves: final application components.

# The Wishbone bus

Bus structure:



# Presentation of POD

"Peripherals On Demand" [2] is an opensource application, written in Python and developed by Armadeus Systems, helping the integration of virtual peripherals (components) in an FPGA

Advantages of POD:

- uses external applications (Xilinx ISE, Altera Quartus) to generate the bitstream for configuring FPGA,
- multi-platform (Windows, Linux, MacOS).
- automatic generation of most mandatory components for using the Wishbone bus, in VHDL or Verilog,
- generates Linux driver templates,
- automated generation and connexion of multiple identical slave components (using different hardware input/outputs)

[2] http://www.armadeus.com/wiki/index.php?title=POD_installation_guide

# Simple application

Blinking LED example to demonstrate the most basic aspects of communicating over the Wishbone bus:

- sending data from the CPU to the FPGA: defines how many times the LED blinks,
- sending data from the FPGA to the CPU: slave component identifier,
- using the FPGA input/outputs: digital input to trigger the continuous blinking of an LED connected to a digital ouput port.

Files needed in the project directory before launching POD:

- wb16.xml: configuration file used by POD,
- hdl/impulse.vhd : top file of the VHDL project,
- hdl/wishbone_interface.vhd: link between the Intercon and the other slave components,
- hdl/diviseur.vhd: clock divider (generates a 2 Hz clock signal),
- hdl/gene_impulse.vhd: LED command.

# Demonstration

Demonstration steps, under POD:

1. project creation
2. automatic generation of the iMx.Wrapper and interrupt manager files
3. pin assignement
4. automatic Intercon file generation
5. automatic project top file generation
6. ISE project generation
7. TCL script generation
8. binary file generation to be loaded in the FPGA

On the Armadeus platform:

1. `ssh` and possibly NFS connections between a PC and the CPU,
2. load kernel module (communication with the FPGA),
3. transfer the binary configuration file to the FPGA,
4. read and write in the registers shared between the FPGA and the CPU.

# wb16.xml

This file includes all the informations needed to generate the project. Mandatory items are:

- generics: the "generic" variables declared in the VHDL files (here the identifier of the component),
- hdl_files: the VHDL files of the project, whose top file is associated to an additional argument (istop="1"),
- interfaces: the various input/outputs of the slave component, grouped as a subset,
- ports: the definitions of the ports used by the interfaces,
- registers: the registers, and their adresses, accessible by the CPU through the Wishbone bus.

Two interfaces are mandatory: candr (**c**lock **and** **r**eset) and swb16 (Wishbone signals). The only modifications to this file are:

- the "generic" variables to be added if needed,
- the files incuded in the project,
- the interface including the "external" input/output signals
- the registers of the swb16 interface

# wb16.xml

## Sample of the file:

```
<generics>
      <generic name="id" public="true" value="1" match="\d+" type="natural" destination="both
            " />
</generics>

<hdl_files>
  <hdl_file filename="impulse.vhd" scope="both" istop="1" />
  <hdl_file filename="gene_impulse.vhd" scope="both" />
  <hdl_file filename="diviseur.vhd" scope="both" />
  <hdl_file filename="wishbone_interface.vhd" scope="both" />
</hdl_files>

<interfaces>
  <interface name="inpout" class="gls" >
      <ports>
      <port name="start" type="EXPORT" size="1" dir="in" />
      <port name="impulse" type="EXPORT" size="1" dir="out" />
    </ports>
  </interface>

  <interface name="candr" class="clk_rst">
    <ports>
      <port name="gls_reset" type="RST" size="1" dir="in" />
      <port name="gls_clk" type="CLK" size="1" dir="in" />
    </ports>
  </interface>

  <interface clockandreset="candr" name="swb16" class="slave" bus="wishbone16" >
    <registers>
      <register name="ADD_ID" offset="0x00" size="16" rows="1" />
      <register name="RCOMPT" offset="0x01" size="16" rows="1" />
    </registers>
```

# wishbone_interface.vhd

This entity includes the declaration of:

- the clock and reset signals,
- the Wishbone communication signals: addresses, distinct data writing and reading, control signals,
- the signals transmitted to the slave component, in our case, a single signal: the number of blinking periods of the LED.

The architecture mainly include two “process”, one for reading and the other one for writing.
The modifications performed on this file are:

- declaring the variable entities to be provided to the subprograms of the component
- the included registers,
- the actions performed when read or write requests are made at a given offset (complying with the offsets defined in the wb16.xml file)
- loading the variables provided to the various sub-programs

# wishbone_interface.vhd

File sample:

```
-------------------------------------------------------------------------------
Architecture wishbone_arch of wishbone_interface is
-------------------------------------------------------------------------------

  signal wb_write      : std_logic ;
  signal wb_read       : std_logic ;
  signal s_compt       : std_logic_vector(15 downto 0);
  constant ADD_ID      : std_logic_vector(1 downto 0) :="00";--identifiant
  constant RCOMPT      : std_logic_vector(1 downto 0) :="01";--nbr impulsions

begin

-- register reading process
  pread : process(gls_clk,gls_reset)
  begin
  if(gls_reset = '1') then--si reset
      wb_read <= '0';
      wbs_readdata <= (others => '0');
  elsif(rising_edge(gls_clk)) then -- si horloge
      wb_read <= '0';
      wbs_readdata <= (others => '0');
      if(wbs_strobe = '1' and wbs_write = '0' and wbs_cycle = '1')then -- si lecture
          wb_read <= '1';
          case wbs_addresse  is
            when ADD_ID =>
              wbs_readdata <= std_logic_vector(to_unsigned(id,16)); --identifiant dans readdata
            when others =>
          end case;

        end if;
  end if;
  end process pread;
```

# Buffer memories

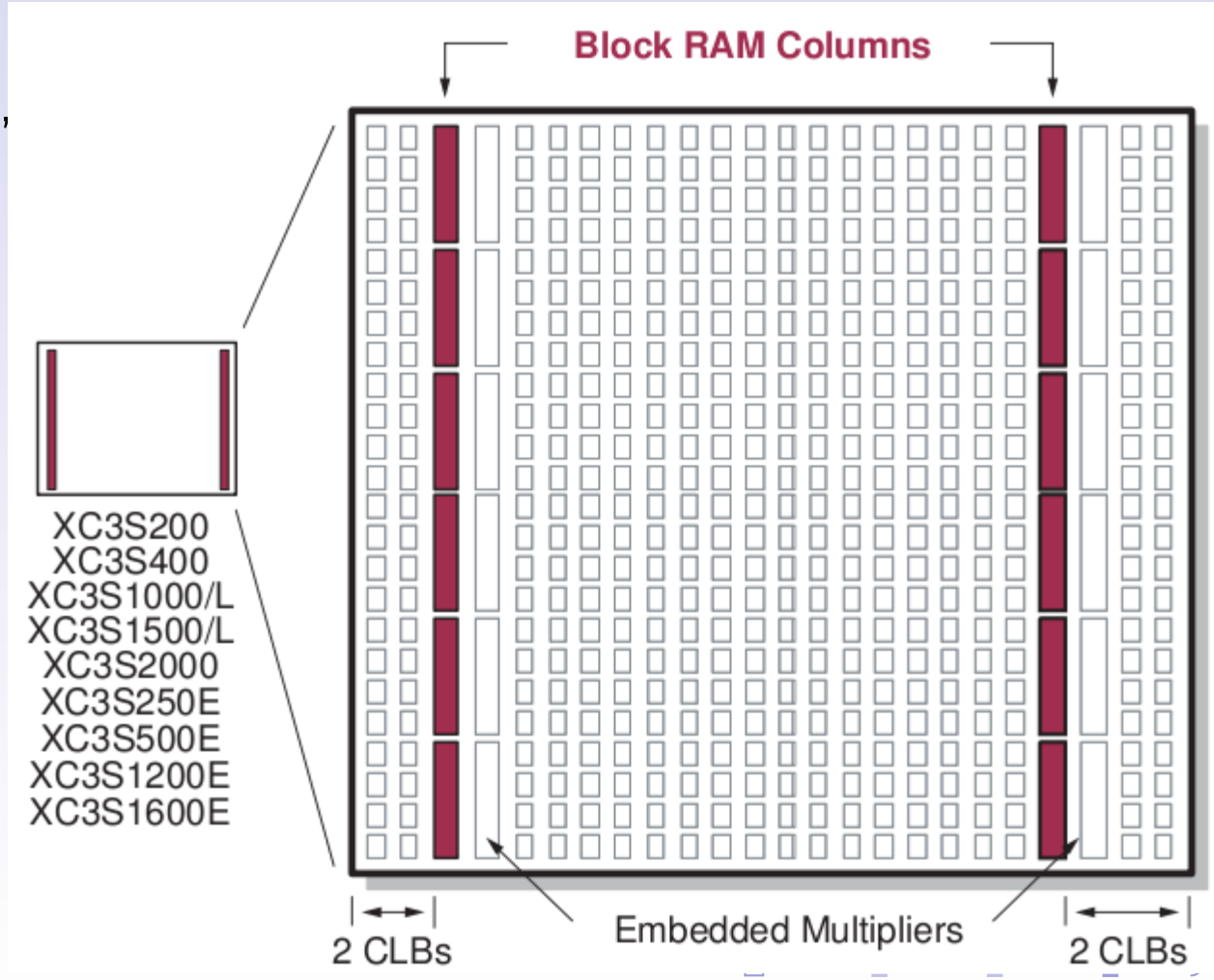
How to continuously transfer data from the FPGA to the ARM9 CPU ?

- implement a buffer memory on the FPGA compatible with POD,
- fetch from Linux the data recorded in the FPGA,
- use of interrupts to optimize the data flow,
- application to the implementation of a frequency counter,
- the limits of the Wishbone bus.

# Xilinx Spartan3 FPGA architecture

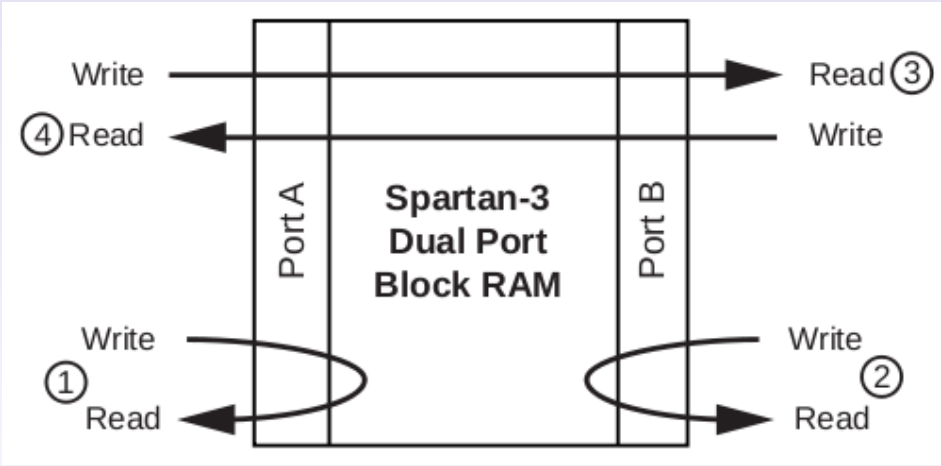
The Spartan 3 XC3S200:

- 12 physical RAM blocks,
- double port RAMs,
- a total of 216 Kb RAM,
- maximum clock speed 200 MHz.



# Double port RAM

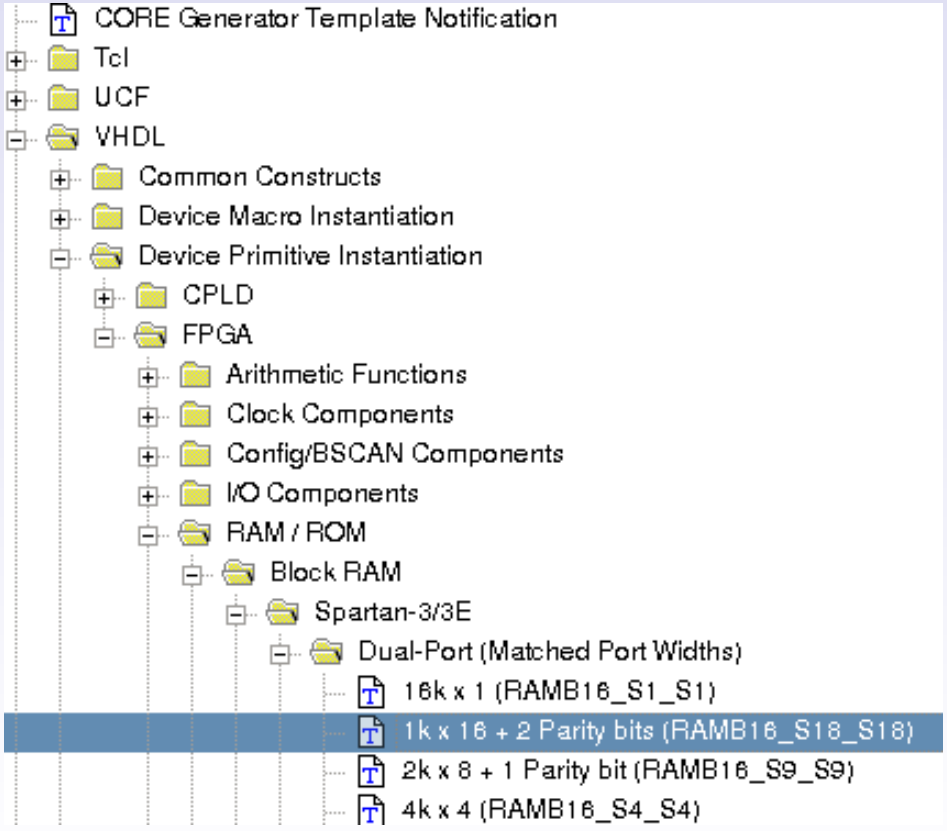
- Read and write address busses are **separate** in a double port RAM.
- Ability to **simultaneously** read and write at different addresses (cases 3 & 4) ...
- ... or to access as a classical single-port RAM (cases 1 & 2) [3]



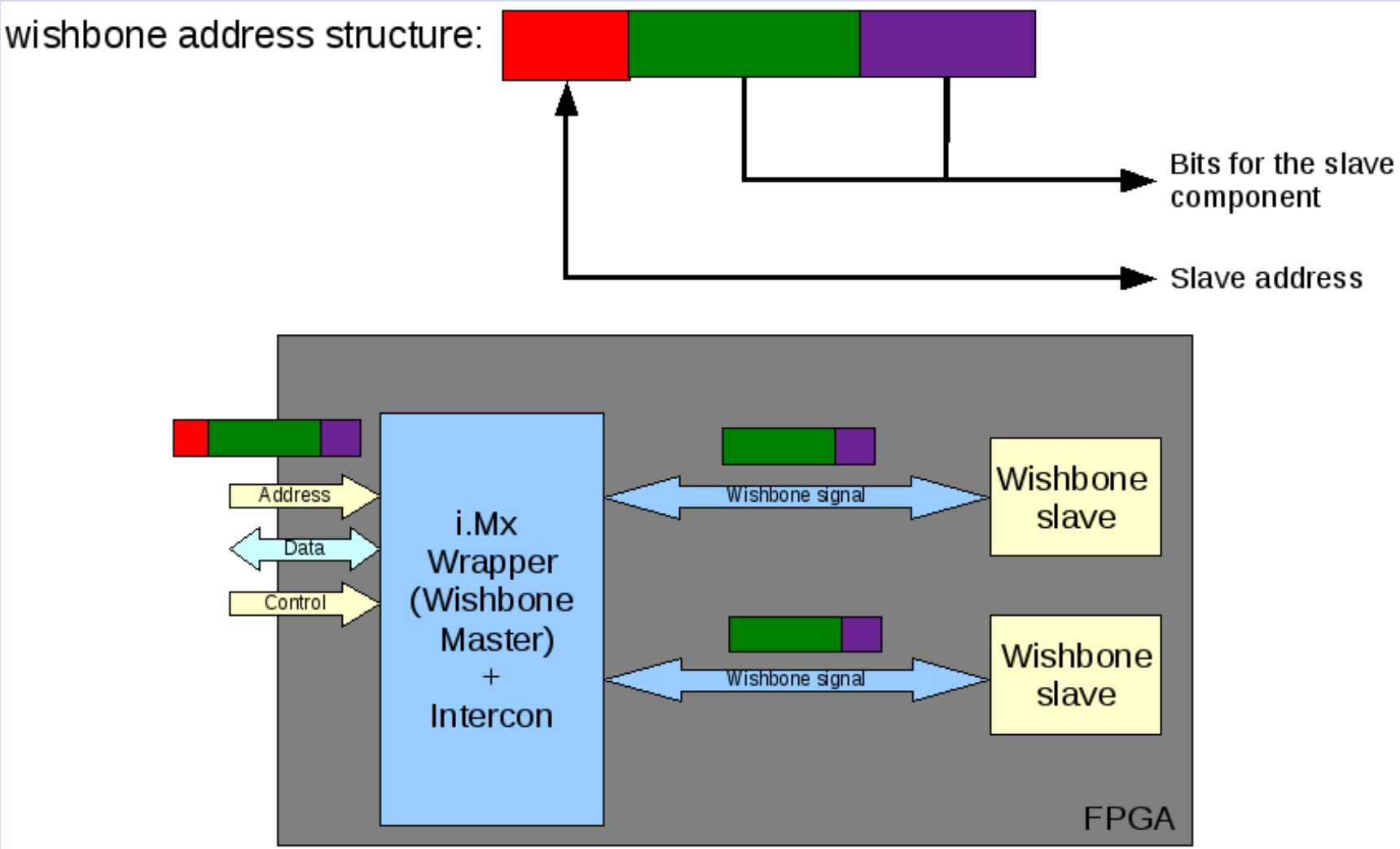

[3] http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf, p.25

# RAM declaration (VHDL)

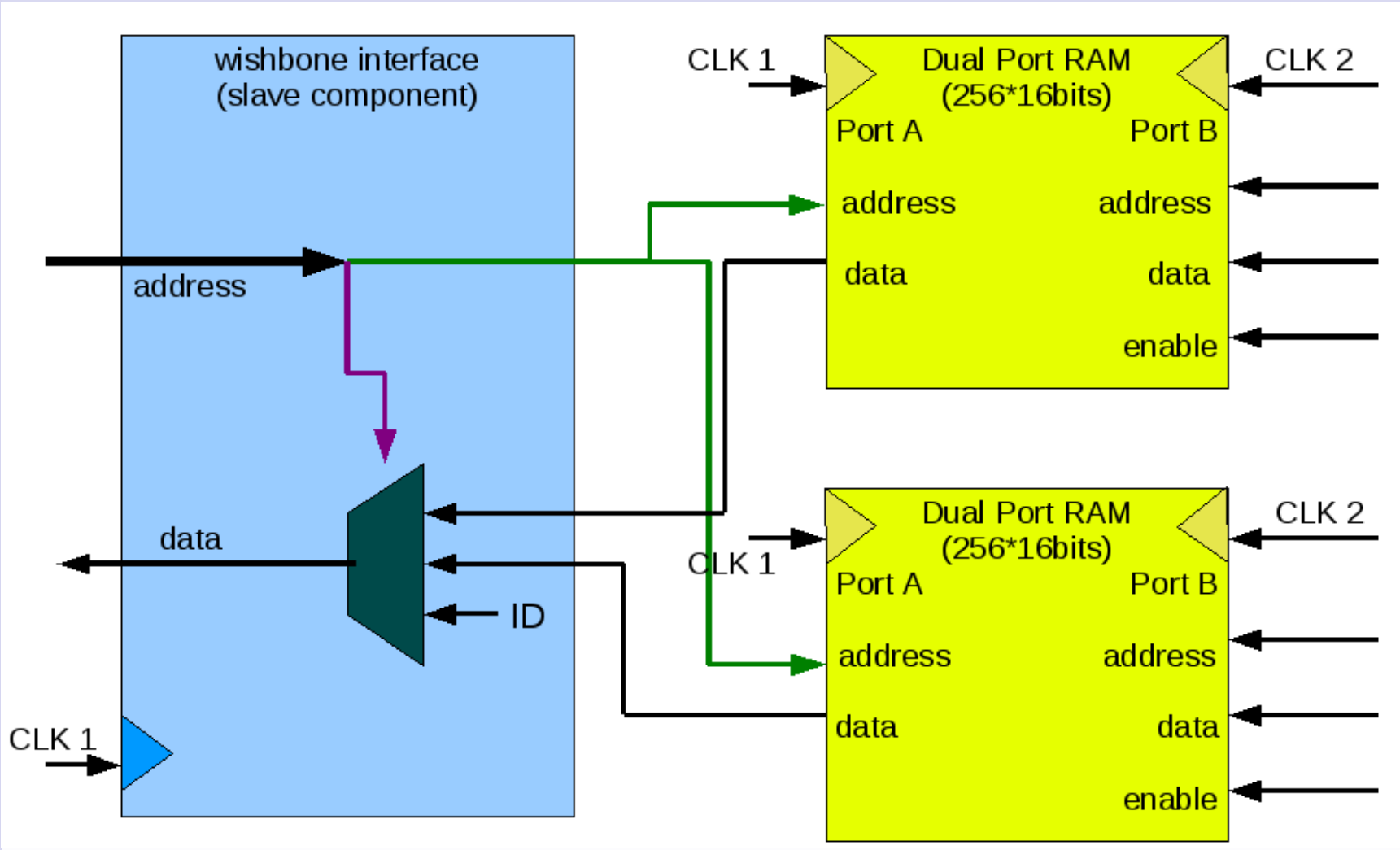
- Hardware resources on the FPGA are defined in a library:
- *Library UNISIM; Use UNISIM.vcomponents.all;*
- In order to use a RAM, import its entity (*VHDL declaration of the component*).
- In ISE: *Edit* $\gg$ *Language Templates.*



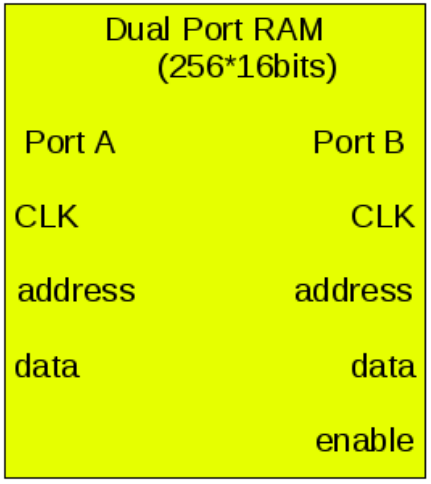
# Wishbone address structure

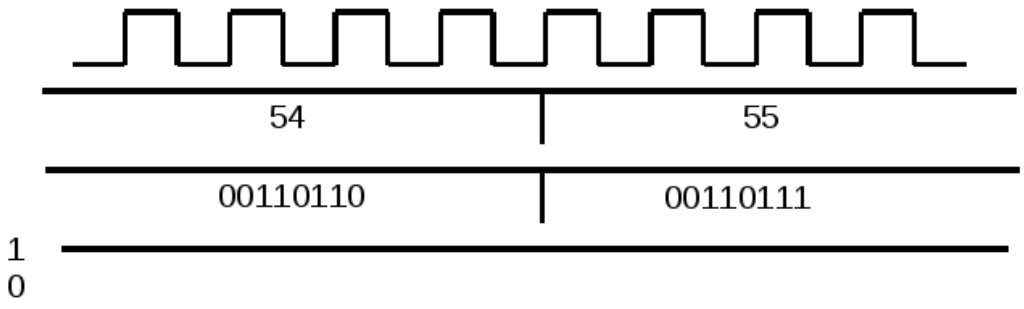


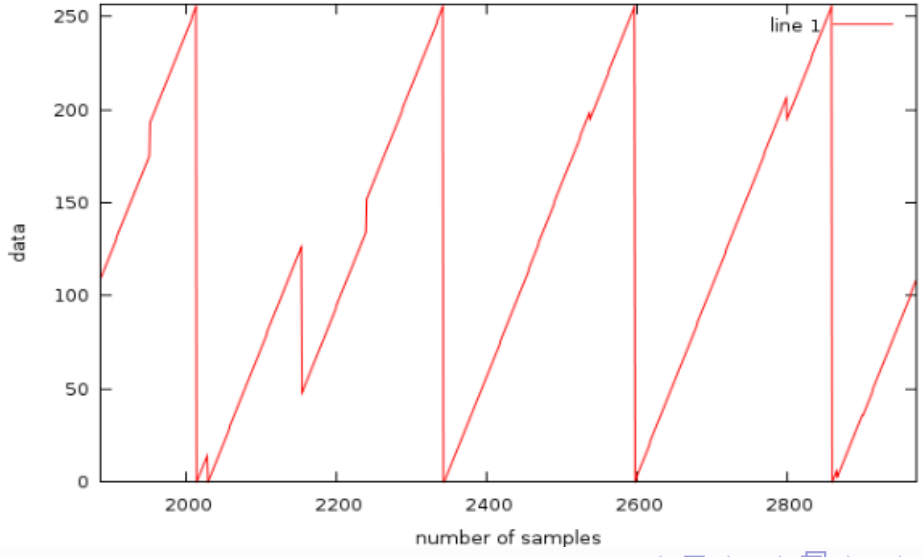
# A buffer memory compatible with POD



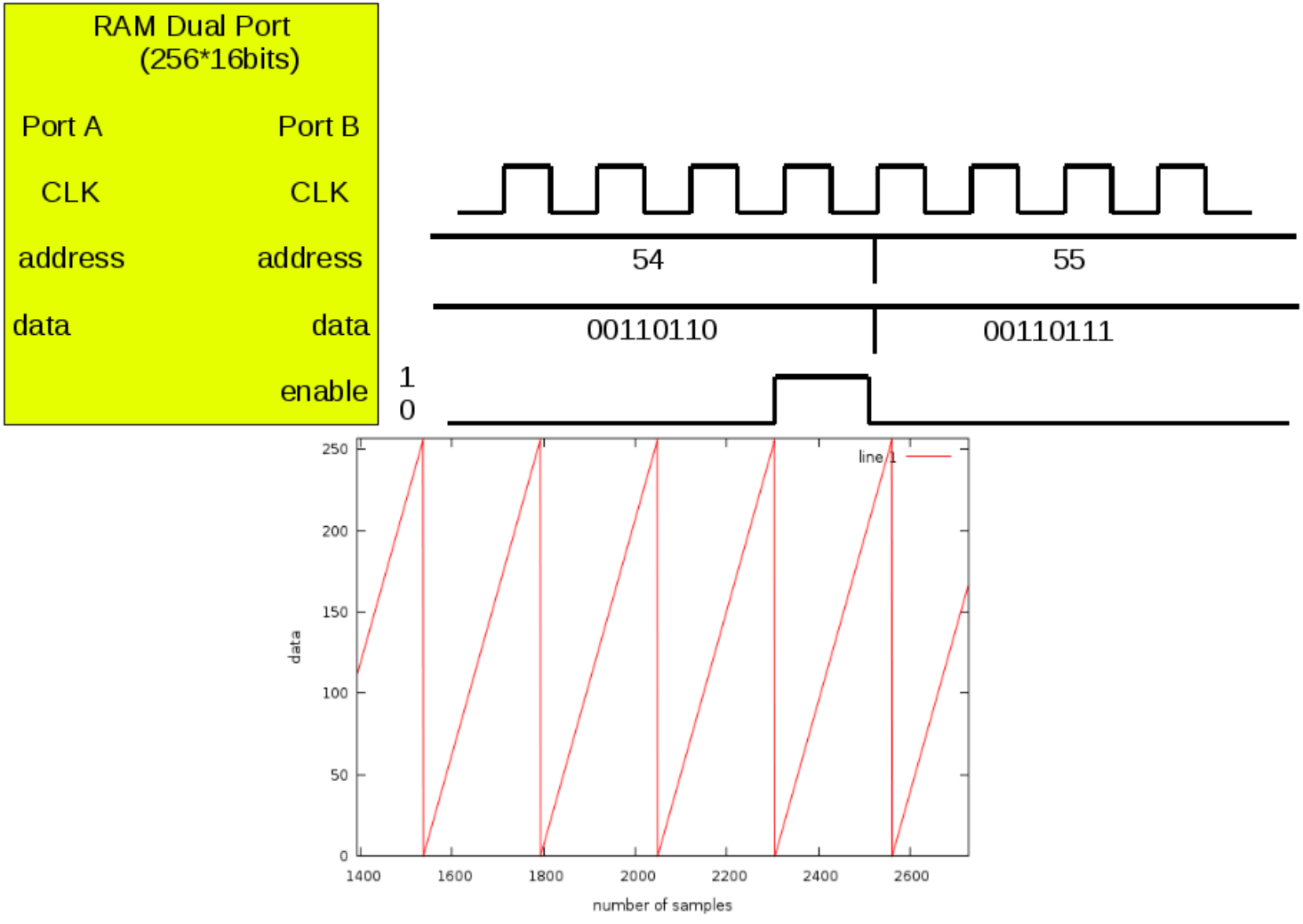
# An FPGA is a digital electronic component (problem)



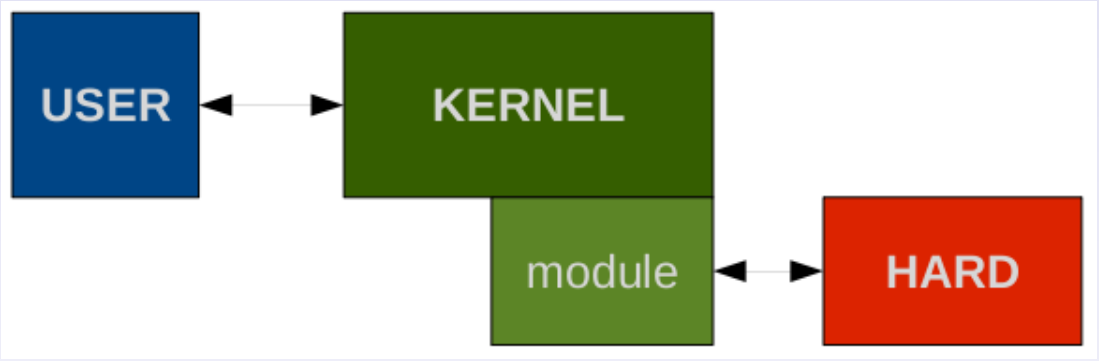



# An FPGA is a digital electronic component (solution)



# User space, kernel space

- User space is protected, and accessing peripherals is restricted,
- kernel space allows access to all peripherals and hardware resources.




The kernel module (driver) provides a link between user space and kernel space.

# The ioread16 function (kernel space)

Communication between the **FPGA and the CPU**

The `ioread16()` function reads the status of the peripheral:

```
1static struct plat_led_port plat_led00_data = {
    .name = "plat_led",
3   .num=00,
    .membase = (void *)(ARMADEUS_FPGA_BASE_ADDR_VIRT + 0x800),
5   [...]
    .data = ioread16(g_current->membase+i_addr);
7}
```

writes the address of the requested peripheral on the Wishbone address bus and returns the 16-bit value read on the data bus.

# The `read()` function (kernel module)

Communication between the **kernel and user space** (Linux)

The `read()` function is one of the basic methods of the module:

```
1 ssize_t led_read(struct file *fildes, char __user *buff, size_t count, loff_t *offp)
  {
3   unsigned short valeur[256];
    [...]
5   copy_to_user(buff, valeur, count);
  }
```

The `copy_to_user()` function requires:

- `buff`: array provided from user space,
- `valeur`: data array the driver sends back to the user through `buff`,
- `count`: number of bytes requested (provided from user space).

# `read()` usage example (user space)

In the user space program:

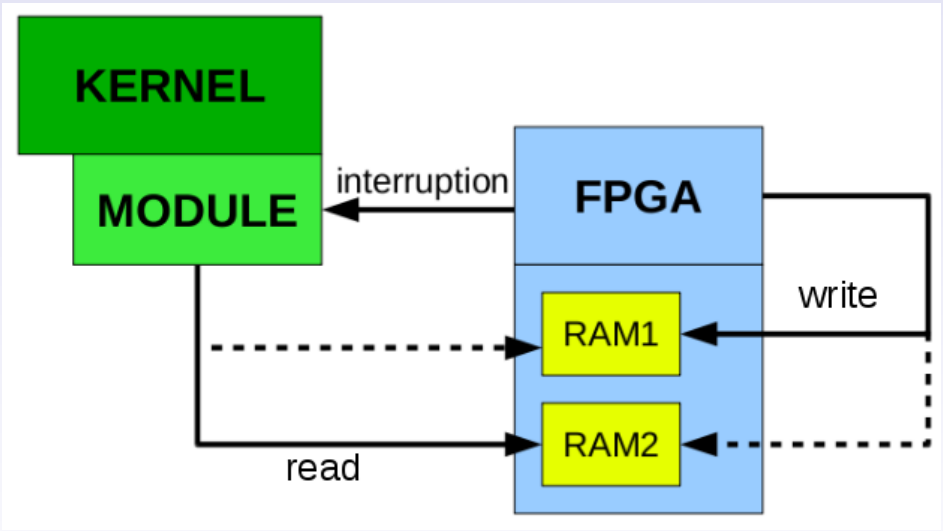
```
unsigned char j[512];
unsigned short *s;
[...]
read(fled,j,512);
s=(unsigned short*)j;
for (i=0;i<=255;i++) fprintf(mFile1,"%d",s[i]);
```

The `read()` function requires:

- `fled`: file descriptor to the device driver acting as communication point between user and kernel spaces (points to `/dev/XXX`),
- `j`: character array in which the module returns the values
- 512: number of bytes to be transfered (RAM=256*16 bits).

# Interrupts

- In order to optimize the data exchange between the FPGA and the Linux driver, use of interrupts to notify the module of the availability of data.
- when the FPGA has completed filling a RAM, trigger an interrupt to wake up the Linux module
- avoid locking the Linux system, efficient data transfer since the CPU reads data from one RAM as the FPGA writes in an other.




Make sure the interrupt rate is low enough to avoid locking the Linux system (depends on processing load)

# Code example

In the `init()` function of the module:

*request_irq(IRQ_GPIOA(4),button_interrupt,0, sdev->name,sdev);*
*set_irq_type(IRQ_GPIOA(4), IRQF_TRIGGER_RISING);*

In the `request_irq()` function:

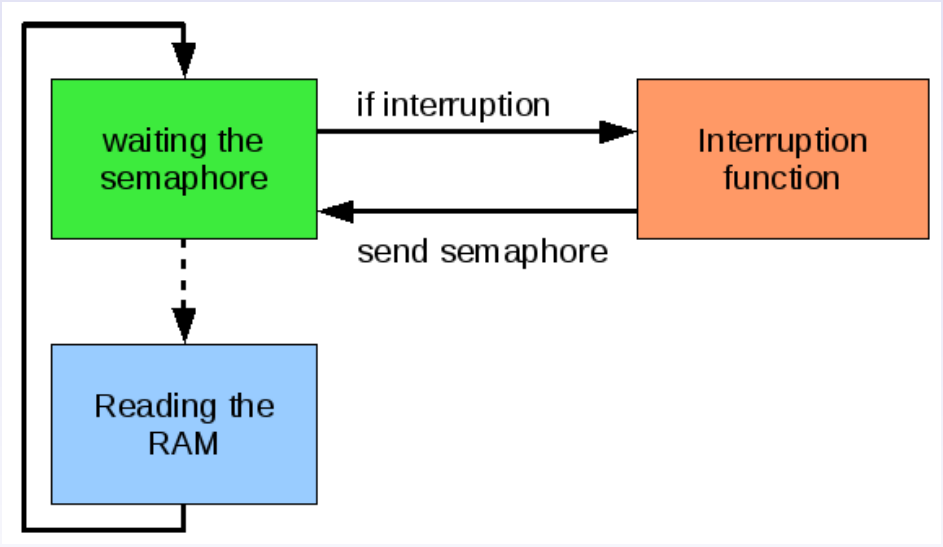
- IRQ_GPIOA(4):
  declares the pin associated to the interrupt request (pin 4 of port A),
- button_interrupt:
  interrupt service routine called upon interrupt trigger.

`set_irq_type` declares the sensitivity of the interrupt (rising edge signal on pin 4 of port A)

# Semaphores

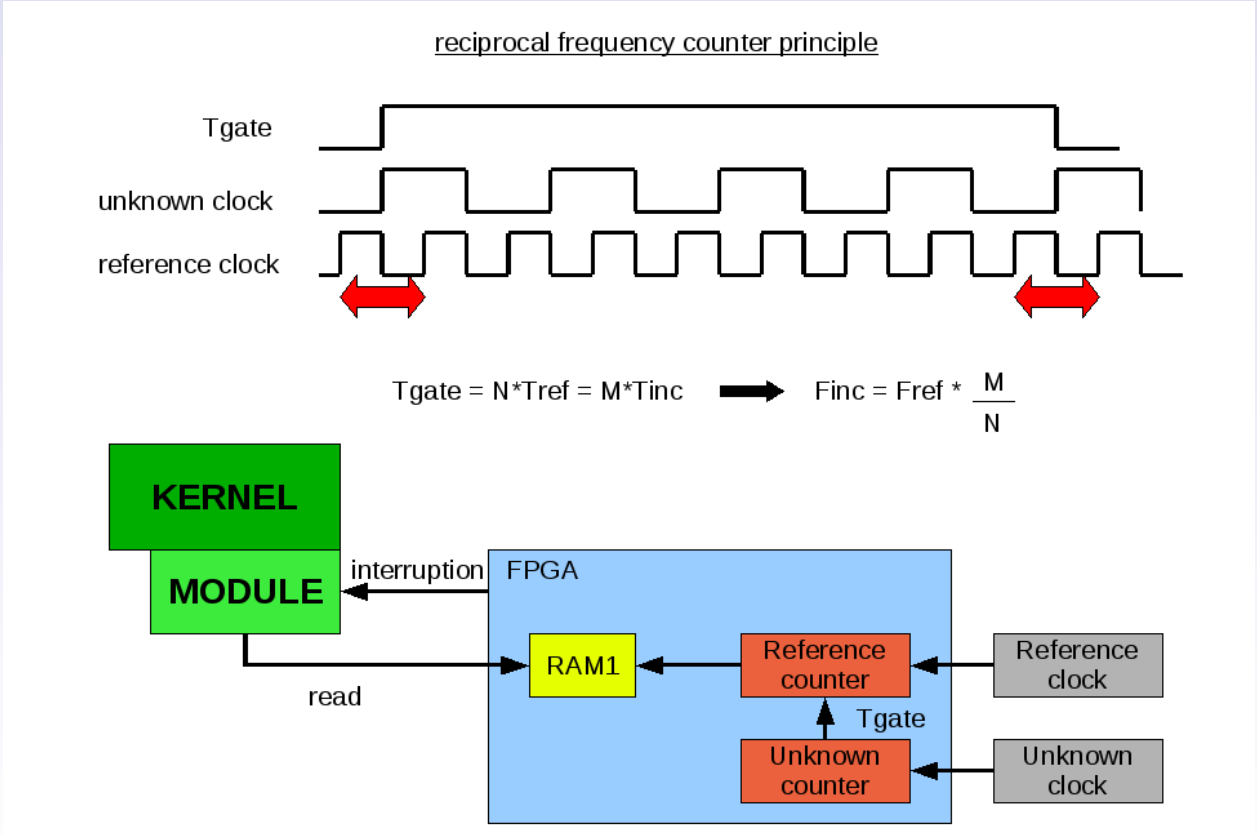
A module sleeps as long as the semaphore has not been cleared
*down_interruptible(&sema);*

Triggering an interrupt cleares the semaphore:
*up(&sema);*



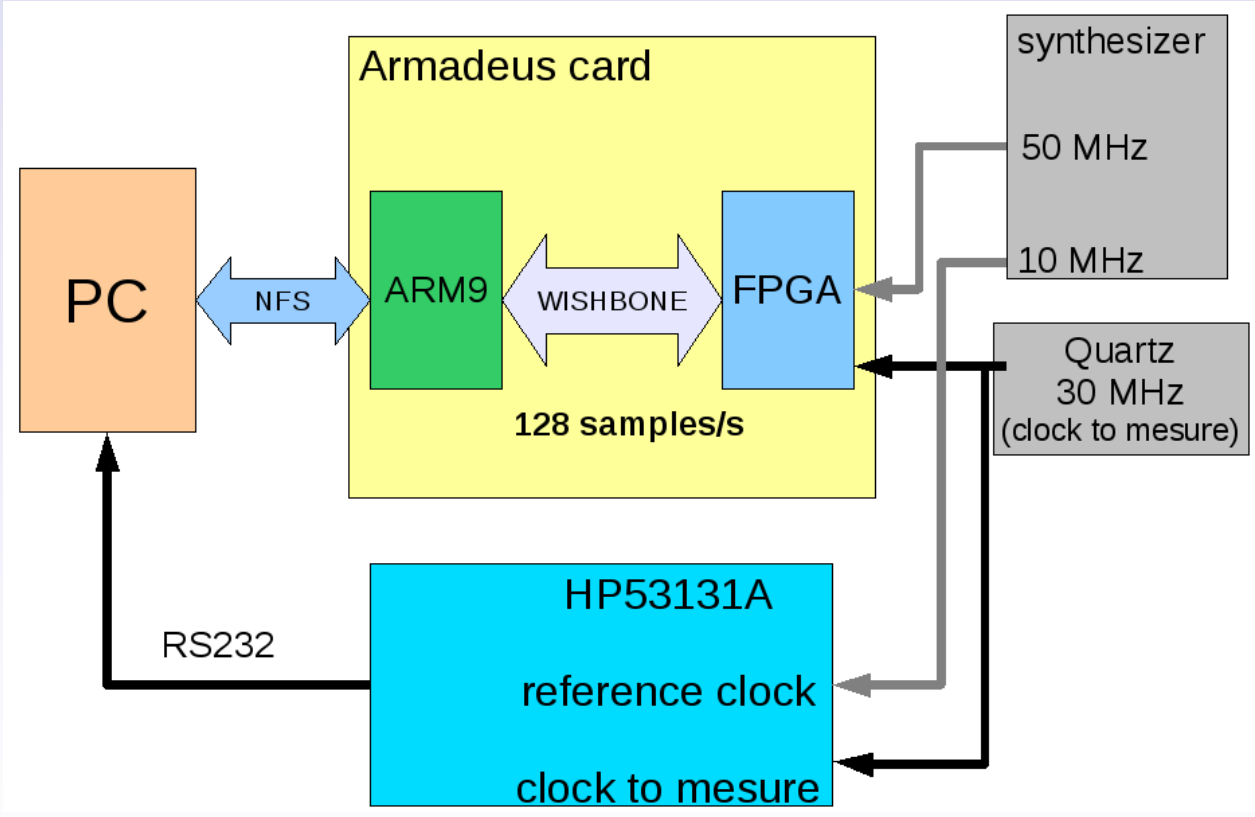
# Frequency counter principle

- Frequency is the physical quantity measured with the greatest accuracy
- FEMTO-ST time & frequency department is specialized in the development of high stability oscillator for source and sensor applications: the frequency counter is our basic measurement tool.



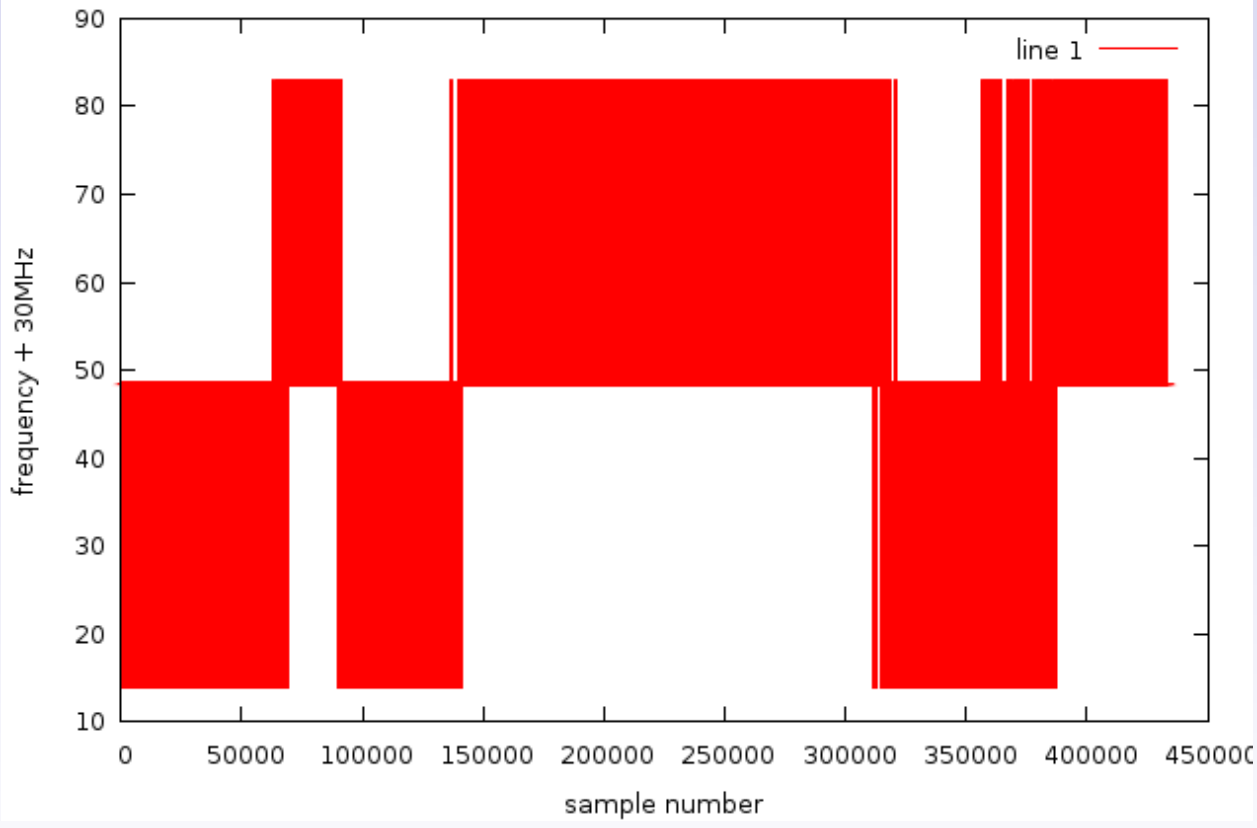
# Experimental test setup

- the FPGA counts clock cycles (real time part)
- the ARM9 CPU collects data from the FPGA and manages data storage and transfer to the PC
- the frequency counter and the FPGA share the same reference clock



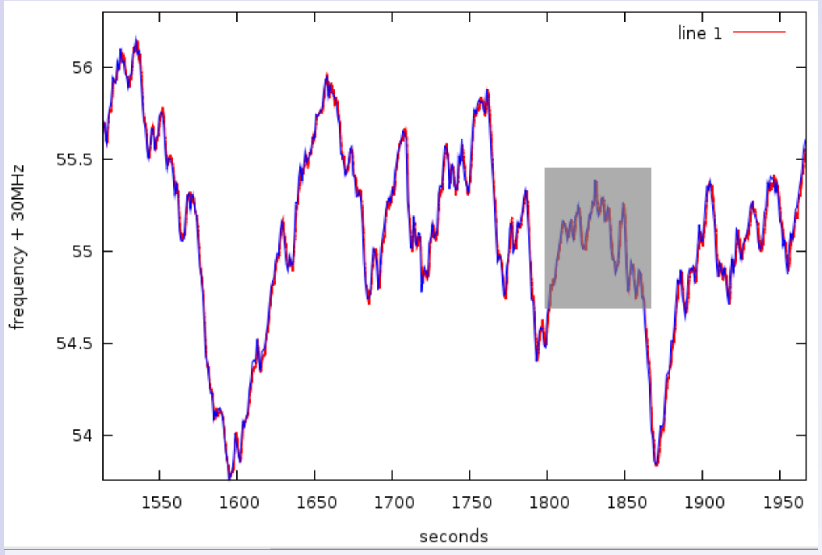
# Frequency calculation

The frequency is *continuously* recorded at a rate of 128 samples/s (8 ms gate time)

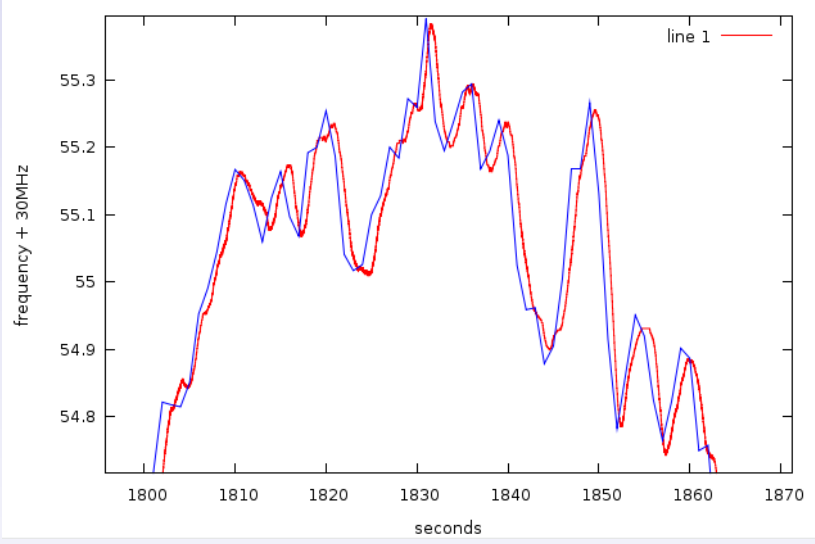


# Sliding average on contiguous samples

Blue: commercial HP 53131A frequency counter,
red: FPGA implementation of a reciprocal counter.




Long term measurement

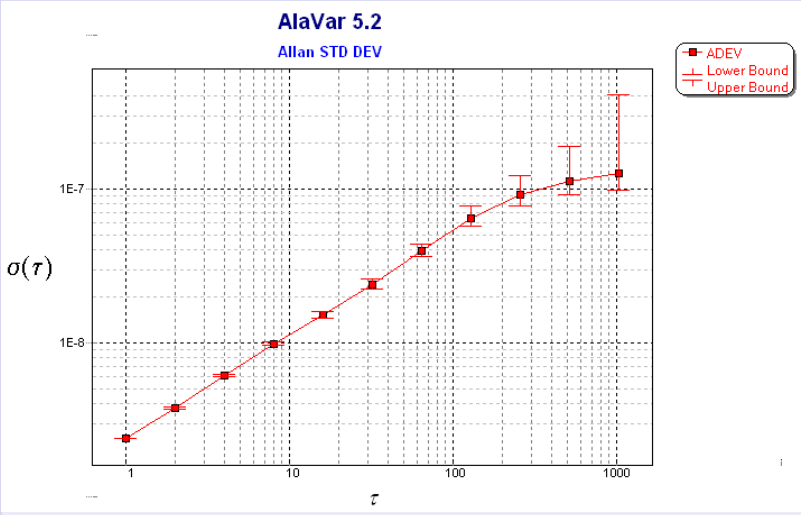



Zoom

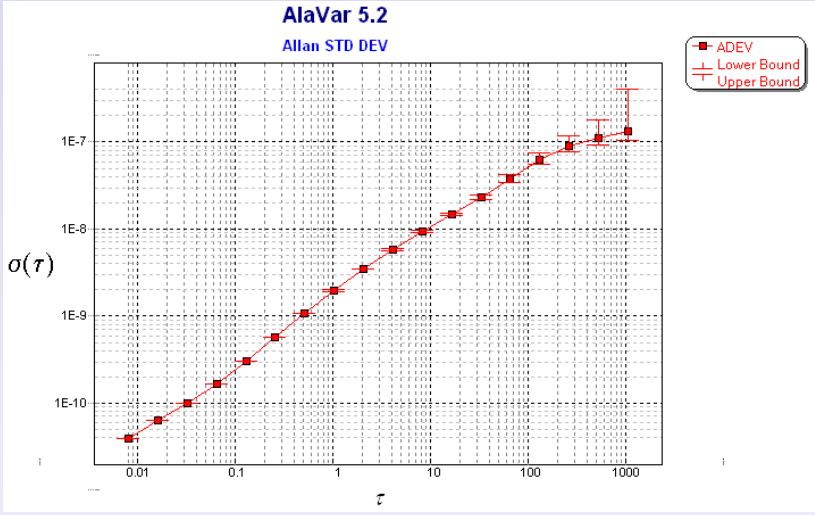
$\longrightarrow$ our measurements are consistent with the results of the (reference) commercial instrument

# Allan deviation

Statistical analysis of an oscillator stability (standard deviation on variable width windows)




HP53131A



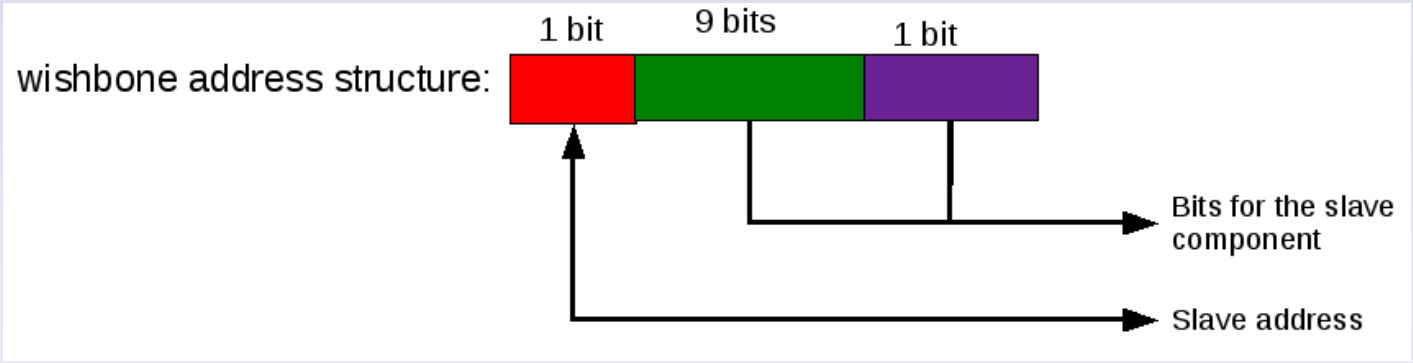

FPGA implementation

- Statistical analysis confirmation of the equivalent results: in both cases the relative stability $\Delta f/f \simeq 2 \times 10^{-9}$ at 1 s integration time
- The short gate time of the FPGA implementation provides results down to $\tau = 10$ ms (impossible to achieve using HP53131A)

# Wishbone bus limitation

The Wishbone bus provides a restricted address bus width ⇒ reduced amount of addressable RAM

Maxium of 9 bits ⇒ 512 accessible addresses.



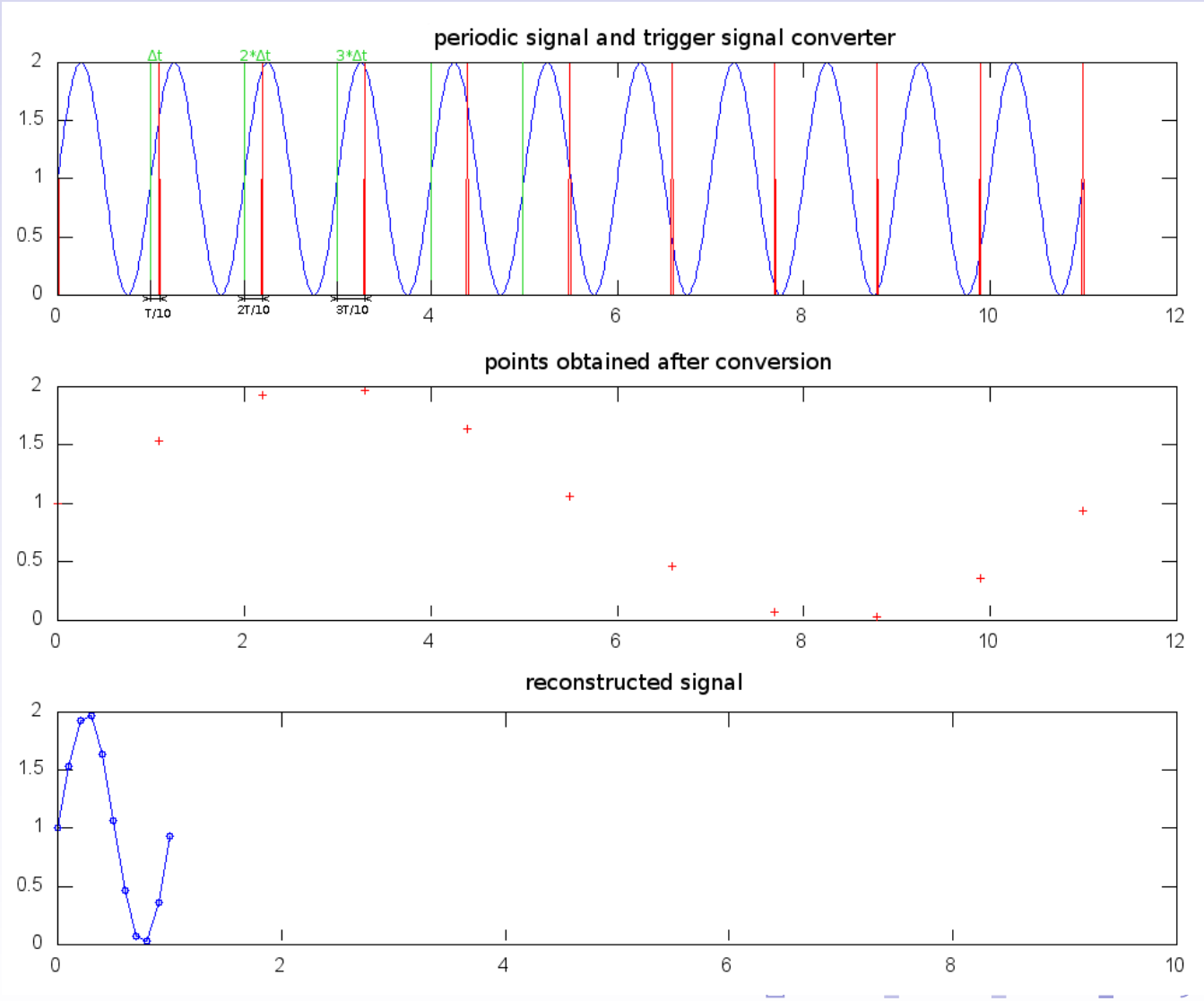

- We know we want to sequentially read the content of all RAM: no need to individually address each register.
- Replace a dedicated address bus with a counter in the FPGA, incremented at each read step
  ⇒ the address bus is free to point to 1024 RAM blocks
  ⇒ strategy similar to DMA (no need to individually point to each memory register)

# Radiofrequency sampling rate: objectives

- The objective is the sampling of ultrasonic and RADAR (GPR) echo signals at rates above the GHz range (practical demonstration: 4 Gsamples/s) ...
- ... under the assumption of a stationary signal (the probed medium does not significantly change between measurements) ...
- ... in order to use an appropriate method: *Equivalent Time Sampling*.
- acoustic and electromagnetic charaterization of passive media probed by series of pulses.

# Equivalent Time Sampling

Principle: in this example, 10 samples are gathered over 1 period at sampling dates $N \times \delta t$ ($N = [1..10]$), yielding an equivalent sampling rate $1/\delta t$

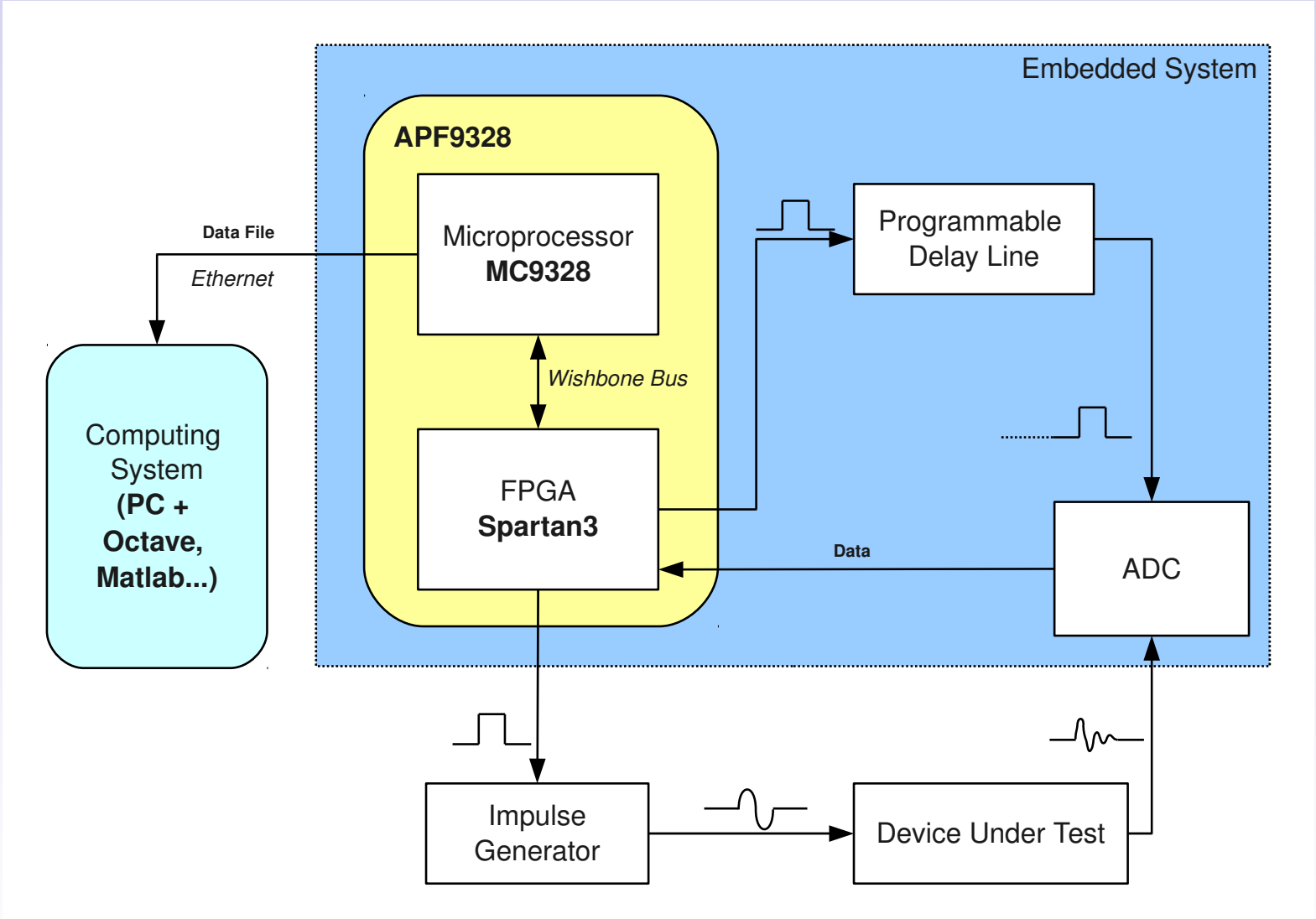


# ETS: implementation

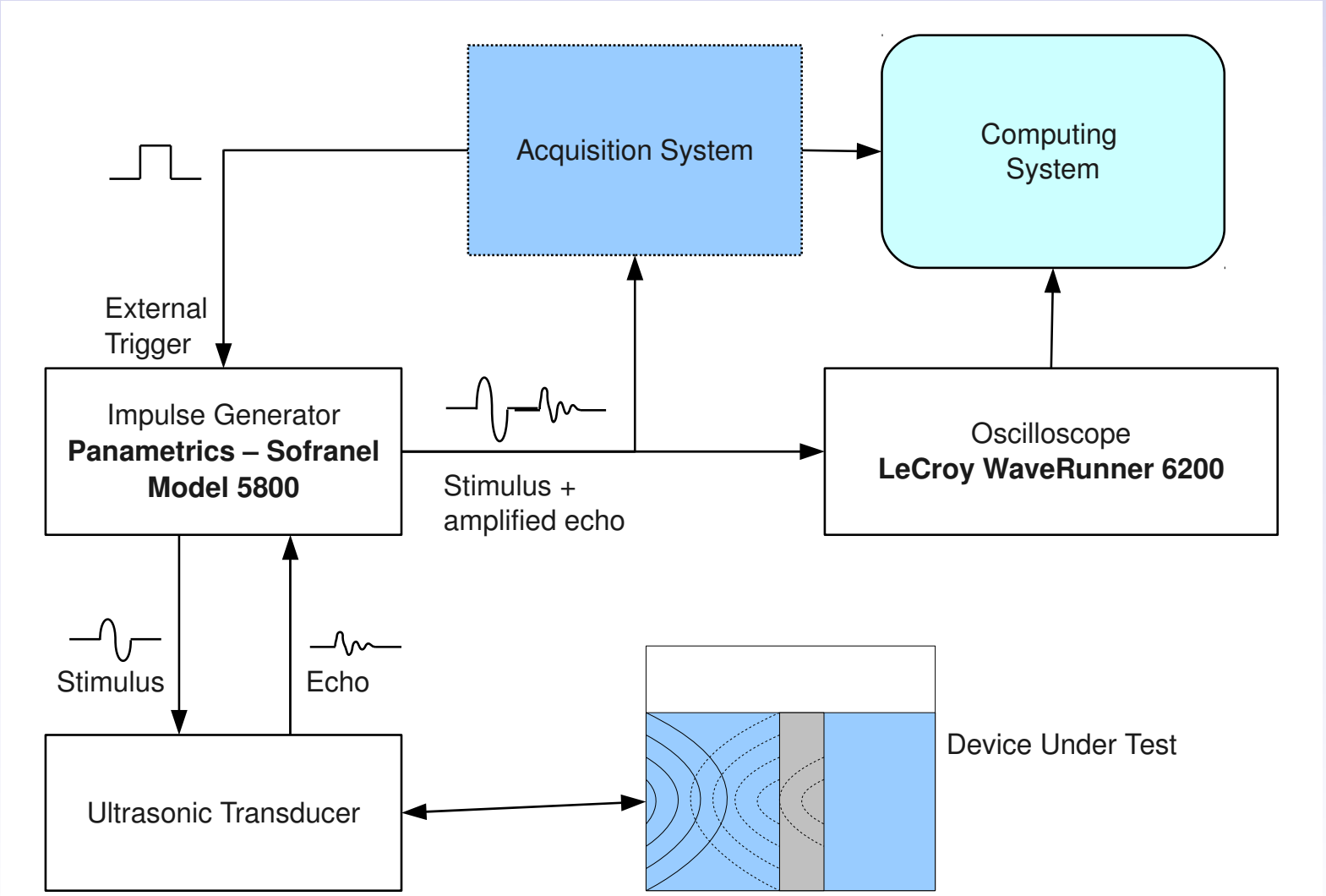
Needed hardware:

- fast A/D converter (low jitter): digitization of a voltage memorized by an externally triggered Sample & Hold (Linear Technology LTC1407),
- sub-ns delay generator (programmable delay line): time delay of the digital signal triggering the Sample & Hold ($\delta t$ steps), Dallas/Maxim DS1023
- Control and synchronization system: provides all components the necessary signals (delay line programming, communication clocks ...)
- data retrieval and storage: low latency constraints compared to the Control system

# Working principle

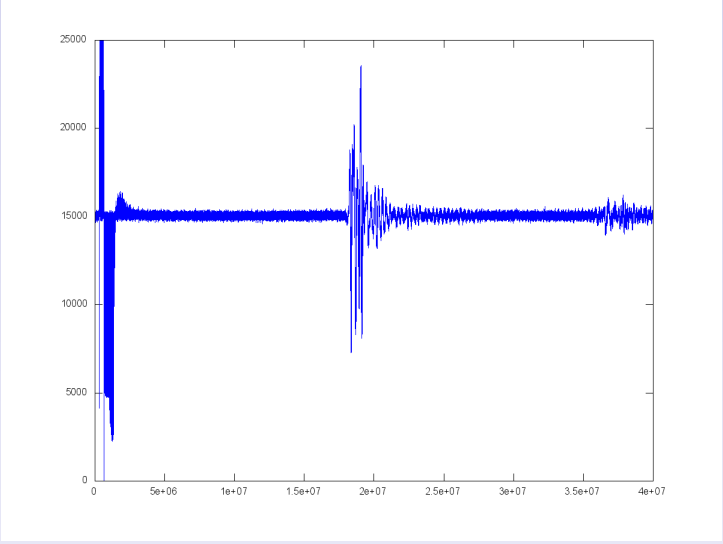


# Application to recording ultrasonic echos (non-destructive testing)

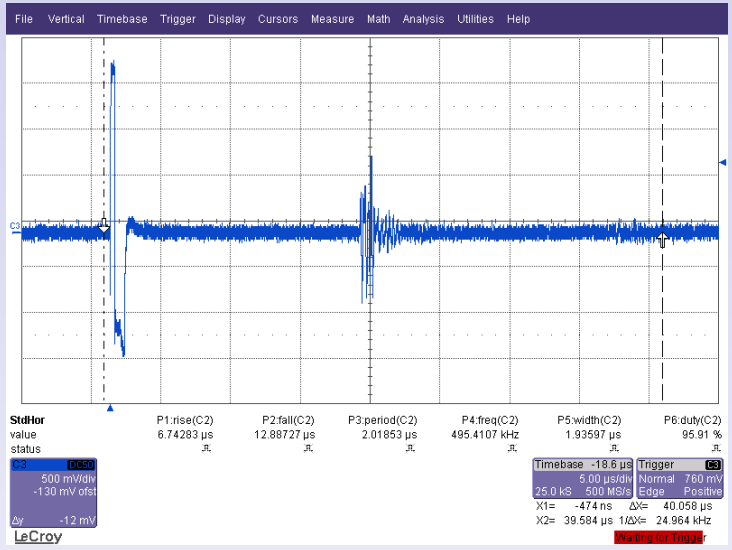


# Obtained results

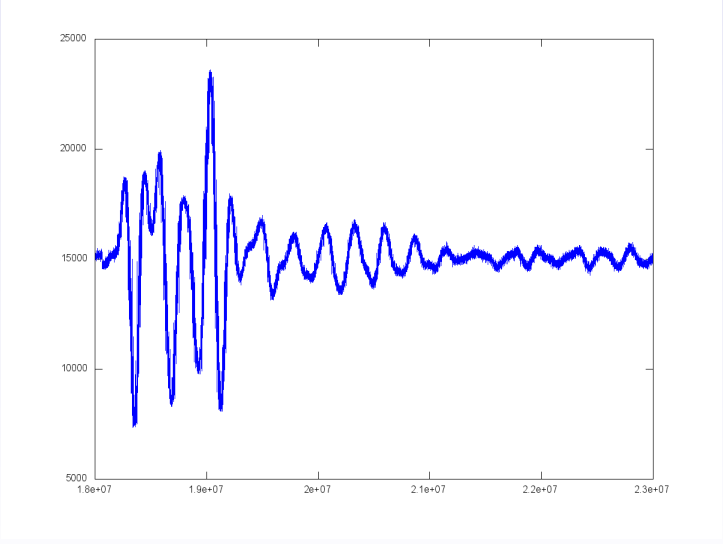
Stimulus and ultrasonic echo: embedded system (0 to 40 $\mu$s)



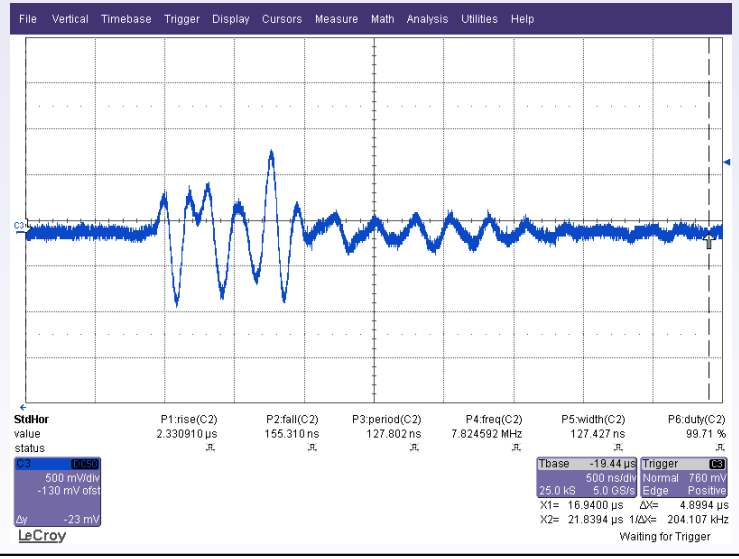

Commercial oscilloscope result




Single echo (18 to 23$\mu$s) recorded using the embedded system




Commercial oscilloscope result



# Conclusion

- Architecture selection justification: complementarity of the general purpose CPU running an operating system + FPGA
- Practical demonstration of the development cycle
  - POD + Wishbone bus for a rational framework to communicate with multiple IPs in the FPGA
  - efficient communication between the CPU and the FPGA
- Experimental results of applications:
  - radiofrequency counters with performances in agreement with the commercial HP53131A (433/16 MHz input signal)
  - high frequency probe non-destructive testing and RADAR