FPGA as real-time Xenomai/Linux co-processor

G. Goavec-Merou, September 25, 2022

Suject available at `http://www.trabucayre.com/enseignement/tp_fpga.pdf`
Codes available at `http://www.trabucayre.com/enseignement/tp_fpga_sources.tgz`

1 Goals

We have seen that the real-time extension of Linux – Xenomai – aims to reduce the variability of a request, and in particular to limit the time interval between the event and the associated action.

To evaluate the various solutions (sharing-time, real-time and FPGA) we will use a `Redpitaya` platform, based on `Zynq7000`, having a software environment where Xenomai is available and an FPGA.

A major benefit of the CPU+FPGA combination is to provide a hierarchy on latencies – low on CPU, good on the Xenomai extension, excellent with the FPGA – at the expense of the complexity of implementing a given algorithm – simple on CPU, more complex on Xenomai, long to debug on FPGA. The time resolution of the operations on the FPGA is 8 ns, since the core of the FPGA is clocked at 125 MHz [1].

This series of practical work has three steps:

1. the first step will consist in becoming familiar with the principle of CPU/FPGA communication and the `AXI` bus;
2. then a resumption of the practical work on Xenomai will be necessary to have the applications in user space necessary for the last part of the practical work;
3. finally we are going to realize a period counter, in the FPGA, in order to evaluate the load stability characteristics of Linux and Xenomai applications.

2 Working environment

All development and compilation are done on the host machine. The binaries must then be moved to a directory on the computer available from the card through a network mount. The manipulations, on the platform, will be done in a terminal, through a serial communication.

2.1 Compilating applications

As part of periodic signal generation tests, scripts are available to automate compilation:

For Linux applications, the following command should be used:

```
make
```

For real-time application the command will be:

```
make -f Makefile.xenomai
```

2.2 Binaries access from the plateform

Instead of transferring files to the Redpitaya card, we will use a network service called **NFS** (*Network File System*). Thanks to this one, it is possible to mount a directory of the computer on the platform and thus to directly access its contents in a transparent manner (fig. 1).

To do that, it's necessary, on the target size, to use the following command:

```
# mount 192.168.2.1:/home/etudiant/nfs /mnt
```

With:

- *192.168.2.1*, the host IP (to be adapted to each workstation);
- */home/etudiant/nfs* the shared host's directory;
- */mnt* local (target) directory where to mount distant (host) directory.

À partir de maintenant tous les binaires (.bit.bin pour le FPGA et applications) devront être copiés dans le répertoire */home/etudiant/nfs* du PC et l'ensemble du travail sur la carte devra être fait dans le répertoire */mnt*

From now, all the binaries (.bit.bin for the FPGA and applications) will have to be copied into the */home/etudiant/nfs* PC's directory and all the work on the board will have to be done in the */mnt* directory

[1] however, we will take care to note that this 125 MHz is the result of a multiplication by PLL and therefore of poor stability – we will prefer provide a good quality external clock for accurate time measurements – with the problem of transferring data between domains clocked by the external clock and the domain clocked by the CPU clock, necessary to synchronize the exchanges on the buses common to the components

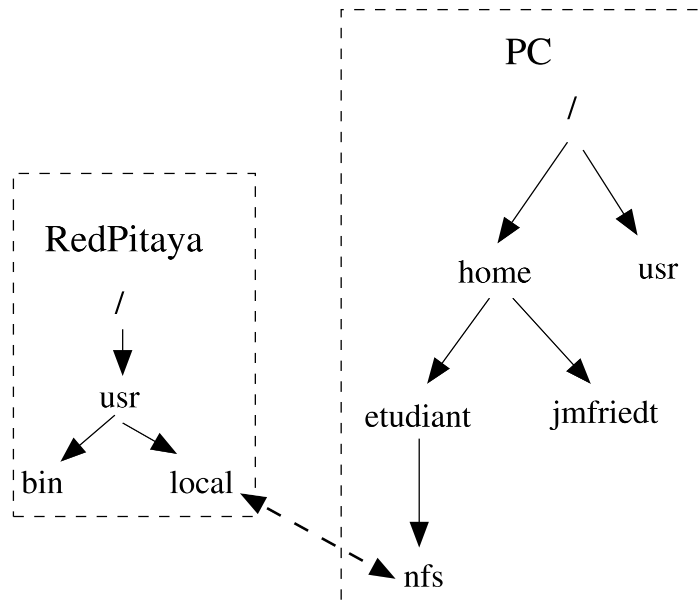


Figure 1: NFS : The contents of the PC's /home/etudiant/nfs directory can be accessed directly from the target board through the /mnt directory.

3 CPU-FPGA communication

We were able to see, in the first part of the lab, the temporal characteristics of Linux and Xenomai, the measurements are done thanks to a dedicated gateware in the FPGA (real-time component by nature). In the second part of this lab, we will focus on the FPGA part. With the final goal of realizing the counter used to qualify the various solutions.

To achieve these goals, we will do this sequence of development:

1. CPU-FPGA communication (Linux kernel, FPGA communication bus),
2. counter implementation in the FPGA and transfer of the value to the CPU,
3. use of the FPGA to measure the jitter of a signal produced by the generic processor, with and without Xenomai.

4 Hardware and software environnment

As mentioned earlier, the hardware used, a `RedPitaya`, based on a `Zynq` has, into the same chip, a generalist processor and an FPGA. Communication between the two zones uses AXI buses (fig 2).

Our first software development concerns the communication protocol between processing blocks in the FPGA: each block is coded in VHDL, and the communication protocol between blocks as well as between CPU and FPGA must be implemented in this language.

Gatewares that will be used in the following differ from the classic approach to development on `Zynq`. This is to avoid the complexity of package updates each time the code is modified. The various applications were generated using *Peripheral On Demand*[2] which is an assembly tool for HDL blocks (not covered in this tutorial). The `block design` containing the `processing system` is confined and the `AXI` bus exported. `POD` generates an address decoder (the `intercon` to communicate independently with each slave (figure 3) integrated into the gateware (figure 4). The role of this component is to divide the memory range into several zones.

4.1 bitstream generation

Bitstream generation from VHDL sources is carried out by the *Vivado* application.

To launch this tool, first shell environment must be updated by:

```
source /opt/Xilinx/Vivado/VERSION/settings64.sh
export LANG=en_US.UTF-8
```

Followed by:

```
vivado&
```

Lastly, opening the project and start the generation by clicking on *generate bitstream*

[2] `http://github.com/martoni/periphondemand`

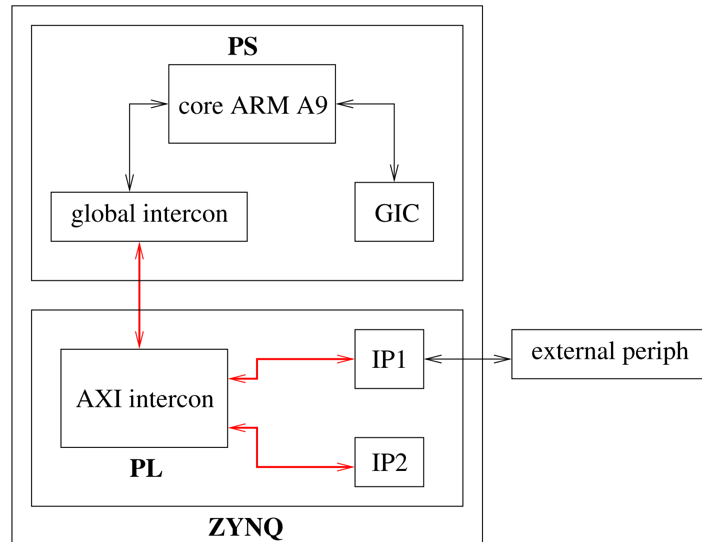


Figure 2: Simplied scheme to the `Zynq` internal structure.

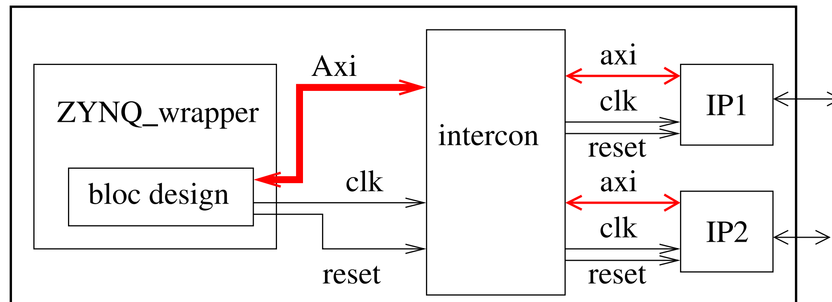


Figure 3: FPGA design example

5 First step with FPGA-CPU communication: adder

project is located into /home/etudiant/tp_fpga_sources/design/exercice_addition directory and project file to use with `vivado` **is in objs/exercice_addition.xpr** sub-directory

This first exercise is an opportunity to become familiar with communication through the **`AXI`** bus. To do this we will create an IP (Intellectual Property), or component, which performs an addition.

5.1 Basic principle of `AXI` communication

Given the relative complexity of the **`AXI`** bus (fig. 5) and the abstraction layer presented previously, the part dedicated to the management of the communication must take into account only a subset of signals (fig. 6). Apart from the two data buses `s00_axi_wdata` and `s00_axi_rdata`, the other signals are created locally and used between the abstraction layer and the `process`.

The management of transactions with the processor presents two control signals:

- **`write_en_s`** : write request (CPU point of view), active high;
- **`read_en_s`** : read request, also active high;

Two data signals and a common address signal:

- **`addr_s`** : register address, common to writing and reading;
- **`readdata_s`** : read data bus. Must be update by IP. A local copy must be present too: required to keep current value (not possible with an output/write only signal):
- `s00_axi_wdata` : write request data bus. Is read by the IP.

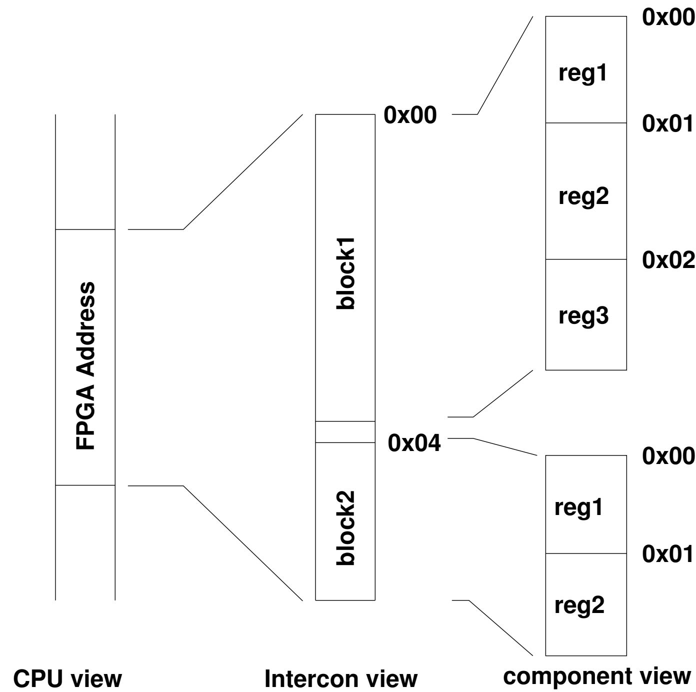


Figure 4: Principle of abstraction of component addresses in the FPGA. The address range dedicated to communication between the CPU and the FPGA is divided into several sub-addresses. The intercon performs the decoding and addresses the component concerned by providing him with the register number corresponding to the transaction.

```
Entity enseignement_addition_axi is
    generic(
        id          : natural := 1;
    -- Parameters of Axi Slave Bus Interface S00_AXI
    C_S00_AXI_DATA_WIDTH   : integer := 32;
    C_S00_AXI_ADDR_WIDTH   : integer := 5);
    port (
    -- axi slave signals
    s00_axi_aclk   : in std_logic;
    s00_axi_reset  : in std_logic;
    s00_axi_awaddr   : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1 downto 0);
    s00_axi_awprot   : in std_logic_vector(2 downto 0);
    s00_axi_awvalid  : in std_logic;
    s00_axi_awready  : out std_logic;
    s00_axi_wdata  : in std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);
    s00_axi_wvalid   : in std_logic;
    s00_axi_wready   : out std_logic;
    s00_axi_wstrb  : in std_logic_vector(3 downto 0);
    s00_axi_bresp  : out std_logic_vector(1 downto 0);
    s00_axi_bvalid   : out std_logic;
    s00_axi_bready   : in std_logic;
    s00_axi_araddr   : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1 downto 0);
    s00_axi_arprot   : in std_logic_vector(2 downto 0);
    s00_axi_arvalid  : in std_logic;
    s00_axi_arready  : out std_logic;
    s00_axi_rdata  : out std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);
    s00_axi_rresp  : out std_logic_vector(1 downto 0);
    s00_axi_rvalid   : out std_logic;
    s00_axi_rready   : in std_logic);
end entity enseignement_addition_axi;
```

Figure 5: Example of the entity part of an IP with AXI communication.

```
Architecture enseignement_addition_1 of enseignement_addition_axi is
  signal addr_s : std_logic_vector(INTERNAL_ADDR_WIDTH-1 downto 0);
  signal write_en_s, read_en_s : std_logic;

  signal readdata_s : std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);
begin
  s00_axi_rdata <= readdata_s;
```

Figure 6: Subset of signals needed for access management from the processor.

5.2 Addition block Implementation

Given the relative simplicty of this component, this one has only 4 registers:

```
constant REG_ID     : std_logic_vector(2 downto 0) := "000";
constant REG_OP1    : std_logic_vector(2 downto 0) := "001";
constant REG_OP2    : std_logic_vector(2 downto 0) := "010";
constant REG_RESULT : std_logic_vector(2 downto 0) := "011";
```

- ID register (RO). Mandatory. Located at address 0x00;
- two operators registers (WO), located at 0x01 and 0x02 respectively;
- the last register is used to retrieve addition result. Located at 0x03 (RO).

To have a more clean code structure, read and write (CPU point of view) requests are implemented using two different `processes`.

5.3 Read management

A first `process` is used to handle read access (CPU point of view). For the two registers with a read access, requests are handled like this (see following listing):

- **REG_ID** for unique id register (0x00) ;
- **REG_RESULT** to provides addition result (0x03).

```
read_bloc : process(clk, reset)
begin
  if reset = '1' then
    readdata_s <= (others => '0');
  elsif rising_edge(clk) then
    readdata_s <= readdata_s;
    if read_en_s = '1' then
      case addr_s is
      when REG_ID =>
        readdata_s <= std_logic_vector(to_unsigned(id,C_S00_AXI_DATA_WIDTH));
      when REG_RESULT =>
        readdata_s <= result_s;
      when others =>
        readdata_s <= (others => '0');
      end case;
    end if;
  end if;
end process read_bloc;
```

In this `process` we check firstly `read_en_s` signal (high during one clock cycle) to determine if a read request is in progress. If the state is low, `readdata_s` remain unchanged. Otherwise the `addr_s` content is evaluated (with a case statement) to know which register is acceded, and read data bus is updated accordingly. When the address didn't match any of supported register the data bus is updated with a default value.

5.4 Write management

For a write request, principle remain, globally the same as a read request: `write_en_s` is sampled and when its state is high the value of the address bus is analyzed.

```
write_bloc : process(clk, reset)
begin
   if reset = '1' then
     op1_s <= (others => '0');
     op2_s <= (others => '0');
   elsif rising_edge(clk) then
     op1_s <= op1_s;
```

```
      op2_s <= op2_s;
    if write_en_s = '1' then
      case addr_s is
      when REG_OP1 =>
        op1_s <= s00_axi_wdata;
      when REG_OP2 =>
        op2_s <= s00_axi_wdata;
      when others =>
      end case;
      end if;
   end if;
end process write_bloc;
```

When address match one of supported registers (using again a case statement) corresponding signal is updated using `writedata_s` signal value. When the address is invalid: nothing is done but for code coverage a `others` statement is present.

5.5 CPU side: FPGA Communication

For this first exercise we will use **devmem** utility. This one allows to do read/write accesses directly to anywhere into the memory area (in our case: the shared memory between CPU and FPGA). It is used like this:

```
devmem 0x10 32 --> read @ 0x10
devmem 0x10 32 0x1234 --> writes 0x1234 @ 0x10
```

Second parameter (*32*) mean an 32 bits access.

Our IP is located at 0x43C00000 (absolute/CPU base address).

Two importants points to keep in mind:

1. the address used by **devmem** est absolue;
2. communication is done using 32 bits data size (4Bytes). So each registers address are multiples of 4 (or shifted by 2) (first is at 0xYYY00, second at 0xYYYY04, ...);
3. data size parameter is optional for read access (default is 32), but is mandatory for write access

Block ID read (address 0x43C00000 + 0x00) :

```
redpitaya> devmem 0x43C00000
0x00000001
```

Writing value 2 into register **REG_OP1** (0x43C00004 = 0x43C00000 + 0x01 << 2)

```
redpitaya> devmem 0x43C00004 32 2
```

Writting value 3 into register **REG_OP2** (0x43C00008 = 0x43C00000 + 0x02 << 2)

```
redpitaya> devmem 0x43C00008 32 3
```

Result access (5) by reading register **REG_RESULT** (0x43C0000c = 0x43C00000 + 0x03 << 2)

```
redpitaya> devmem 0x43C0000c
0x00000005
```

5.6 Exercises

1. complete read and write `processes`;
2. **REG_OP1** and **REG_OP2** registers are write only: based on **REG_ID** and **REG_RESULT** registers, add required code to read, from the CPU, current value;
3. This block, currently, is limited to the addition operation: add a new register to select between addition and substraction (asynchronous affectation must be, also, updated).

Note : one way to have a conditional affectation, in asynchronous mode, is to use something like:

```
signal_destination <= operation_ou_valeur_si_condition_vrai when condition
  else operation_ou_valeur_si_condition_fausse;
```

6 Reading a RAM from the CPU

VHDL project is located into /home/etudiant/tp_fpga/design/exercice_ram directory and vivado's project file is into objs/exercice_ram.xpr sub-directory

The idea to this second application is to start a long treatment from the CPU. This treatment will be simulated by a RAM's filling with some arbitrary data. This block must, also, provoding a status in order to inform CPU if it must wait for treatment completion or if fetching data may be done.

This second exercise will be to realize a block that :

1. following `start` request reception, it begins a pseudo acquisition. This one consist to start a counter and to fill a RAM using the counter's value as RAM's address and data;
2. status bit management in order to inform CPU when the block is busy or ready to new acquisition and/or previous acquisition's data fetching. Reading this status bit has to be done by CPU using a `polling` method; met à l'état haut un signal d'état pour signifier au processeur si l'acquisition est en cours ou fini. Le processeur pourra lire la valeur de ce signal au travers d'un registre de statut et le sondera par *polling*;
3. transmitting to the CPU the RAM's content with an address auto-increment at each new read request (acting as a pseudo *FIFO*).

So we need 4 registers:

31	...	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID															

registre 0x00 : unique block ID. Read only.

31	...	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X															busy

registre 0x01 : status register. Read only. **busy** remain high during acquisition. Low otherwise.

31	...	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X															start

registre 0x02 : Read Write. When `start` is set to 1: start acquisition.

31	...	13	12	11	10	9	8	7	6	5	4	3	2	1	0
data															

registre 0x03 : Read only. This register is used to fetch data one by one from the RAM.

VHDL language is relatively complex and verbose. To keep code readable, this block is splitted into 3 files:

- a wrapper to the RAM;
- a file dedicated to everything about CPU/FPGA communication;
- a **"top"** file containing all sub-entities instantiation and the processing part (counter, RAM write access)

6.1 Pseudo acquisition : synthesis/enseignement_ram/enseignement_ram.vhd

The `process` used to simulate a data stream acquisition has 2 states (represented by **busy** states):

- when `busy` signal has a low level state (l.13-17), the `process` is idle and waits for an acquisition request from the CPU (signal **start_acquisition_s**). When this signal goes high during one clock cycle (l.14-17), the `process` reset counter et set **busy** to high;
- when `busy` has a high level state (l.18-27), the `process` increment the counter to each clock cycle and send a write request to the RAM. When counter value is equal to $2^{10} - 1$, **busy** signal goes to low and the `process` is again into idle state;

Since **busy** signal has an high state during all the acquisition, this one is propagate towards `AXI` communication to be used as status bit.

```
cpt_storage_proc: process(s00_axi_aclk, s00_axi_reset)
begin
  if s00_axi_reset = '1' then
    busy_s <= '0';
    cpt_addr_s <= (others => '0');
    cpt_addr_next_s <= 0;
    cpt_en_s <= '0';
  elsif rising_edge(s00_axi_aclk) then
    cpt_en_s <= '0';
    cpt_addr_s <= cpt_addr_s;
    cpt_addr_next_s <= cpt_addr_next_s;
    busy_s <= busy_s;
    if busy_s = '0' then -- state idle
      if start_acquisition_s = '1' then
        busy_s <= '1';
        cpt_addr_next_s <= 0;
        cpt_addr_s <= (others => '0');
      end if;
    else -- state storage
      cpt_addr_s <= std_logic_vector(unsigned(cpt_addr_s) + 1);
      cpt_data_s <= (31 downto 10 => '0') & cpt_addr_s;
      cpt_en_s <= '1';
      if cpt_addr_next_s = 1023 then
        busy_s <= '0';
      else
```

```
        cpt_addr_next_s <= cpt_addr_next_s+1;
      end if;
    end if;
  end if;
end process;
```

6.2 AXI communication : synthesis/enseignement_ram/ens_ram_comm.vhd

6.2.1 Writing management

Only one register is handled: **REG_START**. It copy `writedata_s` LSB into **start_acquisition_o**. This signal is propagate towards acquisition process seen before, as start acquisition signal.

The signal produce by **REG_START** doesn't keep its value: at the next clock cycle, when write condition are no more validate this signal goes to low level. This behavior is required to avoid to restart acquisition as soon as finished.

6.2.2 Reading management

In addition to handling the id it must handle:

- requests to know the state of the block (busy or inactive). This task consists of concatenating a set of bits in the low state ((31 downto 1 => '0')) and as the LSB the signal **busy_i** corresponding to the signal of the acquisition `process`;
- the requests for obtaining the successive data contained in the RAM. This process consists of copying the value carried by the data bus of the RAM (signal **data_val_i**) in the data bus FPGA towards CPU and to advance the index of the RAM for the request next reading. It should be noted that when the **data_addr_s** signal reaches 0x3ff its increment will reset this signal to 0x000.

Note: For data reading, the `process`, must to each requests, doing to tasks:

1. RAM address increment;
2. writing to `readdata_s` the current value contained to the RAM data read bus.

6.3 CPU communication

Utilities like *devmem* doesn't fit requirement to read a full dataset. Of course, it's always possible to write a script to handle that with this tool but this approach isn't optimal because each calls imply a long serie of subcall between userspace and kernel space.

This why we will uses an userspace application, written in C langage, to communicate between CPU and FPGA.

For Linux point of view, we must convert the physical address to virtual address (due to MMU). Mapping between both address is automatically done by software and hardware. To have access to the virtual address with GNU/Linux, ones must using `mmap()`. But this requires firstly time to open `/dev/mem` pseudo-file to have access to the memory.

So the first step for FPGA access is to open `/dev/mem` in Read and Write mode:

```
int fd = open("/dev/mem", O_RDWR|O_SYNC);
if (fd < 0) {
  printf("can't open file /dev/mem\n");
  return -1;
}
```

With the file descriptor **fd** we now have to request, using *mmap*, a pointer to the memory base address corresponding to the physical one where we wish to access.

```
void *ptr_fpga = mmap(0, 8192, PROT_READ|PROT_WRITE, MAP_SHARED,
        fd, 0x43C00000);
if (ptr_fpga == MAP_FAILED) {
  printf("mmap failed\n");
  return -2;
}
```

and, finally, we can do a read like this:

```
unsigned int pos = fpga_offset+ registre;
unsigned short content = *(unsigned short*)(ptr_fpga+((unsigned short)(pos)));
```

and for writing:

```
unsigned int pos = fpga_offset + registre;
*(unsigned short*)(ptr_fpga + pos) = (unsigned short)value;
```

The full piece of code will look like that:

```
int fd = open("/dev/mem", O_RDWR|O_SYNC);
if (fd < 0)
  return EXIT_FAILURE;
ptr_fpga = mmap(0, page_size, PROT_READ|PROT_WRITE, MAP_SHARED,
```

```
            fd, FPGA_BASE_ADDR);
if (ptr_fpga == MAP_FAILED)
  return -2;

pos = FPGA_OFFSET + REG_ID;
value = *(unsigned short*)(ptr_fpga+((unsigned short)pos));
if (value != 1)
  return EXIT_FAILURE;

pos = FPGA_OFFSET + REG_START;
*(unsigned short*)(ptr_fpga+pos) = 0x01;

pos = FPGA_OFFSET + REG_STATUS;
do {
  value = *(unsigned short*)(ptr_fpga+((unsigned short)pos));
} while((value & 0x01) == 0x01);

pos = FPGA_OFFSET + REG_DATA;
for (i=0; i<1024; i++) {
  value = *(unsigned short*)(ptr_fpga+((unsigned short)pos));
  printf("%hu\n", value);
}
```

- Firstly, it open */dev/mem* (l.1-3), then uses *mmap* (l.4-7) to obtain a pointer to the memory address corresponding to the block into the FPGA mémoire correspondant au bloc dans le FPGA;
- then a read is done to the register containing ID to check the match (l.9-12);
- it sent a start acquisition request (l.14-15) and wait until task is finished (l.17-20);
- At the end it fetch sequentially the dataset contained into the RAM (l.22-26) and displays each value.

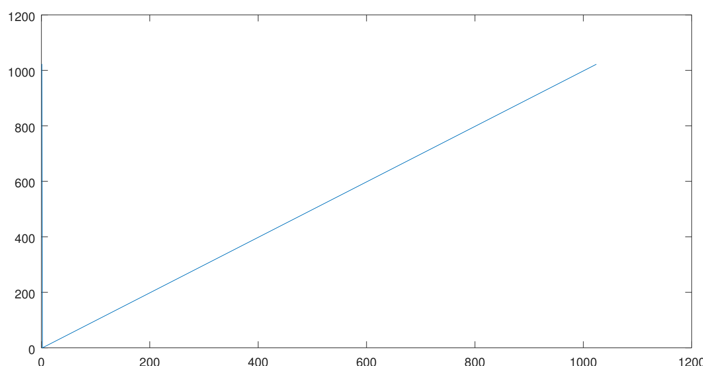
To compile this program, it's necessary at first time, using this command:

```
source sourceme.ggm
```

(This file is located at archive's root).

Then use make into the directory containing sources et finally copied binary file called **app_exercice_ram** into */home/etudiant/nfs* with **cp**.

To execute this application and fetching values to a file, on the platform, into */mnt* directory, you have to use the command ./ app_exercice_ram > test1.dat (> caracter is used to redirect STDOUT towards test1.dat).



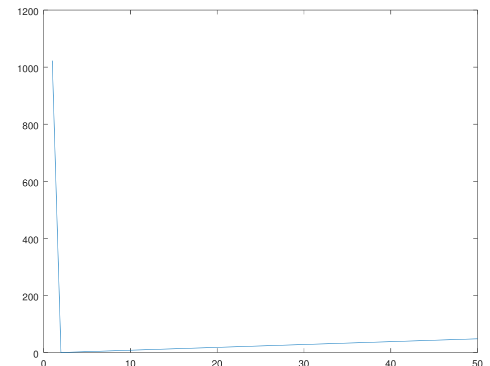


Figure 7: Result seems good (left) ... But shows an incoherency (ramp seems shifted by one sample).

Complete the exercice_ram gateware's code, then run the acquisition. Using octave display the curve. It must be simitar to one shown in figure 7.

```
a = load("test1.dat");
plot(a);
```

We can see that instead of having a sequence between 0 and 1023 (i.e. 1024 points), the first point is worth 1023 then the continuation is logical with 1022 as last value.

Propose a solution so that the data is correctly aligned (the error does not necessarily come from the communication part).

7 Realization of the measurement-storage component

The first two exercises made it possible to acquire the basics of communication between the processor and the FPGA within the framework of communication by AXI bus. It is now possible to create the counter block used to evaluate the latencies of an in GNU/Linux user space application or in the real-time domain of Xenomai.

The purpose of this exercise is to have a block able of measuring the duration between two consecutive rising edges, with a counting depth allowing durations of several seconds.

7.1 Period counter implementation

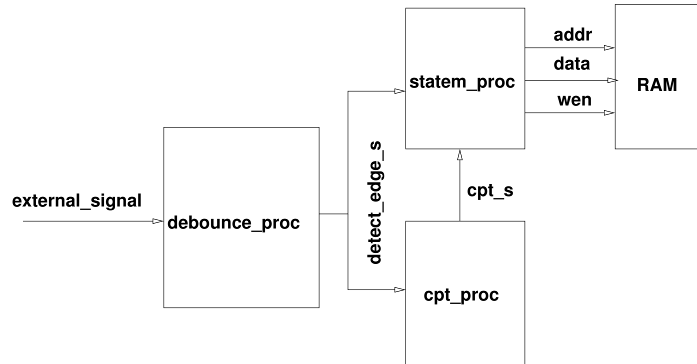


Figure 8: Global scheme of IP to implement

The counter part of this component is divided into three `process` (Fig. 8):

1. a *debounce_proc* `process`: its role is to "clean up" the signal to be studied. Indeed, this signal comes from an external system, **"glitches"** may appear and false the results. This `process` also drives a signal which, if high, warns the other processes of the detection of a rising edge.
2. a `process` *cpt_proc* which counts the number of periods of the FPGA clock between two rising edges of the signal to be studied. His value is reset on detection of the edge and incremented otherwise;
3. a *statem_proc* `process`, similar to the pseudo-acquisition process of exercise 2, comprises a state machine which is by default in waiting for an acquisition order from the processor. When this order is received, the `process` performs n consecutive acquisitions before to return to the initial state.

7.2 debounce_proc

The principle of this `process` is to evaluate a series of past states of the signal to be sampled to ensure that it has remained stable during this time.

This mechanism is based on a shift register, updated at each rising edge of the clock, which maintains the state of the signal over several clock cycles. So if the register is composed of '1' the state is a stable high state, if it is only composed of '0' it is a stable low state.

To determine the state change (low-high or high-low transition), a signal is used. Thus, if the state determined by the register is high and the signal is '0' there was a rising edge and vice versa. This detection is used to propagate the rising edge information for the processing carried out by the other `process`.

7.3 cpt_proc

This `process` sample, at each clock cycle, the state of the detection signal of a rising edge:

- if high, the counter is reset to 0 (use of (others => '0'));
- if low the signal is incremented by cpt_s <= std_logic_vector(unsigned(cpt_s) + 1);

In order to be able to count several seconds, the counter is coded on 32 bits.

7.4 statem_proc

Overall, this `process` is close to the pseudo-acquisition of the previous exercise.

It has three states:

1. an **IDLE** state: the `process` is waiting for the order, coming from the processor, to start a series of acquisitions;
2. a **wait_first_edge** state: in order to guarantee the consistency of the data, the `process` waits for a first edge of the sampled signal before start the acquisition;
3. the last state **acquire_time** also waits for a high level signifying the detection of an edge. When this condition is evaluated as true, a write to RAM is triggered. At the same time, the address is evaluated to determine if the acquisition is finished. In the first case, the state changes to **IDLE** again, otherwise the address value is incremented.

7.5 FPGA's communication layer

Communication handling code, proposed in ths project, is the same as previous exercise.

7.6 Communication: CPU's side

Here again, considering similarities between this exercise with previous, code is the same as previous exercise.

7.7 Measurement of the period of the sampled signal

Results for *gpio_sleep* (Figs. 9, *xeno_gpio_sleep* (Figs. 10) and *xeno_gpio_timer* (Figs. **??**) They present results consistent with the measurements already described using an oscilloscope. Result for *gpio_sigalarm* are not presented because they seem very suspicious (excessive dispersion of measurements).

The histograms were obtained with octave thanks to the commands:

```
a = load('nom_du_fichier.dat');
a = a./125; % conversion base 8ns en 1us
figure;[xx,nn]=hist(a,[1150:100:2000]);bar(nn,log(xx+1))
xlim([1150 2000]);xlabel('intervalle de temps (us)');ylabel('log(occurences+1)')
title('un titre');
```

Limits [1150:100:2000] are to be adapted according to results.

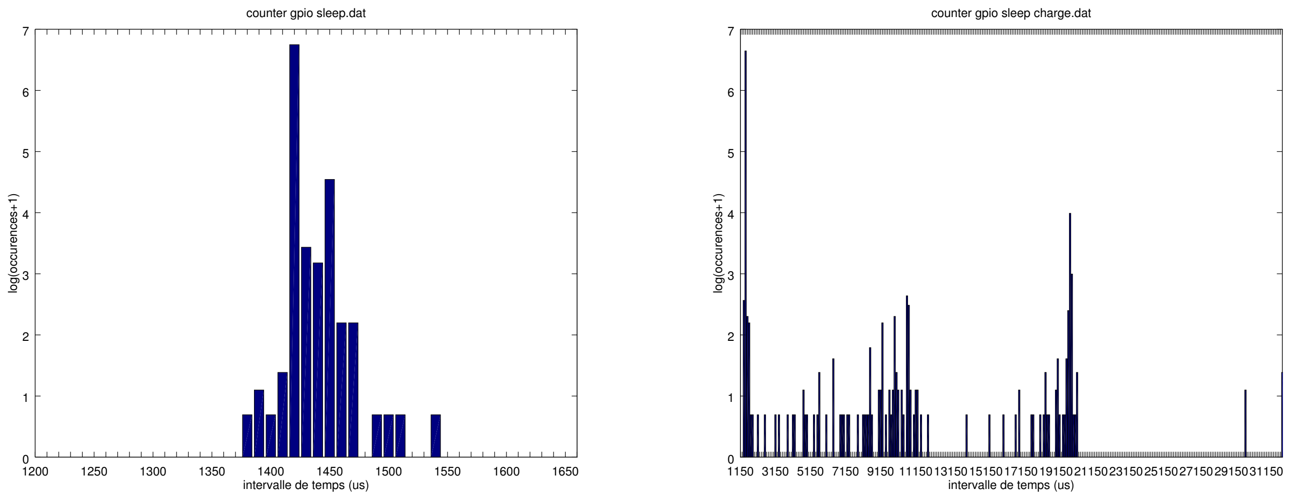


Figure 9: Measurement of a periodic signal generated by a user-space application in which the change of the output state occurs after $500\mu s$ by using the **usleep** command, thus giving a period of 1ms. Left: on an unloaded system the period lasts on average 1440 μs (min 1377 μs, max 1538 μs). Right: on a loaded system, the signal fluctuates between 1385 μs and 40 ms.

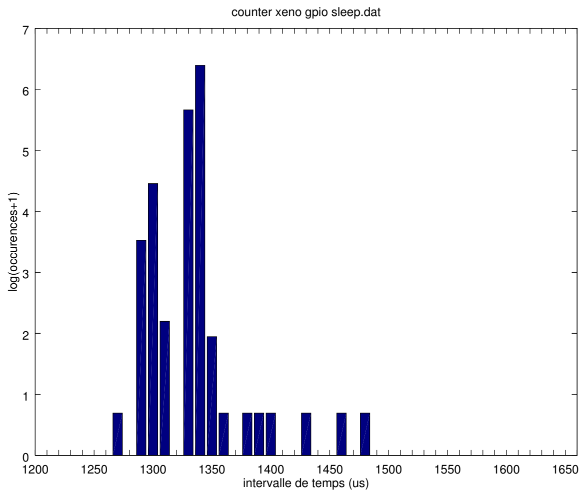
8 Linux and Xenomai qualifications

Applications are stored into */home/etudiant/tp_fpga_sources/apps/gpio_test_xeno* directory

In order to evaluate the behavior of the different domains in the case of an unloaded and loaded system, we will generate a periodic signal by software. The period will be $1ms$. We will therefore have to change the state of a pin (pin `E8/PS_MIO13_500` corresponding to pin 6 of the E2 connector of the `Redpitaya`) every $500\mu s$.

To do that, we will implement this sequence with two differents approaches:

1. suspending the task during a fixed duration;
2. by using a timer.



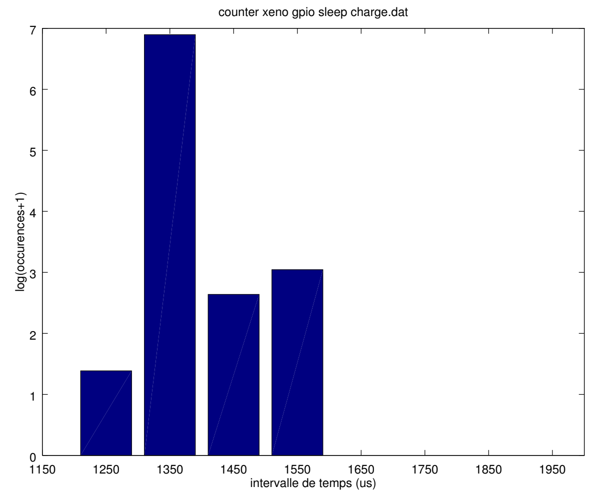
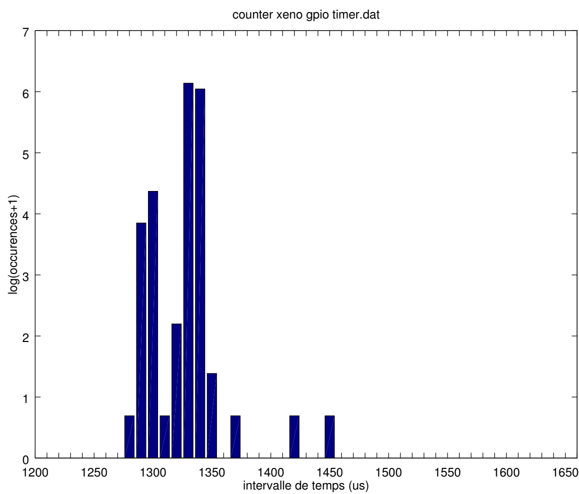


Figure 10: Xenomai : generation of a 1 ms signal on hold. Left: no loaded system, periods have mainly a duration of 1340 μs (min : 1271 μs, max : 1476 μs). Right: the trend remains equivalent (min : 1251 μs, max : 1560 μs).



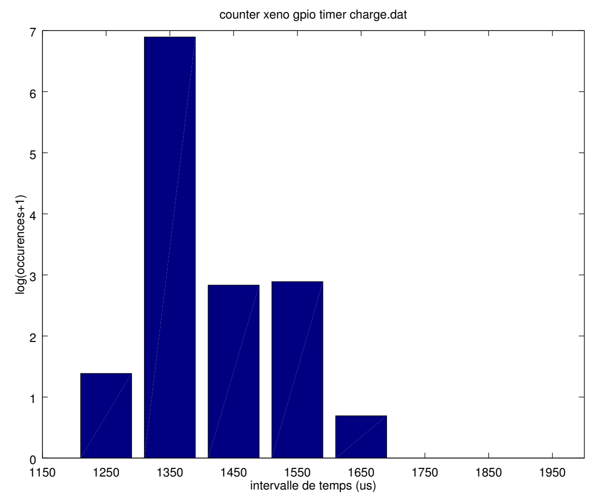


Figure 11: Xenomai : generation of a 1ms signal using a timer. Left: not loaded system, periods are mainly of a duration of 1330 μs (min : 1281 μs, max : 1451 μs). Right: the trend remains the same (1256 μs to 1623 μs).

8.1 Periodic signal generation: sleep mode

```
int main(void)
{
    gpio_config;

    while(1) {
        toggle_pin
        sleep
    }
}
```

gpio_sleep.c : periodic signal generation principle: Linux domain.

```

void blink(void *arg)
{
    while(1) {
        toggle_pin
        sleep
    }

}

int main(void)
{
    gpio_config;
    create_task
    start_task
    wait
    delete_task

}
```

xeno_gpio_sleep.c : periodic signal generation principle: Xenomai domain.

8.1.1 Handling pins from a userspace application

Configuring pins is done in two steps:

Initializing communication layer using the function:

```
unsigned char *red_gpio_init(int pin_num);
```

with `pin_num` the pin number. This function return a pointer toward gpio controller memory area.
then, pin configuration as output:

```
void red_gpio_set_cfgpin(unsigned char *mem, int pin_num);
```

with:

- **unsigned char *mem** pointer returned by `reg_gpio_init` function;
- **pin_num** pin ID, here: 13.

Toggling the pin state is done with this function:

```
void red_gpio_output(unsigned char *mem, int num, int value);
```

with:

- **mem** memory pointer;
- **num** pin ID;
- **value**: new state (0: low, otherwise high)

8.1.2 Putting the process to sleep

For a Linux's userspace application the function to use is:

```
int usleep(useconds_t usec);
```

Where *usec* is the duration (μs) (integer).
For Xenomai the equivalent function is:

```
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

with:

- rqtp a structure already filled:

  ```
  struct timespec tim = {0,TIMESLEEP};
  ```

  This struct is passed by pointer ('&'). TIMESLEEP: *ns* duration (integer)
- *rmtp* unused: **NULL**.

8.1.3 Real-time task creation

A Xenomai application is launched from the time-shared scheduler, to be able to switch to the real-time scheduler it is necessary to create a task that will perform the processing.

This function:

```
int rt_task_spawn(RT_TASK *task, const char *name, int stksize, int prio, int mode,
    void(*)(void *cookie)entry, void * cookie);
```

Allows to create the task and start it, parameters are:

- *task* : a pointer to the task already defined;
- *name* : task name (or NULL).
- *stksize* : stack size. Using 0 lets the system decid. décider;
- *prio* : task's priority, for our tests we will uses 99 (highest priority);
- *mode* : **T_JOINABLE**: main thread will wait until task stop;
- *entry* : a function's pointer. This function is the task behavior;
- *cookie* : to pass something (informations, params) to the function (NULL when unused).

To avoid *main* stop before task stop, it's required to use the function:

```
int rt_task_join(RT_TASK *task);
```

where the parameter is the pointer (*RT_TASK*) to the task (*rt_task_spawn*).

8.2 Signal generation using a timer

```

void blink(int signum)
{
    toggle_pin
}

int main(void)
{
    gpio_init;

    event_handler_init
    timer_init

    while(1) {}
}
```

gpio_sigalarm.c : periodic signal generation using a timer (Linux domain).

```

void blink(void *arg)
{
    timer_init

    while(1) {
        wait_timer
        toggle_pin
    }
}

int main(void)
{
    gpio_init;
    task_create
    start_start
    wait
    task_delete
}
```

xeno_gpio_sigalarm.c : periodic signal generation using a timer (Linux domain).

GPIO configuration and task management, for Xenomai, remain the same.

8.2.1 Timer configuration: Linux scheduler

The management of a timer (call to an interrupt handler) is done in several parts:

Registering a function as signal SIGVTALRM handler

Done using a structure:

```
struct sigaction sa;
```

First: all attributes must be to '0' using

```
memset(void *ptr, int c, size_t n);
```

where:

- *ptr*: structure's pointer (use of '&');
- *c* character to use ('0');
- *n* write size (Byte) (**sizeof(sa)**).

The second step is to pass the function's pointer to this structure. It will be called when the event is triggered:

```
sa.sa_handler = &test;
```

And finally we must register our handler to respond to the right event using the function:

```
int sigaction(int signum, const struct sigaction *act,
                    struct sigaction *oldact);
```

where:

- signum is the event ID (SIGVTALRM);
- act : the tructure previously filled;
- oldact : unused (NULL), allows to save current configuration.

configuration of the generation of a signal at regular intervals on a Linux scheduler
This step is based on filling a structure:

```
struct itimerval timer;
```

4 fields must be filled:

- it_value.tv_sec : initial trigger after 'n' sec (so 0)
- it_value.tv_usec : initial trigger after 'n' usec (so 500)
- it_interval.tv_sec : periodic trigger after (0)
- it_interval.tv_usec : periodic trigger after 'n' usec (500)

And finally the function:

```
int setitimer(int which, const struct itimerval *new_value,
                    struct itimerval *old_value);
```

- which defines which timer compter to use and which signal will be emitted (ITIMER_VIRTUAL);
- new_value : a structure already filled;
- old_value : unused (NULL) to save current configuration;

8.2.2 Timer configuration: Xenomai scheduler

Using a timer to achieve a period task is done in two steps;
configuration :

```
int rt_task_set_periodic(RT_TASK *task, RTIME idate, RTIME period);
```

with:

- *task* a pointer to an already available task. If NULL the current task becomes periodic;
- *idate* starting time. If **TM_NOW**: the start is immediate avant le démarrage;
- *period* period length in nanosecond.

then into the loop the function :

```
int rt_task_wait_period(unsigned long *overruns_r);
```

where overruns_r provides overruns number (ou NULL when unused). This function is used to suspend the task until timer trigger.

8.3 Use

8.3.1 Period counter

A bitstream is provided (available at `/home/etudiant/tp_fpga_sources` and called `top_exercice_counter.bit.bin`). It's, also, required to compile and install application located at `/home/etudiant/tp_fpga_sources/apps/app_exercice_counter`

These applications copied into `/home/etudiant/nfs` are available from board into `/mnt` directory.

The first step is to flash bitstream using:

```
cp top_exercice_counter.bit.bin /lib/firmware
echo top_exercice_counter.bit.bin > /sys/class/fpga_manager/fpga0/firmware
```

counting and results storage is done by:

```
./app_exercice_counter
```

8.3.2 Testing the various solutions

Since there is only one terminal on the target, the applications must be launched in the background (use of an '&' after the name of the application.

To stop application it's required to kill it:

```
# ps aux
PID   USER     COMMAND
  625 root     ./gpio_sleep
#kill 625
```

Once the application is launched, a series of acquisitions will be made by running the command ./app_exercise_counter. When it stop, a **counter.dat** file is created in the current directory. This file must be renamed to be representative of the test.

To load the system, a script, named *charge.sh* is available in */home/etudiant/tp_fpga_sources/apps/app_exercice_counter*.

```
./charge.sh &
```

To stop it you must removes *attente.nop* (with **rm**).

8.3.3 Use of results

This step is done thanks to *octave* (host computer). Commands to use are:

```
% file load
a = load("fichier.dat");
% time base 8ns => /125 for us
a=a./125;
% display histogram
hist(a)
```

To have better details it's also possible to use:

```
figure;[xx,nn]=hist(a,[min(a):10:max(a)]);bar(nn,log(xx+1))
```

Warning : Depending on limits values, and step, this command may be very slow.

References

[1] *Wishbone B4 – WISHBONE System-on-Chip (SoC)Interconnection Architecture for Portable IP Cores*, OpenCores, 2010