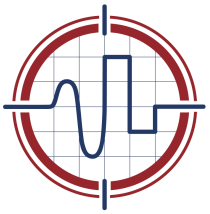


ISOTDAQ

12TH International School of Trigger and Data Acquisition

13-23 June 2022

Catania (Italy)

ISOTDAQ 2022

Lab Book

Index. Tutors and their Labs

Index. Tutors and their Labs **1**

Lab 1: VMEbus programming - Markus Joos **3**

Lab 2: NIM. A simple Trigger Exercise – Andrea Negri, Vincenzo Izzo, Francesca Pastore **5**

Lab 3: NIM & Scintillator. Detector and Trigger – Kostas Kordas, Roberto Ferrari, F. Pastore **16**

Lab 4: Muon DAQ. A small physics experiment – Enrico Pasqualucci, Martin Schwinzerl **23**

Lab 5: FPGA programming – Dominique Gigi, Petr Zejdl **30**

Lab 6: Micro TCA – Hannes Sakulin, Marc Dobson **37**

Lab 7: System Development using LabVIEW – Adriaan Rijllart **48**

Lab 8: ADC basics for TDAQ – Manoel Barros Marin **63**

Lab 9: Networking for Data Acquisition Systems – Vesa Simola **74**

Lab 10: Microcontrollers Exercise – Mauricio Feo, Cristóvão Beirão **82**

Lab 11: Storage Exercise – Tommaso Colombo **89**

Lab 12: DAQ Online Software – Enrico Gamberini, Marco Ceoletta **93**

Lab 13: System on Chip (SoC) FPGA – Johannes Martin Wuthrich **100**

Lab 14: GPU. Introduction to GPU programming – Giuseppe Lamanna **123**

The main page of the 2022 ISOTDAQ School is:

https://indico.cern.ch/event/928767/

For up-to-date information on times and places, please see:

https://indico.cern.ch/event/928767/timetable

Local Organising Committee

Alessandro Di Mattia (INFN)

Cristina Tuvè (UNICT & INFN)

CERN Organising Committee

Paolo Durante

Markus Joos

Hannes Sakulin

Barthélémy von Haller

Laboratory Usage Note

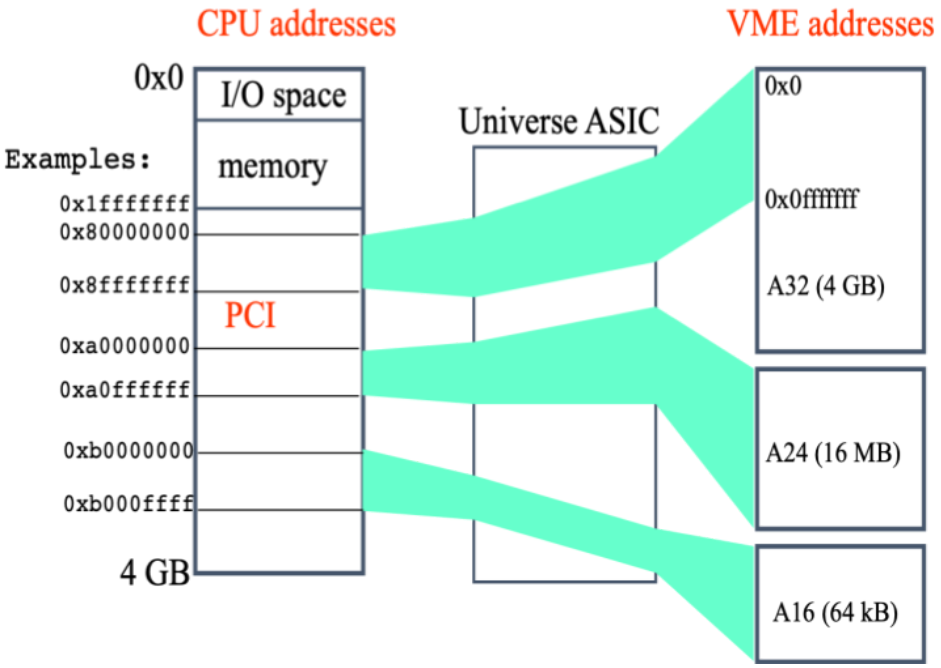
Maintaining a safe working environment in the laboratory is paramount. You should at all times act in a safe and responsible manner. In particular, note that you must not eat, drink, or act unprofessionally in the Laboratory. Any safety concerns should be raised with the local organization committee.

Lab 1: VMEbus programming

Introduction

For the moment forget what you (hopefully) have learnt about the VMEbus protocol and the details of the H/W. For this exercise you have to look at a VMEbus slave as if it was a piece of memory in your PC. The purpose of this exercise is to demonstrate that in some respects there is little difference between internal and external memory; as far as programming is concerned. The exercise also shows the differences between the two types of memory.

What is important to understand is that the VMEbus memory has to be mapped into the (virtual) address space of a user process before it can be accessed. This ties 3 busses together: CPU, PCI and VMEbus as shown in the picture below.



The first part of the exercise is to figure out how to create the appropriate mappings for the type of VMEbus access that you have to do. Then you actually transfer the data. This is done in single cycle mode which means that the CPU controls the data transfer.

In the second part of the exercise you will perform block transfers (DMA). This requires a different programming technique since it is not the CPU that moves the data but an external device (a DMA controller). Such DMA controllers are not VMEbus specific. You find them everywhere (e.g. in Network interfaces, disk controllers, USB devices, etc.).

Before you start you should be able to answer these questions:

1) What does the acronym A24D32 mean?
2) What is endianness and how do you deal with it?
3) What are the advantages of block transfers?

Instructions

1. On the VMEbus single board computer log on with the DAQ school account (daqschool / g0ldenhorn).
2. Run “source setup” and then change directory to exercise1/groupX
3. Open the file solution.cpp with an editor of your choice (vi, nedit).
4. Add the missing code to “solution.cpp” to execute the VMEbus cycles listed below:
    1. Write 0x12345678 to address 0x08000000 in A32 / D32 mode. Use the "safe" cycles
    2. Read the data back from address 0x08000000 and compare it
    3. Write 0x87654321 to address 0x08000004 in A32 / D32 mode. Use the "fast" cycles
    4. Read the data back from address 0x08000004 and compare it
    5. Write a block of 1 KB to address 0x08001000 in A32 / D32 / BLT mode. You have to prepare the data in a cmem_rcc buffer.
    6. Read the data back from 0x08001000 in A32 / D64 / MBLT mode and compare it
5. Run “make” to compile the application
6. Run “solution” and catch the VMEbus transfers with the VMEtro VBT325 analyser

Good practices:

- Check all error codes
- Do not forget to undo all initialization steps (return memory, close libraries) before you exit from an application

Lab 2: A simple Trigger Exercise

Introduction

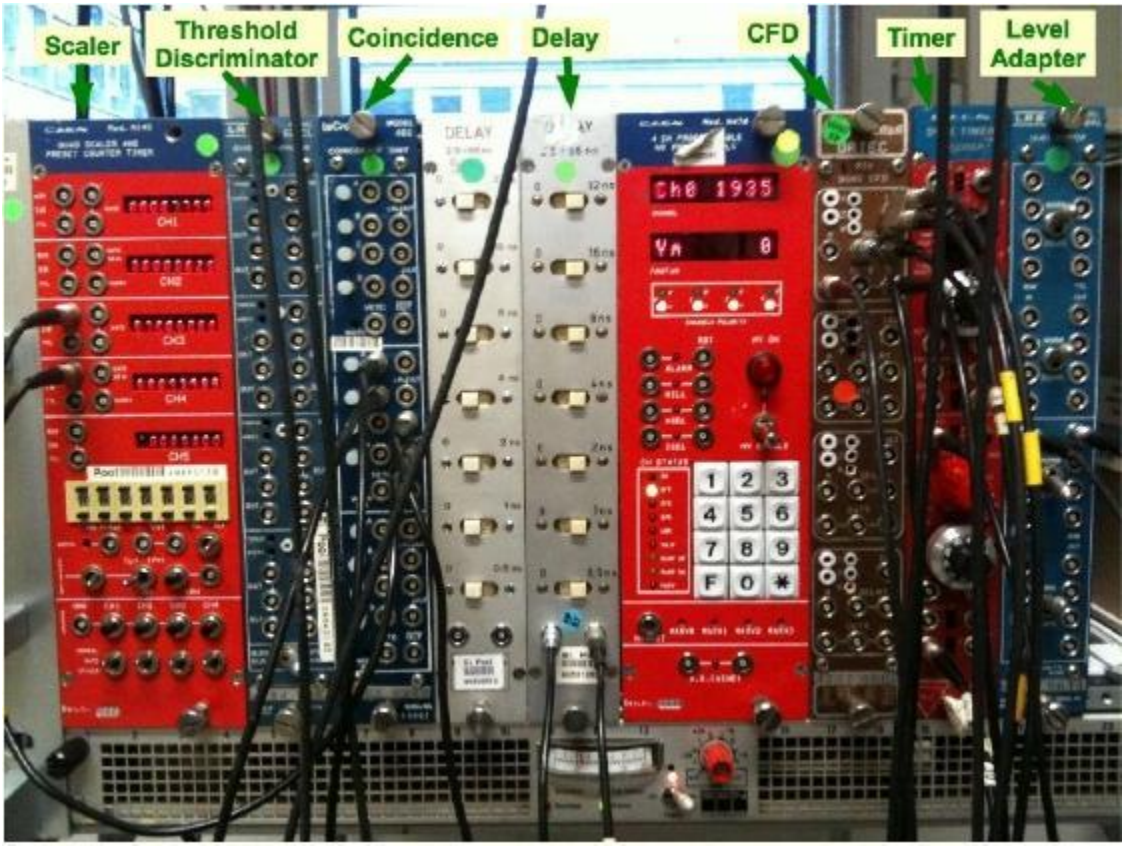


Figure 1: NIM modules.

This is a basic exercise based on the trigger lecture. It introduces all the elements and concepts needed in exercise 3 and 4. The available NIM modules are shown in Fig.1. The exercise is composed of 4 parts. At each step, look at the corresponding schema and follow the instructions.

A trigger is given by the transition of a signal from the logical 0 to 1. Before setting up any trigger system, you must have decided the levels corresponding to these logical levels and all the components of the system need to be correctly configured.

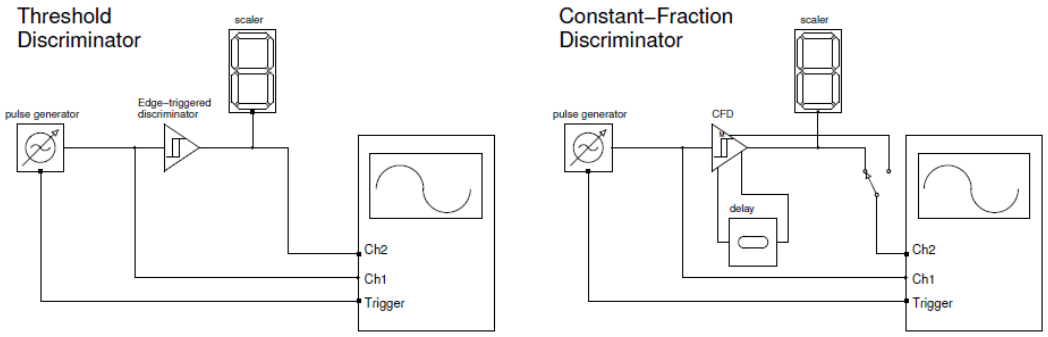


Figure 2: Scheme of threshold and constant fraction discriminators.

Part 1a: Threshold Discriminator

The Signal Generator is pre-configured to provide a triangular pulse with a period of 300 µs. Look at the signal (Channel Output) with the oscilloscope (CH1), using the Trigger Output of the generator as oscilloscope trigger (EXT). The Trigger Output is TTL signal.

What do you expect?

Why do we use it?

Now try to characterize the signal:

Leading edge time:	
Trailing edge time:	
Width:	

Using the LEMO cables, try to implement the schema shown in the left part of Fig. 2, i.e.:

- Split the generator output signal: connect the two parts to the input of the **Threshold Discriminator** and to the oscilloscope.
- Connect one output signal of the discriminator to the scaler module and a second output to the oscilloscope (CH2).

We have set-up a simple trigger system: you have a digital answer based on the amplitude of a signal. Reproduce the oscilloscope display shown in Fig. 3 and observe the amplitude of both the signal and its corresponding trigger.

Can you modify their amplitude?

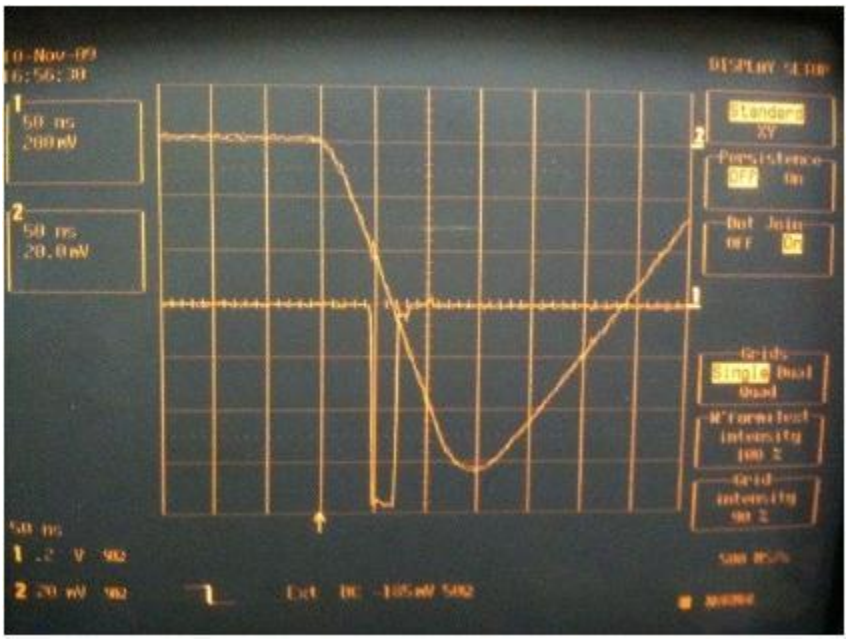


Figure 3: Input signal and threshold discriminator output.

The threshold set on the discriminator can be measured with a Voltmeter (x10 output) and changed with a screwdriver. Change the threshold value: observe the behavior of the discriminated signal on the scope and its rate on the scaler.

Can you relate them to the threshold values?

In real experiments, how is the best threshold value found?

Part 1b: Threshold Discriminator, the jitter

Using the above set-up, set the discriminator threshold to 60 mV and change the amplitude of the input signal.

What is the effect on the discriminated signal?

How does it affect a timing measurement?

Measure the discriminated signal delay with respect to the reference as a function of the amplitude of the input signal (-100, -150, -200, -250 mV) and fill up Table 1 with your numbers.

Input signal amplitude (mV)	Threshold D (ns)	CFD (ns)
100		
150		
200		
250		

Table 1: Measured delays on the discriminated signal with respect to reference.

Part 2: Constant Fraction Discriminator (CFD)

Using the above set-up, set the discriminator threshold to 60 mV and change the amplitude of the input signal.

Now use the Constant Fraction Discriminator to make a trigger from the generator signal and implement the layout shown in the right diagram of Fig. 2.

Using the Voltmeter and the screwdriver, set these CFD parameters:

- threshold (T): 60 mV Measure with Voltmeter (x10 output)
- walk (Z): 2 mV Measure with Voltmeter
- delay (D): 80 ns Set with delay module + 2x10ns cables

Connect the CFD monitor output (M) to the scope CH2 and reproduce Fig. 4.

Figure 4: Input signal and CFD monitor output.

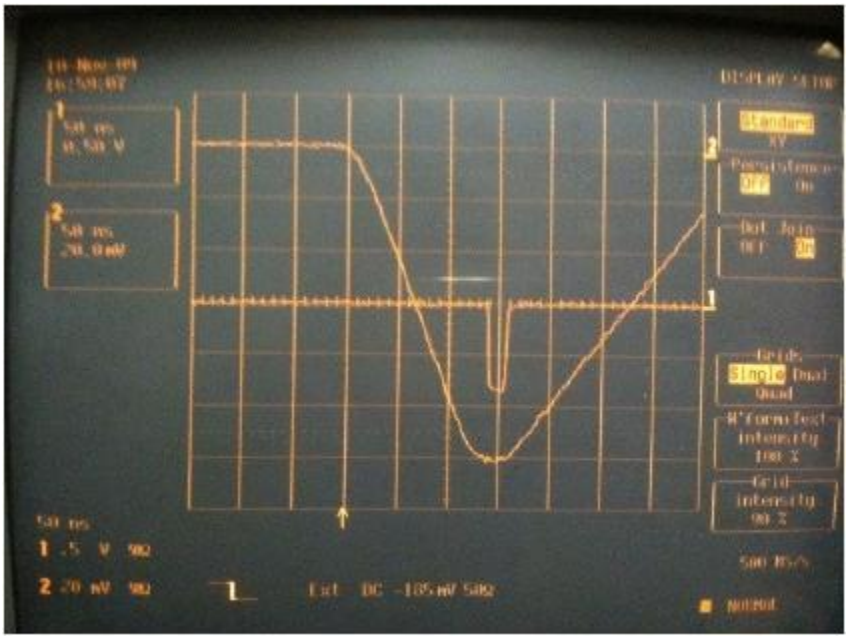


Figure 5: Input signal and CFD output.

Can you recognize the CFD technique?

What is the effect of varying the value of the delay D?

Now connect the CFD output to the scope (CH2) and change the amplitude of the input signal.

What happens to the output of the discriminator?

Measure the discriminated signal delay with respect to the reference as a function of the amplitude of the input signal (-100, -150, -200, -250 mV). Fill up Table 1 with your numbers. Compare the results with the previous measurements.

Can you see the advantage?

Can you make the CFD behave like a normal threshold discriminator?

Which configuration parameter has to be modified?

Part 3: Making a timing coincidence

We now try to simulate the coincidence of two different trigger signals, in a simplified way. For that, use an additional output of the signal generator, which is configured to generate a triangular pulse similar to the first one. Use a standard threshold discriminator unit (attached to ch. 2 of the generator) and a CFD (ch. 1) to discriminate both signals, as described in Fig.6.

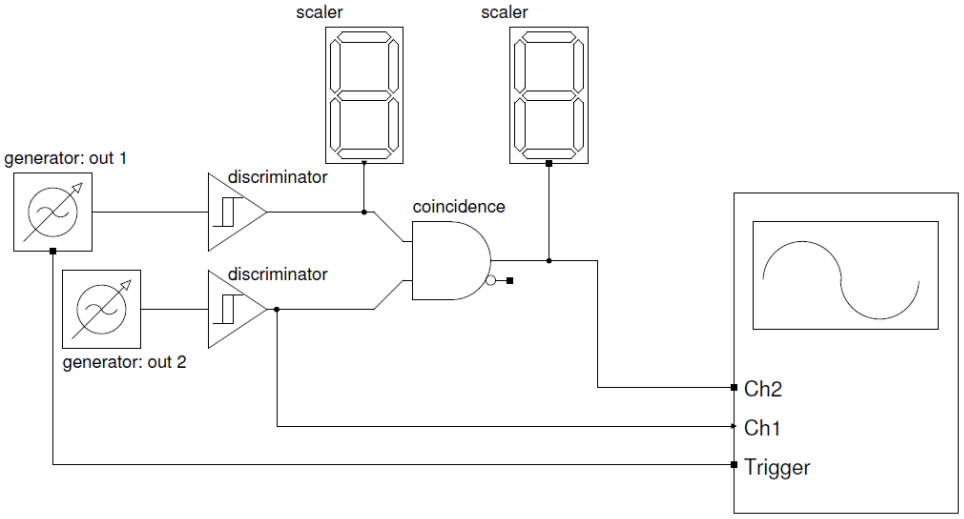


Figure 6: Coincidence layout.

We now have two independent trigger signals with similar characteristics. Look at them in the scope.

Which parameters are important when making a coincidence?

Use one unit of the **Coincidence Module**, which is able to generate the logical AND of its input signals. The module has two outputs: OUT and LIN-OUT.

Can you guess the timing behaviour of the AND output?

When are you expecting the AND output to rise?

The Scaler Module is a simple and useful tool in a trigger system: it allows you to simply count the triggers and verify if your system is behaving correctly. Use the scaler to measure the counting rate of your coincidence and try to answer these questions:

Can you count any trigger? How can you recover the coincidence rate?

After your adjustments, what is the width of the coincidence signal?

Can you explain the different behavior of the OUT and the LIN-OUT signals?

Which is better to use in a real trigger system?

How can you preserve good trigger efficiency if one of the signals has a large jitter?

Which is the drawback?

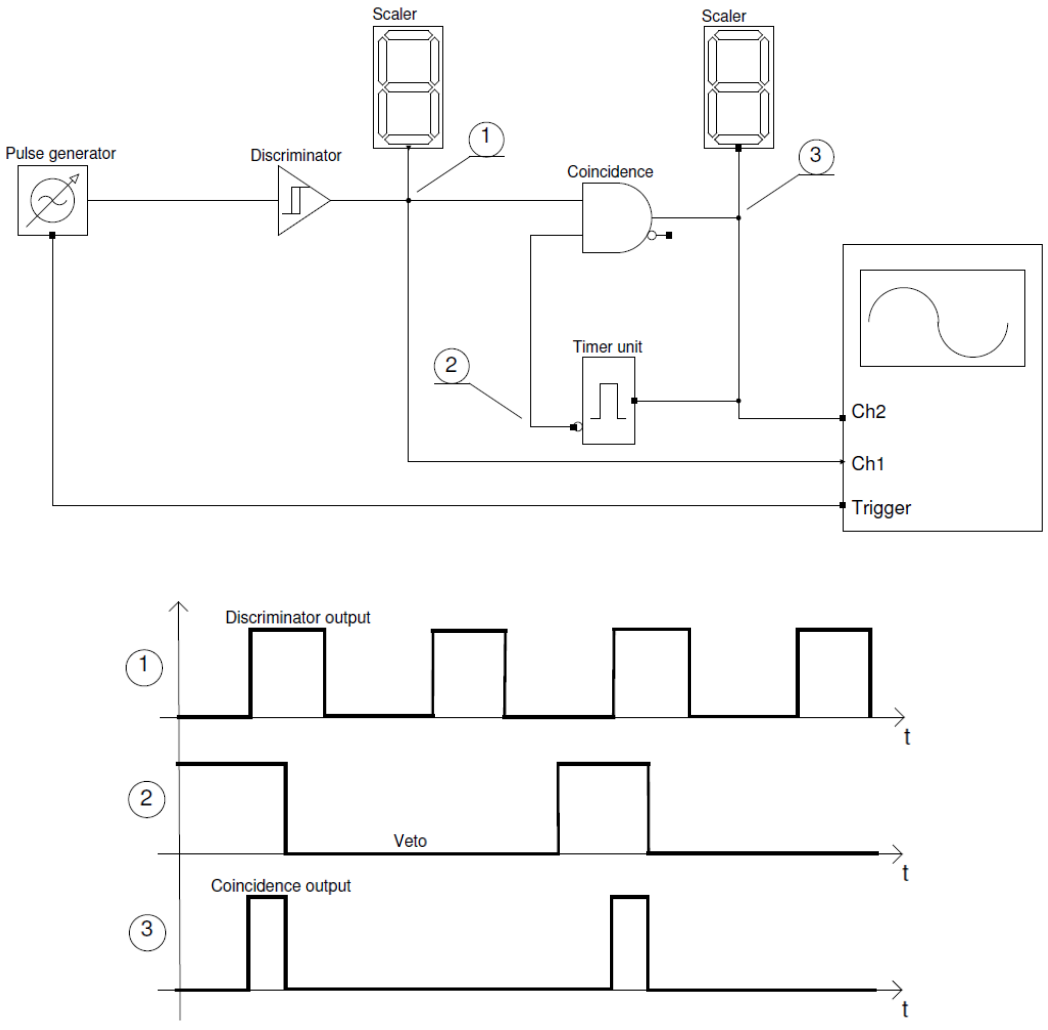


Figure 7: Top: busy logic schema with readout processing time simulated via a dual timer module. Bottom: time diagram of signals at the discriminator output (1), after the veto (2) and at the coincidence output (3).

Part 4: Trigger veto and dead-time

A busy logic can be implemented using the coincidence module and a Dual-Timer Module which simulates a readout system with a fixed processing time (readout dead-time). Configure one stage of a dual timer module to generate signals with 10 ms width. Then implement the busy logic in a second stage of the coincidence unit as shown in Fig. 7

- one input of the coincidence unit is the trigger signal;
- to simulate the start of the readout, and so the trigger ACCEPT signal sent to the readout system, use the output of the busy coincidence to drive the timer module (START);
- use the output of the timer as the VETO of the busy coincidence: this is the BUSY signal sent back to the trigger system;
- connect the trigger signals before and after the busy logic to the scaler and check the correct logic

You can easily make a rate measurement configuring the Scaler to work with a time gate of 1s with a GT+CLR configuration. Compare the trigger ACCEPT rate and the readout rate (after the BUSY) on the scalers.

How do they relate to the timer module setting?

Can you reproduce the numbers using the LIN-OUT of the coincidence unit?

Alternatively you can make an AND between the trigger and the output of the timer (with inverted logic) and not as a veto.

Where is the difference in the logic? What risk are we taking?

Can you explain the behaviors observed disabling either one or the other input of the coincidence unit?

Appendix: the Constant Fraction Discriminator

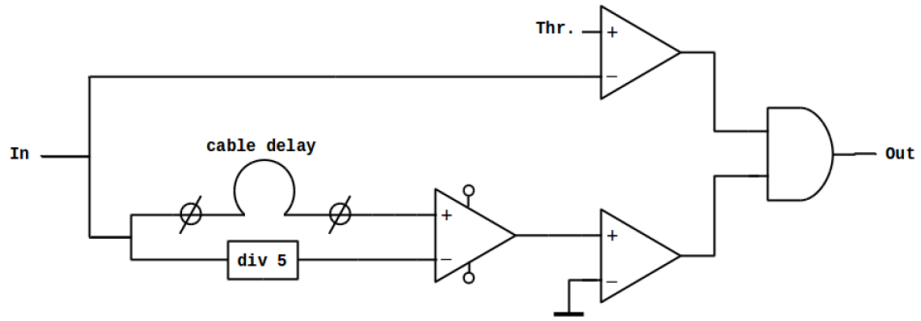


Figure 8: CFD function diagram.

The CFD functional diagram is shown in fig. 8. The input signal is split in two different discrimination branches, whose results are then merged by the final AND gate. The top branch is a standard threshold discriminator, where the input signal is compared against a (configurable) threshold Thr.

The bottom branch instead implements the constant fraction technique. Technically, the input signal is split: one copy is delayed, while the other is attenuated by a factor of 5. The two copies are then subtracted and the final result is compared with a threshold of (close to) zero. In fact, the zero-crossing time of the resulting signal in nearly independent from the input signal leading edge gradient (i.e. the source of time jitter in a standard threshold discriminator).

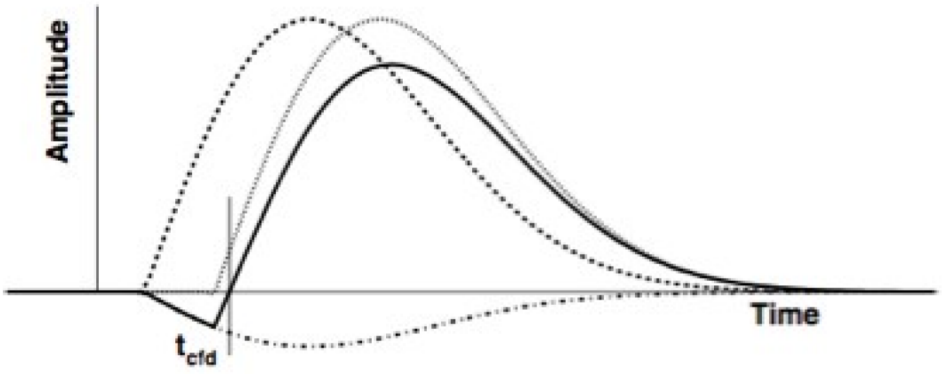


Figure 9: CFD function diagram.

Fig. 9 shows in detail the signals in the bottom branch of the CFD. The input pulse (dashed curve) is delayed (dotted) and added to an attenuated inverted pulse (dash-dot) yielding a bipolar pulse (solid curve). The output of the bottom branch fires when the bipolar pulse changes polarity which is indicated by time t_{cfd}. From a practical point of view, a small threshold, as close as possible, is actually used in the final comparator of

the bottom branch. This is needed to avoid fake signals possibly caused by noise. Such a small threshold is normally called walk (Z).

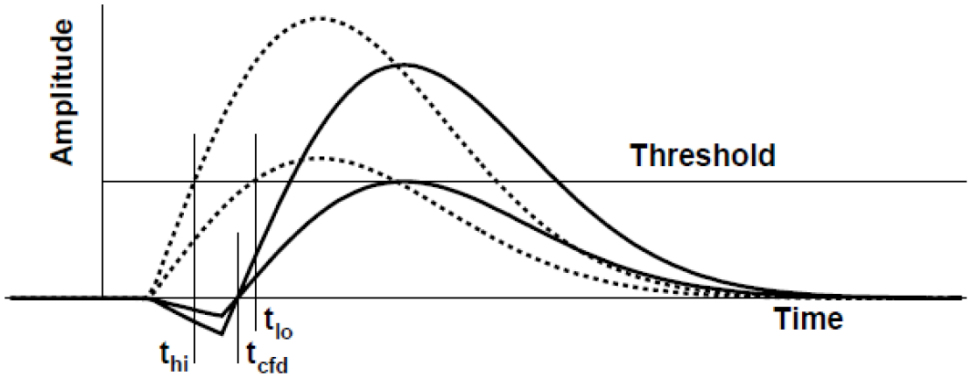


Figure 10: CFD function diagram.

In order to complete the CFD description, the merging of the top and bottom branch signals has to be considered, with the help of fig. 10. In the top branch, the threshold discriminator fires at time t_{hi}, that depends on pulse leading edge characteristics. The bottom branch instead fires at a time t_{cfd}, as discussed above, which is almost constant, due to the delay introduced in the bottom branch, normally $t_{cfd} > t_{hi}$. Therefore, the overall CFD, defined as the signal generated by the final AND gate, will fires at t_{cfd}, achieving both our requirements:

- only select signal above a given amplitude Thr;
- provide an output trigger whose timing is independent from input signal amplitude.

As can be seen in the above figure, the CFD operating principle is not retained for all the possible combinations of configured delay, threshold and input signal amplitude. As the top branch timing depends on the signal amplitude, a small enough signal can make it fire at a time $t_{lo} > t_{cfd}$. In this case the CFD will behave like a normal threshold discriminator, as the output AND gate will be driven by t_{lo}.

Lab 3: Detector and Trigger

Scintillators, trigger logic, input to readout modules (ADC & TDC)

Introduction

This exercise consists in building the trigger logic and the input signals to the VMEbus readout modules for a detector (exercise #4) using the experience with NIM electronics acquired in exercise #2. The detector comprises two scintillation counters detecting cosmic rays (muons). A schematic diagram of a scintillation counter is shown in Figure 1. When a charged particle traverses the scintillator, it excites the atoms of the scintillator material and causes light (photons) to be emitted.

Through a light guide the photons are transmitted directly or indirectly via multiple reflections to the surface of a photomultiplier (PM), the photocathode, where the photons are converted to electrons. The PM multiplies the electrons resulting in a current signal that is used as an input to an electronics system. The PM is shielded by an iron and mu metal tube against magnetic fields (of the Earth). The scintillator and light guide are wrapped in black tape to avoid interference with external light. The scintillation counter setup is shown in Figure 2.

The NIM modules used to build the trigger and the input to the readout system and provide the high voltage are shown in Figure 3.

Outline:

The aim of the exercise is to get an understanding of the detector and trigger logic used in Exercise 4. The signals from two scintillation counters are analyzed using an oscilloscope and transformed into logic NIM signals that allow to build a trigger based on a coincidence between the signals. The coincidence rate i.e. the rate of cosmic muons is counted using a scaler and the charge content of the scintillator signals is measured on the oscilloscope. In addition the inputs to the readout modules (QDC and TDC) are set up.

A schematic diagram of the full trigger and readout electronics is shown in Figure 4.

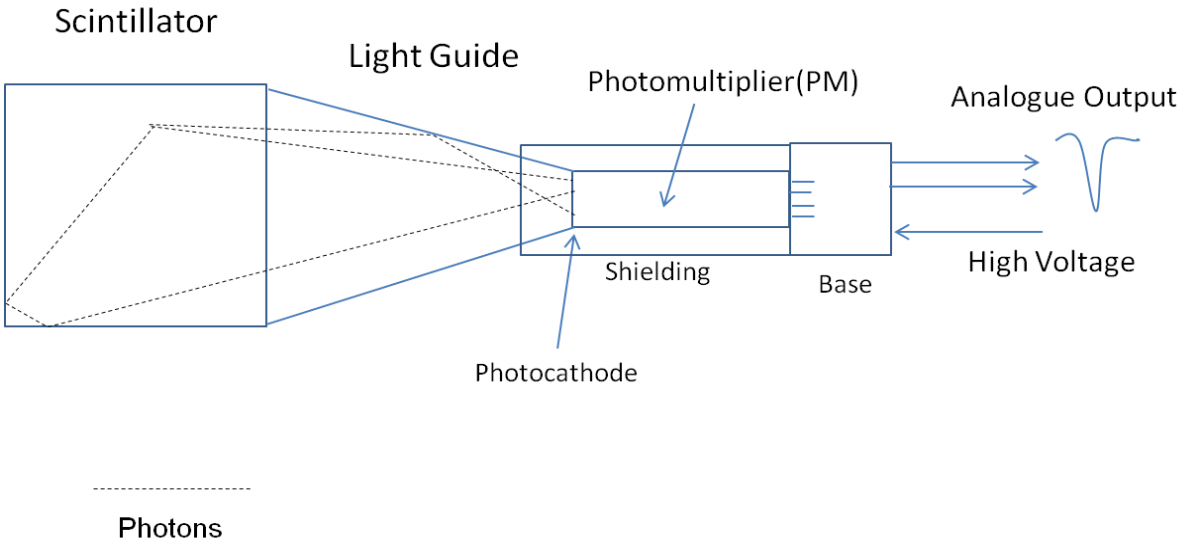


Figure 1. Schematic diagram of a scintillation counter.

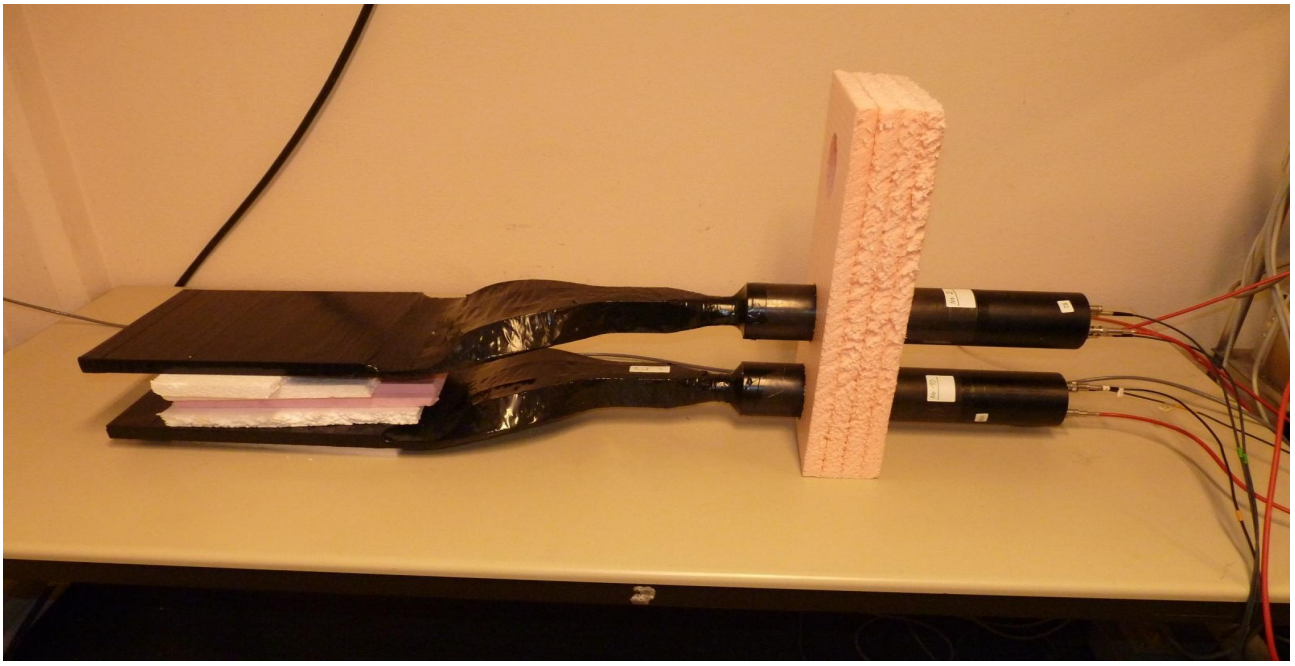


Figure 2. Scintillation counter setup

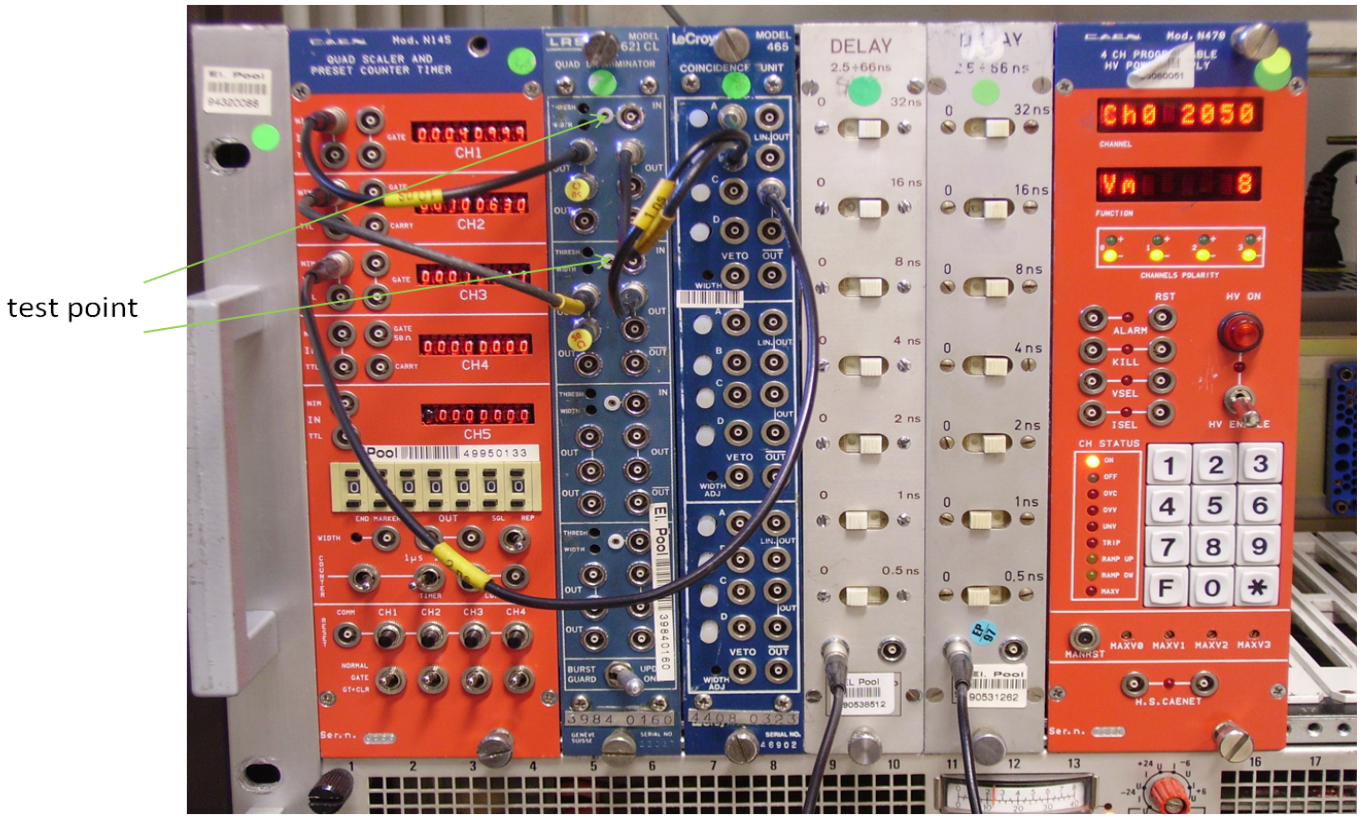


Figure 3. NIM trigger electronics. From left to right: scaler (counter), discriminator, coincidence unit, delay modules and high voltage power supply.

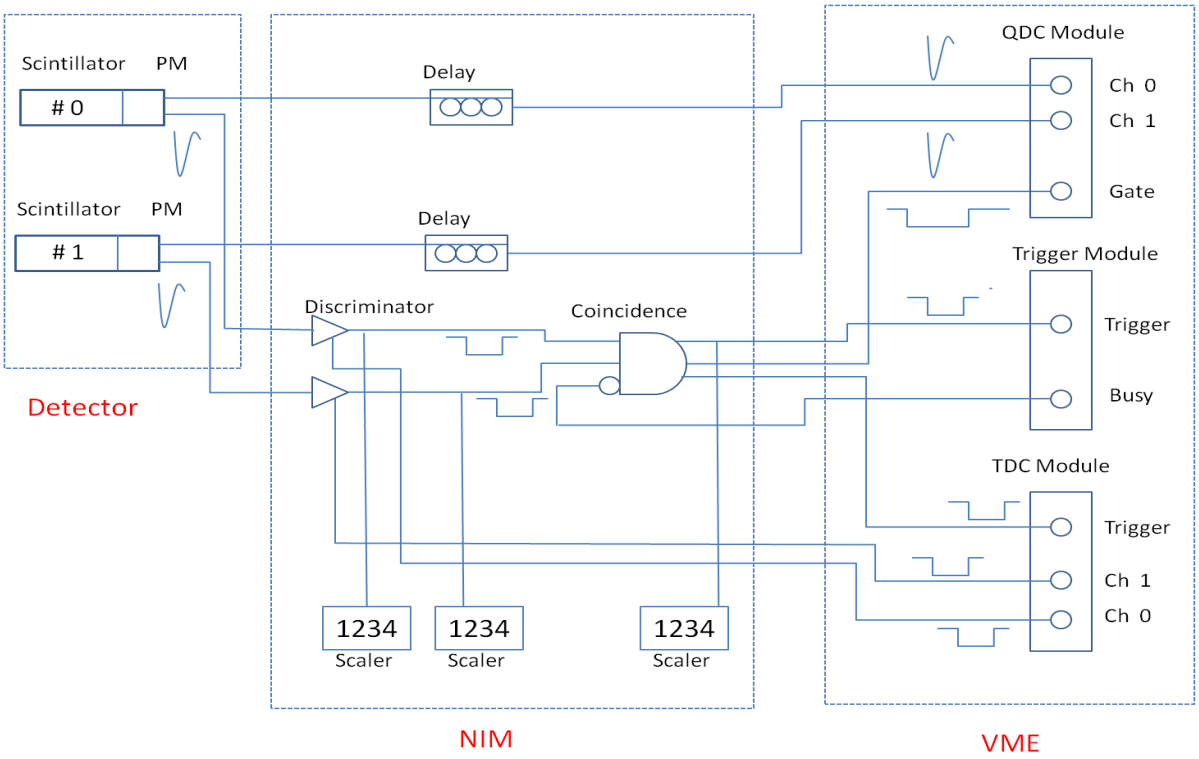


Figure 4. Diagram of the electronics for the detector, trigger and readout of the scintillator counter setup.

Work plan:

Note: whenever there are two parallel outputs from a (NIM) module one needs to make sure that they are both cabled, i.e. either terminated with 50 Ohm or connected to another unit. This ensures that the pulses have the correct NIM voltage levels: 0 and -0.8 Volts.

1. Install the scintillation counters close to each other with maximum overlap between the scintillator areas.
2. Check that the scintillator photomultiplier bases are connected to the N470 NIM high voltage supply.
3. Switch ON the NIM crate.
4. Connect an output from scintillator 0 (the upper one) to an oscilloscope (10ns LEMO), terminate the other output with 50 Ohm.
5. Set the nominal high voltage on scintillator 0 using channel 0 of the N470 HV supply. The voltage is marked on the label glued onto the base. Refer to 0 at the end of this exercise for a short guide to using the N470 HV supply.
6. Look at the signal on the oscilloscope (volts/div ~ 50 mV, time/div ~ 20ns). What is the maximum voltage of the signal?

7. Connect the cable to the input of the first channel of the discriminator.

Connect an output to the oscilloscope (0.5 Volts, 50 ns) and adjust the pulse width to around 100 ns using a small screwdriver (terminate the other output with 50 Ohm), see

8. Figure 3.
9. Connect the output to the first channel of the NIM scaler (N415) using a short LEMO cable (1ns).

Set the discriminator threshold to 50 mV: adjust the voltage on the test point using a DC voltmeter and a small screwdriver, see

10. Figure 3. The voltage is 10 times the threshold value i.e. the voltage should be around 0.5 Volts. This step may require teamwork.
11. What is the scaler rate?
12. Vary the threshold around 50 mV and check the variations in scaler rate.
13. Repeat points 4 to 11 above for scintillator #1 (the lower one), connecting this scintillator in addition to the one already connected.
14. **Given the scaler rates measured above, what is the probability of random (unphysical) coincidences between pulses from the two scintillators?**
15. Connect an output from each of the two discriminator channels to the oscilloscope and check that they have a timing overlap i.e. are coincident.
16. Connect the cables from the discriminators to the first inputs of the coincidence unit (LeCroy 465) using short LEMO cables (1ns).
17. Connect an output from the coincidence unit to a scaler input. What is the rate? Given that the rate of cosmic muons is about 100 per second per square meter, does the rate make sense?
18. Connect an output of the coincidence unit to channel 1 of the oscilloscope.
19. Connect the (other) analogue output from scintillator 0 to a delay unit (LEMO 10ns) and the output of the delay unit to channel 2 of the oscilloscope.
20. Using channel 1 as a trigger, observe the analogue signal on channel 2. Channel 2 will then show the scintillator signals for the cosmic muons. Assuming that the signal is triangular, what is the charge of the signal? See Figure 5. **Note down the charge. You will need it again in exercise 4**
21. Adjust the delay unit such that the analogue signal falls within the NIM pulse from the coincidence unit: inputs to the charge to digital converter (QDC) in Exercise 4 are now ready (analogue signal and gate).
22. Repeat point 21 for scintillator 1.
23. Connect a cable from the first discriminator to channel 2 of the oscilloscope and check the timing with respect to the output from the coincidence (channel 1). The signal from the discriminator should precede the coincidence. Similarly for the second discriminator. The inputs to the time to digital converter (TDC) in Exercise 4 are now prepared (trigger and

timing signals).

24. The signals from the discriminators are sometimes about twice as long as expected. What could the reason be?

Appendix 1: Short User's Guide to the CAEN N470 High Voltage Supply

This is a short list of the most common operations for the N470 High Voltage Supply used in Exercises 3 and 4. The manual can be found at
http://www.caen.it/nuclear/product.php?mod=N470#

- To select a channel: F0*(channel number)* e.g. F0*0*
- To set the High Voltage on the selected channel: F1*(type value)* e.g. F1*2000*
- To read the voltage on the selected channel: F6*
- To read the current on the selected channel: F7*
- To turn the selected channel ON: F10*

Notes:

The maximum voltage on the channels has been set to around 2300 Volts (on the potentiometers). These can be checked via F13*. The current limits have been set to 2mA (via F2*).

Appendix 2 : Charge of scintillation counter current pulse

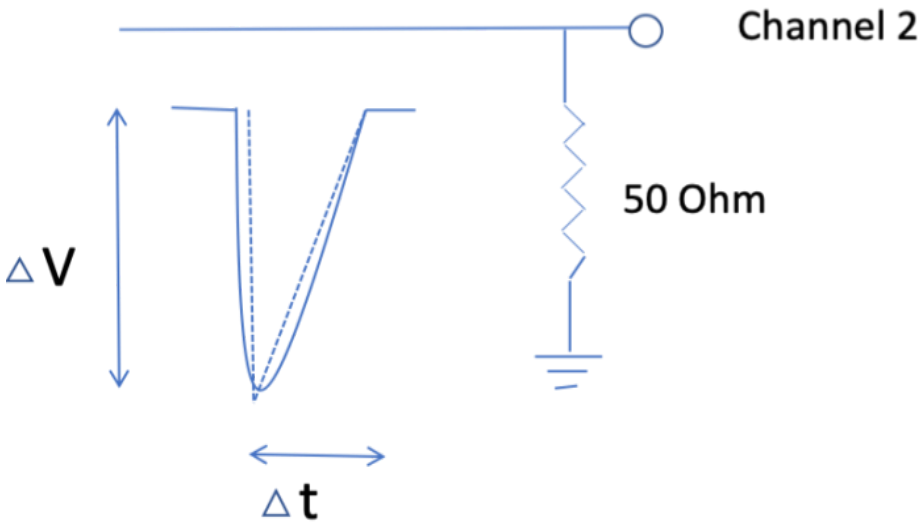


Figure 5. Input to the oscilloscope from a scintillation

Lab 4: A small physics experiment

Detector, trigger and data acquisition

Introduction

This exercise comprises all the components of a typical experiment in high energy physics: beam, detector, trigger and data acquisition. The "beam" is provided by cosmic rays (muons) and the detector consists of a pair of scintillation counters, see Figure 2 in Lab 3. The trigger logic, built from NIM electronics, forms a coincidence between the signals from the scintillation counters which indicates that a muon has traversed the detector, see Figure 3 in Lab 3. A data acquisition system based on VMEbus is used to record the pulse heights from the scintillation counters and measure the time of flight of the muon. The VMEbus crate is shown in Figure 6 and the VMEbus modules shortly described in Appendix 1, Appendix 2 and Appendix 3. The overall run control and monitoring is provided via software running on a (Linux) single board computer (SBC).

Outline

This exercise is a continuation of Lab 3. First, standalone programs are executed to give an understanding of the QDC and TDC VMEbus modules. A full DAQ system is then run on a multi-processor configuration, with the readout, run control, GUI and infrastructure on a VMEbus SBC. Event rates and dumps are examined. An event monitoring program produces histograms of the QDC and TDC channel data which allow to compute the charges of the input signals to the QDC and the speed of the cosmic muons.

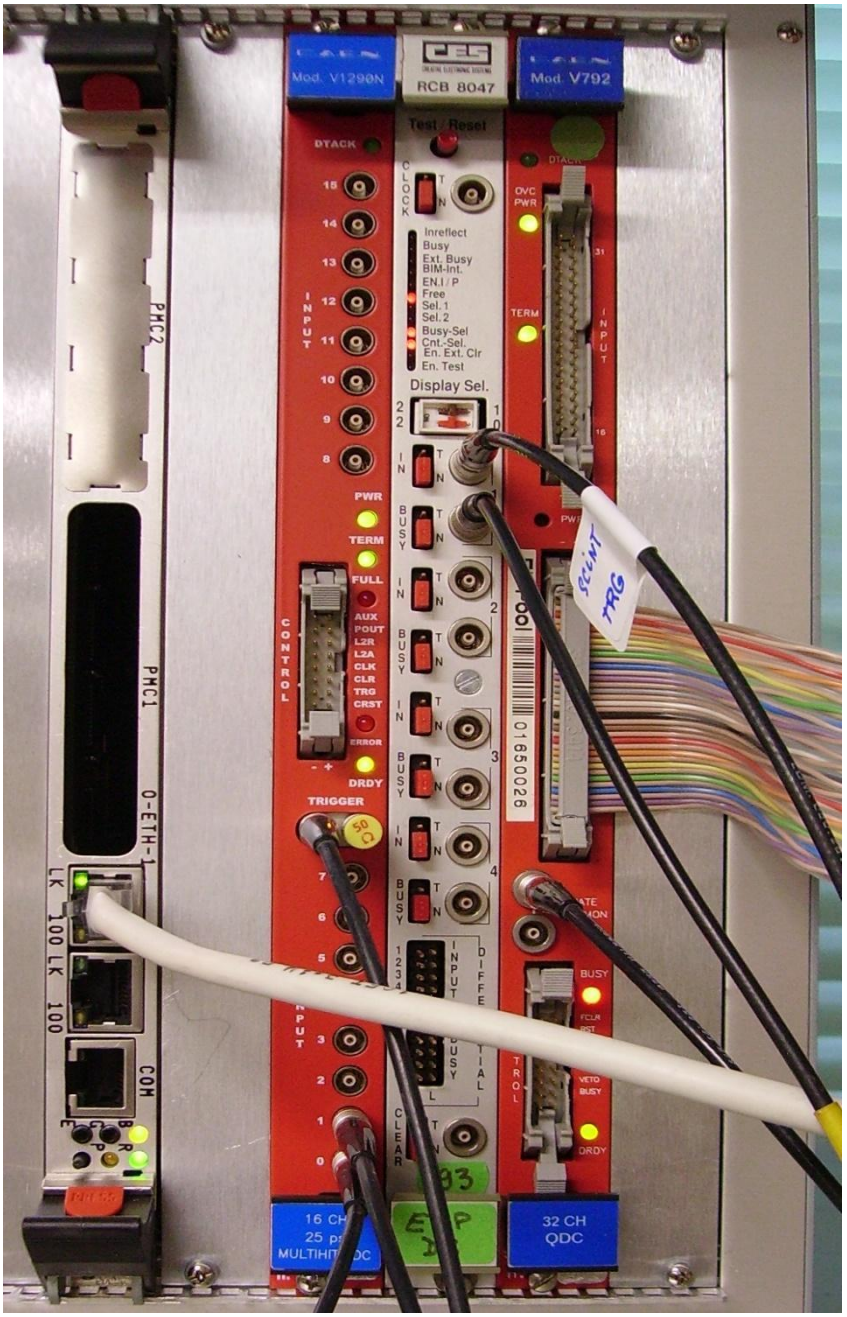


Figure 6. VMEbus data acquisition system: SBC (Single Board Computer), TDC (Time to Digital Converter, Trigger Module (CORBO), QDC (Charge to Digital Converter)

Work plan

- Verify that the detector is working i.e. the scaler counts for scintillator 0, scintillator 1 and the coincidence are counting such that the TDC and QDC receive signals (note for the tutor: if the coincidences are not counting, remove the CORBO busy from the trigger coincidence by pushing the button).
- Login to the SBC as user `daqschool`, password `g0ldenhorn`
- Start a Terminal window
- Go to TDAQ directory: **`cd ~/TDAQ`** and run the command **`source ./setup_RCDTDAQ.sh`** to define the environment
- Run the program **`v1290scope`** which is a low-level test and debug program for the CAEN V1290 TDC
    1. Run command: **`v1290scope`** (Use defaults for the command parameters)
    2. Use VMEbus base address = `0x4000000`
    3. Dump and decode the registers (option **2**). Is data ready? (bit `DREADY` in the status register). What are the values of the match window width and the window offset? See Appendix 1.
    4. Configure the TDC (option **3**)
    5. Read an event (option **5**). How many words are read? (Check in the global trailer). What are the values of the TDC measurements (in ns). Do they make sense? See Appendix 1.
    6. Exit from the program by choosing menu option **0**.

1. Run the program **`v792scope`** which is a low-level test and debug program for the CAEN V792 QDC
    1. Run command: **`v792scope`** (Use defaults for the command parameters)
    2. Use VMEbus base address = `0x0`
    3. Dump and decode the registers (option **2**). Is data ready? Check also the LED on the module.
    4. Read an event (option **5**). How many words are read? Which channels have data and which are pedestal (empty) values?
    5. Exit from the program by choosing menu option **0**.
- We now run the full DAQ system
    1. Start the DAQ system: **`./setup_RCDTDAQ.sh start`**. This script will read the configuration database and start a number of processes on the server: run control, GUI and a number of infrastructure SW components. This is a somewhat long procedure and should result in a message 'OK!'.
    2. Now start a GUI display: **`./start_Igui.sh`**. The "folders" in the infrastructure panel should be green! You may need help from the tutor here ...
    3. We now go through the run states in order to start a run. But first please obtain a 'Control' access by selecting the 'Control' radio button in the top menu 'Access Control'. The initialize button should become active. Now, click on INITIALIZE and then wait for the RCDApp (in RCDSegment) to reach the INITIAL state. The readout application is now loaded on the VMEbus processor.

4. Click the CONFIG button followed by OK on the "Remember to ..." dialog box. This configures the VMEbus modules, the CORBO, QDC and TDC.
5. If you don't see the DFPanel tab close to the top of the GUI, click LOAD Panels and load the first panel: DFPanel should now appear in the bar above the Run Control panel.
6. Click START in the control panel (on the left)
7. Data taking should now start. Click on the DFPanel and the L1 check-button to display the event rate. Is it what you would expect after Lab 3? Check also the LEDs on the VMEbus modules (the event rate is computed by the Information Service (IS) which periodically sends a command to the Readout Application to obtain the rate which is then retrieved by the GUI).

- Event Monitoring
  This part demonstrates event monitoring. An event monitoring program obtains a sample of events from the readout application and analyses them, in this example by producing histograms of the values from the QDC channels as well as the time difference between the two TDC values. The histograms can then be viewed via the GUI. The code for the monitoring program can be found in `~/RCDTDAQ/RCDMonitor/`
  1. Open another terminal window.
  2. Run **`cd ~/TDAQ`**
  3. Run **`source ./setup_RCDTDAQ.sh`** to define the environment.
  4. Run the event monitoring task: **`./event_dump.sh -e -1`**
     (`-1` means to run forever. I you want only one event, please change it to `1`.). Once you have seen the raw data output of the even_dump you can terminate this application with `Ctrl+C`.
  5. The first nine words of the data constitute an Event (ROD) header. The following words are the data from the QDC and the TDC. Do you recognize the data?
  6. On the terminal start the monitoring program by executing **`monitor`**. This program monitors data, like the event_dump program, publishing measurements to the histogramming service.
  7. In the GUI click on the `OH` button (Online Histogram). Click on Histogram Repository, partRCDTDAQ, RCDMonitor. Double click on the histograms to view them.
     i. Alternatively, using a new terminal execute **`source ./setup_RCDTDAQ.sh`** followed by **`./start_ohp.sh`**. This is the online histogramming presenter. In the panel Histograms (on the left) select SCMonitor to view TDC histograms or RCDMonitor to view also the QDC histograms that we are producing.
  8. Record the mean values of the QDC histograms and the mean value of the time difference histogram. The time histogram is not centered around zero. Why?
  9. The charge that you find in the histogram is not the charge delivered from the PMT to the QDC. What is the reason for that and how can we measure the proper charge?

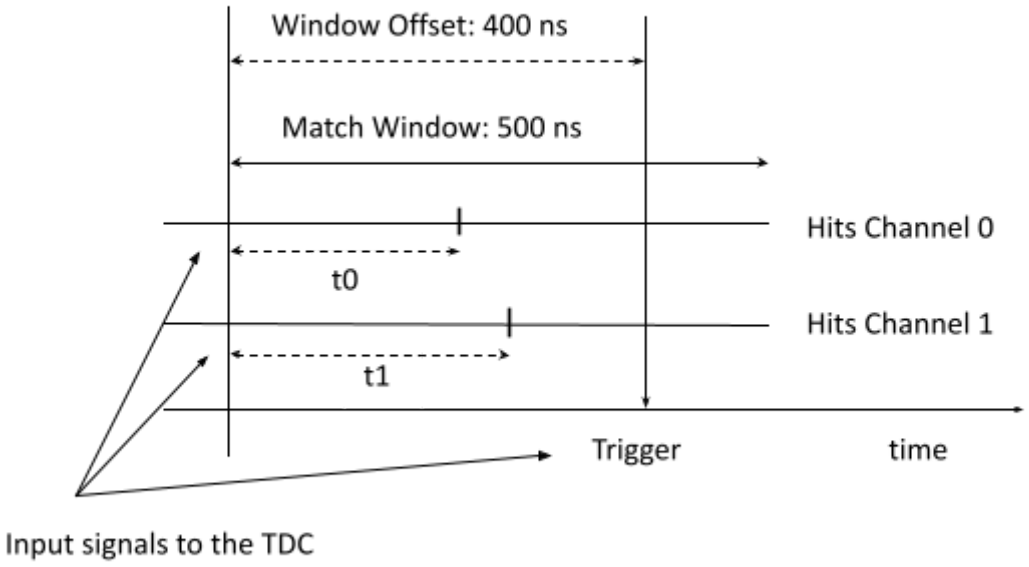
10. The monitoring of the statistics can be reset by stopping and starting the monitoring program (`Ctrl+C` to terminate). This restarts the monitoring program described in point 6.
11. Display the histograms of the QDC channels. Record the pedestal values.
12. Using the formula shown in Appendix 2, compute the mean charges of the signals from the scintillators. Do they agree with the results obtained in Lab 3?

- We now want to measure the time of flight of the muons between the two scintillators.
    1. In the histogram for the data from the TDC we already get a Δt. This value, however, is not the time of flight of the muon. Why? How can we modify the set-up in such a way that we can correct the Δt for systematic errors and measure the actual time of flight?
    2. Restart the monitor program from the IGUI per point 10 above. Record the new mean value of the Δt histogram.
    3. What is the difference with respect to the value measured before? Compute the speed of the cosmic muons.

Appendix 1: TDC CAEN V1290 VMEbus module

The TDC is operated in *trigger matching* mode. This means that the TDC measures the time of arrival of the hits on a channel within a *match window*. The TDC receives a trigger and the channel signals as shown in the diagram of the complete setup, Figure 4 of Lab 3 and seen in the picture of the VMEbus crate, Figure 6. A trigger match window is then defined by a window offset with respect to the trigger and a match window size as shown in the figure below. The hits occurring on channel 0 and channel 1 within the match window are recorded by the TDC and the values in units of 25ps are stored in the memory of the module.

The module is shown in the photo of the VMEbus crate and the manual for the module can be found at https://www.caen.it/products/v1290n-2esst/ (registration required)

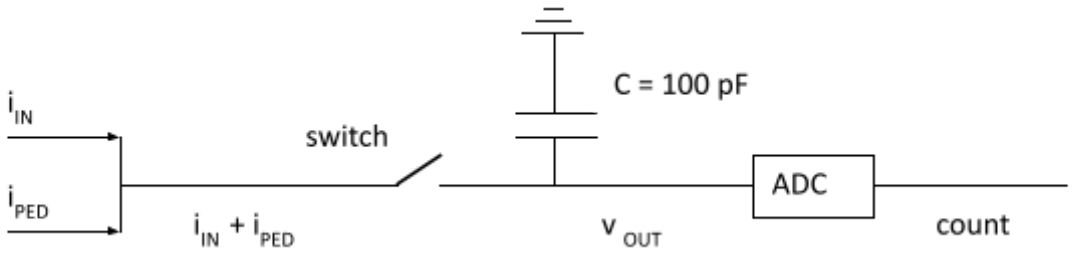


Appendix 2: QDC CAEN V792 VMEbus module

This page explains briefly how to calculate the charge of the input signal to the QDC from the data readout from the module over VMEbus. The module is shown in Figure 6.

The manual for the module can be found at https://www.caen.it/products/v792/ (registration required)

The circuitry of a channel is shown below, schematically.



The switch is closed as long as the gate input signal is present. The input current is the sum of i_{IN}, the current input to the module via the front panel (from the scintillator), and i_{PED}, a bias (or pedestal) current which is generated internally. The bias current allows to handle input signals with small positive voltage components. When the switch is closed during the time of the gate signal, the input current charges the capacitor C. When the switch is opened again, the voltage across C, v_{OUT}, is converted by an ADC and stored in the memory of the module. The ADC has the property that **one count = 1 mV**.

We now have for the charge of the capacitor:

$Q = C * v_{OUT} = 100\ (pF) * count\ (mV) = 0.1 * count\ (pC)$

To compute the charge in the signal input to the channel, corresponding to i_{IN}, we have to correct for the pedestal value:

$Q_{IN} = 0.1 * (count - count_{PED})\ (pC)$

count = channel data with input signal present

$count_{PED}$ = channel data with input signal removed ($i_{IN} = 0$)

Appendix 3: CES RCB 8047 CORBO VMEbus trigger module

When a NIM signal is sent to a channel on the CORBO, a bit is set in a status register and an interrupt on VMEbus is generated, optionally.

The DAQ process on the VMEbus processor can then execute the code to readout the data from the QDC and TDC modules. In addition, the CORBO generates a busy signal which allows to block further triggers until the readout code is terminated.

The CORBO module is shown in Figure 6.

Lab 5: FPGA programming

(Ver2014_v01)

INTRODUCTION

In a lot of digital designs (DAQ, Trigger, …) the FPGAs are used. The aim of this exercise is to show you a way to logic design in a FPGA. You will learn all the steps from the idea to the test of the design.

In this exercise you will:

- discover how we can do parallel applications
- program a FPGA from the design up to the implementation and the test

The boards used are ALTERA development kit (Figure 1) based on a small FPGA (CYCLONE) with multiple additional interface components like audio CODEC, switches, button, seven-segments display, LEDs, ….

and a home-made board (named detector in the following pages) connected to the development kit with a flat cable (figure 2)

The initial design is loaded into the board.

You will follow the example to understand the design flow. Four exercises are proposed to modify the original design functionality.

Figure 1: development kit

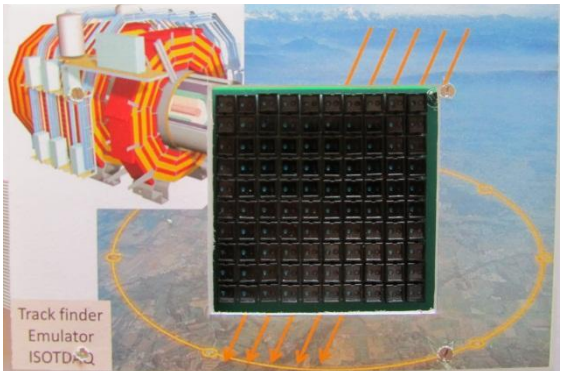


Figure 2: detector

QUICK START

1) Programs used are: QUARTUS (FPGA tool), ModelSim (simulator), LabView

2) Ask the tutor if you have question(s) or problem(s)

EXERCISE (example)

When you switch on the kit, the initial design is loaded into the FPGA.

On the LabView window, you can see the progression of the marker on the detector.

At the same time, you can see on the two 7-segments LED (the right ones on ALTERA kit) the column and the line number over which the maker is positioned.

DESIGN ENTRY

The design file is named "CII_Starter_Default.bdf" (for all exercises you should work with the same design file).

The design is divided in three parts:

a) A green rectangle which is used to transmit the information to the computer via the RS232 connection to display the trace on LabView.
b) A blue rectangle in which the design generates the clock and the logic to control the detector (see Appendix A for detailed functionality).
c) A red rectangle, which contains the logic to detect the trace. You will change the logic in this rectangle in the following exercises.

The idea of all exercises is to detect a trace. As soon as the trace is detected one 7-segment LED blinks (the third for the right side).

Click on key0 (Altera kit) to stop the blinking. Now generate another trace.

Spend some time to understand how this design works.

Do you understand it?

COMPILATION

This design is the entry of your logic, it should be compiled now; go to QUARTUS *Processing->Start Compilation*.

The design is compiled for the chosen component (Cyclone II).

The compiler executes multiple tasks:

✓ logic optimization
✓ generates a binary file used to program the FPGA (memory array),
✓ extracts the timing between each logic elements used for the timing analyses

✓ generate an output VHDL file used for the simulation

SIMULATION

When the compilation is finished, you can check the design with a simulator. To do this you will use ModelSim.

Check in the "Project" TAB if there is a file marked with a bleu "?", if YES, compile it (right-clic on it, Compile-> compile selected)

In the "Transcript" tab, type 'source sim.tcl', ENTER. The simulator opens the waveform, loads the signals, and starts the simulation.

At the end, stimuli and results are displayed in the wave window.

This simulation emulates a trace starting from the top left and finishing at bottom right describing a straight line on the detector.

(The tutor will give you some explanations on the results and the signals shown in the waveform)

Remember where the signal OK goes to "TRUE".

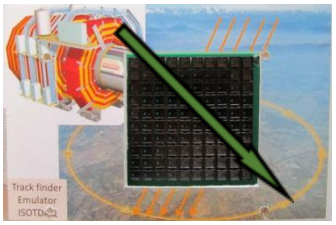


Figure 3: straight line

When you have finished with the simulator type 'quit –sim ' ENTER in the "Transcript" tab.

PROGRAM THE KIT

To download the design on the board, (QUARTUS program) go to on *Tools->Programmer* (Check that the Hardware is USB-Blaster, if not ask the tutor).

One file is shown in the window: it is your design. Click on Start .The programmer takes few seconds. At the end, a message appears to inform you that the programming is completed (or not successful: in this case usually the board is switched OFF, or the cable is not well connected).

TEST

Draw a straight line from top left to bottom right to see if the design works well!

Now, you are ready to do the other exercises by yourself.

Good Luck!

EXERCISE I

The exercise above uses the graphic to describe the design. In this exercise, we want to do the same with a text design entry (VHDL).

In the QUARTUS design entry (file "CII_Starter_Default.bdf"), delete the line between inst_graph and JKFF inst_result and connect the output 'result' of "track1"box to the JKFF inst_result with a line.

- Compile the design
- Simulate the design
  - Go to ModelSim:
    - ✓ Compile the file marked with a ? in the "Project" tab (select the file to be compiled – Menu Compile-> Compile selected)
    - ✓ Type "quit –sim" in the "Transcript" tab.
    - ✓ Type "source sim.tcl " in the "Transcript" tab.
  - Find out the difference with the previous result (check where the signal OK goes to "TRUE").
  - Can you explain the difference? Can you modify the file "track1.vhd" to have the same result as in the previous exercise?
- Download the design
- Test the design

EXERCISE II

In this exercise we want to detect a curved trace.

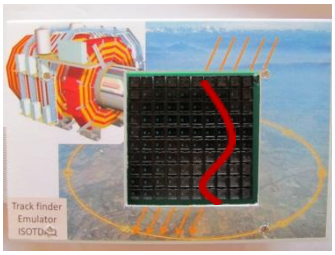


Figure 4: MISSING

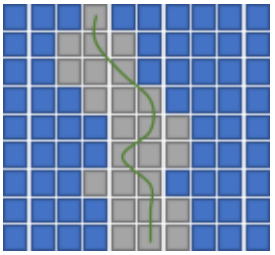


Figure 5: example of trace expected.

In the QUARTUS design entry (file "CII_Starter_Default.bdf"), delete the line between output 'result' of "track1" box to the JKFF inst_result, and connect the output of the "trck_fnd01" box to JKFF inst_result.

The "trck_fnd01" box logic detects only a straight trace. Compile the design and do a simulation:

- Compile the design (QUARTUS)
- Simulate the design

- Go to ModelSim, compile the file marked with a ? in the "Project" tab (click on the file to compile – Menu Compile-> Compile selected)
- To simulate:
  - ✓ Type "quit –sim" ENTER in "Transcript" tab to exist any running simulation.
  - ✓ Type "source sim2.tcl" ENTER in "Transcript" tab to start the simulator.
- A signal OK becomes true if the logic detects the expected trace (here a straight trace).

In this exercise, you will examine the implementation of the design in the FPGA and see how we can change the results (max. frequency …)

1. In QUARTUS open TimeQuest (Tools -> TimeQuest timing Analyser)
   - double click on Report Fmax Summary ("Tasks" window)
   - You can see the maximum frequency of each clocks implemented in the design (Note the max frequency that "scan_clk" can reach)
2. Go back to QUARTUS,
   - Open the partition window (Assignments -> Design partitions window)
   - Right-click on the partition named "trck_fnd01:instzigzag" (Locate-> Locate in Chip Planner)

Now, you will specify the place where your logic will be implemented:

There is a blue rectangle in the Chip planner (named "trck_fnd01:instzigzag").

Place it where you want (not at the place where the logic is actually implemented) to implement the logic at the next compilation.

Compile the design (Quartus), and execute the TimeQuest (see point 1). Normally the maximum frequency will change.

This give you an idea of the importance of the place of you logic or how to reserve a place if you work in a team (each person will have a reserved place to implement his logic).

NB: For your information, for each clock of the design, the frequency to reach **has to be specified** in a **constraint file.**

EXERCISE III

The exercise consists to modify the "trck_fnd01" box logic to detect any curve trace as in figure 4.

The trace should start at any pixel in the first line and goes to next line going to a pixel adjacent to the pixel of the first line and so forth (figure 5).

To help you, you have to change code in the “mask_build” entity (beginning of the “trck_fnd01.vhd”).

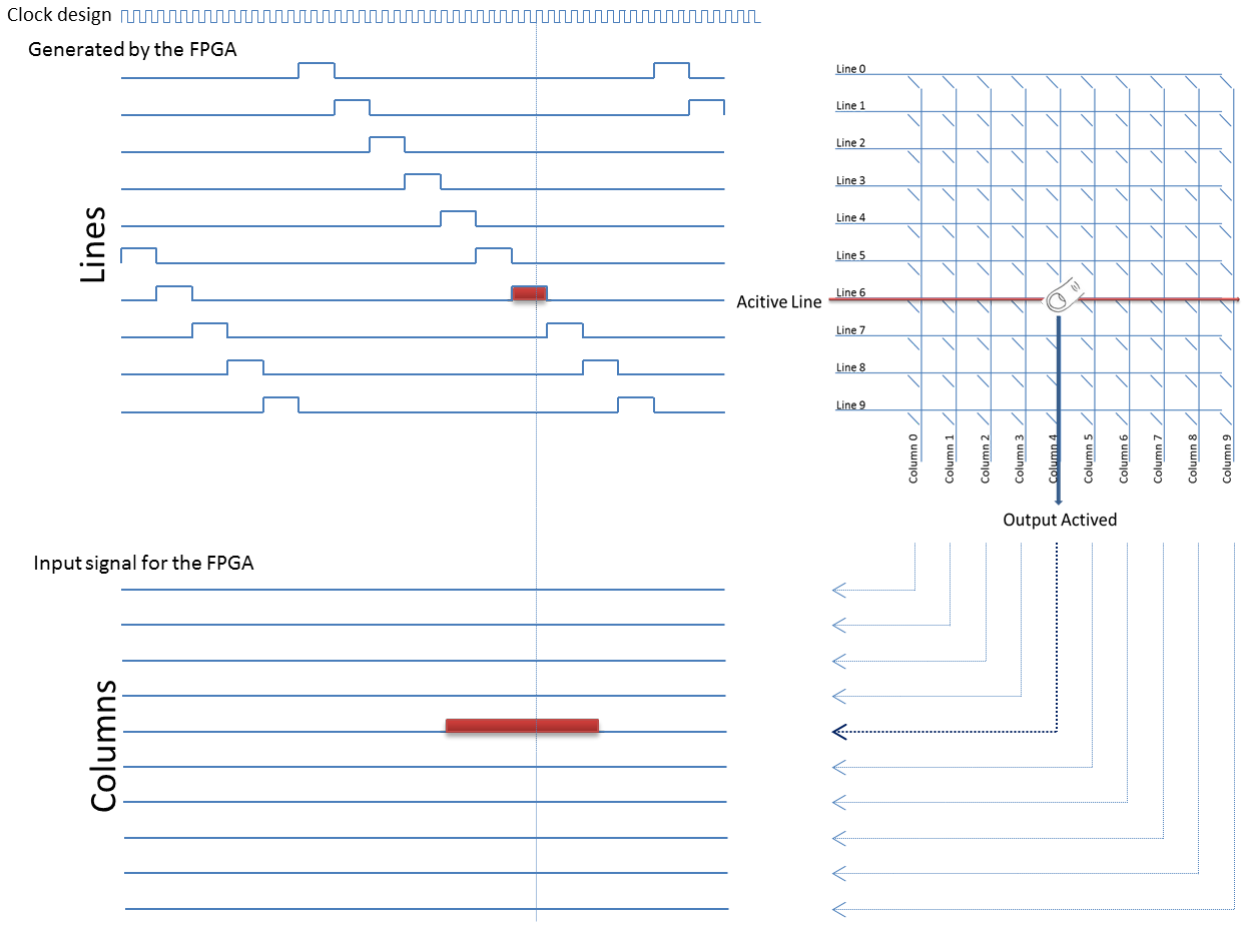
- Compile the design
- Simulate the design
    - Go to ModelSim, compile the file marked with a ? in the “Project” tab (click on the file to compile – Menu Compile-> Compile selected)
    - To simulate:
        - ✓ type “quit –sim’ ENTER in “Transcript” tab to exist any running simulation.
        - ✓ type ‘source sim2.tcl’ ENTER in “Transcript” tab to simulate in this exercise.
    - A signal OK becomes true if the logic detects the expected trace.
- Download the design
- Test the design

EXERCISE IV

If you have time, you can modify the previous file to detect only the curve trace on right or left (not in zigzag like the red trace in figure 4).

APPENDIX A

The detector is a matrix of 10 lines and 10 columns (100 pixels). Only one line is activated at a time.



When a line is activated the result of each column indicates if the marker is over a pixel. Each line is activated one after the other (0, 1, 2… 8, 9, 0, 1, …) . Each line is activated during 4 clock cycles. The detection logic checks the result (if pixel is masked by the marker) only during the third clock cycle (signal “check” in the design).

Lab 6: Micro TCA

Overview

In this exercise you will …

- explore the Micro-TCA technology
- learn about the PCI (express) bus
- write data acquisition software to sample music and display the wave forms

Introduction

In this exercise you will work with modular electronics based on the Micro-TCA standard. TCA stands for Telecommunications Computing Architecture, an architecture that is used in Telco industry to provide high-bandwidth, high-availability solutions. Both Micro-TCA and its bigger brother Advanced TCA (ATCA) are being used in the upgrades of the LHC experiments. Like VME or Compact PCI, Micro-TCA defines a rack—called *shelf* in TCA speak—and boards, called Advanced Mezzanine Cards or *AMCs*. Typical shelves have space for 12 AMCs but we will work with a slightly smaller version. A *Micro-TCA Carrier Hub (MCH)* performs management functions, such as monitoring temperatures and regulating fan speed to provide the necessary cooling. A Micro-TCA shelf may contain a second MCH for redundancy (but we will work with only one). The *backplane* contains high-speed serial links that are suitable for transferring data at rates of 10 Gb/s or more using various protocols. Typically the backplanes have a single or dual-star layout with all high speed-links going from each AMC to the MCH slot(s). The MCH then contains a switch for the desired protocol—in our case PCI Express (PCIe). Other backplane layouts exist with high-bandwidth links between neighboring AMC slots.

The test setup

We are using a small ELMA Micro-TCA shelf containing:

- a built-in power module and a built-in fan
- a backplane with star and mesh connections
- an MCH by Samway (IP 137.138.63.22)
- an AMC containing a Processor running Linux (IP 137.138.63.15)
- an I/O AMC (AMC-ADIO24) providing digital and analog IO
- optionally, an AMC that can generate a programmable load on the crate

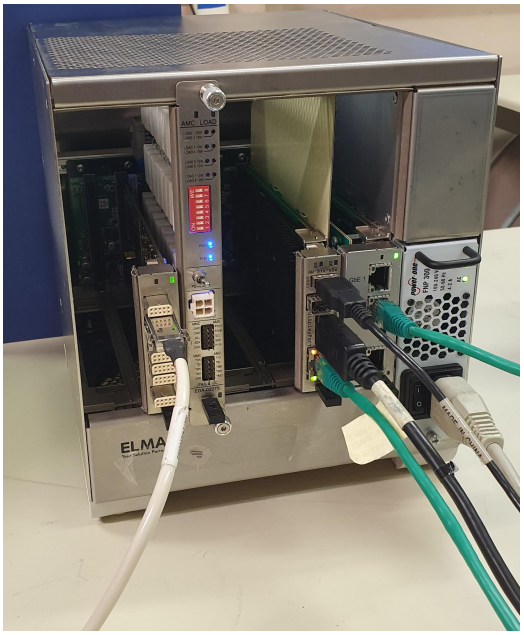


Figure 1. The exercise setup

We will be working directly on the processor AMC. Keyboard, mouse and screen are directly connected to this card which runs a standard Scientific Linux CERN (SLC) distribution. We will use the network port of this card to communicate with management port of the MCH.

Get to know the setup

Log into the processor AMC:

User: student (ask your tutor for the pwd)

Explore the system using the Webserver

- open Firefox
- Connect to the Samway MCH webserver at IP 137.138.63.22

You can use the webserver to browse the different Field Replaceable Units (FRUs) in the system. You can get information about the units, their voltages and temperatures as well as the valid operating ranges for all these quantities.

Turn on all load groups of the load AMC as shown in the next section. See how the MCH reacts.

Connect to the MCH by telnet

You can see details of the processes in the MCH by connecting to it with telnet (*telnet 137.138.63.22*). Login: user

Try it. (use command *help* to display help).

Note that the backspace may not work. In this case you can use ctrl+h.

To see the fan-speed and fan-levels:

sensor cu 1
cu

As an alternative way to looking at the webserver, you can check the operating ranges and current readings of all sensors via the telnet connection:

sensor amc <slot>

IPMI

The web server communicates with the MCH through IPMI (Intelligent Platform Management Interface) commands. The MCH either answers to the IPMI commands itself or it forwards the request to a FRU using a dedicated I2C (Inter-Integrated Circuit, often pronounced I-squared-C) link.

You can also directly use the IPMI protocol to talk to a card in the sytem. For example, we can program the load of the load board (produced at CERN) through IPMI. For this we use the program *ipmitool* with the following syntax:

ipmitool -I lan -H <ip_address> -U admin -P ADMIN -T 0x82 -t <AMC_address>
-b 7 -B 0 raw 44 7 0 0 <group_number> <action> 0 15

Where:

<ip_address> is your MCH address

<AMC_address> is the target AMC address (Slot1 = 0x72, Slot2 = 0x74, Slot 3= 0x76 ....)

<group_number> is the LED/Load group number from 4 to 11

<action> if 0xff group ON, if 0x00 group OFF

- Try switching on all the load-groups of the AMC
- See the reaction on the temperature of the AMC by repeatedly running the sensor command (see above)
- When a non-critical threshold is reached the MCH should increase the fan-speed of the crate
- Check this by running the cu command
- Now you should turn off the load groups quickly to avoid that the system overheats (the fan is not powerful enough in this crate)

Explore the Backplane

The telnet prompt has commands that allow to display the links of all FRUs.

- *bpppc* # backplane connectivity
- *pcie* # PCIe status

Try them. The backplane connectivity information needs some decoding. There is some information given at the top of the output. Some further information may be found in the Samway MCH user manual (ask your tutor for a printed copy or find it in /ISOTDAQ/doc/).

AMCs usually have 21 ports, MCHs up to 84 ports. An MCH connects to multiple connectors and is composed of a number of printed circuit boards stacked on top of each other. The boards are called the *tongues* of an MCH. Figures 2 and 3 show the block-diagrammes of the two tongues of the Samway MCH. Figure 4 shows another block-diagram of a MCH with more functionality compared to the one used in the test setup.

Ports are grouped into fabrics. MCHs usually provide *switches* for a certain fabric.

In our test setup, the MCH contains

- a Gigabit Ethernet Switch on Fabric A going to ports 0 and 1 of each AMC.
- a PCI-express Gen3 Switch on fabrics D-G supporting up to 4 lanes, going to ports 4-7 of each AMC.

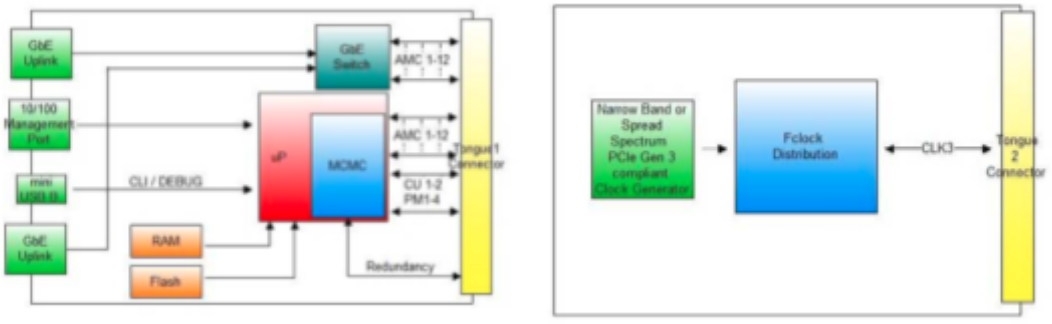


Figure 2. Block diagram of the Samway MCH

Figure 4 shows the backplane of our test shelf, figure 5 shows a typical backplane of a larger shelf with 12 AMCs and redundant MCHs.

Take a look at the backplane with the bpppc command, which AMC has which connectivity? You can also have a look with the pcie.

For some MCH dedicated tools are available to illustrate the backplane connectivity, figure 6 shows one such example, the NATView Backplane viewer.

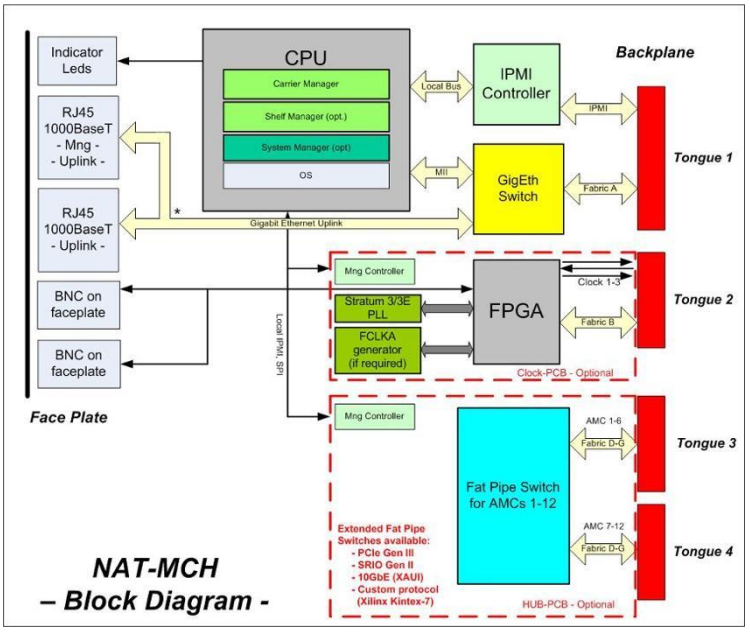


Figure 3. Block Diagram of the NAT MCH.

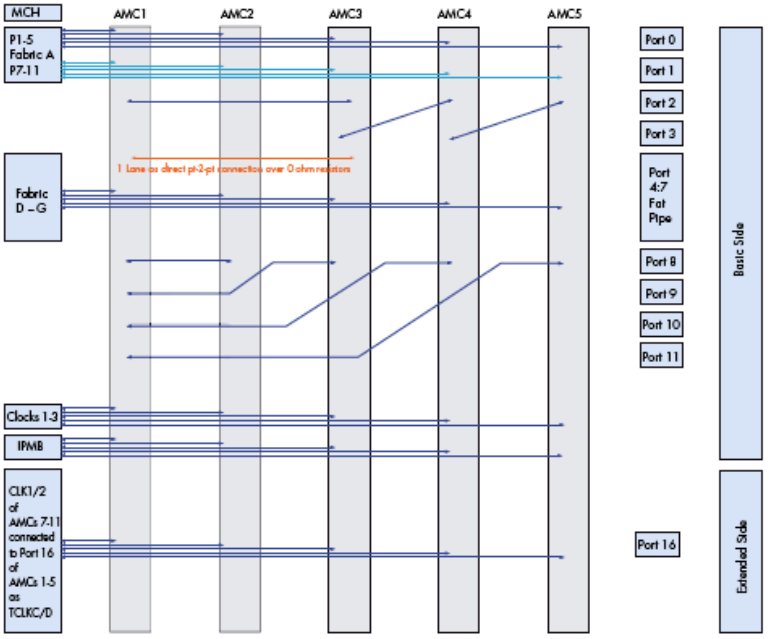


Figure 4. Backplane of the ELMA blue eco shelf used in the exercise.

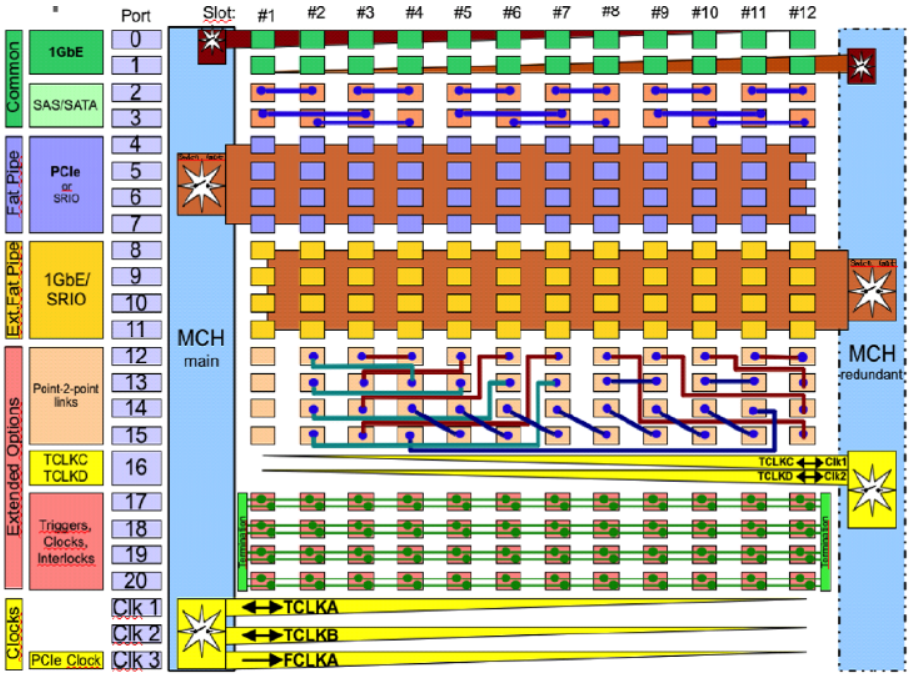


Figure 5. Backplane of a typical larger Micro-TCA crate with 12 AMCs. (not used in the exercise)

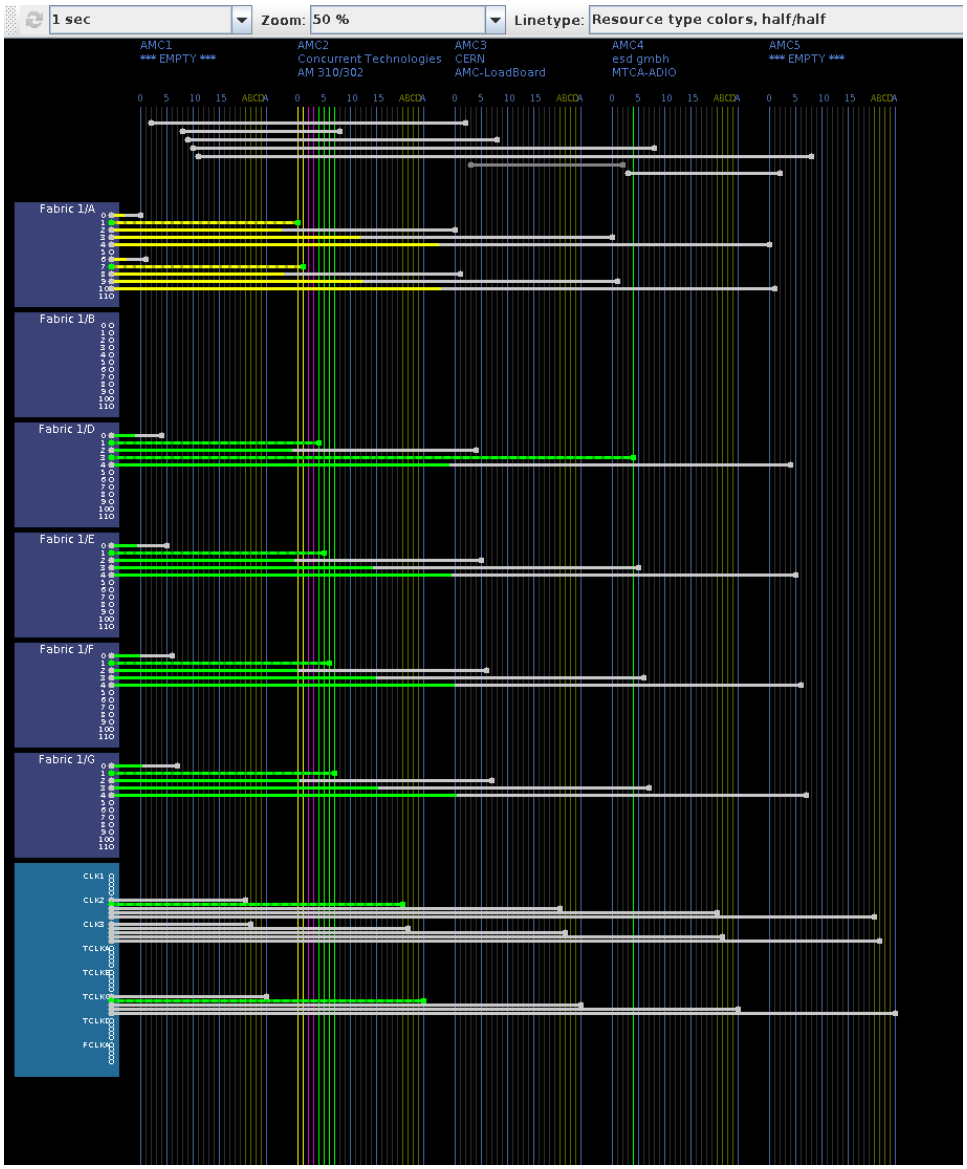


Figure 6. NAT Backplane viewer

PCI Express

The MCH in our test system provides a PCIe switch. Cards in the shelf may use it to communicate with each other. In our system, the Processor AMC communicates with the IO AMC via PCIe. Unlike its predecessor, PCI, PCI express is a serial link. Up to 32 serial links may be combined to form a link. Table 1 shows the speeds in Giga-transfers (GT) per second per lane. Physically, PCIe links are point-to-point, as opposed to a bus topology used in PCI. Data are transferred in packets (like in Ethernet) with data integrity checks, re-transmissions and flow control. PCIe switches are used to connect multiple devices to a single controller (called *root complex* in PCIe). Despite all these differences in the lower link layers, PCI-express is software compatible to PCI.

Table 1. Speed of PCI-express

	per lane	**per lane**
Gen-1	2.5 GT/s	250 MB/s
Gen-2	5 GT/s	500 MB/s
Gen-3	8 GT/s	985 MB/s
Gen-4	16 GT/s	1970 MB/s

To discover the bus structure and devices, you can use the *lspci* tool.

For example, try: *lspci -tv* to display the bus in a tree structure
or try options *–v* and *-vv* to display detailed information about the devices.

Try to locate the IO AMC. Find its bus address and its base address.

Hot Plugging (demo by your tutor)

In a Micro-TCA system, AMCs may be hot-plugged (i.e. exchanged without switching the shelf off). We will try hot-plugging the IO AMC card. Since the IO AMC is a PCI device connected to the Processor AMC, we have to let the Processor AMC know.

Together with the tutor, try these steps (you need to be *root* on the machine):

- show the PCI devices with *lspci*
- Pull gently on the black lever
- wait till the blue light is on
- [pull the card out by its lever]
- repeat *lspci* (did anything change ?)
- [push the card back in]
- wait till the blue light is on
- push the black lever in
- *echo 1 > /sys/bus/pci/devices/[address]/remove*
- (*address* is the address of this device: *PCI bridge: PLX Technology, Inc. PEX 8111 PCI Express-to-PCI Bridge*)
- repeat *lspci* (did anything change ?)
- *echo 1 > /sys/bus/pci/rescan*
- repeat *lspci* (did anything change ?)

Build your own digital scope

Now let's get our hands dirty and do some programming. You will use the Analog-to-Digital converter on the IO/AMC to repeatedly sample an analog input channel and to display the waveform of the sampled signal. The A/D converter continuously samples its input at a programmable frequency. The acquired data may either be polled or transferred to host memory by Direct Memory Access (DMA).

- First have a look at the provided example program *adio_scope.cpp (in /home/student/amc_adio/src).* See how the program maps the address space of the IO AMC into the Processor AMC's memory space and how it then addresses the IO AMC's registers by simple read and write operations to a data structure. Run the program (from */home/student/amc_adio/bin*) and play with it. You can recompile it by running *make* in */home/student/amc_adio.*

- The scope should sample Analog Input 0 at 44.1 kHz. Have a look at the documentation (Hardware Manual) of the IO AMC and find out how to set up the ADC to sample at this frequency.

- You just need to fill in a few missing pieces in the method *acquire_shot*():

    - **First set up the timestamp counter to provide timestamps in microseconds (register DIVMODE)**
    - **Now set up the IO AMC to sample Analog Input 0 at 44.1 kHz You will need to set up registers FGENAB and ADCMODE.**

- After setting up the card, the program acquires a few hundred samples from the ADC by reading register ADC0ACT.

- You need to **fill in code to decode the timestamp and voltage from the value read from register ADC0ACT.**

- The program produces a file with lines of two columns, containing a timestamp in microseconds and the voltage in volts (separated by a space).

- **Build & run the program and have a look at the timestamps. Verify if the make sense. Is polling fast enough to do a scope working at 44.1 kHz?**

- A ROOT program to plot you file is available. So you don't need to worry about producing the graphics. The program will let you choose the file name to plot.

root –l

root [0] .x /home/student/gui_cint.cpp

- As an input signal, you can connect the provided head-phone jack to your smart phone and play some music (you'll have to turn up the volume). Or you download a function generator onto your phone – for example RADONSOFT Signal Generator. (No smart phone in your group? Ask your tutor.)

References (.pdf files available in /ISOTDAQ/doc)

- Short intro to uTCA. *MicroTCA_ShortOverview.pdf*
- http://en.wikipedia.org/wiki/PCI_Express - *PCIExpressWikipedia.pdf*
- More info about PCIe: http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1 *PCIExpress_Xillybus[1-3].pdf*
- Info about the shelf: *ELMA_BlueEco_Shelf.pdf*
- MCH doc: *Samway_MCH_Usermanual_Rev_1.6*
- Manual for the IO AMC: *HardwareManual_IO_AMC.pdf*

Lab 7: System Development using LabVIEW

Gary Boorman, RHUL (Gary.Boorman@rhul.ac.uk)

Adriaan Rijllart, CERN (Adriaan.Rijllart@cern.ch)

Introduction

LabVIEW is systems engineering software for applications that require test, measurement and control, with rapid access to hardware and data insights. It uses a graphical approach to visualize every aspect of the application, including hardware configuration, measurement data and analysis. LabVIEW excels at acquiring data from electronic measurement systems, such as that shown schematically in Fig. 1.

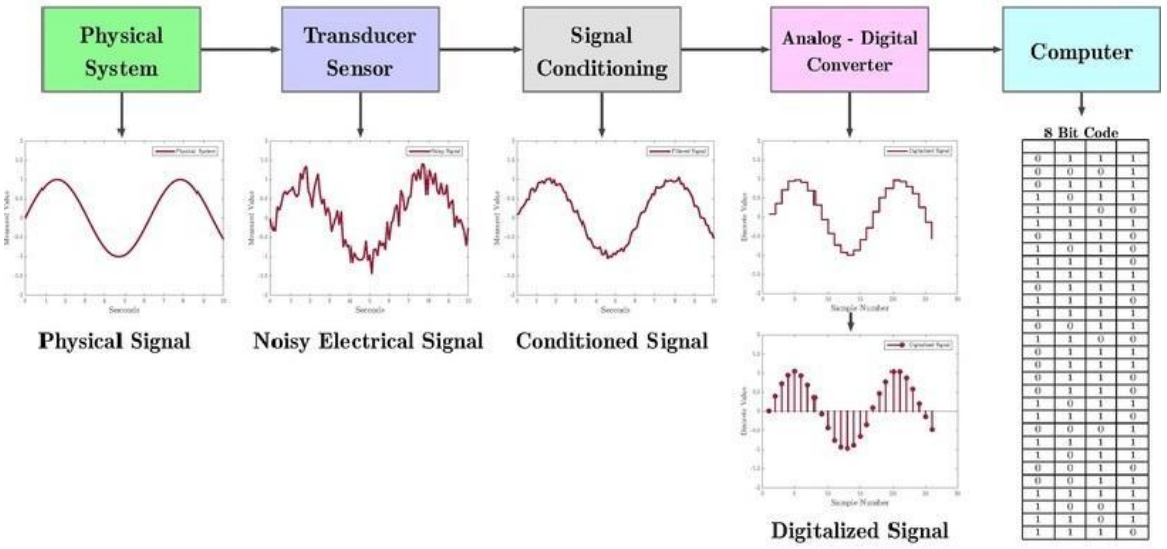


Figure 1: Data acquisition system (from Wikipedia)

The sensor, or transducer, converts a physical quantity (temperature, pressure, acceleration, position etc) into an electrical signal. Conditioning of this signal, usually with analogue electronics such as amplifiers or filters, is sometimes necessary. The resulting signal is converted into the digital domain using an analogue-to-digital converter (ADC) and a computer system can then process and store these digital data. The numbers and the physical types of the measured signals can all be different, as can the processing algorithms. The computer can also generate digital data, such as an output of a control system, which can drive actuators (mover systems, valves etc.) or be converted into an electrical signal via a digital-to-analogue converter (DAC). If all types of signals could be processed then any measuring problem could be solved. Modular instrumentation allows a system to be built that can process many different types of signal.

National Instruments (NI), the producer of LabVIEW, has been developing measurement equipment since the 1980s. They make modular, programmable hardware for a wide range of applications, from acquiring simple analogue signals at a few Hz, to complex FPGA-based high-speed control systems capable of running real-time software for high reliability.

This laboratory session makes use of both LabVIEW and a NI Compact DAQ chassis (cDAQ-9178), into which are installed several types of input/output module, both analogue and digital, summarized in Table 1.

Table 1: Function of Data Acquisition modules

Device	*Number of channels*	*Application*
NI-9211	4	Thermocouple readout 24 bits
NI-9474	8	High-speed digital output
NI-9263	8	General-purpose analogue +/- 10V output
NI-9205	16/32	General-purpose +/- 10V input (16 differential, 32 single-ended)

The cDAQ chassis has the following (programmable) internal electronics system: four 32-bit general-purpose counters, FIFO for analog/digital inputs (with 127 depth per slot), FIFO for digital outputs (2047 samples), clock generators (base clocks of 20 MHz, 10 MHz and 100 kHz with divisors: 1 to 16), digital trigger circuit. Using LabVIEW, the chassis can be programmed to control the modules (a typical set-up is shown in Fig. 2), and a computer can then visualize the acquired data and perform analysis on it.

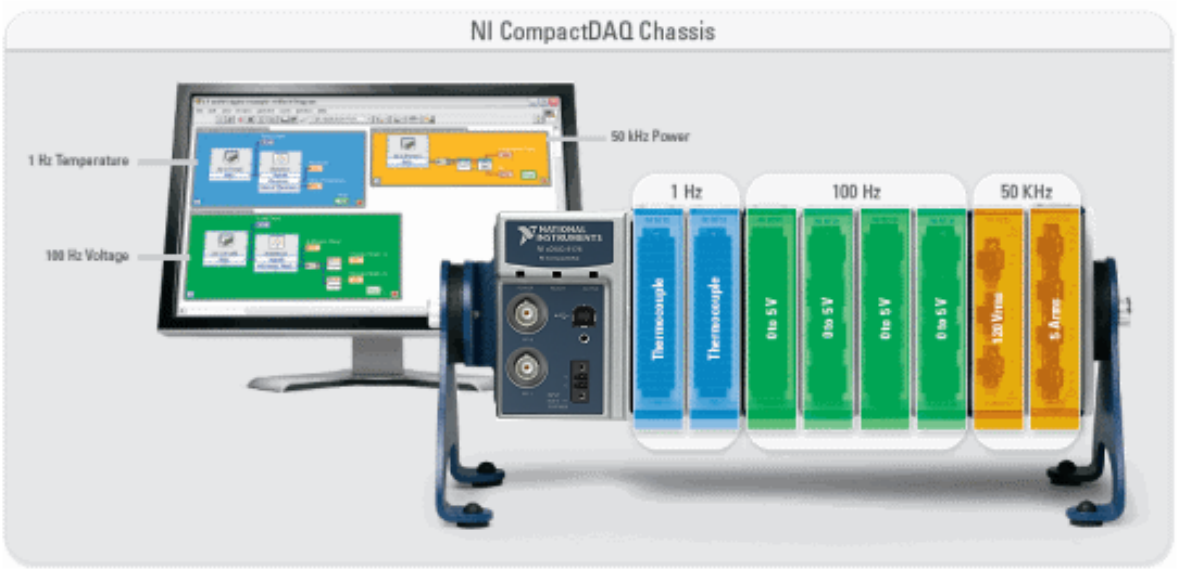


Figure 2: Typical CompactDAQ setup controlled with LabVIEW

Discuss the following question:

What is the advantage of modularity when developing a measurement system?

- The cDAQ, and all other NI hardware, has the advantage of self-discovery. Plug it in and chassis, and any modules/cards present, will be automatically detected
- Common interface to all modules as well as external equipment
- No knowledge of individual busses or communication methods is required, since this is abstracted within the LabVIEW software

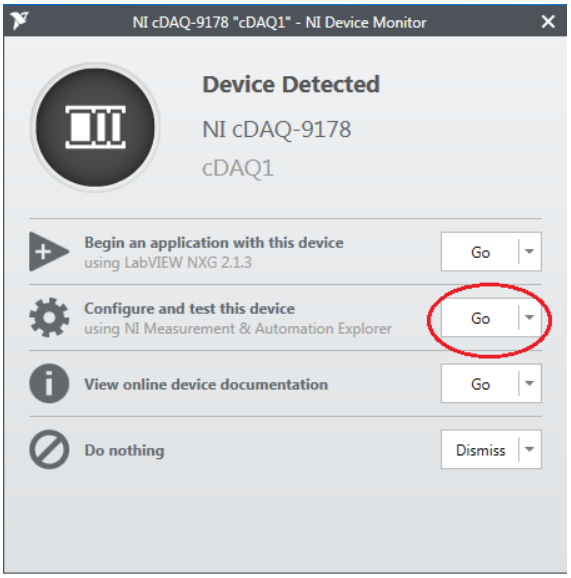
- Scalability – code can work on any number of modules, or even chassis, with minimal changes

Exercise 1: Configure Hardware

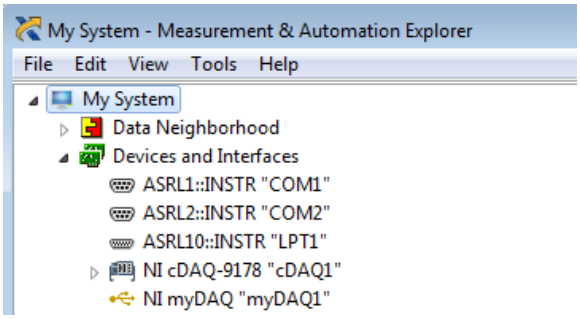
This exercise tests the configuration of the CompactDAQ system.

Set up the Hardware

1. Ensure the NI CompactDAQ chassis (cDAQ-9178) is powered on.
2. Connect the chassis to the PC using the USB cable.
3. The NI-DAQmx driver installed on the PC will automatically detect the cDAQ chassis and bring up the following window.



4. Select *Configure and test this device using NI Measurement & Automation Explorer*. NI Measurement & Automation Explorer (MAX) is a configuration utility for all National Instruments hardware. It can take several seconds to start, depending on how many devices are connected to the PC.
5. Within MAX, the *Devices and Interfaces* section under *My System* shows all the National Instruments devices installed and configured on the PC. By default, the NI CompactDAQ chassis NI cDAQ-9178 shows up with the name 'cDAQ1'.

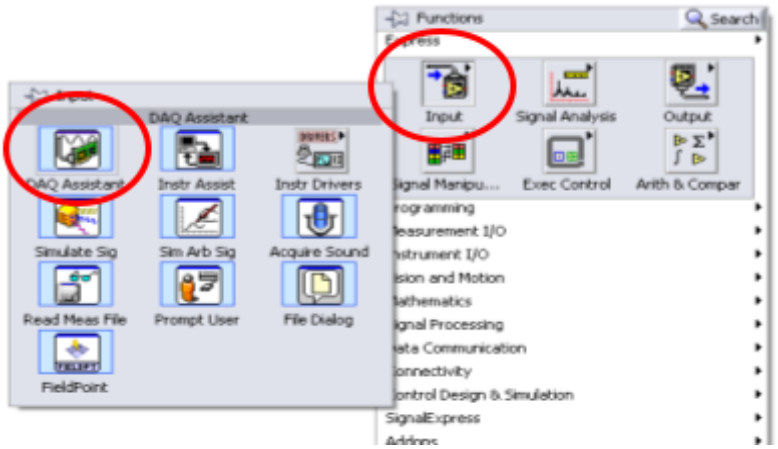


6. Click the triangle to the left of the cDAQ entry to show the modules contained in the chassis.
7. Right-click on NI cDAQ-9178 and select *Self-Test*.
8. The cDAQ passes the self-test, meaning it has initialized correctly and can communicate with the modules, and is ready to be used in a LabVIEW application.

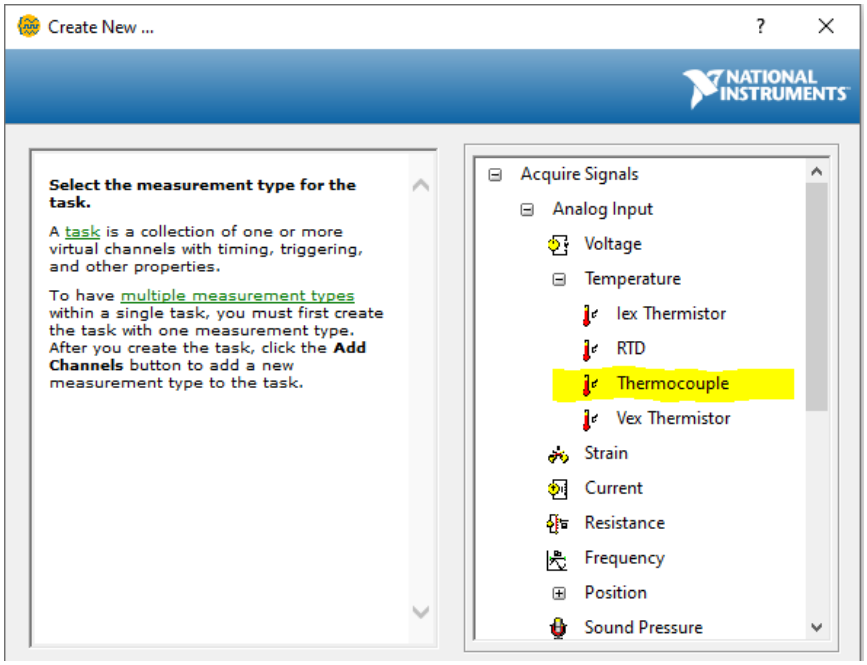
Exercise 2: Create a LabVIEW Application

This exercise creates a VI that can read a temperature sensor (thermocouple) and display both the current temperature and a history of the previous values.

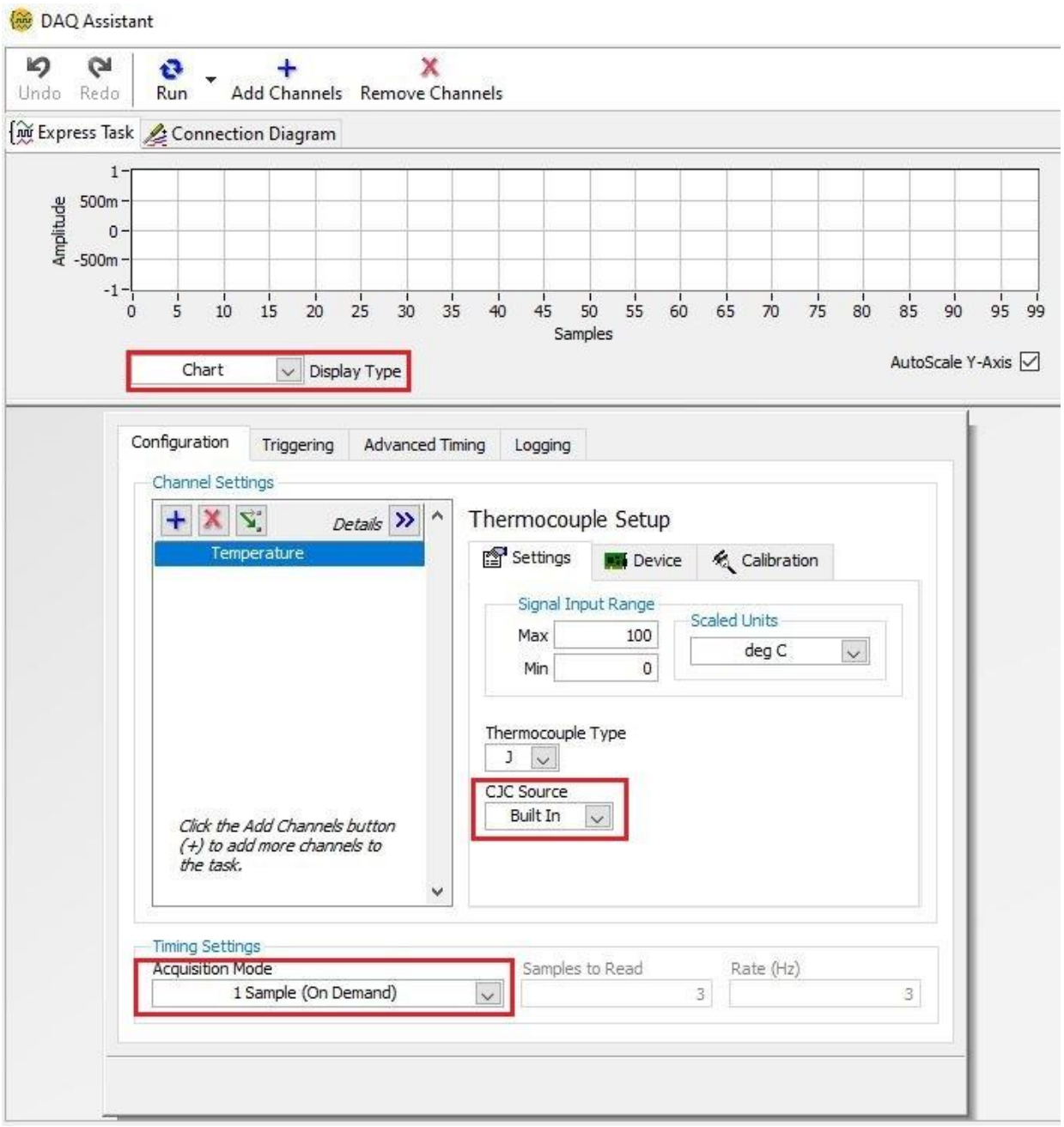
1. Start the LabVIEW application (NI LabVIEW 2018 SP1).
2. Create a new VI, by going to **File» New VI**.
3. The VI consists of the *Front Panel*, the window with which the user interacts, and the *Block Diagram*, where the code is written. The two panes can be displayed side-by-side using <Ctrl +T>.
4. Display the Functions Palette by right-clicking on the white space on the LabVIEW block diagram window.
5. Select the **Express» Input** palette and click the *DAQ Assistant Express VI*. Place it on the block diagram.



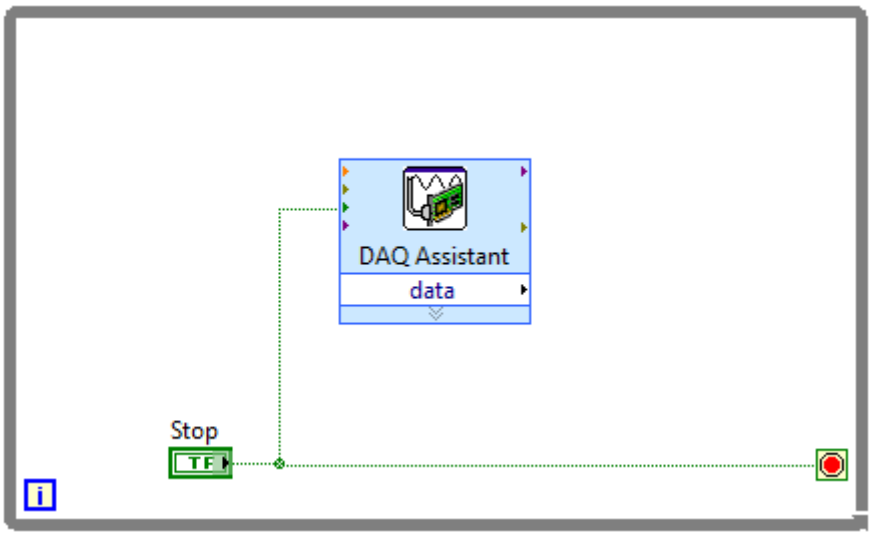
6. Once the DAQ Assistant VI is placed the *Create New Express Task…* window then appears:



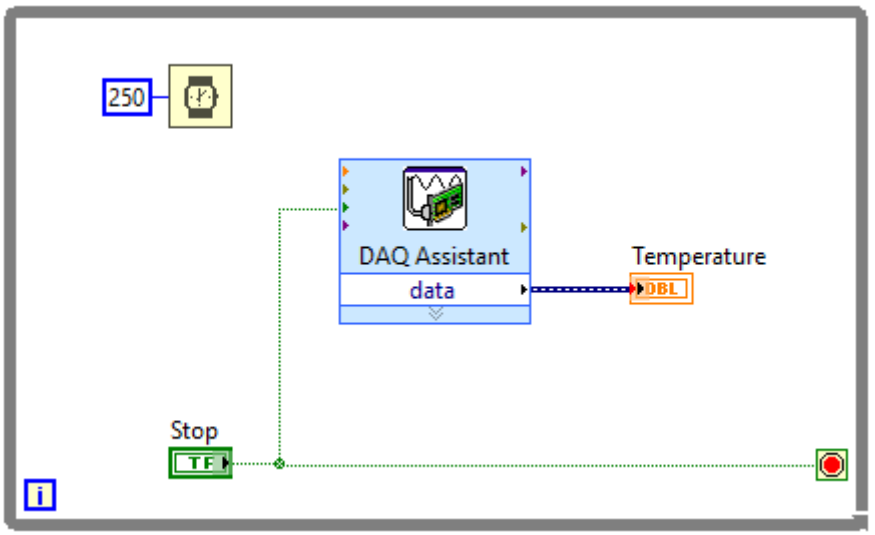
7. To configure a temperature measurement application with a thermocouple, click on **Acquire Signals» Analog Input» Temperature» Thermocouple**. Expand cDAQ1Mod1 (NI 9211), highlight channel ai0, and click Finish. This adds a physical channel to your measurement task.
8. Ensure the CJC Source is set to Built In and Acquisition Mode to 1 Sample (On Demand). Click the Run button to see live data from the thermocouple. The Display Type can be changed to show either a Chart or Table.



9. Click Stop and then click OK to close the Express block configuration window to return to the LabVIEW block diagram. It takes a few seconds for LabVIEW to build the code.
10. A *While Loop* is also required: right-click on the block diagram and select **Structures» While Loop** and draw a loop around the Express VI.
11. Right-click the *Condition Terminal* in the bottom right corner of the While Loop and select *Create Control*. A *Stop* button is placed on the Front Panel, and its terminal is linked to the While Loop. Move the Stop terminal as shown, and also connect it to the *Stop (T)* terminal of the DAQ Assistant.



12. On the Front Panel select **Numeric» Thermometer** and adjust its size to ensure comfortable viewing. Rename the its label to 'Temperature'. On the Block Diagram, wire the DAQ Assistant *data* terminal to the *Temperature* terminal.
13. Finally, add **Timing» Wait (ms)** inside the While Loop and create a constant with a value of 250. The while loop will now run, reading the thermocouple and displaying the temperature every 250 ms.



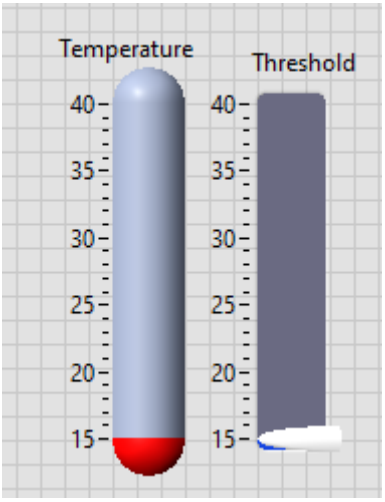
14. To view the temperature history, a chart is required. On the Front Panel, right-click and select **Graph» Waveform Chart**. Rename it *Temperature History* and adjust its size.
15. On the diagram, connect the *Temperature History* terminal to the DAQ Assistant *data* terminal.
16. Save the VI and then run it. Note the Y-scales of both the Thermometer and the Chart can be auto-scaled, and the maximum and minimum values defined by the user. The data points/line on the Chart can also be altered. These changes can be done either during code editing *or* when the code is running.

End of Exercise 2

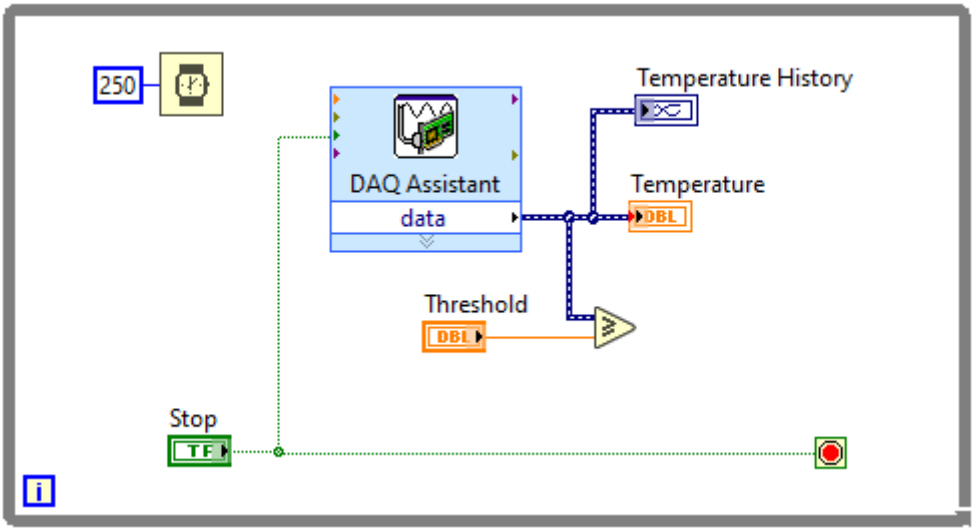
Exercise 3: Add a Threshold to the Temperature Reading

This exercise extends the VI from the previous exercise to add an indication of when the input temperature exceeds a user-defined threshold.

1. On the Front Panel, select a Numeric» Vertical Pointer Slide and position it next to the Temperature indicator. Adjust its position and size, and set the Threshold maximum and minimum values to the same as that of the Temperature indicator. Name the new control *Threshold*.



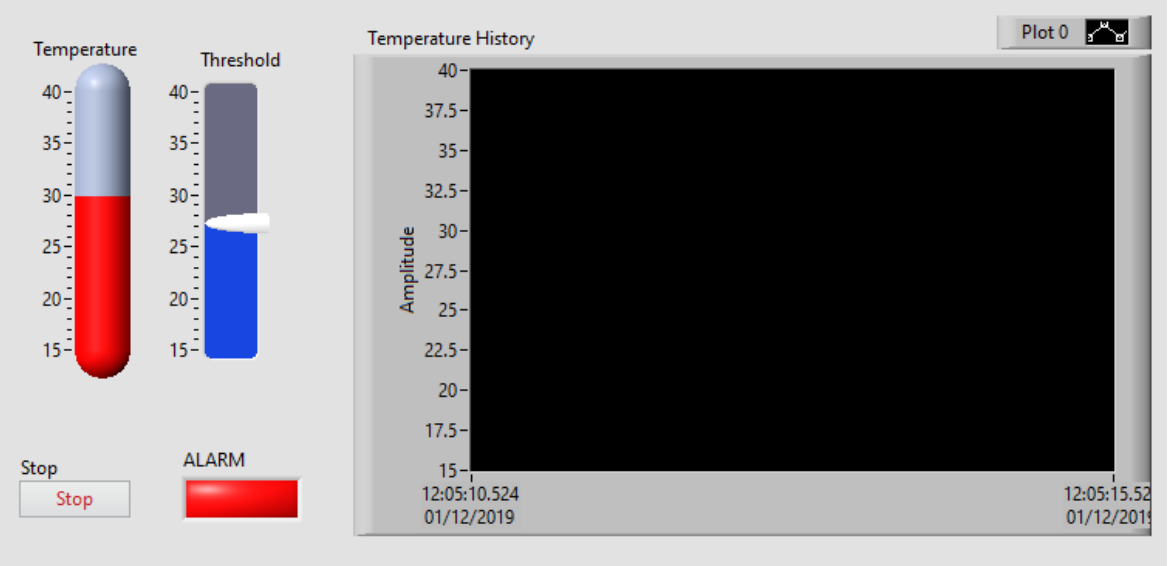
2. On the Block Diagram, from the *Comparison* palette, select a *Greater or Equal?* function, and wire as shown.



3. On the Front Panel, select **Boolean» Square LED** and rename it *ALARM*. Adjust the size. On the block diagram wire the output of the comparator to the ALARM terminal.
4. Save and run the VI. Notice the ALARM indicator lights up when the current temperature exceeds the threshold.
5. The default colour for a Boolean indicator is green, which is not useful for indicating when a

process is in an *Alarm* or *Fault* state. Right-click the ALARM indicator, select *Properties* and adjust the colours to suit.

6. The Front Panel should look similar to the following:



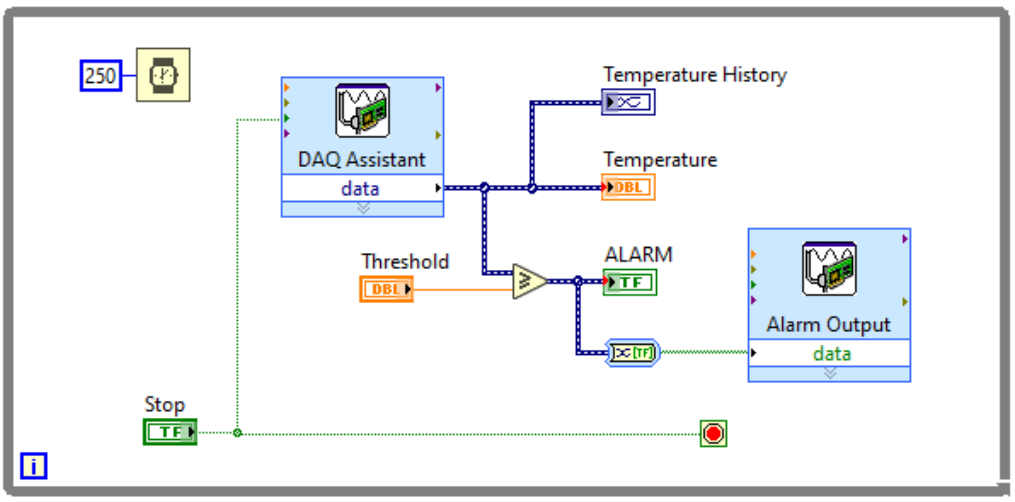
7. Ensure the VI is saved after any further changes are made.

End of Exercise 3

Exercise 4: Output the Alarm

This exercise builds on the previous VI, adding code to write the current Alarm value to a Hardware digital indicator.

1. Place a second *DAQ Assistant* within the While Loop on the block diagram. Configure it **Generate Signals» Digital Output» Line Output**.
2. Select the channel *NI-9474 port0/line0* and press Finish when done.
3. Ensure the *Generation Mode* is set to *1 Sample (On Demand).*
4. The 9474 Digital output module can be demonstrated before the DAQ Assistant is closed. Click Run, and change the *DigitalOut* button state, and observe the LED on channel 0 of the 9474 module. Stop the DAQ Assistant and click OK for the code to be generated.
5. On the block diagram, rename the DAQ Assistant2 to *Alarm Output*.
6. Ensure the *Alarm Output* is moved within the While Loop, and wire to its *data* input from the output of the *Greater or Equal?* comparison function.



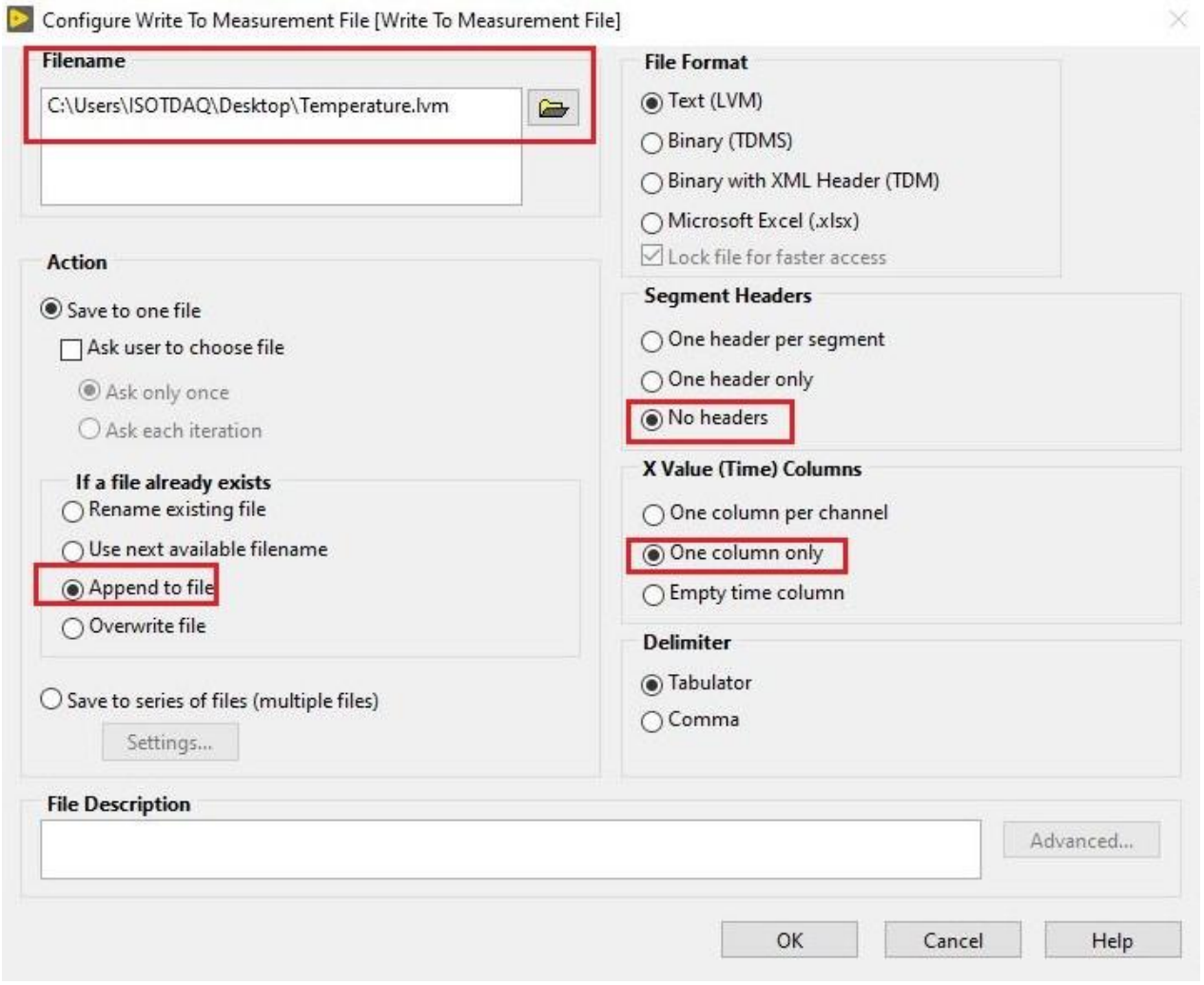
7. Save and Run the VI.
8. Adjust the threshold value to ensure the Alarm can be triggered.
9. The digital output module can drive relays or interface with other industrial control systems, but for demonstration it is only driving a red LED.

End of Exercise 4

Exercise 5: Data Logging

This exercise writes to file the current temperature, using the VI from the previous exercise.

1. On the Block Diagram, place **File I/O» Write to Measurement File**.
2. Make changes to the configuration as shown. The file location may need to be adjusted, depending upon the setup of the computer and directory write permissions.



3. Wire the output of the original DAQ Assistant to the *Signals* input of the File VI.
4. Save and then run the VI for a few seconds, halting the VI using the Stop button. Open and examine the newly created data file.
5. Change the *Write to Measurement File* configuration to include one header, and again inspect the data file.
6. The location of the data file can be displayed on the Front Panel. On the block diagram right-click the top-right terminal of the *Write to Measurement File* function, then select **Create» Indicator**. Expand the indicator on the front panel to enable the path and filename to be displayed.

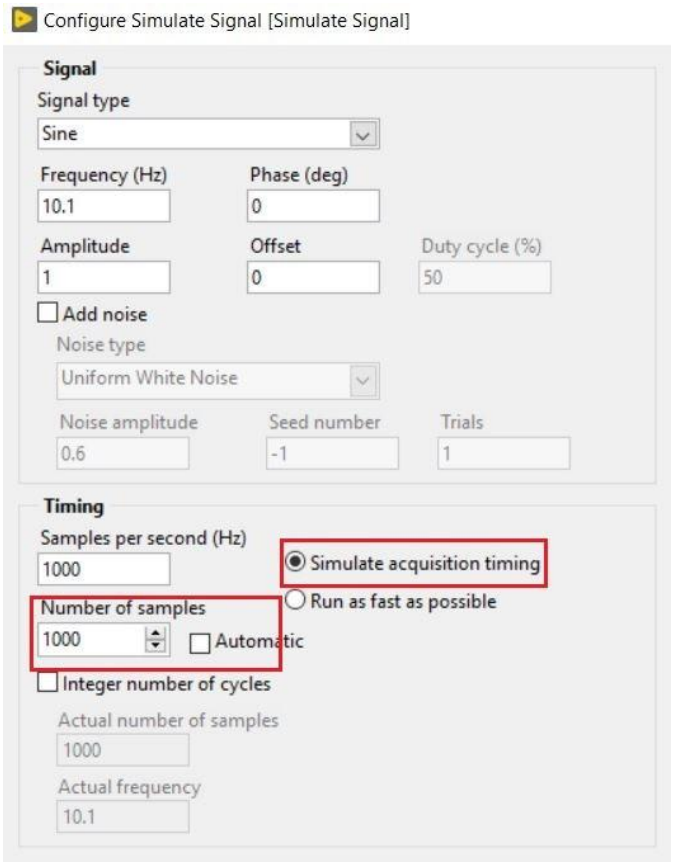
End of Exercise 5

Exercise 6: Generate and Acquire a Waveform

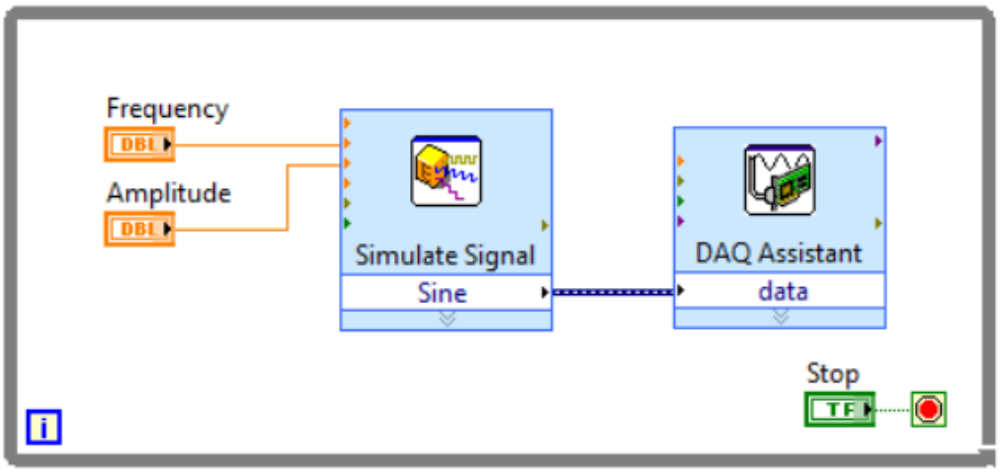
This exercise sets up one of the cDAQ modules to output a sine wave of user-defined frequency and amplitude, and another module to read the generated waveform. If time permits, some basic analysis of the acquired signal can be performed. Different VIs can be made for each function, but the VIs can be run concurrently.

Signal Generation

1. Generate a sine wave using the analog output module (9263). Create a new VI and place a *DAQ Assistant* Express VI on the block diagram and make it generate an analogue voltage. The default settings will be sufficient.
2. Add a *Simulate Signal* Express VI and change the number of samples to 1000, using Simulated Acquisition timing. This VI will generate 1000 samples, at a rate of 1000 samples per second.



3. Connect the *Sine* output of this VI to the *Data* input of the DAQ Assistant.
4. Create controls for the Frequency and Amplitude of the *Simulate Signal* VI. These appear on the front panel and can be changed to knobs or sliders to make them more usable.
5. On the Block Diagram enclose the VIs and Controls in a *While Loop* and add a *Stop* button.



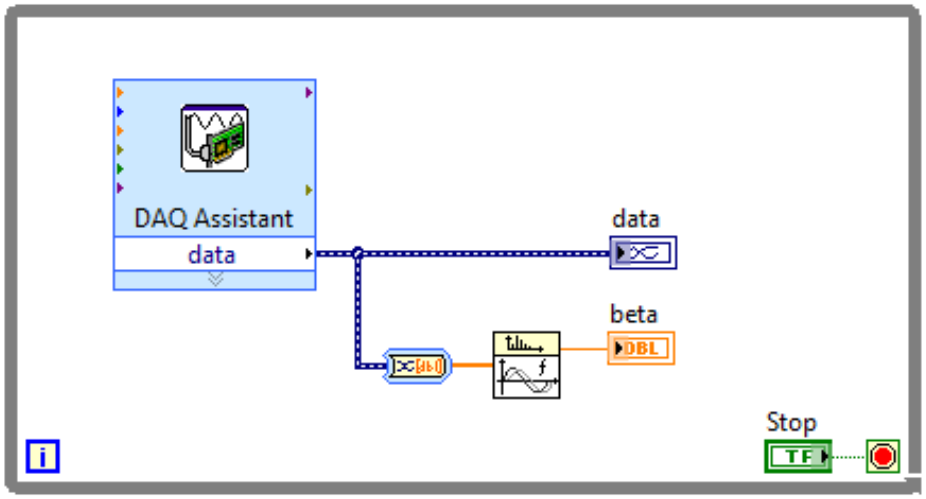
6. Change the maximum value on the *Frequency* control to 1000 (Hz).
7. Save the VI (Sine Generation.vi) and then run it. The output can be viewed on an oscilloscope if one is available.

Signal Acquisition

1. Create a new VI and add a *DAQ Assistant* Express VI on the block diagram to acquire an analogue voltage using the 9205 module.
2. Change the module configuration to read 1000 samples, but leave the other parameters as default.
3. Right-click the output of the *DAQ Assistant* and create a graph indicator. Adjust the size graph on the Front Panel.
4. Add a *While Loop* and *Stop* button and save the VI as *Sine Acquistion.vi*
5. Ensure the two modules, 9205 and 9263, are connected correctly (each module uses channel 0 as their input and output).
6. Run both the *Sine Generation* and *Sine Acquisition* Vis. Observe the graph output as the frequency is increased from a few Hz to 1 kHz. Is the output correct, ie what is expected?

Aliasing

1. Stop both VIs. Edit the configuration of the *Simulate Signal* in *Sine Generation.vi* to generate 10,000 samples at 10 kS/s. This ensures a smooth sine wave up to 1 kHz can be generated.
2. Edit the *Sine Acqusition.vi* block diagram. Add a *Buneman Frequency Estimator* function. Use ctrl-space to open the *quick drop* menu and type *Buneman* to quickly find the function.
3. Wire from the DAQ Assistant *data* output to the *Buneman* input (the *DDT to Array* function is automatically inserted) and create an indicator for the *Buneman* Beta output.



4. Set the Sine frequency generation to 100 Hz and run both VIs. Verify the Buneman function returns the correct value.
5. Change the sine frequency generation in steps of 100 Hz (ie 200, 300, 400 …) up to 1000 Hz, and record the value of Beta.
6. Is this the expected outcome? Where does the linear relation between the measured and generated break down?
7. The *Nyquist* frequency is defined as half the sampling rate, in this case 500 Hz.
8. This is an example of aliasing, where the measured (apparent) frequency is an *alias*, or disguised value, of the generated frequency. This occurs because of under-sampling of the input signal – the acquisition rate is too low to acquire all of the frequency information in the input signal.
9. How can aliasing be overcome in a signal acquisition system? Are there occasions where aliasing can be used to your advantage?

End of Exercise 6

Lab 8: ADC basics for TDAQ

Tutor: Manoel Barros Marin (manoel.barros.marin@cern.ch)

Table of Contents

Concepts of this lab……64
Lab setup……65
Architecture of a Linux device driver for the PCIe card……66
Introduction to Analogue to Digital Conversion (ADC)……68
Acknowledges……74

Concepts of this lab

Figure 1 below shows the generic signal flow of a Data AcQuisition (DAQ) system performing Analog to Digital Conversion (ADC), as well as trigger, data readout and slow control.

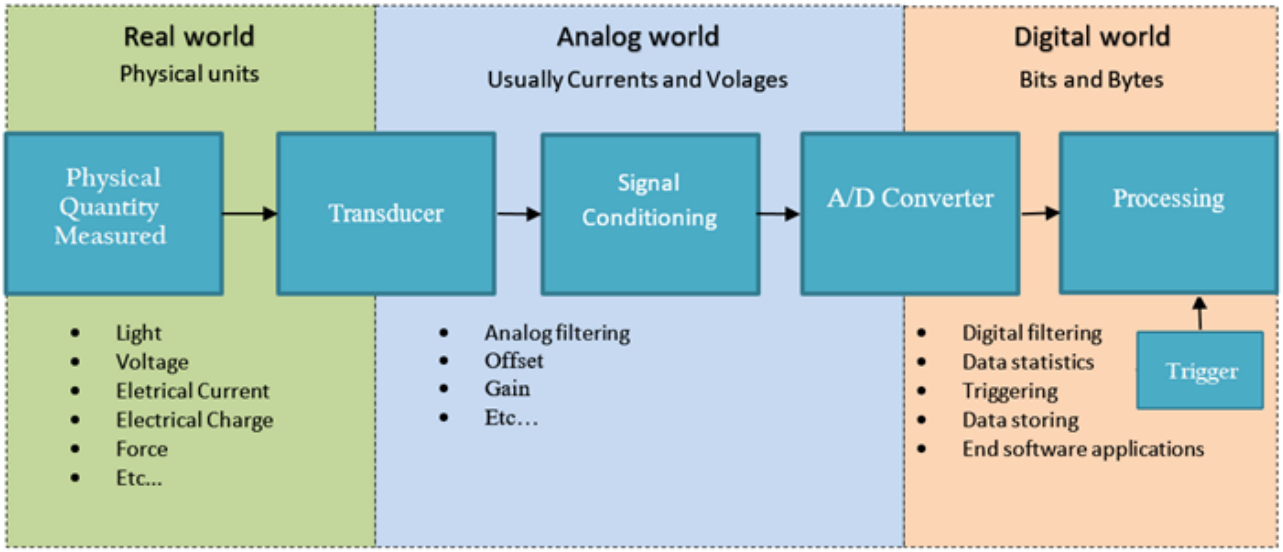


Figure 1- Basic DAQ chain featuring ADC

The scope of this laboratory experience is to understand and experiment with the following components of a Trigger and Data AcQuisition system:

1. Triggers:

   1.1. External triggers: accelerator like type of triggers

   1.2. Triggering on the signal: physics detector like type of triggers

2. Analog to Digital Converter (ADC): we will try to understand fundamental parameters that come into play when measuring with ADC; as for example its resolution, speed, bandwidth, acquisition window, etc.

Lab setup

Theory

Figure 2 below depicts the setup for Lab 8.

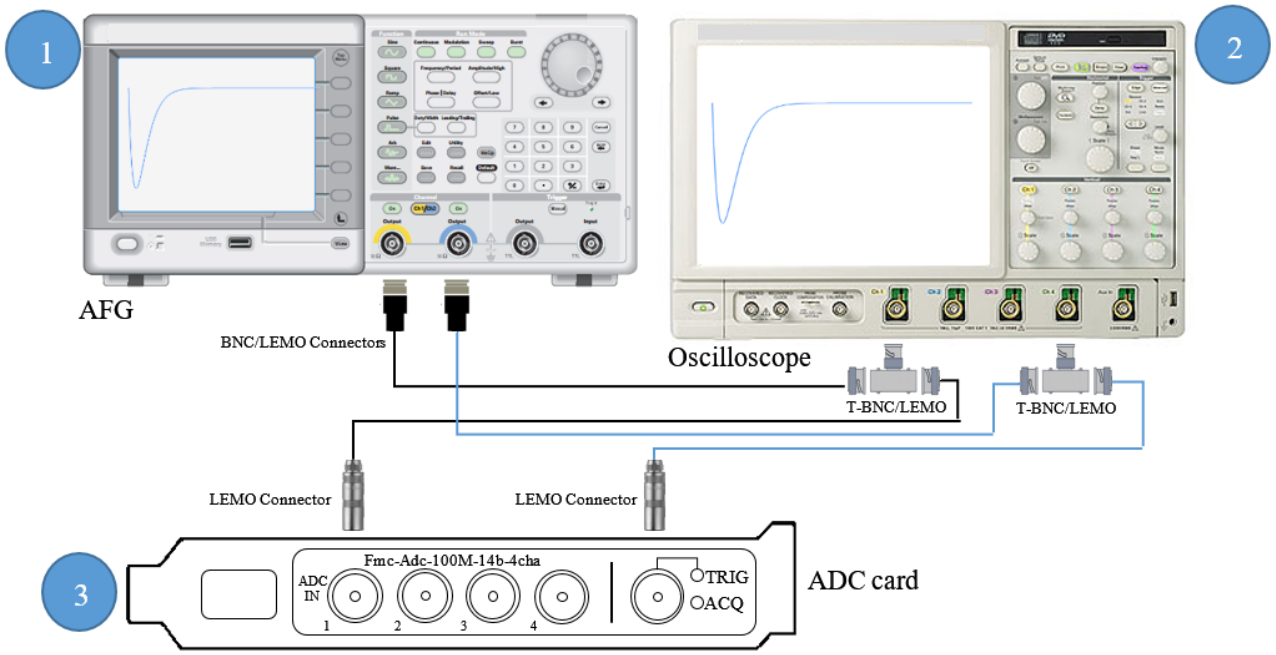


Figure 2 - Equipment setup

1. Arbitrary Function Generator (AFG), is the **input** of our system (acquired signal and trigger)

2. Oscilloscope, is the **monitor** for qualifying our system (to cross-check that the signals fed to the system are truly what intended)

3. SPEC+FMC ADC card plus host PC: this is the **core ADC DAQ**
   3.1. FPGA Mezzanine Card (FMC) ADC: http://www.ohwr.org/projects/fmc-adc-100m14b4cha
   3.2. Simple PCI Express Carrier (SPEC) card: http://www.ohwr.org/projects/spec/wiki
   3.3. Linux based host PC

Practice

1) Construct/confirm the experimental setup according to the sketch you find at the beginning of this document. **Check that both outputs of the AFG are switched OFF.**

   The signal generated by the Channel-1 (**source input**) of the AFG unit is supplied to the oscilloscope, and to the Channel-1 input of the ADC card. Use a T-BNC to split the signal at the AFG output. Also check if the Channel-2 output (**trigger input**) of the AFG is connected to the Trigger input of the ADC card. Again, you can monitor the trigger on the scope by spitting with a T-BNC on the AFG output.

2) Using the AFG, **while keeping the outputs OFF,** set Channel-1 to generate a sine waveform with a frequency of 1 MHz and amplitude of 1 Vpp. Completed this operation, set Channel-2 to generate a pulse waveform with a frequency of 1 KHz and amplitude of 4 V. **Check again that the amplitudes of the channels are below 5 Vpp and the outputs are OFF**.

3) Set the scope to clearly show both the sine (source input) and the pulse (trigger input) waveforms on the screen.

4) **Switch ON** both outputs of the AFG

5) Compare the waveforms on the scope with the setting of the AFG. Are the waveforms stable? Are the amplitudes as expected? If not, what can be the issue? How can we solve it?

6) **Switch OFF** both outputs of the AFG

Architecture of a Linux device driver for the PCIe card

Theory

Before running the DAQ system, an **overview on the communication between the software and the hardware layers and an introduction to the role of device drivers**, is fundamental to understand. *Figure 3* shows a simplified diagram of the Layers of a Linux Operating System (OS).

The basic control is on the hardware peripheral itself. The lowest level software for this system resides in the kernel of the OS as a “device driver.” There are certain control/status registers on the ADC card. These registers can be accessed just like a regular memory access in a C program.

As seen earlier in this document, in our context, the ADC DAQ hardware used is composed of two electronic boards: the SPEC carrier board and the FMC ADC module. The first board is attached directly to a PCIe slot connector on the PC and is the host for the ADC module. The SPEC card acts as a bridge for the electrical signals of the ADC FMC to be interfaced and converted to PCIe. For the scope of this lab, the “bridging” is done by some “black box” electronics. Thus, from a software perspective, one kernel module is instantiated to use the SPEC card, another one for the FMC ADC module.

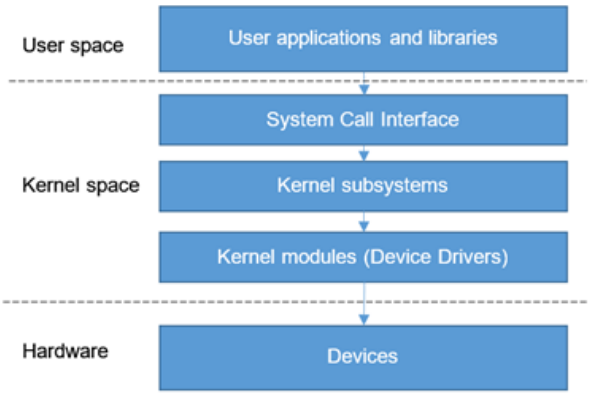


Figure 3 - Layers of a Linux OS

For simple sensors and devices vendors provide information about their operation and use, which can be seen as a map of internal registers and/or procedures. They are required to perform operations or change their current states (more about in Exercise 12 "DAQ Online Software"). For the sake of simplicity and reusability of software code, software engineers created the concept of frameworks. Among others, this concept was used at CERN for creating a flexible interface for the development of input and output drivers. The target is for very-high-bandwidth devices, with high-definition time stamps and a uniform metadata representation. Such framework is named the ZIO (with Z standing for "The Ultimate I/O" Framework).

In short, the way ZIO provides to talk to our hardware is through two different channels, one for control and one for data. Figure 4 shows the two data flows.

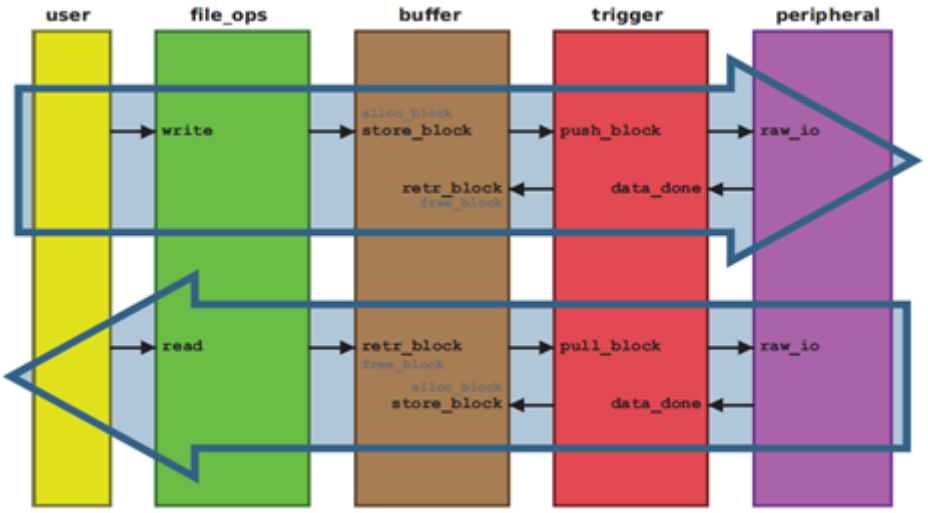


Figure 4 - ZIO data pipeline

When designing a piece of low-level software to communicate to an acquisition device, it is important to choose the way data streams should be passed back and forth the hardware device. To illustrate this, they can be qualified in three types: **Polling mode**, **Interrupt based**, and **DMA (Direct Memory Access)** transfers.

- **Polling mode:** the CPU checks the state of the device's registers every time it can, this has the advantage of providing a fast and low latency communication and data transfer between them, but at the cost of high CPU load.
- **Interrupt based**: this type of transfers eases the CPU load time as it only have to care about the hardware when it tells the CPU to do so; it has the advantage of a lower impact on CPU loads and fast transfer, but with variable latency.
- **DMA transfers**: relies on a shared memory area the CPU provides to the DMA controller to work on the transfer with the hardware device, this frees completely the CPU from controlling the hardware transfer representing a true concurrent system. Highest of all transfer rates allied to an acceptable latency.

Practice

7) Login to the PC (user: adc_*lab* | pass: *isotdaq*) and **reset the directories** with the files of the lab by executing following script, so all the work/changes done by the previous group are removed and a fresh copy of the files are installed.

   $ cd ~/lab8_adc

   $ ./reset_adc_folder.sh

8) To **load the drivers**, change to the required directory and execute the following script. *You need the root password to execute the command above:*

   $ cd adc

   $ ./load_drivers.sh

Spying the content of our script shows the correct order for loading the different kernel modules and set the user permission to talk with the ADC board. It is also possible to check the current kernel modules loaded by issuing the command *lsmod*.

Introduction to Analog to Digital Conversion (ADC)

Theory

An Analog to Digital Converter (ADC), also known as "digitizer", is a device which periodically converts the level of an analogue signal into an integer number in base 2. In other words, an ADC is a device that converts a continuous signal into a discrete representation, both in signal level (**quantization**) and time (**sampling**).

The number of choices of this integer for representing the level of the analogue signal (**quantization**) is determined by the number of bits used for the digitization.

Example: We have a voltage signal whose range is [0 - 10] Volt and we have a digitizer which has just 3 bits. We can have 8 different integer numbers with 3 bits. Each of these numbers will correspond to a voltage. See table below.

Voltage	Digitizer bits	Integer
0.00 - 1.25	000	0
1.25 – 2.50	001	1
2.50 - 3.75	010	2
3.75 – 5.00	011	3
5.00 - 6.25	100	4
6.25 – 7.50	101	5
7.50 – 8.75	110	6
8.75 – 10.00	111	7

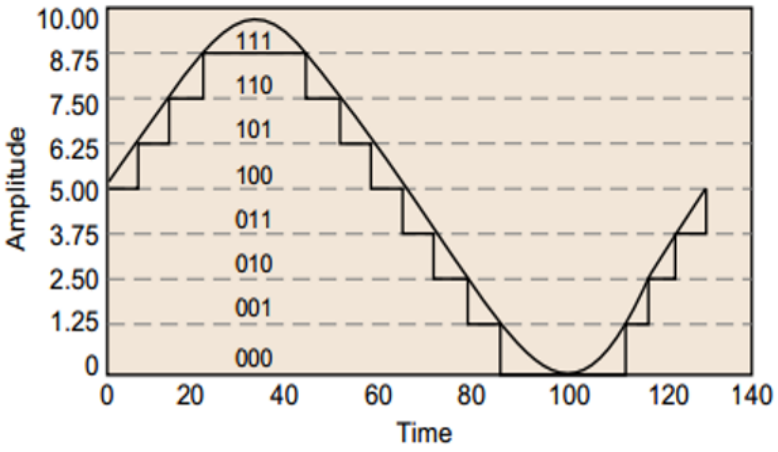


Figure 5 – Quantization & Sampling

As previously mentioned, this kind of conversion is performed at regular intervals (**sampling**) and the repetition (frequency, Hz) is called **sampling frequency** (normally expressed in samples per second). The spectrum of an analogue signal (above) and its sampled version (bellow) are shown in Figure 6.

Theoretically, the limit in how fast may vary the signal that we are measuring is set by the **Nyquist frequency**. The **Nyquist theorem** states that: the minimum sampling rate of an acquisition device must be at least two times the maximum frequency of the original signal, otherwise information would be lost in the process. If this criterion was not respected, the positive and negative frequencies (see in the spectrum of the sampled signal) would overlap and the signal information would be lost. This effect is called **aliasing**.

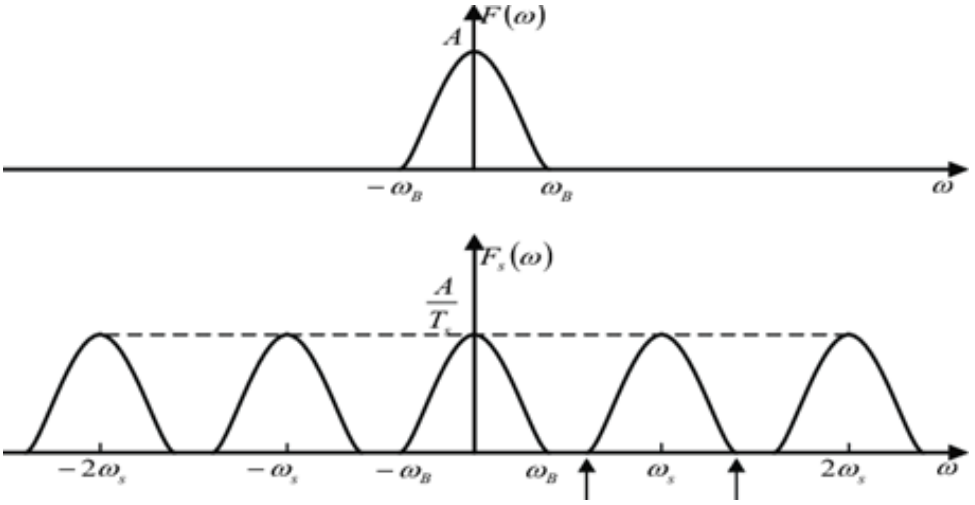


Figure 6 – Spectrum of an analogue signal (above) and its sampled version (bellow)

Also note that to improve the precision of the measurement one should:

- Increase the number of bits, so each corresponding range is small with respect to the signal to be sampled
- Increase the sampling frequency, respecting the maximum frequency allowed (see Nyquist frequency). Please note that a precise sampling clock should be used, such that the deviation between consecutive samples is kept as constant as possible.

If you are interested in more details about ADC parameters, please check:

http://www.analog.com/en/analog-to-digital-converters/products/index.html

If you are interested in understanding more in detail the fundamentals of ADCs and the importance of clock stability (jitter), please check:

https://www.analog.com/en/education/education-library/data-conversion-handbook.html

http://anlage.umd.edu/Microwave%20Measurements%20for%20Personal%20Web%20Site/Tek%20Intro%20to%20Jitter%2061W_18897_1.pdf

Qualify in the best possible way a set of signals mimicking real experimental equipment. The tutor will provide some pre-cooked ones (sine waves, positive pulses, scintillator like pulses) but you can try to think of your real world input.

In general terms, a digitization process is characterized by the following:

- Signal range (dynamic range): how much the signal can vary to be correctly interpreted by the device
- Resolution: The step between consecutive levels of the analogue signal when represented as integer number
- Sampling frequency: the rate at which it can convert an analogue value to a discrete signal
- Noises: physical quantities responsible for the signal deterioration (e.g. added noise, intrinsic noise, quantization errors, etc.)
- Latency: How long does the device takes to finalize the acquisition process in the chain

The best measurement is achieved by understanding and controlling those parameters. **The designers (so YOU) have to decide how to setup your TDAQ:** choose a trigger type, define the acquisition windows, find the signal in the window, maximize the scaling for increasing the accuracy of the measurement, etc.

The ADC card we will be using has 14-bit voltage resolution and a minimum sampling period of 10 ns (100 Msps). In this exercise, we will operate the ADC in order to understand and push its limits.

Practice

9) Now it is time to test our ADC, **turn ON only Channel-1** of the AFG and check if the signal is correctly displayed by the oscilloscope. Run the acquisition program which will subsequently show an acquisition plot:

   $ cd tdaq

   $ ./fald-acq -b 0 -a 1000 -n 1 -e –r 10 -l 1 -g 1 -X 0500

   Where acq-program has the following parameters:

   --before|-b <num> number of pre samples
   --after|-a <num> n. of post samples (default: 16)
   --nshots|-n <num> number of trigger shots
   --delay|-d <num> delay sample after trigger
   --under-sample|-u|-D <num> pick 1 sample every <num>
   --external|-e use external trigger
   --threshold|-t <num> internal trigger threshold
   --channel|-c <num> channel used as trigger (1..4)
   --range|-r <num> channel input range: 100(100mv) 1(1v) 10(10v)
   --negative-edge internal trigger is falling edge
   --loop|-l <num> number of loop before exiting
   --graph|-g <chnum> plot the desired channel
   --X11|-X Gnuplot will use X connection
   <LUN>

   Did the program make the acquisition?

10) Turn ON Channel-2 of the AFG. Does it run now?

11) Now that we know that the ADC is working properly, we can go to some real time data analysis. Bearing this in mind, there is a small program called V_t_continous.C which uses the ROOT framework functionalities and runs on top of fald-acq. To run V_t_continous.C issue the command:

   $ root V_t_continous_Ext.C

12) One issue you may get while changing the frequency is the ‘non-stopping’ graph, meaning it is continuously sweeping horizontally. What causes this?

   Since we are using Channel-2 of the AFG as an external trigger, its triggering frequency dictates how, or at which point in time, the acquisition of Channel-1 signal starts. In practice: if the frequency of our sine wave is not a harmonic of Channel-2 pulse, i.e. not an integer multiple, the ADC doesn’t capture the signal at the same phase.

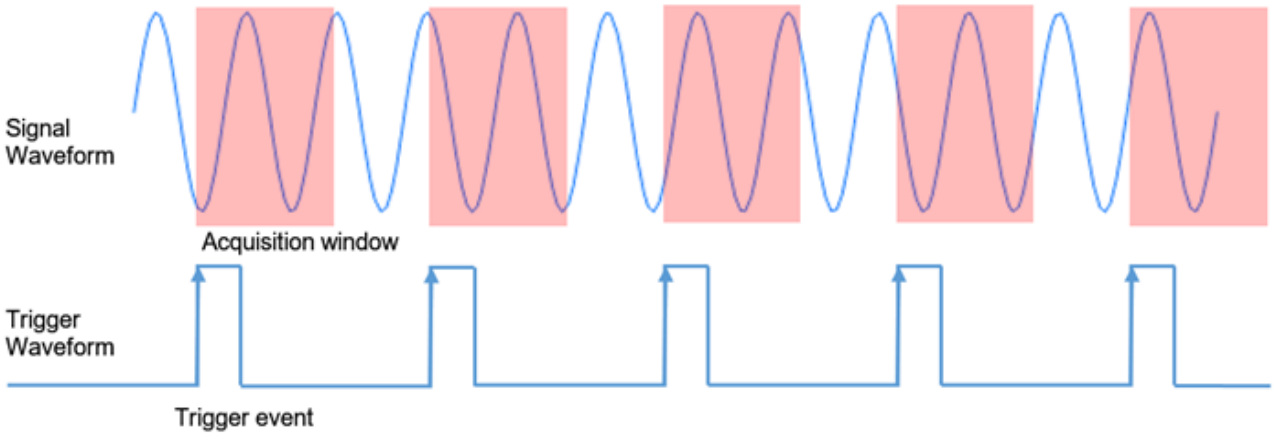


Figure 7 – Signal frequency non multiple of trigger frequency

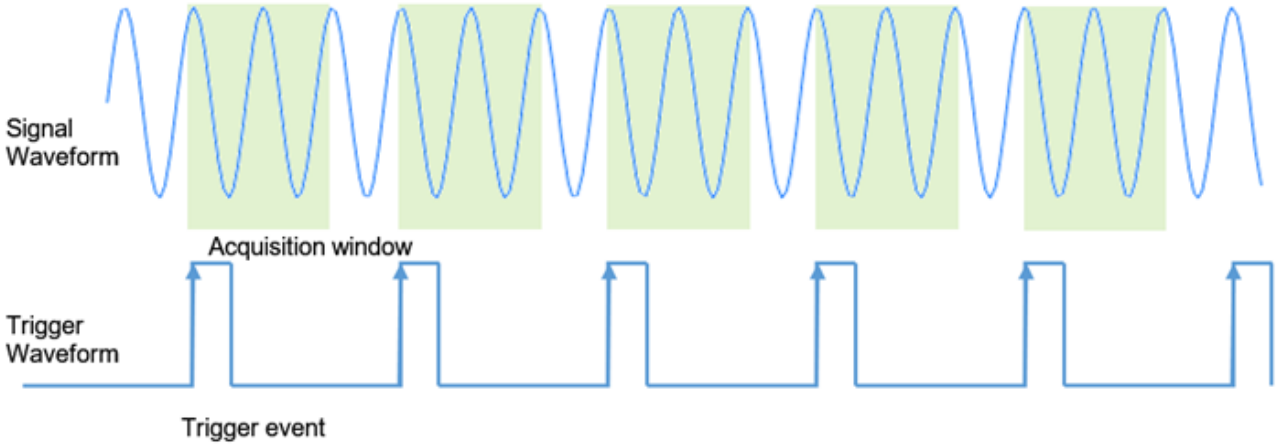


Figure 8 - Signal frequency multiple of trigger frequency

13) Now that you know the trigger requirements for having a "static" plotting of the waveform, let's do a new acquisition, but this time using the following script:

    $ root V_t_continous_Ext_CH2.C

    Do you see any difference with respect to the previous acquisitions done the other script? If so, are you able to understand what is going on?

14) So far, we have applied external triggers to our acquisitions. In this point we are going to trigger on the signal (Figure 9 shows a diagram of this type of acquisition). For this type of acquisition, **turn OFF the trigger channel on the AFE** and run the following script:

    $ root V_t_continous_Int.C

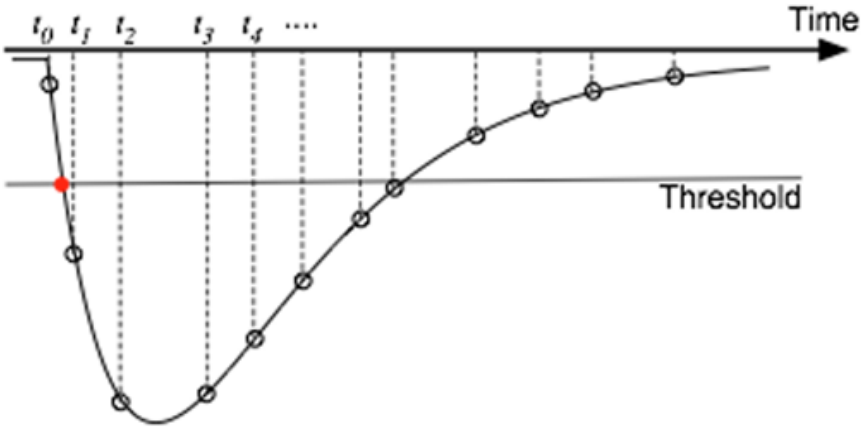


Figure 9 - DAQ triggering on the signal

Do you see any difference with respect to the previous acquisitions done the other script? If so, are you able to understand what is going on?

15) Going further, let's push the frequency of our signal to the limits of the ADC capabilities. Recalling that our ADC specifications say that the default sampling rate is 100 Msps try to increase the signal frequency in steps of MHz (**make sure you are changing the frequency and the NOT the amplitude**). *Please note: it is recommended to decrease the number of samples in our acquisition window*; otherwise it would become hard to analyze the signal in a cloud of points.

    For this, open the V_t_continous_Ext_CH2.C file with your favourite editor and change the number of post samples (-a parameter) to 100 or less, it is located on the line which calls fald-acq.

    Turn ON the trigger channel on the AFE

    Run V_t_continous_Ext_CH2.C file again.

    Can you see the relation between the acquisition window and the signal speed?

16) You can see that the closer you get to 100 MHz the worst the acquisition signal looks like. Can you define the maximum AFG signal frequency where our sine wave keeps its shape in a single acquisition shot (i.e. you can still see a sine wave with the same frequency as the original signal)? Does maximum signal frequency match our expectations regarding the Nyquist theorem? It not, do you know the reason of this mismatch?

    In the example of Figure 10 below: the original signal is in blue, the sampling points are pointed by the black arrows and the acquired data is represented in orange. As previously mentioned, when the Nyquist criteria is not respected an effect called **aliasing** appears. This effect is illustrated with the orange signal in the time domain.

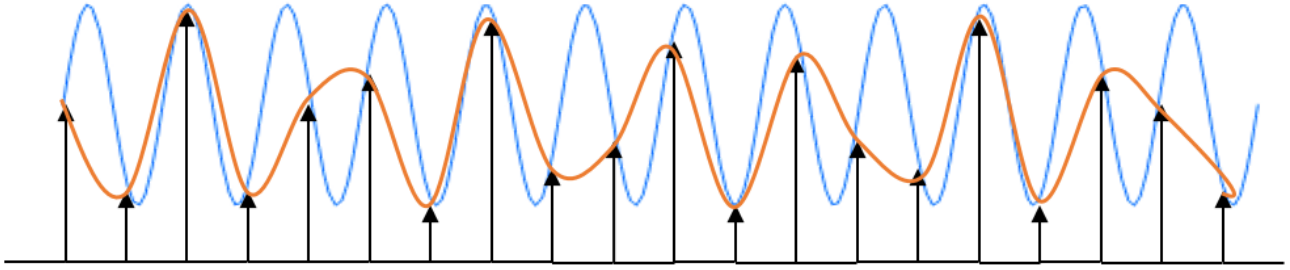


Figure 10 - Aliasing effect

Does maximum signal frequency match our expectations regarding the Nyquist theorem? It not, do you know the reason for this mismatch?

17) The maximum Nyquist frequency is computed on a signal ideally composed of a single sinusoid. In fact any real function of a complex signal is mathematically described as the sum of a series of trigonometric functions in the **frequency domain** (called *signal spectrum*) instead of time. **Fourier series and transforms** base their analyses on this concept. The spectral representation of a sine wave is rather simple: it is only a single line centred on the frequency it converts from the time domain. The spectral content is more complex for different waveforms signals such as square, saw-tooth and other more complex signals. **To qualify ADCs the most used method are based in the frequency domain**.

    In order to check how our acquisition systems behaves with complex signals go back to the frequency of 1 MHz but change the type from sine-wave to square signal. Increase the frequency to values below Nyquist criteria.

18) In case of having some time left, ask the tutor to present a real time Fast Fourier Transform (FFT) and **ask further questions!**

Acknowledges

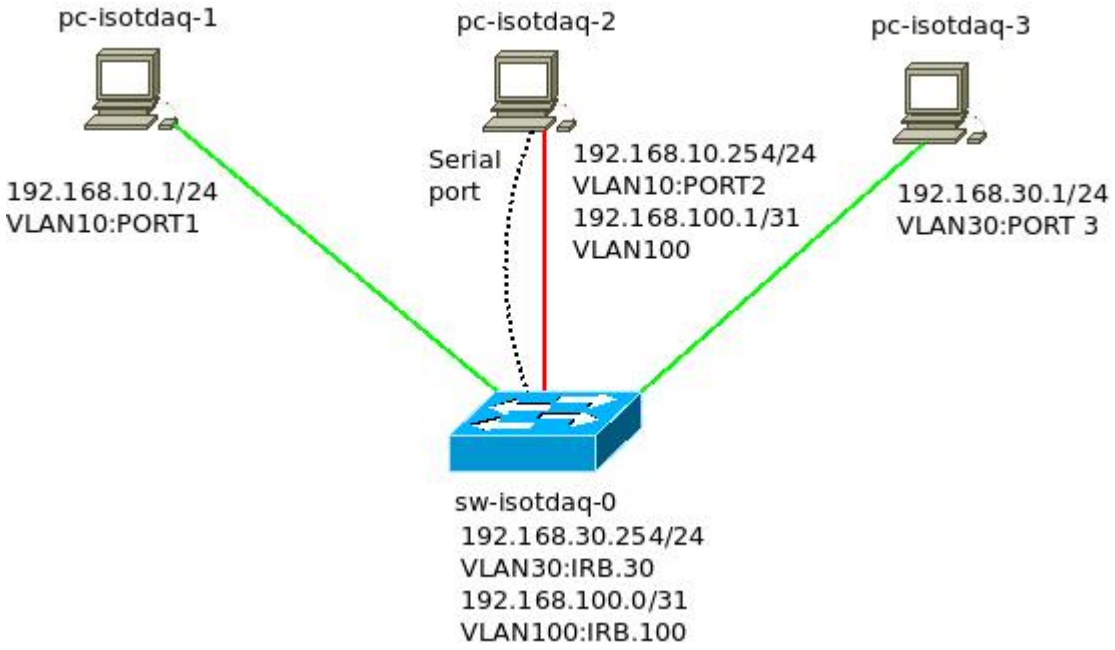
Andrea Borga (andrea.borga@nikhef.nl, initiator and tutor of this lab in ISOTDAQ2016)
Cairo Caplan (cairo.caplan@cern.ch, initiator and lab assistance in ISOTDAQ2016)
Diogenes Gimenez (diogenes.gimenez@usp.br, initiator and lab assistance in ISOTDAQ2016)
Jan Pospíšil (Jan.Pospisil@fosfor.cz, tutor of this lab in ISOTDAQ2018)

Lab 9: Networking for Data Acquisition Systems

I. Introduction

Through the use of a simplified network setup this lab aims at getting you familiar with the following notions involved in networking:

- configuration: MAC and IP addresses, switch management, VLAN, routing, DNS
- monitoring: SNMP, traffic analysis
- testing: performance benchmarking, TCP/IP
- troubleshooting: tcpdump, netcat
- Use several terminal windows as we'll need to swap from host to host often, so no need to close the terminal windows.



The lab setup consists of:

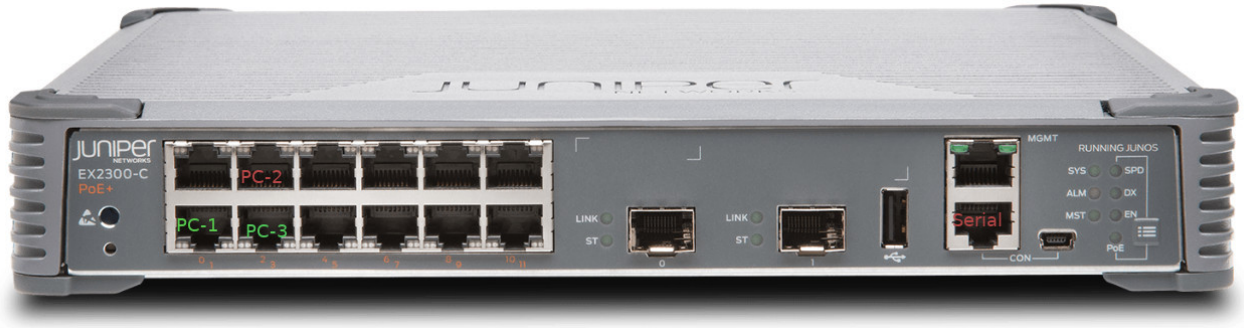
- 2 headless desktop computers mimicking servers that talk to each other.
- 1 desktop computer (pc-isotdaq-2) acting as a control station and router and while also providing services essential for network operatons.

- 1 enterprise-grade network switch mimicking a data collection network.
- Separate VLANs to illustrate routing and separation of control and data traffic.

II. Exercises

1. Switch Configuration

1. The switch has just been powered on and has default factory configuration. Network cabling is as follows:



2. Login to the pc-isotdaq-2 with username student and password Student
3. By default the switch has hardly any functional configuration and we have to deploy the configuration. There are few ways to do this, including via dhcp/tftp, serial port or usb. We're going to use the serial port approach. Login to the switch using serial from pc-isotdaq-2 by using "minicom" application:

```
[student@pc-isotdaq-2]$ sudo minicom
You'll be prompted for a username / password: student / Student
```

4. Let's start by creating interface configuration that allows us to talk to the switch over IP protocol instead of relying on the serial port:
   1. student@sw-isotdaq-0> configure
   2. student@sw-isotdaq-0# set vlans VLAN_20 vlan-id 20 l3-interface irb.20
   3. student@sw-isotdaq-0# set interfaces irb.20 family inet address 192.168.20.254/24
   4. At this point you may want to review what we've done so far:
   5. student@sw-isotdaq-0# show | compare
   6. We still need to assign the physical port 2 to the correct VLAN we just created:
   7. student@sw-isotdaq-0# set interfaces ge-0/0/2 native-vlan-id 20
   8. student@sw-isotdaq-0# set interfaces ge-0/0/2 unit 0 family ethernet-switching interface-mode trunk

9. student@sw-isotdaq-0# set interfaces ge-0/0/2 unit 0 family ethernet-switching vlan members VLAN_20
10. You could now review your changes and commit
11. student@sw-isotdaq-0# show | compare
12. student@sw-isotdaq-0# commit
13. After this point you could open another terminal window and you should be able to ping the switch from the pc-isotdaq-2:
14. student@pc-isotdaq-2: ping 192.168.20.254 (ctrl+c to stop it)

5. Before proceeding further, let's have a quick look at the arp cache on pc-isotdaq-2 and the mac address tables on the sw-isotdaq-0
    1. issue "arp -an" or "ip nei" on pc-isotdaq-2
    2. the switch: you can review the mac addresses by issuing: "run show ethernet-switching table"
6. You can now try accessing the switch via ssh from pc-isotdaq-2
    1. ssh 192.168.20.254
    2. This is not working, can you think of a reason for this based on the error message received?
    3. To get the ssh going, we need to do something on the switch side using the serial port
        1. student@sw-isotdaq-0# set system services ssh
        2. student@sw-isotdaq-0# commit
7. Once you are logged in via ssh we can configure couple of more VLANs that are used in the lab.
    1. First, change over to configuration mode in the ssh session by typing "configure" followed by enter.
    2. Enter the "vlan" hierarchy of the configuration by typing "edit vlans" and start creating the vlans.
        1. set VLAN_10 vlan-id 10
        2. set VLAN_30 vlan-id 30 l3-interface irb.30
        3. set VLAN_100 vlan-id 100 l3-interface irb.100
        4. issue "show" to see the current configuration hierarchy (vlans) and note that the switch is complaining that the l3-interfaces are missing, so lets create those next by issuing the following commands.
        5. top (This changes to the top of the configuration hierarchy)
        6. edit interfaces irb (This changes to the irb sub hierarchy inside interfaces)

7. show (To review the current irb subhierarchy configuration
8. set unit 30 family inet address 192.168.30.254/24
9. set unit 100 family inet address 192.168.100.0/31
10. up (This moves one level up in the hierarchy, allowing us to modify the physical ports)
11. set ge-0/0/1.0 family ethernet-switching vlan members VLAN_10
12. set ge-0/0/3.0 family ethernet-switching vlan members VLAN_30
13. set ge-0/0/2.0 family ethernet-switching vlan members [VLAN_10 VLAN_100]
14. show | compare
15. commit
16. After this we should be able to talk to the switch via both VLAN_20 and VLAN_100, also pc-isotdaq-1 (192.168.10.1) is now reachable, can you think of why this is?

3. You should be able to ssh to to pc-isotdaq-1. While you were able to ssh to the switch (192.168.100.0 or 192.168.20.254) from pc-isotdaq-2, this does not appear to work from pc-isotdaq-1, can you think of a reason?
    1. on pc-isotdaq-1 try executing “ip route” or “netstat -rn” to investigate
    2. add a default route pointing to the pc-isotdaq-2 on pc-isotdaq-1 and try pinging the switch:
        1. sudo ip route add default via 192.168.10.254
        2. ping 192.168.100.0
        3. Communication to the switch is still not functioning but you can verify from the pc-isotdaq-2 that it is trying by running “sudo tcpdump -i eno1.10 icmp” (ctrl+c to stop)
        4. Running “sudo tcpdump -i eno1.100 icmp” on pc-isotdaq-2 shows us that the switch is not replying? Can you think of why this is?
        5. Lets see if we can fix this somehow by loading a configuration patch on the switch from the top of the configuration hierarchy
            1. student@sw-isotdaq-0# top
            2. student@sw-isotdaq-0# load patch ospf.patch
            3. student@sw-isotdaq-0# show | compare
            4. student@sw-isotdaq-0# commit
        6. After 10-15 seconds the ping starts working against the switch and also to the pc-isotdaq-3 (192.168.30.1) , lets quickly review what we did with the patch.

2. Network Monitoring, Routing and Traffic Analysis

Before we further investigate the connectivity in our network, let’s setup a simple monitoring method called snmp to understand how we could monitor bandwidth and other metrics of interest.

1. Start by enabling snmp service on the switch:

   i. student@sw-isotdaq-0# configure (in case you exited the configuration mode)
   ii. student@sw-isotdaq-0# set snmp community public authorization read-only
   iii. student@sw-isotdaq-0# commit
2. After this we can use a commands from the NET::SNMP family on pc-isotdaq-2 to pull data metrics and other data from the switch:
   i. snmpget -c public -v 2c 192.168.100.0 sysDescr.0
   ii. snmpwalk -c public -v 2c 192.168.100.0 ifInOctets
   iii. you can dump the full table with: snmpwalk -c public -v2c 192.168.100.0 (takes a while to download)
3. In the above example, we could see the interface index number. This is essential information to have for each of the interfaces we like to monitor. Use the following to figure out the index number for the ge-0/0/2 that connects to pc-isotdaq-2:
   i. snmpwalk -c public -v 2c 192.168.100.0 ifDescr|grep “ge-0/0/2.0”
4. In the home directory of student user on pc-isotdaq-2 there is a script “monitor.pl” that can be used to illustrate how the monitoring over snmp could be used. Script takes one argument, the snmp interface index (the one you collected in the previous step), for example: ./monitor.pl 528
   i. You should start to see the input/output counters of the corresponding interface to appear. You can see how the counters react if you start an aggressive ping while the script is running: sudo ping -i 0.1 192.168.100.0 (ctrl+c to stop)

Data pulled via snmp could be stored to a time series database and visualized by tools such as Grafana, MRTG and thelike.

3. Troubleshooting and diagnosing using tcpdump, netcat and iperf

1. Start by creating a tcp connection from pc-isotdaq-02 to pc-isotdaq-1 on port 3010
   1. [student@pc-isotdaq-2]$ nc 192.168.10.1 3010
      - Why does it fail? TIP: look for glues in “netstat -ln” and “netstat -ln | grep 3010” output on pc-isotdaq-1.
2. Create a tcp server on pc-isotdaq-1 on port 3010 and attempt to establish a connection from pc-isotdaq-2
   1. ssh 192.168.10.1
   2. nc -l 3010
   3. Now, take another terminal from pc-isotdaq-2 and try to connect to the tcp server on pc-isotdaq-1
   4. nc 192.168.10.1 3010

   You can now send data by typing something on the client session followed by enter.
3. Stop (ctrl+c) tcp client and the server from the previous step and open up two new terminal sessions on pc-isotdaq-1 and pc-isotdaq-2 and start tcpdump on both machines.
   1. sudo tcpdump -i eno1 tcp port 3010
   2. try running the same tcp client while observing what you can see on the tcpdump

windows. You should see the connection attempt from pc-isotdaq-2 and a reset from pc-isotdaq-1 since there is nobody listening.

3. start the tcpserver on pc-isotdaq-1, nc -l 3010 and try the same thing again. You should see the tcp handshake taking place.

4. Stop both the client and the server and both tcpdump sessions and modify the tcpdump to also look for udp packets and tell tcpdump to show the payload of the packets. Start a udp server on pc-isotdaq-1 and udp client on pc-isotdaq-2.
    1. student@pc-isotdaq-[1-2]: sudo tcpdump -i nn -X -vv -i eno1 tcp port 3010 or udp port 3010
    2. student@pc-isotdaq-1: nc -u -l 3010
    3. student@pc-isotdaq-2: nc -u 192.168.10.1 3010
    4. Do you see any handshake taking place in the tcpdump?
    5. If you send some data over the nc session, can you read the payload in the tcpdump?
    6. If you stop the nc session, do you see any packets stating that the disconnection took place? Is this any different to tcp based nc session we used before?
5. Start iperf server on pc-isotdaq-1 to measure network throughput
    1. student@pc-isotdaq-1: iperf -s
    2. student@pc-isotdaq-2: iperf -i1 -c 192.168.10.1
    3. You can also run the previously mentioned monitor.pl while the iperf run ins going to see how the snmp counters react. You can use -t 30 to tell iperf to run for 30 seconds. Note that you have to use the physical interface index (ge-0/0/2 - not ge-0/0/2.0) as the pc-isotdaq-1 is in the VLAN 10.
6. On the pc-isotdaq-2 node, start an iperf client using TCP traffic but lowering the TCP window's max size. Analyze the command output: what differences can you observe in comparison to default TCP?
    a. student@pc-isotdaq-2: iperf -i1 -w 8k -c 192.168.10.1

4. Bonus Exercises

1. In the first exercise the pc-isotdaq-3 was configured with a static IP address through the configuration file /etc/sysconfig/network-scripts/ifcfg-eno1. Another way of doing this is by using DHCP (dynamic host configuration protocol). DHCP allows the client machine to request for an ip configuration from a central location. We can experiment with this by running the dhcp client on pc-isotdaq-3 against the eno1 interface while observing what is happening on the dhcp server pc-isotdaq-2):
    a. student@pc-isotdaq-2: sudo tcpdump -XX -vv -i eno1 udp port 67
    b. student@pc-isotdaq-3: sudo dhclient -v eno1

c. It is not working? Can you think of why this might be? Pro tip: IPv4 client relies on broadcast to reach the server.
d. On the switch, enter configuration mode and issue: load patch dhcp_relay.patch
e. issue “show | compare” to review the change
f. issue “commit” on the switch and try the client again (you may have to stop the previous client with “killall dhclient”).
g. Can you think of what the patch did?
h. Investigate the dhcp relay binding table on the switch with “run show dhcp relay binding”
i. You can issue “ip addr” on pc-isotdaq-3 to verify that the ip address was added to the eno1 interface.

2. Let’s have a quick look at DNS (domain name system) that is used for name resolution in networking. By name resolution we mean resolving names, such as pc-isotdaq-01 to an ip address, ip addresses to names and to find various types of infrastructure information. Let’s explore this functionality briefly from pc-isotdaq-3
    a. student@pc-isotdaq-3: host pc-isotdaq-1
    b. student@pc-isotdaq-3: host pc-isotdaq-01
    c. What can you decipher from the response? Can you think of a reason as to why the query went against the FQDN (pc-isotdaq-1.isotdaql.lab?
    d. How does the dns client know from who to ask about isotdaq.lab? Query for the NS record for isotdaq.lab -domain
        i. student@pc-isotdaq-3: host -t ns isotdaq.lab
    e. What if I wanted to send an email to someone at isotdaq.lab? How would I know to which machines to send the SMTP traffic?
        i. student@pc-isotdaq-3: host -t mx isotdaq.lab

III. Bonus Questions

- What happens if you have a static entry in the ARP cache and the NIC for that target computer is changed?
- How could you find the physical address of the Ethernet card installed on your computer?
- What is the purpose of the TTL field in the IP packet header?
- From pc-isotdaq-1, if you ping pc-isotdaq-2 using: ping -s 1800 pc-isotdaq-02 this works, but ping -M do -s 1800 pc-isotdaq-02 does not work?

IV. Useful definitions and glossary

MAC address: (Media Access Control) unique identifier associated with a physical network interface. Also named hardware address or Ethernet address in the case of an Ethernet device.

IP address: (Internet Protocol) numerical identifier associated with a device connected to an IP network.

Most of the time the MAC address is provided by the device manufacturer and never changes, and the IP address is the logical identifier associated to this device by the network manager according

to the purpose and location of the device.

Switch: network device that interconnects devices using frame-based switching at the data-link layer (layer 2). For Ethernet, the frame header contains the destination MAC address which allows the switch to determine the physical port to send frames to.

Router: network device that interconnects LANs using network-layer (layer 3) mechanisms. IP makes use of IP addresses and routing tables to implement such mechanisms.

LAN: (Local Area Network) limited-area computer network. It usually consists of devices interconnected via a network switch. Any device belonging to a specific LAN can communicate with any other within this LAN without the need for routing mechanism. A LAN is equivalent to a broadcast domain: broadcast messages reach every device in the LAN.

VLAN: (Virtual Local Area Network) broadcast domain logically created at the data-link layer from a larger physical broadcast domain. For Ethernet, VLANs are implemented with a specific Tag field in the frame header (802.1Q).

SNMP: (Simple Network Management Protocol) protocol to monitor and control network devices. SNMP defines a structured organization of network-related information and the ways to request it from a device. Typically network switches and routers implement SNMP to enable access to monitoring information (traffic metrics, errors, etc.).

RRD tool: (Round-Robin Database Tool) time series database implementing a circular buffer strategy to enforce constant footprint. Widely used to store monitoring information, especially for networks.

QoS: (Quality of Service) the whole set of mechanisms used to monitor and control the performance of networks. Metrics usually include throughput, latency, packet loss, etc.

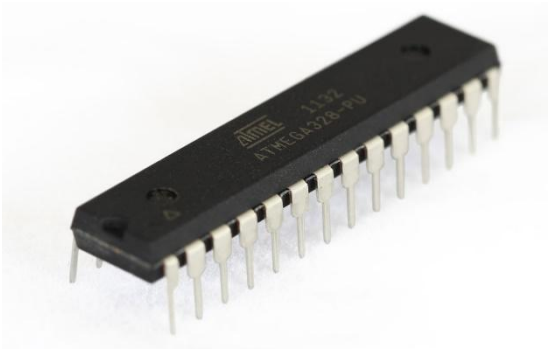
DSCP: (Differential Service Code Point, or DiffServ) feature of the IP protocol to classify and manage network traffic. DSCP uses 6 bits in the 8-bit DS field of the IP header to indicate the class of a packet, and network devices may use this information to handle different classes of packets with different policies. Examples: low latency, bandwidth constraint.

DHCP: Dynamic Host Configuration Protocol. Protocol that dynamically allocates unique IP addresses and other network parameters (e.g. hostname, default gateway) to hosts on demand.

Lab 10: Microcontrollers Exercise

Tutor: Cristóvão Beirão da Cruz e Silva *(c.beirao@cern.ch)*

Developer: *Maurício Féo Rivello (m.feo@cern.ch)*



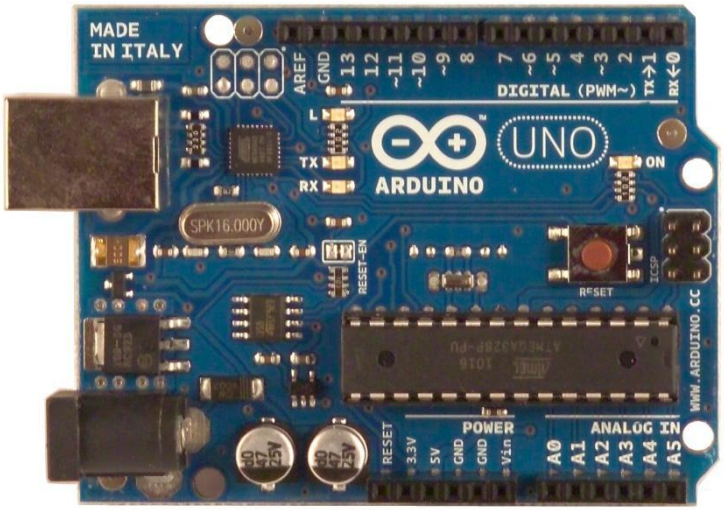
Introduction:

Microcontrollers are small computers with all its components (CPU, memories, peripherals) integrated on the same chip. Apart from their capability of processing data, they are low power, usually inexpensive devices that easily interfaces with sensors and actuators, making them perfect to use in embedded systems.

In this lab we are going to learn the basics about microcontrollers, how to use and program it, as well as common applications and explore the most relevant peripherals through the hands-on exercises and a challenge.

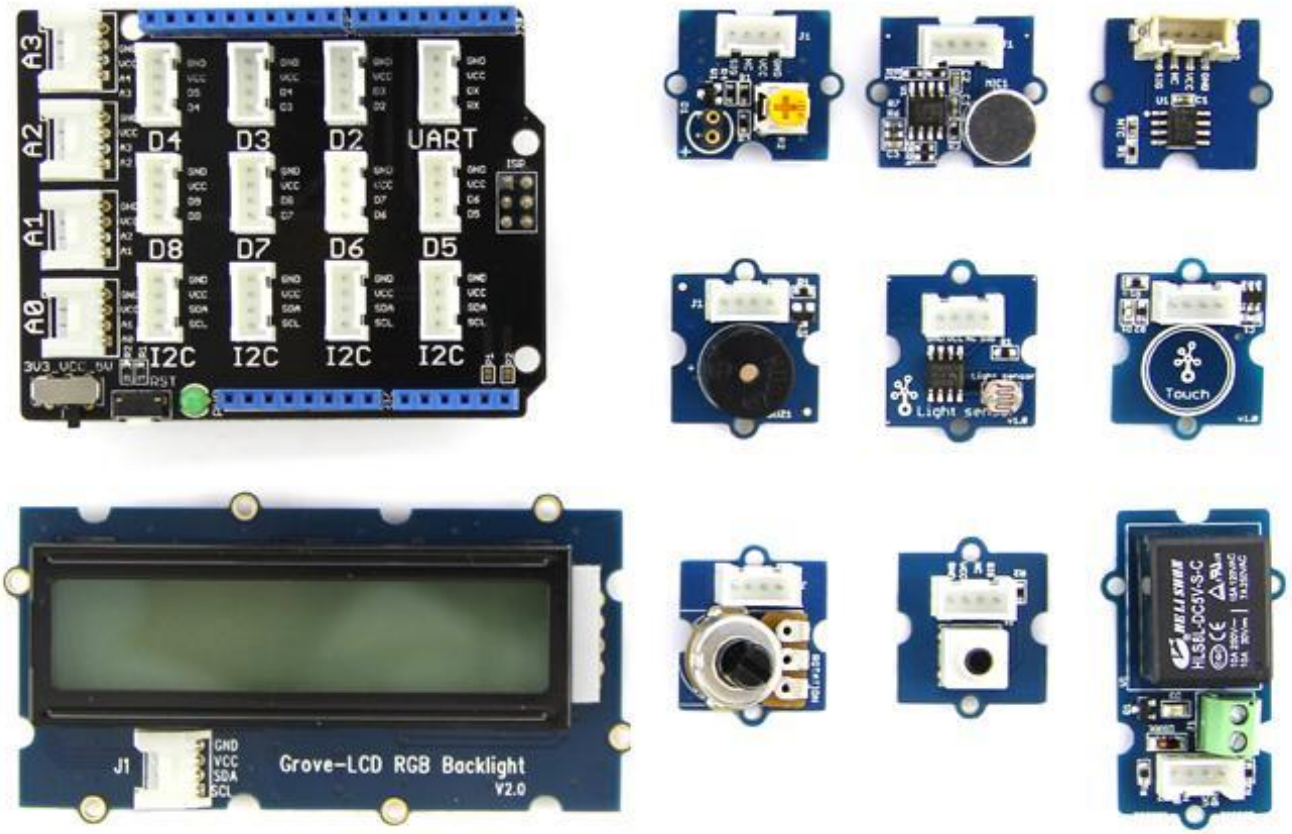
Arduino:

Arduino is an open-source electronic prototyping platform based on easy-to-use software and hardware. In short, it is the most popular microcontroller development board, with a lot of "shields" (extension boards that you pile up on top of the Arduino) to add functionalities and libraries to use the most common sensors and devices used with microcontrollers.



Grove Starter Kit:

Together with the Arduino, we are going to use the Grove Starter Kit, which provides a few gadgets (sensors, actuators and a LCD display) together with a shield and cables that provides a nice plug-and-play interface to ease and speed up the process of wiring the gadgets together.



Setting up the Arduino.

In case you want to set up the Arduino on your own laptop, the Arduino official webpage provides a very simple quick start guide. Basically you just have to download the software, plug the board on the USB port and install it. For further details, visit: http://arduino.cc/en/Guide/HomePage

Before you start:

The Arduino software and drivers will be already configured on the lab PC. Plug the Arduino board on the USB port of the PC and start the Arduino IDE (desktop shortcut). Select the Board you are using (Uno) under the IDE Menu "Tools". Then select the Port under the same menu. The Port number depends on which USB the Arduino is connected to and it is listed after you plug it in.

Hands On Exercises

We are going to do a few exercises to learn the basic functions of Arduino and then, using the hardware provided, we are going to implement a simple project to solve a challenge with what we have learned so far. As you do the exercises, save your code because you might reuse it in the project ;-)

Visit the site feo.dev/isotdaq for a summary of the functions that you will mostly use during the exercises. For detailed information about the main functions from the Arduino libraries, please refer to: www.arduino.cc/en/Reference

Exercise 1: Blinking a LED.

Blinking a LED is the microcontroller equivalent of printing "Hello World". The basic structure of an Arduino program is very simple: It must contain a **setup()** and a **loop()** function. Open the Arduino software, start a new program and write the following structure:

```
void setup() {
  ...
}

void loop() {
  ...
}
```

The **setup()** function is executed once every time the Arduino is powered on. As the name suggests, it should be used for setting up your application, like initializing classes and variables, declaring pin modes, etc.

The **loop()** function keeps being executed in loop ad eternum. This is where the main logic of your program should be.

To get started, let's build a program to blink a LED. There is already a LED attached to pin 13 on every Arduino. You should first of all declare the mode (output or input) of the pin to be used with the following function:

```
pinMode( [pin_number] , [OUTPUT/INPUT] );
```

In our case, the pin_number is 13 and the mode is OUTPUT. So to blink the LED we can write a HIGH and then a LOW signal to the pin 13, adding a delay between each command.

```
digitalWrite( [pin_number], [HIGH/LOW] );
delay( [milisseconds] );
```

Now try to make yourself a program to blink the LED on pin 13 once every second.

Exercise 2: Reading the state of a Push-Button.

The same way we can declare a pin as output and write a state (HIGH or LOW) to it, we can also declare one as INPUT and read its state with the following function:

```
boolean_variable = digitalRead( pinNumber );
```

It returns TRUE or FALSE (HIGH or LOW). Let's use it to read the status of the Push-Button from the Grove kit, which can be connected to any of the digital pins of the Arduino. In the Grove shield, these are the connectors marked with the letter D.

Use the code from exercise 1 and the LED to identify the pressing of the button.

Exercise 3: Serial communication.

In this exercise, we're going to use the Arduino Serial library to send and receive characters from the PC using the Serial Monitor tool of the Arduino IDE. First thing to do in your code is to initialize the Serial with the following command:

```
Serial.begin( 9600 );  // (9600 is the baud rate).
```

As it only needs to be executed once, it should be on the setup() function. Now there are 3 more important functions to learn. The available() returns whether there is a character available to be read:

```
boolean_variable = Serial.available();
```

The read() reads into a variable a single character from the serial buffer, and the println() works like in C, printing on the serial port a string. It also converts numbers into characters. The Serial library is very handy and there are more functions. For reference visit: http://arduino.cc/en/Reference/Serial

```
byte inByte = Serial.read(); // reads into inByte a character from the buffer.
Serial.println("Hello World");  // Writes a string to the serial port.
```

Now let's write a code that does the following:

1) Toggles the LED whenever the button is PRESSED DOWN. (Not when released)
2) Prints to the Serial port for how long the LED has been ON whenever it is turned OFF.

You can make use of one of the time functions, like micros(), which returns the amount of microseconds since the microcontroller started:

```
unsigned long var = micros();
```

Once you succeed, what about controlling the LED from the Serial Monitor as well? Use the described functions to try to control the LED from bytes sent to the Arduino from the Serial Monitor.

Exercise 4: Reading an analog input.

The Arduino UNO has 6 analog inputs. Reading one of them is as easy as reading a digital pin. It returns an integer value ranging from 0 to 1023. The scale varies from 0V to a reference voltage, which is by default 5V (but can be changed). In short: 0 -> 0V; 512 -> 2.5V; 1023 -> 5V.

To read an analog pin, use the following function:

```
int integer_variable = analogRead( pin_number );
```

The Grove Starter Kit has four analog sensors: a sound sensor, a light sensor, a temperature sensor and a rotary switch. Plug any of them into one of the connectors labeled with the letter 'A' and try the following code (don't remove the code from the previous exercises as it will be used again):

```
int analog_value;
void setup() {
  Serial.begin(9600);
}
void loop() {
  analog_value = analogRead( plug_number );
  Serial.println( analog_value );
  delay(200);
}
```

Open the Serial monitor and see the results and how they change when you interact with the sensor used. Try the Serial Plotter as well.

Exercise 5: Interrupts

Note what happens when you try checking the time between presses of the button from exercise 3 while running the code from exercise 4. Did you note that the code on the loop cannot identify properly when the button is pressed when the processor is “stuck” on the delay(200) function?

It happens because you are reading the button by polling it’s state at every loop. For critical applications where you need a precise timed response, interrupts should be used instead.

I/O interrupts are implemented in a very simple way by Arduino. You can set up an interruption in a pin using the following function (Arduino UNO only supports interrupts on the pins 2 and 3):

```
attachInterrupt(      digitalPinToInterrupt(      pin      ),      function_To_Be_Called,
RISING/FALLING/CHANGE );
```

Now whenever the value of the pin rises or falls or changes (depending on your choice) the function passed will be called. The name of the function is arbitrary and you should define it in your code:

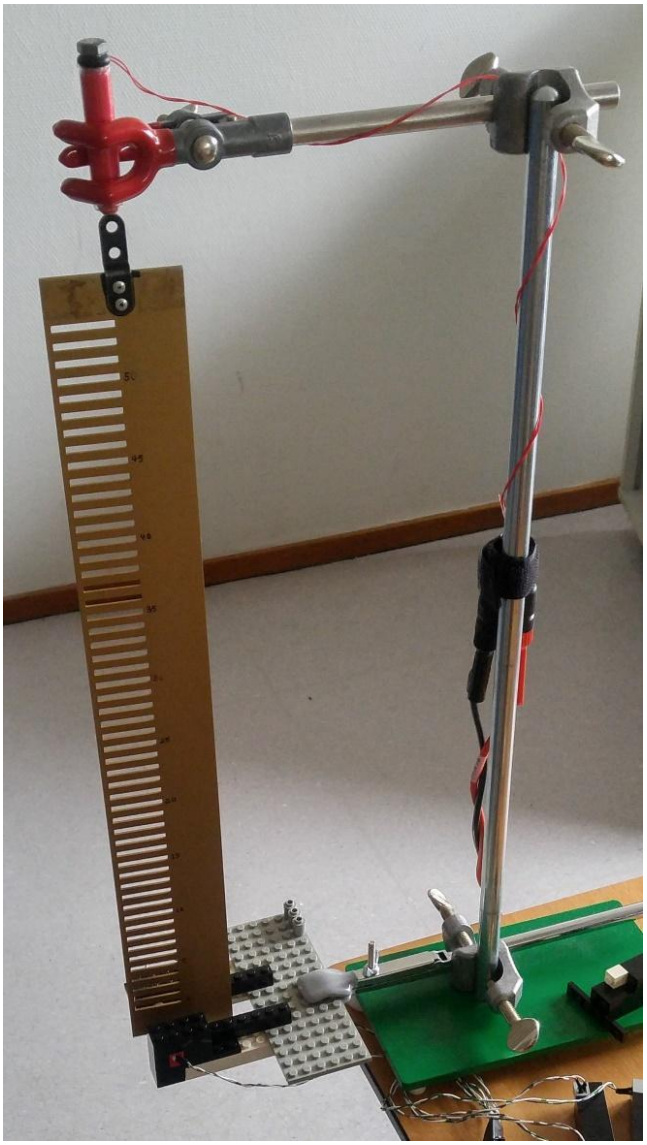
```
void function_To_Be_Called(){
// code to be executed on the interrupt
}
```

Reimplement the printing of the time in between presses of the button but now using interrupts instead of polling.

Challenge: Measuring the acceleration of gravity.

In this challenge we are going to use all what we learned above to measure the acceleration of gravity using a drilled bar that falls through an infrared beam. The space in between the first edge of each hole is 7.2mm on average. The precise measurements can be found on feo.dev/isotdaq
We have an infrared LED and an infrared phototransistor that works like a digital input for the Arduino (just like the push button). For a smooth drop of the bar, we can use an electromagnet controlled from the relay board of the Grove Starter Kit.

Given all that we learned and the apparatus available, how could one calculate the acceleration of gravity? For any help setting up the equipment, ask the tutor. And good luck! ☺



Lab 11: Storage Exercise

Overview

The aim of this lab is to provide an overview of the configuration and the evaluation of a storage setup.

Objectives

- partition storage units
- setup raid systems
- performance measurements
- evaluate different raid strategies
- evaluate different storage technologies

Tools

dd

The linux dd tool allows you to make copies of files at a block level. Its basic syntax is :

```
dd if=/path/to/input/file of=/path/to/output/file bs=X count=Y
```

Where X is the block size of the individual transfers, and Y the amount of blocks you want to copy. We recommend 32M as X value.
The seek option allows you to skip a certain amount of blocks at the start of the output.
The skip option allows you to skip a certain amount of blocks at the start of the input.
The oflag is used to set particular flags that are used on the output stream. In our case the 'direct' option is useful. It forces the operating system to not use the write behind cache on the stream.

fdisk and sfdisk

Be very very careful. These tools can very easily wipe out the entire operating system if used on the wrong disk. Make sure you are only working on /dev/isotdaq/XXX

The fdisk tool is used to manipulate the partition table of a disk. The tool has an interactive shell and the important commands for the following exercises are:
n: create a new partition
d: erase a partition
w: write the new partition table to disk
p: show the current partition layout
h: help

The sfdisk tool allows you to dump the partition table of a disk into a file using the -d option, and

then to apply the same schema to another disk using the redirect operator (<).
For example: sfdisk -d /dev/isotdaq/disk1 > file //to dump the partition table into file

sfdisk /dev/isotdaq/disk2 < file //to read a partition table from a file

mdadm

The mdadm tool is used to manipulate the Linux software raid devices. Its main running modes are 'Create' and 'Manage'. In order to create a new raid, the 'Create' mode is obviously to be used. You need to provide it with information about the raid level you want to create, and on which device it will reside.

Create a raid array:

```
mdadm --create=md<x> --level=<x> --raid-devices=<N> <device1> <device2> …
<deviceN>
```

Stop a raid array:

```
mdadm --misc --stop /dev/md/md<x>
```

fio

Fio is an advanced tool for characterising IO devices. It can be used to simulate different IO loads and profiles and evaluate disk performances. In our case we will use it to measure iops at a fixed block size. The following syntax will be enough for all of the exercises:

```
fio --rw=<opt1> --bs=<opt2> --runtime=<opt3> --filename=<opt4> --direct=1
--ioengine=libaio --name=isotdaq
```

opt1: randread or randwrite
opt2: 4096
opt3: 60
opt4: /dev/isotdaq/disk<X> or /dev/md/md<Y>

Exercises

Exercise 1: Determine the raw throughput of a single disk

For this exercise use dd to measure the throughput of one of the hard disks in the machine for read and write performance.
For write performance use /dev/zero as input file and /dev/isotdaq/disk<N> as output.
For read performance use /dev/isotdaq/disk<N> as input file and /dev/null as output.

Use the seek and skip options of dd to measure the performance at the end of the disk too.
Use the count option to write/read only 1GB of data.
Use oflag=direct during writing.
Hint: If you use an I/O block size of 32M the end of the disk should be around 7000

Questions:

- What is the read/write throughput of the disk in MB/s?
- The oflag=direct option circumvents the operating system cache for the disk. Why is this important for this measurement?
- Which disk corresponds to which physical disk inside the enclosure?
- Optional: Usually, for disk based storage, the write throughput is the same as the read throughput. Do you have any idea why it is different in this case?

Exercise 2: Determine the IOPS of a single disk

For this exercise use the fio tool to measure the random read and write Input Outputs Per Second (IOPS) of a single disk.
Good values for the parameters are a run time of 60s and IO size of 4096. For reading use --rw=randread. For writing use --rw=randwrite.

Questions:

- What are the values for random reading and random writing for these disks?
- Why are these important values?
- Why are the reading and writing values different?
- Using the result from Ex.1: Calculate the IOPS of the throughput measurement. Why are the values for random IO so much smaller?

Exercise 3: Partitioning the disks

For the purpose of this exercise, we will create 4 partitions on each disk. They will later be used to host different raid types.

Create 4 partitions on /dev/isotdaq/disk1 using fdisk. The partitions should be of type ‘primary’, and 2 Gb each.

After this you can either use fdisk to create the same partitions on the other 3 disks or use sfdisk to dump the layout of the first disk and import it to the other three disks.

Questions:

- Make sure that /dev/isotdaq/disk<0-3>part<1-4> exist

Exercise 4: Creating the raid arrays

You will now create 4 different kinds of raid sets to measure their different properties. Use mdadm to create the following raids:
Raid0 on disk1part1, disk2part1, disk3part1, disk4part1
Raid1 on disk1part2, disk2part2

Raid5 on disk1part3, disk2part3, disk3part3, disk4part3
Raid6 on disk1part4, disk2part4, disk3part4, disk4part4

Hint: To create a raid set of a particular kind use
mdadm --create md<x> --level=<x> --raid-devices=<N> <device1> <device2> … <deviceN>

Raid levels are 0, 1, 5 and 6. For easier recognition you can use the same number <x> for the raid level and the device name.

You can create and initialize multiple arrays in parallel.

Questions:

- Use 'cat /proc/mdstat' to follow the initialisation of the raid arrays.
- Why do raid1, raid5 and raid6 need initialisation and raid0 does not?
- Explain the different sizes of the finished raid sets.

Exercise 5: Performance Measurements

In this exercise you are going to explore the different performance values of the different raid types. Use dd and fio like in exercise 1 and 2 to determine the throughput and IOPS of the four raid sets you created earlier. For the throughput you can skip the measurement for the end of the device (Why?).

Remember to use /dev/md/md<x> for your measurements and not /dev/isotdaq/...
Questions:

- What values did you expect for the different raid sets?
- Is what you got coherent with what you expected?
- Which raid would you use for a data acquisition system?
- What would you use for a normal file system/database?

Exercise 6: Failures

Remove disk1 and check if you can still access all your raid sets. Repeat the performance measurements for the raid sets that still work.

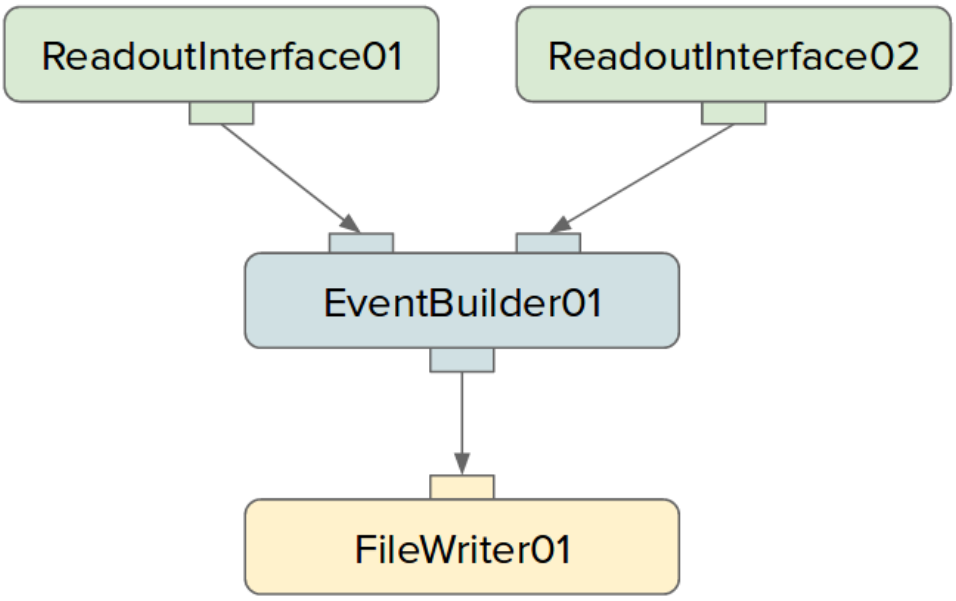
- Explain why the performance of all raid sets has deteriorated.
- Would you still choose the same raid type for your data as in exercise 5?
- Why does raid 6 exist?

Lab 12: DAQ Online Software

1. Outline

The aim of this laboratory is to develop tools to **control, configure** and **monitor** a DAQ system. The system is composed of different independent modules that simulate data acquisition applications. The control system steers the behavior of these applications via the Run Control.

Students will develop the system relying on the control, configuration and monitoring capabilities provided by a simplified version of a real DAQ system. This consists of two readout modules (ReadoutInterface01 and ReadoutInterface02), one event builder application (EventBuilder01) and one data logging system (FileWriter01).



Readout modules generate random data and write them to a buffer, waiting to be processed by the event builder application. This is similar to emulating a readout system from the DAQ of any particle physics experiment. Finally, once the events have been built, the FileWriter application writes them to permanent storage for offline analysis.

In addition, students will learn about the most common situations in controlling DAQ applications in a distributed environment and how they can be addressed. They will also learn about the main capabilities provided by the online software framework in a DAQ system.

2. DAQling framework

For this laboratory the *DAQling* framework is used. *DAQling* is an open-source lightweight C++ software framework, to be used as core for data acquisition systems of small and medium-sized experiments such as NA61/SHINE or FASER.

The framework offers a complete DAQ ecosystem, including a communication layer based on the widespread *ZeroMQ* messaging library, configuration management based on the *JSON* format,

control of distributed applications, extendable operational monitoring with web-based visualization, and a set of generic utilities. The framework comes with minimal dependencies, and provides automated host and build environment setup based on the *Ansible* automation tool. Finally, the end-user code is wrapped in so-called "Modules", that can be loaded at configuration time, and implement specific roles.

The *DAQling* framework together with its detailed documentation can be found at:

https://gitlab.cern.ch/ep-dt-di/daq/daqling

https://daqling.docs.cern.ch/

3. DAQling modules

The *DAQling* building block is the "Module", a plugin that can be loaded at run-time by the bare `daqling` executable, which then acquires its functionalities.

A Module is a class, inheriting a set of default methods from the `DAQProcess` base class, such as the `runner()`, `start()` and `stop()`.

Since we want a uniform way to control different processes, *DAQling* modules implement the Finite State Machine (FSM) paradigm with both transition commands and states that uniquely identify a specific behavior. All the applications in the system must behave according to this FSM.

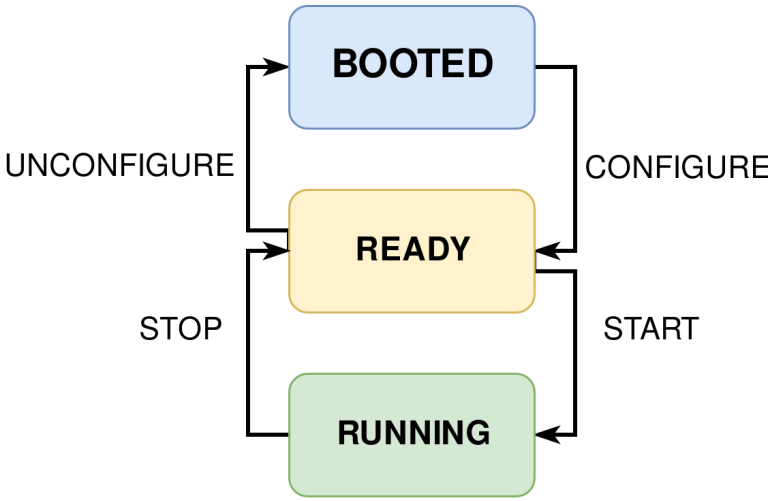
Thus, the two main concepts of the FSM are:

1. **Transition commands**

`add, boot, configure, start, stop, unconfigure, shutdown, remove`

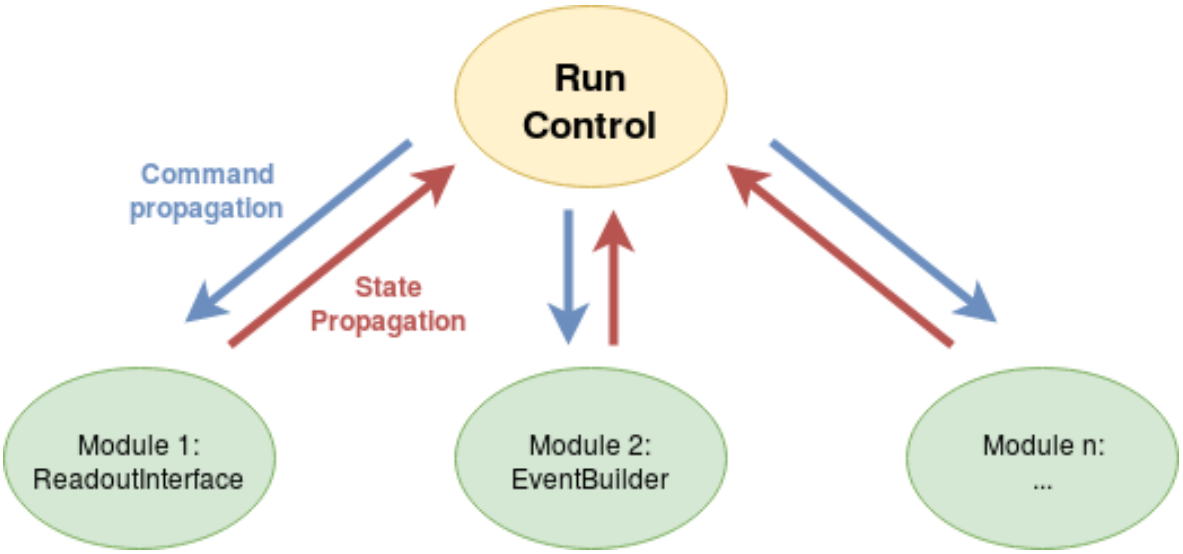
2. **States**

`NOT_ADDED, ADDED, BOOTED, READY, RUNNING`



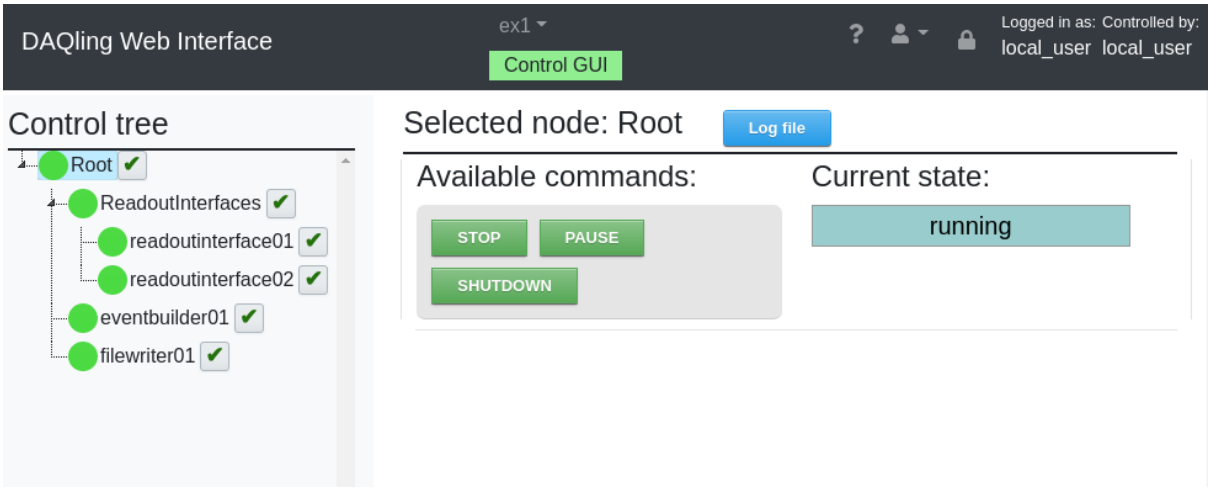
3. Run Control

The distributed control system has to manage and gather information from a set of modules running on different machines. All these modules have to be properly configured and running at the same time to provide meaningful data. This introduces the need for an entity to manage the control flow: Run Control - an application that receives FSM commands and forwards them to a set of children applications. The Run Control application must also check the proper execution of the FSM transitions, deal with common problems, etc.



The Run Control implements the same interface as every other application and is able to receive commands.

During this exercise the Web based application will be used as a Run Control. It uses the "*Supervisor*" tool (http://supervisord.org/) as process control system. The Web Interface gives the possibility to send commands to all the processes as well as monitor their states.



4. Monitoring tools

It is extremely important to monitor the data that has been collected as well as the operational information (e.g. event rates, health of the computing cluster, etc.) during data taking.

Each *DAQling* Module has its own instance of `Statistics` class which provides the interface for metrics monitoring. While defining the Module you can register the metric using the template function:

```
void registerMetric(T* pointer, std::string name, metrics::metric_type mtype)
```

in the `configure()` function of your Module, where:

- `pointer` - pointer to the variable storing the metric value
- `name` - name of the metric published to database
- `mtype` - type of the metric

For example:

```
m_statistics->registerMetric<std::atomic<int>>(&m_metric6, "RandomMetric1-int",
daqling::core::metrics::LAST_VALUE);
```

The acceptable types of variables are:

- `std::atomic<int>`
- `std::atomic<float>`
- `std::atomic<double>`
- `std::atomic<bool>`
- `std::atomic<size_t>`

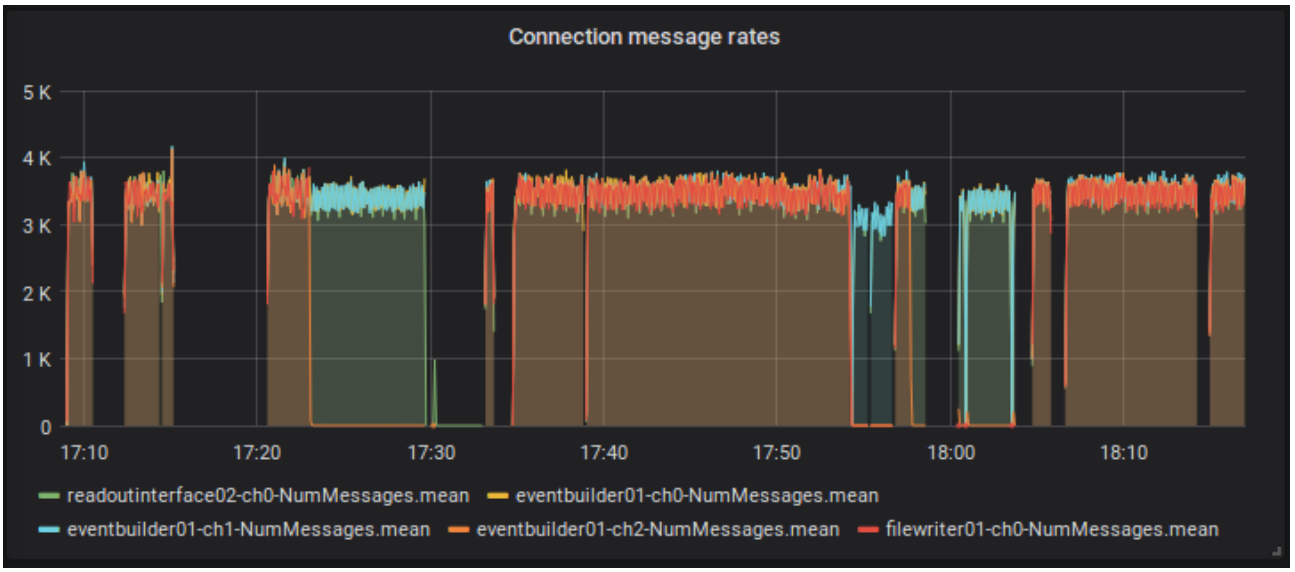
There are 4 possible metrics types:

- `LAST_VALUE` - measure the current value of metrics`
- `ACCUMULATE` - accumulate the current variable value to metric value and reset the variable

- AVERAGE - calculate the average value of metric over given time interval
- RATE - calculate the rate over a given time interval

All of them are defined in the `daqling::core::metrics` namespace.

The default metrics visualization tool is *Grafana* (https://grafana.com/) connected to *InfluxDB* (https://www.influxdata.com).



Controlling data acquisition modules

The goal of the following two exercises is to understand how to control different data acquisition applications (e.g. readout interfaces, event builder, file writer, etc.), and understand the concept of FSM and Run Control application.

Exercise 1:

Run the example application, get familiar with the code and investigate the behavior of the modules.

```
[student@isotdaq]$ cd online-software
[student@online-software]$ source cmake/setup.sh
[student@online-software]$ cd scripts/ControlGUI/
[student@ControlGUI]$ ./run.sh
```

Open *Firefox* and open the "DAQling Web Interface" at http://localhost:5000.

Choose the configuration named "ex1" and click on the open lock pad on the top-right to take control of the system.

Explore the “Control tree”, “Available commands” and “Log file” and bring the system to “Running” state.

The selected configuration can be found at:

```
[student@isotdaq]$ cd online-software/configs/ex1/
```

The `config.json` file defines which modules are loaded. It is composed of any number of “`components`”. The `"type"` field of component is the name of module which you can find here:

```
[student@isotdaq]$ cd online-software/src/Modules/
```

All the modules inherit the DAQ`Process` class. This can be found here:

```
[student@isotdaq]$ vim online-software/src/Core/DAQProcess.hpp
```

Exercise 2:

Repeat the previous exercise with a different configuration file: “ex2”.

Localize the bugs and fix them. Remember that all the modules should follow the transition commands and states defined in the FSM. Therefore, any inconsistent state should be fixed.

Instruction for compiling DAQling:

```
[student@isotdaq]$ cd online-software
[student@online-software]$ source cmake/setup.sh
[student@online-software]$ cd build && make -j
```

Operational monitoring of the data acquisition system

Exercise 3:

Select now the *InfluxDB* and *Grafana* configuration file: “ex3”.

Open a new tab in *Firefox* and open the *Grafana* dashboard (http://localhost:3000) named “DAQling demo”.

Go back to the Web Interface application (http://localhost:5000) and start the modules (go to “Running” state).

Check the plots on the *Grafana* dashboard and their behavior in different modules states.

Exercise 4:

This exercise contains CPU and Memory consumption variables already prepared. The goal is to create a new *Grafana* dashboard to visualize them.

Select the configuration file “ex4”.

The following are guidelines on how to add a new dashboard into *Grafana*:

- Create a new dashboard
- Add a new panel
- Choose variable to query

Exercise 4b (bonus):

With the help of the *Grafana* dashboard, compare the results of the configuration files “ex4” and “ex4_bonus”.

Tip

There is something wrong with the data displayed on *Grafana*. Find the origin of the difference and implement a solution to fix it.

Configuring a distributed system

Exercise 5:

The goal of this exercise is to combine several applications running on different host machines, controlling them coherently under a unified framework and monitor the overall status of the system.

Modify the configuration file from exercise 3 to have the readout applications running on two different host machines. Use the Web Interface to start the data acquisition.

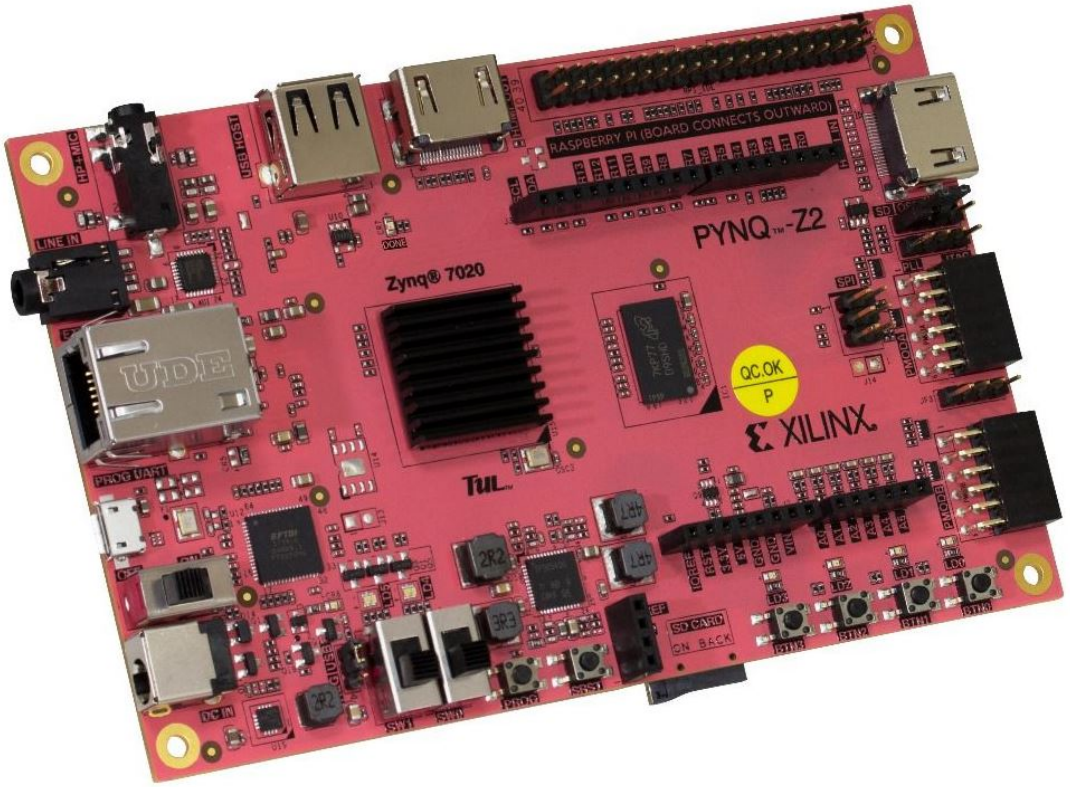
Lab 13: System on Chip (SoC) FPGA

(Version: 2.0)

Tutor:
Johannes Wüthrich (CERN) (johannes.wuethrich@cern.ch)

Lab Developer (Version 2.0):
Johannes Wüthrich (CERN) (johannes.wuethrich@cern.ch)

Lab Developers (Version 1.2):
Manoel Barros-Marin (CERN) (manoel.barros.marin@cern.ch),
Elena-Sorina Lupu (Caltech) (eslupu@caltech.edu)



1. Introduction

The aim of the **SoC FPGA laboratory** at the **International School Of Trigger Data AcQuisition (ISOTDAQ)** is to provide students a brief overview of the different stages in the SoC FPGA design workflow, as well as the knowledge when a SoC FPGA is the most appropriate core for their project. After the completion of the lab, the students should be able to understand the interaction between the two main building blocks of a SoC FPGA, the FPGA fabric and the embedded Processor (embedded CPU), and assess the challenges of implementing such a system. The students should also gain a high level understanding of the **AXI** and **AXI-Stream** interfaces used in the project and the main differences between the two protocols.

1.1 *SoC FPGA*

Processors (CPU) and **Field Programmable Gate Arrays (FPGAs)** are at the core of most Trigger DAQ systems. By integrating the high-level management capabilities of a processor and the real-time capabilities, extreme data processing, and interface functions of an FPGA into a single device we end up with a more powerful embedded computing platform. **System on Chip (SoC) FPGA** devices integrate both processors and FPGA architectures into a single chip. Consequently, they provide higher integration, lower power, smaller board size and higher bandwidth communication between the processor and FPGA. They also include a rich set of peripherals, on-chip memory, customizable logic fabric and high speed transceivers. As a result of these qualities, SoC FPGA devices are becoming more and more popular among digital electronics and software designers, especially because they also exist in a wide range in terms of cost and performance.

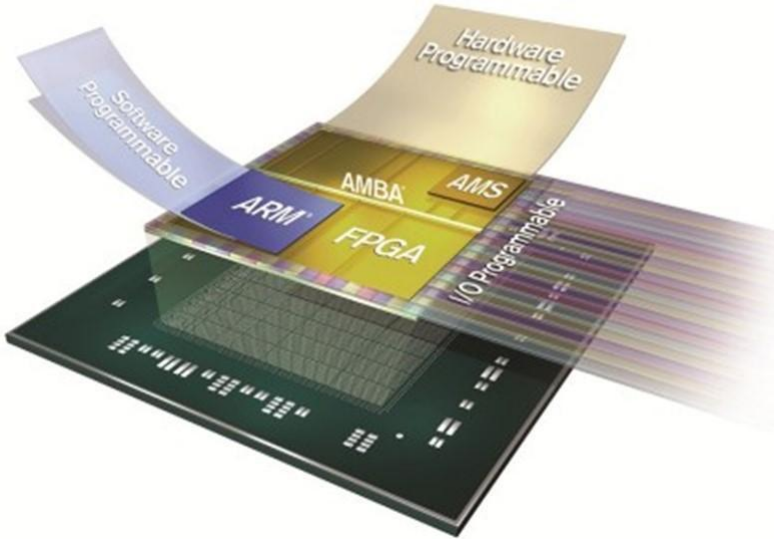


Figure 1: *Model of SoC FPGA*

1.2 *SoC FPGA workflow*

A typical SoC FPGA workflow is illustrated in Figure 2.

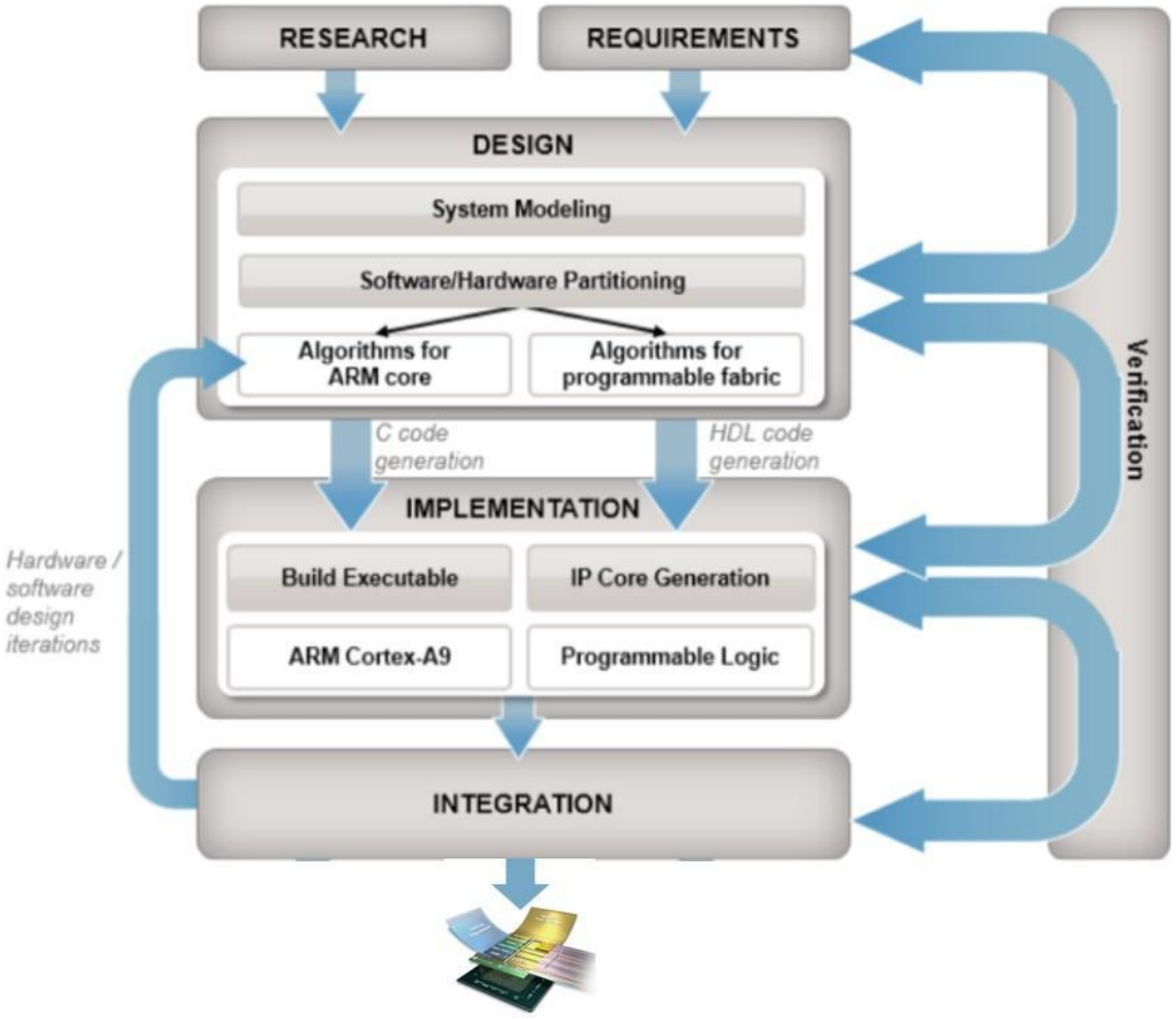


Figure 2: *SoC FPGA workflow*

2. Lab Setup

2.2 Specifications

The goal of this lab is to implement a data processing chain for a small energy dispersive X-Ray spectroscopy (EDXS) experiment. In this experiment a material sample is targeted by a particle beam. These particles kick out electrons from orbital shells. When electrons from higher orbitals relax into the empty spots, they emit characteristic X-Ray photons. Figure 3 left shows the atomic principle of EDXS.

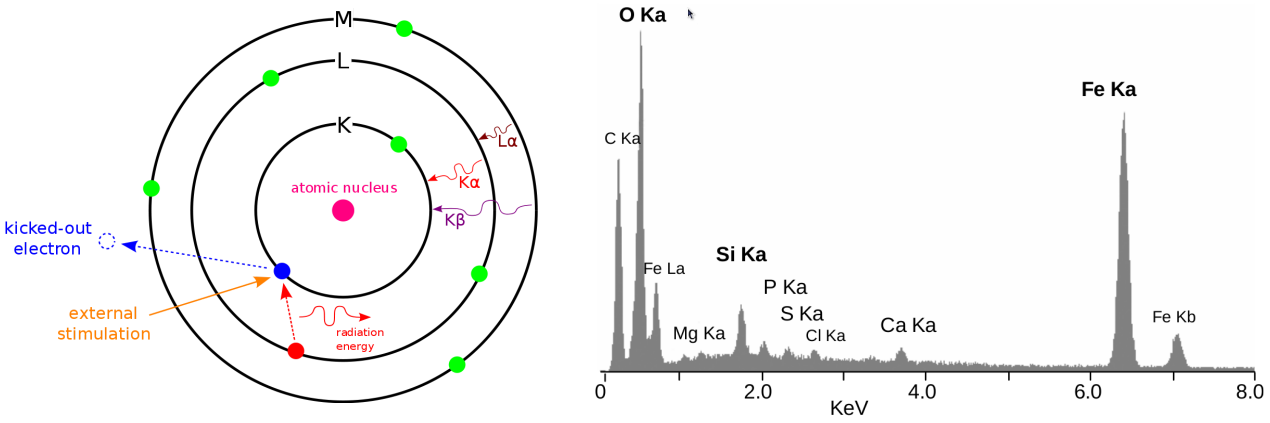


Figure 3: *Left: Principle of EDXS[1] – Right: Example EDXS spectrum[2]*

The energy of these X-Ray photons are clearly defined for each type of atom. Therefore, by measuring the energy spectrum of emitted X-Rays, the different atomic elements in our sample can be identified. Figure 3 right shows an example spectrum obtained from an EDXS measurement.

In our experiment, the emitted X-Rays are detected with a SiPM coupled to a scintillator crystal. The resulting SiPM pulses are then digitized and further processed in a digital data processing chain, which is made up of the following components:

- **Pulse Threshold**: Applies a threshold to the incoming signal in order to detect pulses.
- **Pulse Integrator**: Calculates the area under the pulse curve, which is proportional to the X-Ray photon energy.
- **Multi Channel Analyzer (MCA)**: Generates a histogram of the integrated pulse areas. This will be our EDXS spectrum.
- **CPU**: The embedded processor will control all the data treatment blocks, read out the spectrum from the MCA and plot the resulting spectra.

Figure 4 shows a high-level view of the system to be implemented.

[1] Published under the GFDL – https://commons.wikimedia.org/w/index.php?curid=3241031
[2] Published under CC BY 3.0 – https://commons.wikimedia.org/wiki/File:EDS_-_Rimicaris_exoculata.png

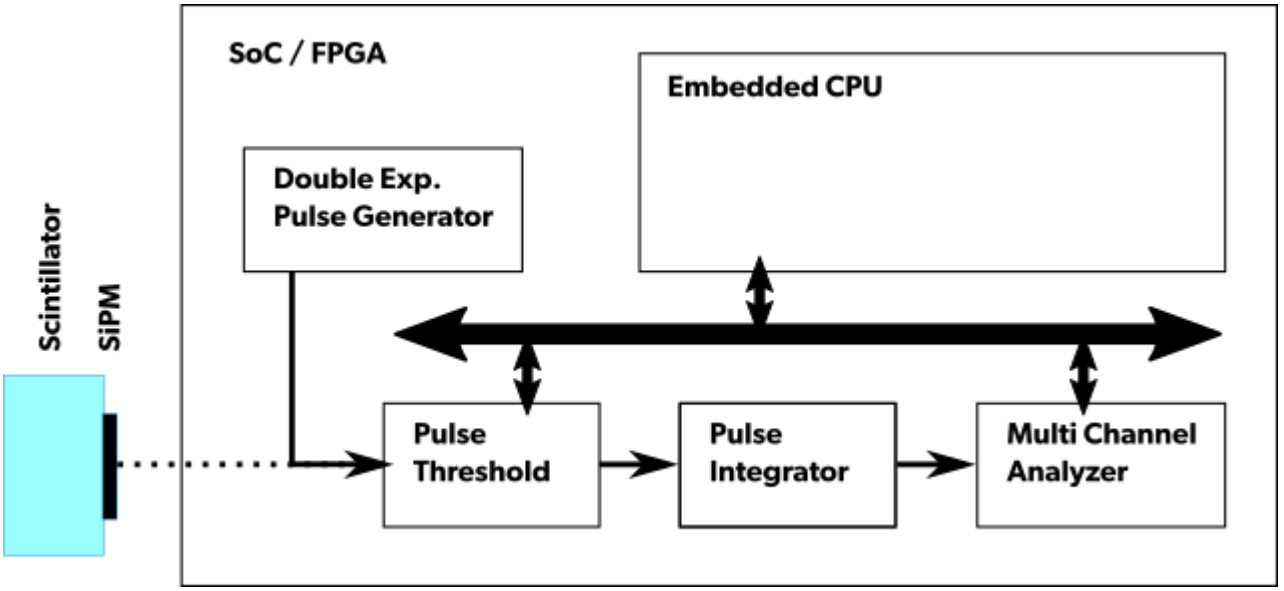


Figure 4: *High-level view of the system to be implemented*

As this is ISOTDAQ we can sadly not have an entire particle beam setup to do EDXS. Therefore, instead of measuring real SiPM pulses, we have an additional block in our FPGA (**Double Exponential Pulse Generator**) which generates fake pulses, emulating the SiPM. Appendix A) shows how the generated signals look like, and the function of each block in the data processing chain.

In the rest of the lab, we will implement the system shown in Figure 4 as a System-On-Chip on a Xilinx FPGA. The communication between the block will happen via **AXI** and **AXI-Stream** interfaces. In Figure 4, **AXI** interfaces are represented by the bold arrows, while **AXI-Stream** interfaces are represented by the thin arrows. We will later see what these interfaces do in detail.

2.3 Hardware

2.3.1 PYNQ-Z2 Board

The **data processing electronics** of our emulated HEP experiment is based on the PYNQ-Z2 board[3], which is a low-cost and high-performance System On Chip FPGA devkit. This board is based on the Xilinx Zynq-7000 family, featuring an integrated dual-core ARM Cortex-A9 processor with Xilinx7-series Field Programmable Gate Array (FPGA) logic.

The PYNQ-Z2 Board takes full advantage of the features of the Zynq-7020 SoC. It has 512MB DDR3 SDRAM and 16MB QSPI Flash on board and a rich set of peripherals including USB-to-UART, USB-Host, 10/100/1000Mbps Ethernet, HDMI, JTAG and RGB LEDs. The board also makes many of the ZYNQ IO-Pins available either via Pmod ports, Arduino compatible headers or a Raspberry Pi compatible header. This allows the direct use of many Ardiuno shields or Raspberry Pi hats. The

[3] https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html

PYNQ-Z2 Board is capable of running the Linux operating system. A Linux distribution based on Ubuntu with all the necessary drivers is provided as part of the PYNQ environment and can be downloaded from the manufacturer website. An image of the PYNQ-Z2 Board, highlighting its main components, is illustrated in Figure 5.

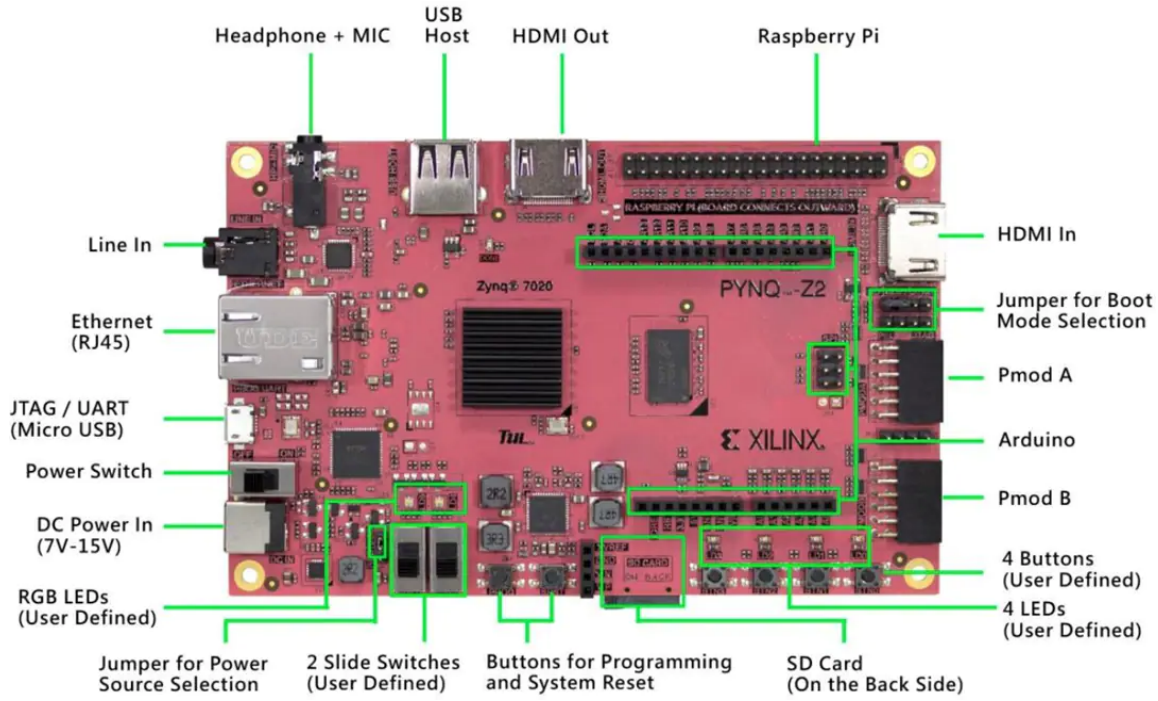


Figure 5: *PYNQ-Z2 board overview*

2.3.2 Laptop

A laptop / computer is used for running the **EDA** software necessary for designing our System-on-Chip. The laptop is further used to access the online interface running on the embedded CPU.

2.3.3 Miscellaneous

The rest of the hardware components required for this lab are the following:

- **Micro-USB to USB cable** (PYNQ-Z2 power) (see Figure 6)

Figure 6: *Mini-USB to USB cable*

- **RJ45 CAT5e cable** (PYNQ-Z2/Laptop communication) (see Figure 7)

Figure 7: *RJ45 CAT5e cable (Ethernet cable)*

2.4 Gateware

A typical SoC FPGA GateWare (GW) is composed of two main parts: On one hand, a representation of the SoC with the embedded CPU and its peripherals (e.g. I2C, timers) and on the other hand, the FPGA fabric and its hard blocks (e.g. BRAMs, Multi-Gigabit Transceiver, Multipliers etc.).
For this lab, you have to implement the GW for the PYNQ-Z2 board. As previously mentioned, the PYNQ-Z2 features a Xilinx Zynq SoC FPGA[4]. Therefore the Vivado development tools are used throughout this lab, provided by Xilinx for FPGA and SoC FPGA development.

2.4.1 SoC

The modern approach for implementing the SoC on a high level, is through **graphical schematics and wizards**. The use of graphical representations facilitates the design of complex and large architectures. An example of an SoC FPGA schematic and wizard is illustrated in Figure 8.

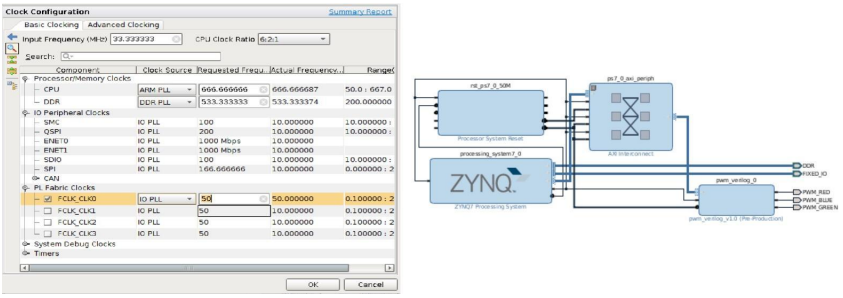


Figure 8: *SoC FPGA wizard (left) & schematic (right) example*

2.4.2 FPGA fabric

The individual modules in the FPGA fabric (also called IP cores[5]) are usually implemented using a **Hardware Description Language (HDL)**, such as Verilog or VHDL. While these languages look a bit like programming languages, they are not. Consider the following fundamental differences between programming and writing hardware:

[4] There are other vendors available, such as Altera (Intel) or Microsemi.
[5] IP here stands for *Intellectual Property*. Usually developing such blocks requires a large amount of effort and therefore companies heavily guard the source files of these blocks.

- **Programming**: You define instructions which are executed by a processor / CPU one after another.
- **Writing Hardware**: You are describing a logical circuit, where all the elements are running in parallel.

In some specific cases, it may be interesting to implement these modules using other techniques (e.g. schematics, high-level synthesis). An example of VHDL code is illustrated in Figure 9.

```
25  architecture arch_imp of PulseIntegrator_v1_0 is
26
27      signal SumxD : signed(SUM_WIDTH-1 downto 0);
28
29      type STATE_TYPE is (INTEGRATE, SEND);
30      signal StatexS : STATE_TYPE;
31
32  begin
33
34
35      FSM: process(ACLK)
36      begin
37
38          if rising_edge(ACLK) then
39
40              if ARESETN /= '1' then
41                  StatexS <= INTEGRATE;
42                  SumxD <= to_signed(0, SumxD'length);
43              elsif StatexS = INTEGRATE and s_axis_tvalid = '1' then
44                  SumxD <= SumxD + signed(s_axis_tdata);
45
46                  if s_axis_tlast = '1' then
47                      StatexS <= SEND;
48                  end if;
49              elsif StatexS = SEND and m_axis_tready = '1' then
```

Figure 9: *VHDL code example*

2.4.3 AXI Interfaces

Our final system will be made up of multiple blocks, each block with a specific function. As mentioned before, these blocks need a way to communicate with each other. We can in general distinguish two different types of data transfers between blocks:

- **(High data rate) Point-to-Point communication**: In this case, the data will follow a predefined path and flow from one block (sender) to the next block (receiver). You might think of this as a single (unidirectional) high-way, going from one city to another city.
- **(Low data rate) N-to-M communication**: In this case, many blocks are connected together on the same communication channel and a data transmission might happen between any two blocks[6]. You could think about this as a street network in a city, where you can go from any address (any building) to any other address in the city.

[6] This is not strictly true, as in some implementations (including AXI), each interface either has the role of a controller (which can initiate transactions) or a responder (which only responds to transactions).

In modern Xilinx based systems, these two forms of communication are implemented via interfaces defined in the Advanced eXtensible Interface (AXI) protocol, developed by ARM as part of their Advanced Microcontroller Bus Architecture (AMBA). The AXI protocol defines two different types of interfaces[7]:

- **AXI Stream Interface**: This implements the point-to-point communication mentioned above. **AXI Stream** interfaces are very simple to implement. In their most simple form they consist of two hand-shake signals plus N data signals. You would use a stream interface, when you want to make a fixed and unidirectional link between two blocks which should support high data rates. In our project, we for example use an **AXI Stream Interface** to transmit the data samples from the **Pulse Threshold** to the **Pulse Integrator** block. Properly implemented, **AXI Stream Interfaces** allow transmitting data on every clock cycle, hence the high data rate. Figure 10 shows a simple **AXI Stream** transaction.

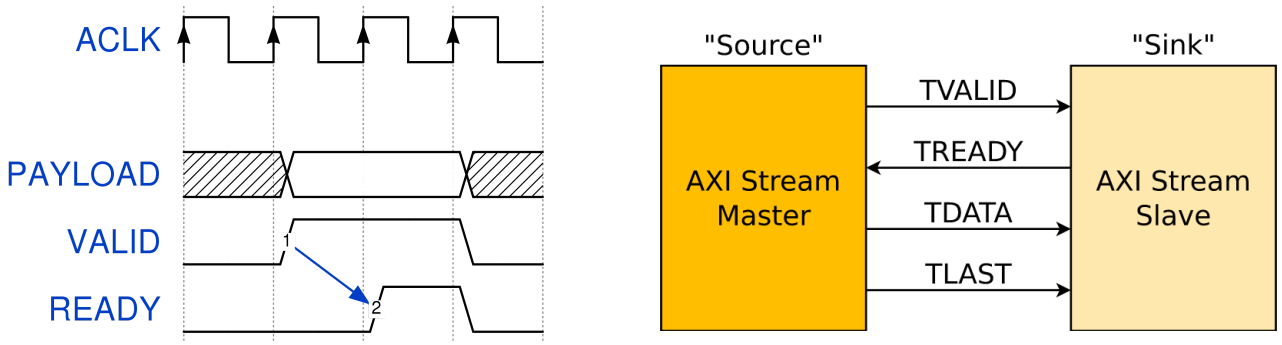


Figure 10: *Left:* ***AXI Stream*** *timing diagram[8] – Right:* ***AXI Stream*** *connections[9]*

- **AXI Interface**: This implements the N-to-M communication mentioned above. **AXI** implements a bus structure, which means that transactions include not only data, but also an address corresponding to the data to be accessed. There are two different types of data transfers: **Read** operations and **Write** operations. During a **Read** operation, a controlling block sends out an address from which it wants to read data. A responding block, to which this address belongs, then responds with the corresponding data. During a **Write** operation, a controlling block sends out some data and an address, to which it wants to write this data. The responding block, to which this address belongs, then receives the data to be written and stores it locally. Given this type of question and response style communication, **AXI** data transfers usually take up multiple clock cycles. Figure 11 shows a high level view of **AXI** read and write transactions.

[7] Technically there are three protocol definitions in the AXI specifications, the third one being the **AXI Lite Interface** which implements a subset of the full **AXI Interface** functions. But the exact differences between **AXI** and **AXI Lite** go beyond the scope of this lab. If you are curious, ask your tutor.
[8] Published under CC BY-SA 4.0 – https://en.wikipedia.org/wiki/File:AMBA_AXI_Handshake.svg
[9] From https://zipcpu.com/dsp/2020/04/20/axil2axis.html

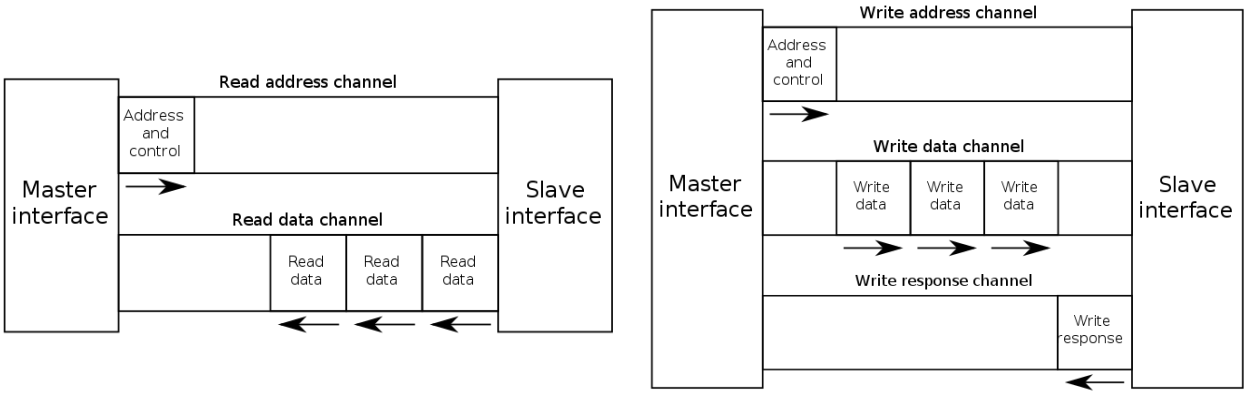


Figure 11: *Left:* ***AXI*** *read transaction – Right:* ***AXI*** *write transaction*[10]

2.5 Software

The embedded CPU of the SoC requires software for performing the tasks assigned to it. There are two main approaches when implementing software for a SoC. This software can be either **stand-alone** or executed over an **Embedded Operating System (EOS)**, such as embedded Linux. As previously mentioned, the SoC FPGA of our emulated experiment is based on a Xilinx Zynq SoC/FPGA. Xilinx, as part of the Vivado design suite, supplies a dedicated SDK for developing software in the stand-alone approach. In our case we will be using the EOS approach, making use of the Linux operating system supplied as part of the PYNQ environment. PYNQ is an open-source project from Xilinx, which makes it easy to get started with Xilinx based SoC, and which can be easily programmed in Python (hence the P in PYNQ)[11].

2.5.1 Stand-alone

In the stand-alone approach, user software is directly executed on the embedded CPU, just having the software drivers between the hardware and the software scripts. This approach is very useful when working with **single-threaded** or **real time** systems, as it simplifies the implementation and enables time determinism. The different tiers of the stand-alone software approach are illustrated in Figure 12.

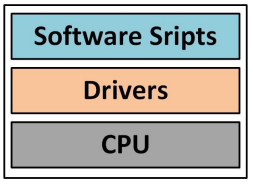


Figure 12: *Different tiers of the stand-alone software approach*

[10] Published under CC BY-SA 4.0 – https://en.wikipedia.org/wiki/File:AXI_read_channels.svg and https://en.wikipedia.org/wiki/File:AXI_write_channels.svg

[11] http://www.pynq.io/home.html

Stand-alone software is often written in C or C++, sometimes making use of a (very slim) real time operating system (RTOS), such as FreeRTOS[12]. This gives you the biggest amount of flexibility, but comes at the price of increased effort for software development.

2.5.2 Embedded Operating System

The use of an embedded operating system (EOS) is required (or highly recommended) when implementing systems **running several software processes in parallel**. In this case, the integrated arbitration capabilities of the EOS will handle the execution of the parallel processes without requiring interaction from the developer. This approach comes at the cost of a more complex setup. Adapting for example a Linux kernel for a SoC is not a trivial task and requires extensive previous knowledge. Luckily SoC / FPGA manufacturers often supply example implementations of such embedded operating systems. This is the case for the PYNQ environment used in this lab, which is based on Linux. The different tiers of the EOS approach are illustrated in Figure 13.

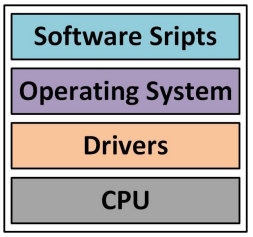


Figure 13: *Different tiers of the EOS approach*

[12] https://www.freertos.org/

3. Lab Exercises

3.1 Hardware assembly

The hardware connection of the SoC FPGA lab is illustrated in Figure 14.

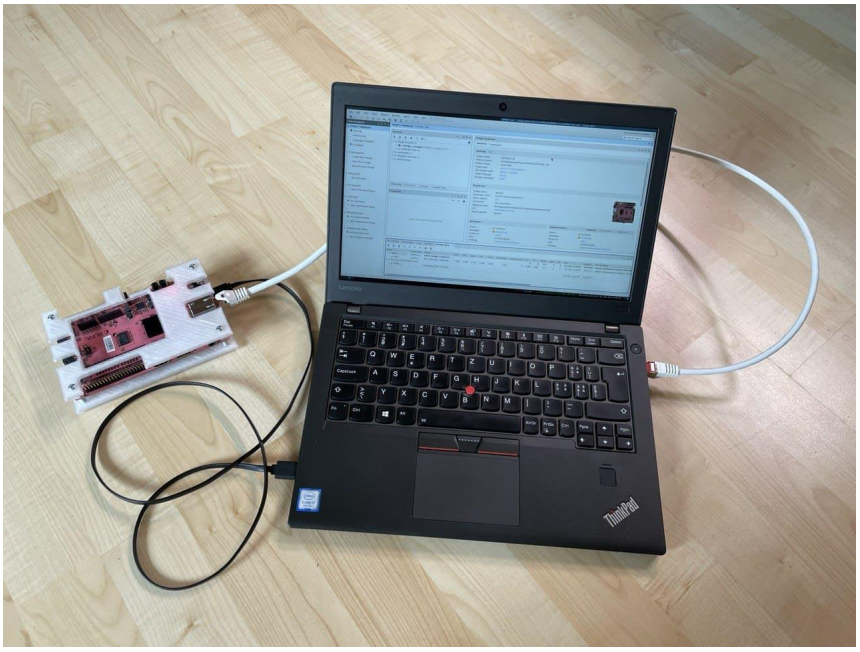


Figure 14: *SoC FPGA lab hardware*

- Ensure that the PYNQ board is connected to the Laptop / Computer via the Ethernet cable. We will later access and program the FPGA via the Jupyter web interface running under the Linux operating system on the embedded ARM CPU.
- Ensure that your board is powered on by connecting the Micro-USB to USB cable from the computer to the JTAG / UART Micro-USB socket. This connection provides power to the PYNQ board and would allow us to interact with the serial interface connected to the CPU (See Figure 15).

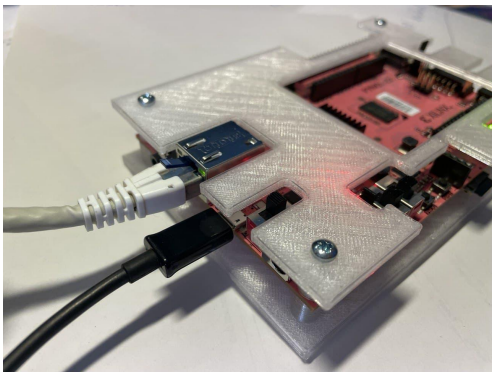


Figure 15: *Ethernet and Micro-USB connection to the PYNQ-Z2*

3.2 Gateware development

The Gateware for this lab is developed using **Vivado**, the vendor specific EDA software from Xilinx.

3.2.1 Project setup

1. Open an Ubuntu terminal (CTRL+Alt+T).
2. First we create a new folder for your project, by using the following terminal commands:

```
mkdir -p /home/isotdaq/SoC-Lab/Group[N]
cd /home/isotdaq/SoC-Lab/Group[N]
```

   Please replace **[N]** with the number of your lab group!

3. Type the following to launch the Vivado Design Suite:

```
vivado &
```

4. Create a new project and call it **SoC-LAB-Group[N]**.

   The location of your project should be `/home/isotdaq/SoC-Lab/Group[N]`

   Click **Next** and tick **RTL Project** as well as **Do not specify sources at this time**.

   Press **Next**, and now you will be asked to specify the hardware for which you are developing a project. Under **Boards**, select the **pynq-z2** entry (as shown in Figure 16).

   Press **Next** and you see a summary of the new project as well as the selected hardware. Continue with **Finish**.

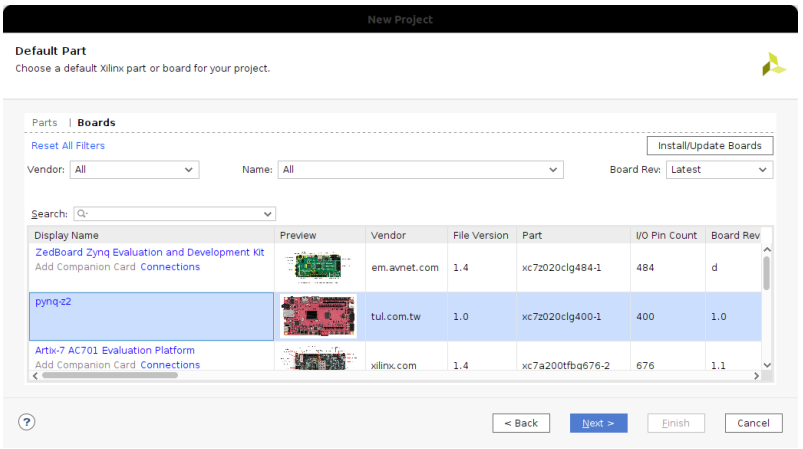


Figure 16: *Hardware Selection*

The starting page of the project should be like in Figure 17.

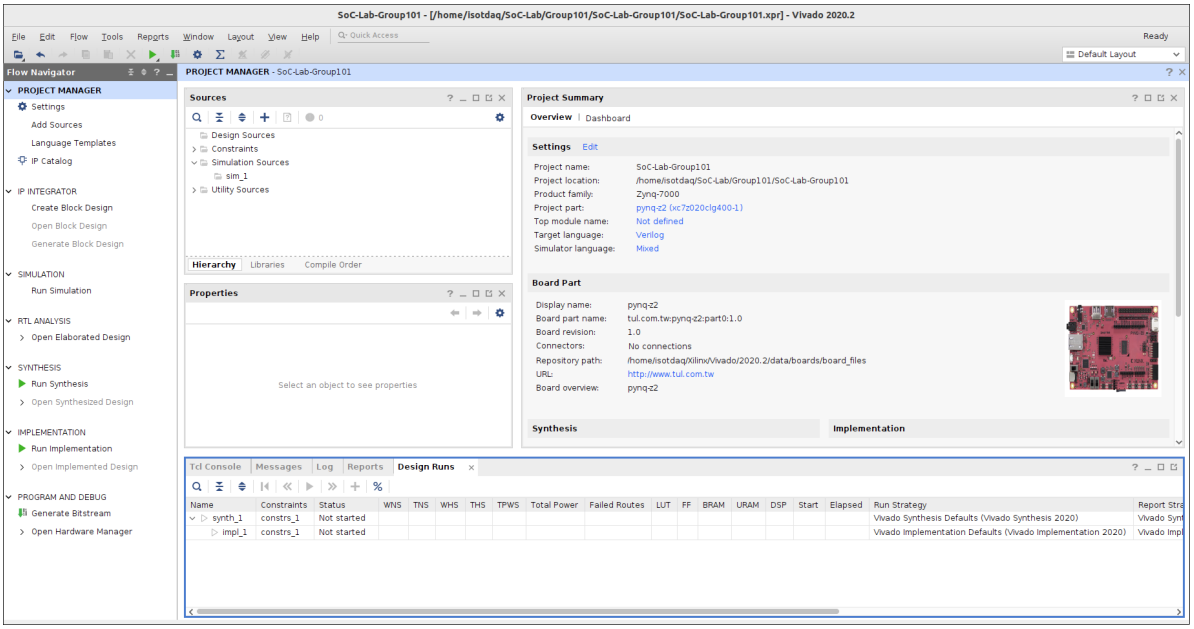


Figure 17: *Initial Project Structure*

3.2.2 Block design for SOC

1. Create the Block Design

   On the left of the window, in the **Flow Navigator**, choose **Create Block Design**. You can leave the default name and click **OK**.

2. We prepared multiple IP cores for you, which will implement the functions of the data path.

   In order to use them later on, we need to add the prepared IP cores into the repository.

   In the menu go to **Tools -> Settings** and in the settings window to **IP -> Repository**.

   Use the **+** button to add the following folder to the available repositories:

   `/home/isotdaq/IP-Repository`

   Click **OK** twice to close the settings window.

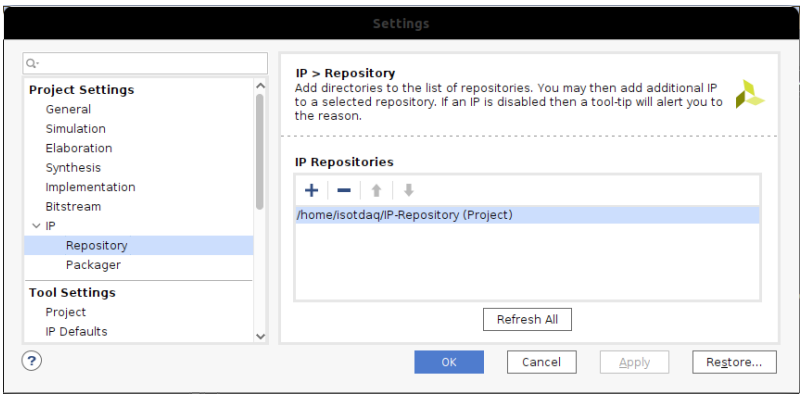


Figure 18: *Settings view to add an IP Repository*

3. First we will add the representation of the embedded ARM CPU to our block design.

   In the **Diagram** panel, use the **+** button to add our first block, the **ZYNQ7 Processing System**. Select it and press **Enter**.

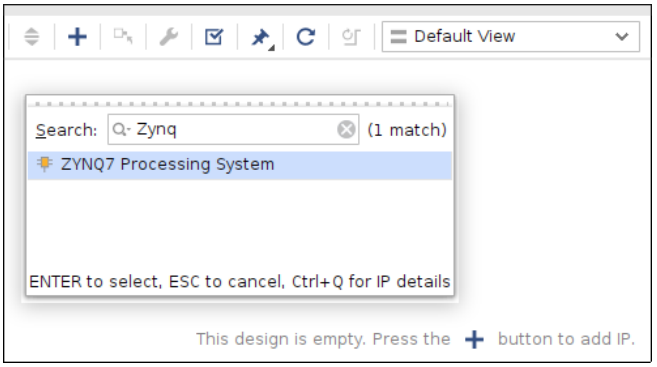


Figure 19: *Adding the ZYNQ7 Processing System block*

   At the top, Vivado now proposes to **Run Block Automation**, which will do some default configuration of the system. Run this and confirm with **OK** in the dialogue.

   Double click the **ZYNQ7 Processing System** block, which opens its configuration window.  Have a look at the presented schematic view of the embedded CPU and discuss it with the tutor. Also have a look at the **Clock Configuration**. Finally, close the window with **OK**.

4. Creating the AXI interconnect structure which allows us later to communicate from the CPU to the data processing blocks.

   Add an instance of the **AXI Interconnect** block. This block will later enable the communication from the CPU to the data processing blocks.

   Double click the **AXI Interconnect** block and set **Number of Master Interfaces** to **3** as well as **Interconnect Optimization Strategy** to **Minimize Area**. Confirm with **OK.**

   Now connect the **M_AXI_GP0** port on the **ZYNQ7 Processing System** to the **S00_AXI** port on the **AXI Interconnect**.

   At the top Vivado now proposes to **Run Connection Automation**. Click on it and select **All Automation** at the top of the list. Start the process by clicking **OK**.

   The **Run Connection Automation** makes a lot of default connections. In our case of a simple system these connections are ok. But in a more complex system, these connections would need to be made manually!

Now your block diagram should look like in Figure 20.

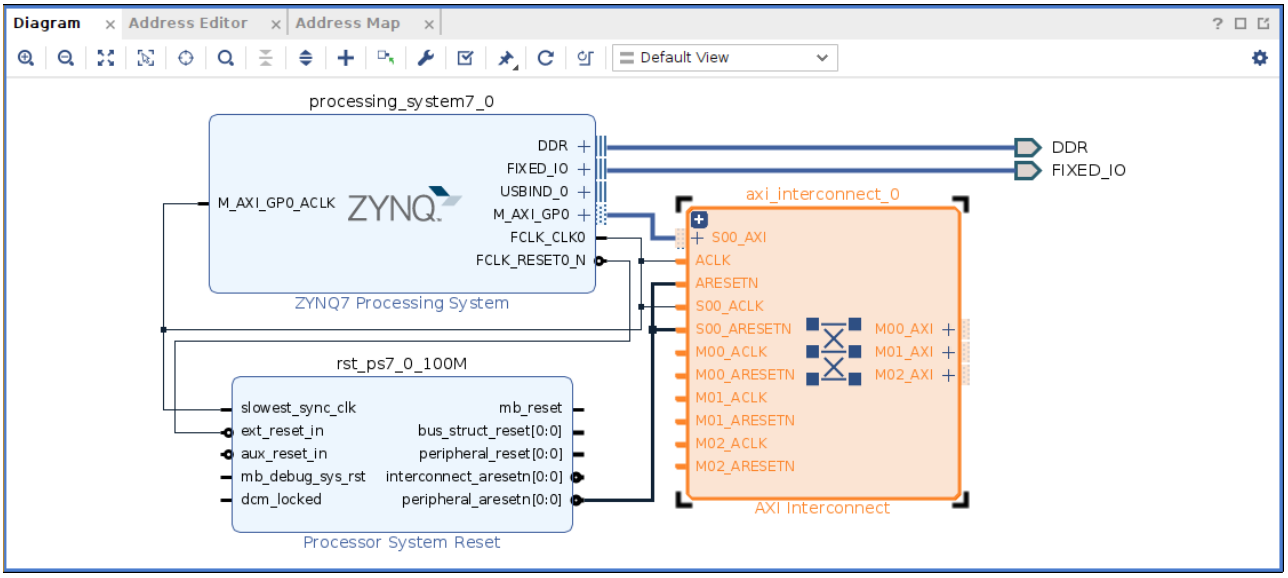


Figure 20: *Block diagram after adding the interconnect and the connection automation*

Note: If your block diagram looks very messy with many lines crossing each other, you can manually move the blocks around to re-arrange them better. You can also use the **Regenerate Layout** button at the top of the **Diagram** panel.

3.2.3 Block design for the data processing in the FPGA fabric

Now it is up to you to add the necessary blocks to create the data processing chain. We implemented the following blocks for you to use:

- **DoubleExpPulseGenerator**: Generates the fake SiPM signal pulses.
- **PulseThreshold**: Applies a threshold to the recorded pulses.
- **PulseIntegrator**: Calculates the area under the curve for each pulse passing the threshold.
- **MultiChannelAnalyzer**: Creates a histogram of the calculated pulse areas.

In order to implement the data processing chain, follow these steps:

1. Add all the blocks to the block diagram and arrange them in the correct order.
2. Properly connect the AXI Stream interface in the correct order. Always connect a transmitter port **M_AXI** to receiver port **S_AXI**.
3. Do not connect the control ports (**S_CTRL**) and the clock (**ACLK**) and reset signals (**ARESETN**) signals manually. But do **Run Connection Automation** at the end which should connect all these signals automatically.

Your block diagram should now look like in Figure 21.

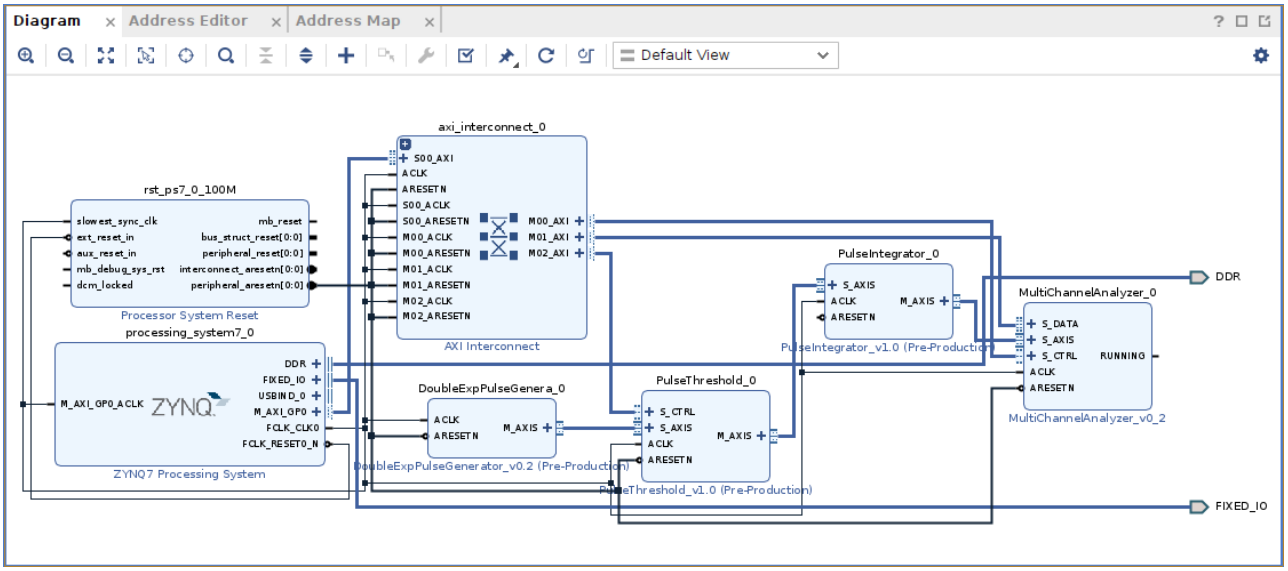


Figure 21: *Block diagram after adding the full data chain*

 Once you are done, show and discuss your block diagram with the tutor.

3.2.4 Assign addresses, create external ports & add constraints

1. On the main AXI bus we now need to verify that each component has an address assigned.

   In the same panel as the **Diagram** view, go to the **Address Editor** tab.

   If there are entries under **Unassigned**: Right click on one of them and select **Assign All**. This will attribute addresses to each AXI interface.

   In the end, the **Address Editor** view should look similar to Figure 22.

2. Configure the **RUNNING** signal of the **MultiChannelAnalyzer** block to connect to an LED.

   Go back to the **Diagram** tab.
   Right click on the unconnected **RUNNING** pin of the **MultiChannelAnalyzer** and select **Make External**.

   In the **External Port Properties** panel (on the left), rename the new external port to **RUNNING_LED**.

   In the **Source** tab (above), right click on **Constraints -> constrs_1** and click **Add Sources**.

   Select **Add or create constraints** and click **Next**. Click **Add Files** and select the following file

   `/home/isotdaq/Constraints/Pins.xdc`

   Click **OK** and make sure to check **Copy constraints files into project** before clicking **Finish**.

3. Create the VHDL wrapper of the block design. This is necessary, as the synthesizer only works with text input files. So the graphical block design needs to be translated into a textual representation.

In the **Sources** tab under **Design Sources**, right click on the **design_1** block design.

Select **Create HDL Wrapper** and click **OK**.

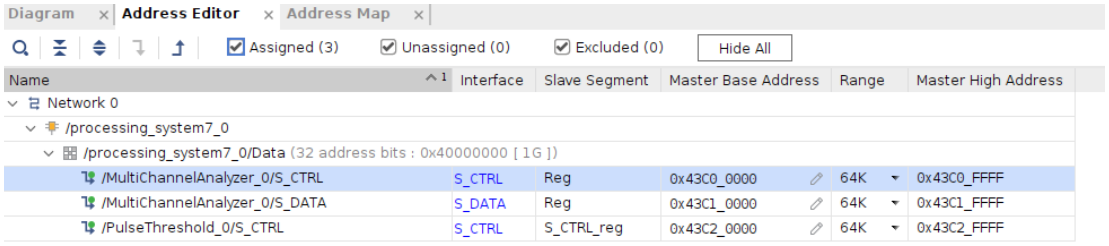


Figure 22: *Address Editor when all blocks have an assigned address range*

3.2.5 Synthesis, implementation & static timing analysis

Finally it is time to run the Synthesis, Implementation and Bitstream generation. You can think about these steps like compiling a C-Program, but for FPGA designs.

 Before you do this next step, ask the tutor to have a quick look at your system.

In the **Flow Manager**, click on **Generate Bitstream**.

If asked for it, click **Save**.
Click **Yes** when asked about **No Implementation Results available**.
Finally, click **OK** in the **Launch Runs** dialogue.

The entire process will take 15-20 minutes, depending on the speed of your computer.

 Time to grab a snack / drink and discuss the following points with your tutor:

- The use of the constraints file (`Pins.xdc`)
- Synthesis
- Implementation (also called Place & Route)
- Bitstream generation

You can now also have a look at how the **PulseIntegrator** is implemented using the **VHDL** hardware description language. To do so:

1. Right click on the **PulseIntegrator** block.
2. Select **Edit in IP Packager** which opens a new Vivado project with the source files of the **PulseIntegrator** block.
3. In the **Sources** panel under **Design Sources** open the **PulseIntegrator_v1_0.vhd** file[13].

[13] The file might have a slightly different name, depending on the exact version of the **PulseIntegrator** block. In doubt, ask your tutor.

4.  Have a look at the **VHDL** code. Can you understand something? Discuss the implementation with the tutor.
5. To return to the block diagram, close the IP Packager project by clicking the **X** on the right end of the blue bar.

3.3 Software development

Note: You can already start working on the Python readout software while the synthesis and implementation are still running. In this case, you just have to postpone the copying of the **.bit** file until after the implementation has finished.

ISOTDAQ Lab 13: SoC on FPGA

In [ ]:
```
import time

# Packages for plotting
import numpy as np
import matplotlib.pyplot as plt

# Package to interfacing with the FPGA / programmable logic
from pynq import Overlay
```

In [ ]:
```
# Program the FPGA
overlay = Overlay('./design_1_wrapper.bit')

# Retrieve AXI control objects
THRESHOLD = overlay.PulseThreshold_0
MCA_CONTROL = overlay.MultiChannelAnalyzer_0.S_CTRL
MCA_DATA = overlay.overlay.MultiChannelAnalyzer_0.S_DATA
```

Interaction with the AXI Bus

In order to write or read data from a block of our design, we need to talk to it over the AXI bus.

Figure 23: *A view of the template Jupyter notebook.*

3.3.1 Basic functionality

We will implement the control and readout software with Python. The PYQN system comes with a pre-prepared Linux operating system, which runs on the embedded ARM processor. The Linux system also includes a **Jupyter** server, which we can use to edit and run our Python code.

Follow these steps to create your own **Jupyter** notebook:

1. Open **Firefox** and go to the following IP-Address: **http://10.10.0.100**. The password is **xilinx**.

   You will see the home screen of the **Jupyter** instance running on the embedded ARM processor.
2. Go into the **ISOTDAQ-Lab13** folder and create a new folder named **Group[N]** with **[N]** the number of your group.
3. Duplicate the **Lab13-Template.ipynb** and move the new copy to your folder **Group[N]**.
4. Now go into your **Group[N]** folder.

Upload the generated **design_1_wrapper.bit** file.

You can find it under:
`/home/isotdaq/SoC-Lab/Group[n]/SoC-Lab-Group[N]/SoC-Lab-Group[N].runs/impl_1/design_1_wrapper.bit`

Upload the generated **design_1.hwh** file and store it as **design_1_wrapper.hwh**.

You can find it under:
`/home/isotdaq/SoC-Lab/Group[n]/SoC-Lab-Group[N]/SoC-Lab-Group[N].gen/sources_1/bd/design_1/hw_handoff/design_1.hwh`

5. Now open the template **Jupyter** notebook (the file with the extension **.ipynb**) and have a look at the pre-prepared code. Follow the instructions within the file to test the basic functionality of your SoC.

In order to control the **PulseThreshold** block and the **MultiChannelAnalyzer**, as well as to read out the data from the **MultiChannelAnalyzer** you can refer to the following register map.

Block	**Register**	**Offset**	**Comment**
PulseThreshold	THR_START	0	If the signal rises above this threshold, a pulse is detected. Possible values: 0 - 65535
	THR_STOP	4	If the signal falls below this threshold, the end of a pulse is detected. Possible values: 0 - 65535
MultiChannelAnalyzer Control	RESET	0	Writing **1** to this register clears all the MCA data.
	ENABLE	4	Writing **1** to this register enables the operation of the MCA. Writing **0** to this register pauses the MCA operation.
MultiChannelAnalyzer Data		0 - 8188	The 2048 histogram bins. Bin **N** corresponds to register offset **N*4**.

 Ask your tutor if you are stuck, or if you need help with Python.

3.3.2 Additional software features

Here are two more challenges to be implemented in software, depending on the available time. Pick one to start and discuss between yourselves how to implement this.

 You can ask your tutor for advice at any point in time.

Medium: Identification of the spectrum components

The spectrum recorded by your system contains a total of 6 peaks. The two middle peaks correspond to the **Kα** and **Kβ** emission lines of Iron (**Fe**). Can you figure out what the other two components of the recorded spectrum are?

A table with **Kα** and **Kβ** emission lines of most elements can be found under:
https://xdb.lbl.gov/Section1/Table_1-2.pdf

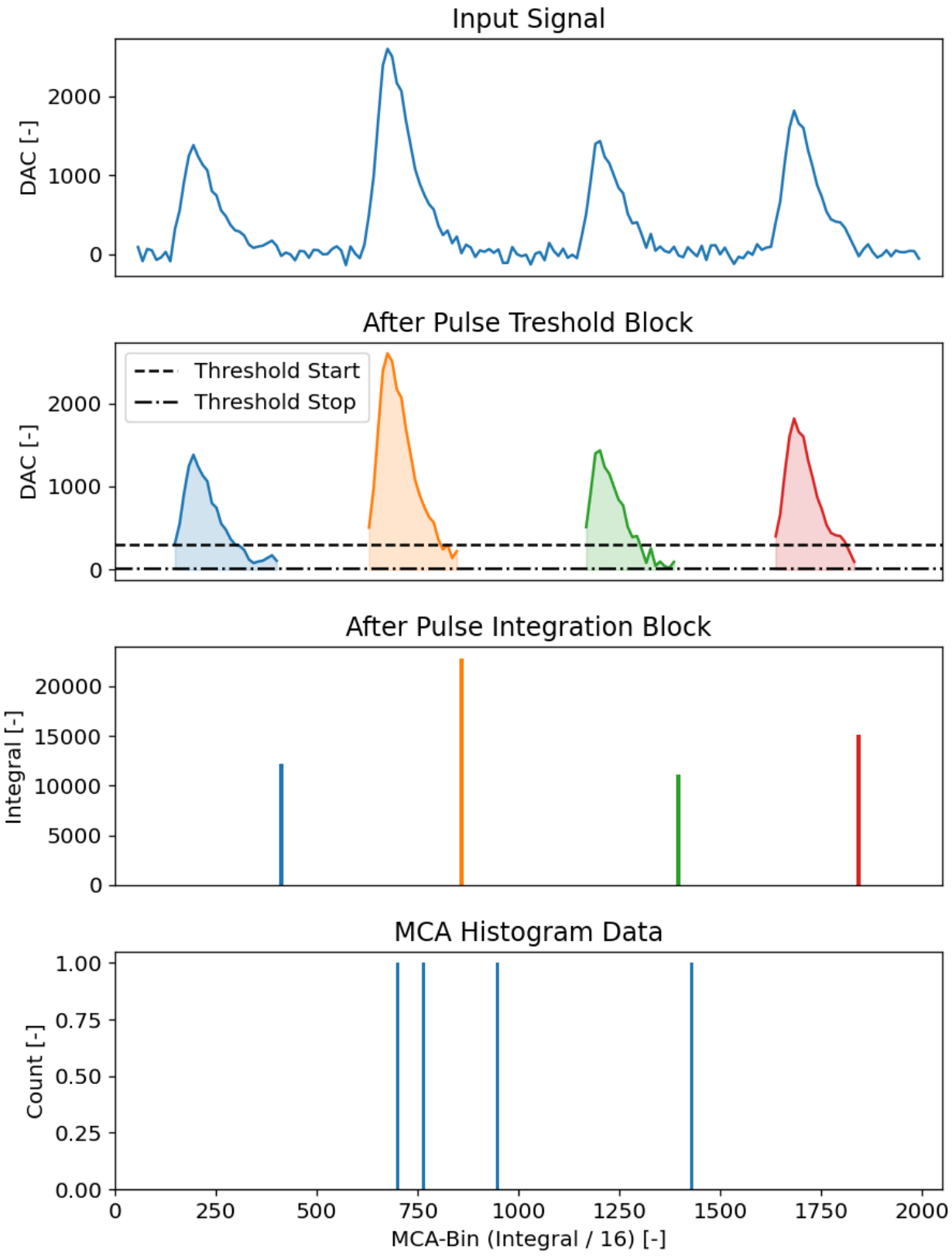
Hard: Identification of the noise floor

The pulse signals generated by the **DoubleExpPulseGenerator** are not ideal, but have a certain amount of white noise added. Can you identify the amount of noise which is present? Or formulated differently: what is the minimal threshold which should be set to not record any noise?

This is the end of the guided lab. Have fun playing around with the system and discuss any question you have with your tutor. Thank you for your participation!

Appendices

A) Pulse Processing Pipeline



B) Acknowledges

- Markus Joos (CERN) & other organizers of ISOTDAQ
- Ton Damen (NIKHEF)
- Peter Jansweijer (NIKHEF)
- The other members of the ISOTDAQ SoC FPGA Lab Team:
  - Elena-Sorina Lupu (Caltech/Caltech)
  - Patryk Oleniuk (Hyperloop One)
  - Andrea Borga (NIKHEF)
  - Manoel Barros Marin (CERN)

Lab 14: Introduction to GPU programming

V.1.0 - Gianluca Lamanna (gianluca.lamanna@cern.ch)

Overview

This lab aims at introducing the main concepts of GPU programming. After simple exercises to become familiar with the video card and software environment, we will study the influence of the GPU (Graphics Processing Units) architecture on code optimization. The students are invited to use the pieces of code provided in this lab book as a source of inspiration and to write is own code.

Introduction

Historically most of the code, written for standard processors, runs sequentially. To speed-up the software execution most of the developers relied on the continuous increase in hardware system performances with small or null code optimizations. In the last years, the processors' speed is gone towards a kind of saturation due to integration limits and power constraints. Parallel processors, such GPUs, have shown to overcome the limits of single-core processors, thanks to a different architecture. The GPUs are designed to have many-threads running in parallel with the purpose to increase the computing throughput for parallelizable problems. In contrast, the CPUs (Central Processing Units) are designed with a multi-core architecture, in which each core is optimized to maximize the execution speed of sequential programs. Nowadays commercial GPUs are designed to have an order of 10 TFlops (Floating point operations per second) computing throughput while first class multi-core CPUs rarely exceed 1 TFlops.

Fig.1

One could ask why there is such a large difference in performance between many-threads GPUs and general-purpose multi-core CPUs. As shown in figure 1 the main reason is due to the very different structure between the two types of processors. In CPU most of the “transistors” are devoted to control logic and large cache to reduce instructions and data access latencies, while in GPU most of the space is used for real computing. Another important reason is the difference in memory bandwidth since several applications are limited by the rate at which the data are delivered to the processor for computing: in the GPU the memory bandwidth is at least x5 higher than in CPU. This is due to the fact that in standard CPU the memory management is conditioned by legacy operating systems, applications and I/O, while in GPU only data movement from memory to computing units occurs.

From a software point of view, to exploit the power for these GPUs the application is expected to be written with a large number of parallel threads. The hardware takes advantage of the large number of threads to find work to do when some of them are waiting for long-latency memory

accesses or arithmetic operations (throughput oriented). On the other hand, the CPUs are designed to minimize the execution latency of a single thread (latency-oriented). For programs that need one or very few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs. When a program has a large number of threads, GPUs with higher execution throughput can achieve much higher performance than CPUs. It should be clear now that GPUs are designed as parallel, throughput-oriented computing engines and they will not perform well on some tasks on which CPUs are designed to perform well.

A heterogeneous computing model, in which both serial and parallel processors are used at the same time, is able to speed-up several computational problems. This kind of philosophy is used, for instance in the CUDA model on NVIDIA video cards. Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphics API (Application Programming Interface) functions to access the processing units. Starting from 2007 CUDA has been introduced to allow developers to direct access to computing resources, in an environment integrated with standard programming languages.

The CUDA model is not the only way to exploit GPU power for computing. Nowadays several other possibilities are present on the market. OpenCL is a low-level API, equivalent to CUDA but suitable for several types processors (including FPGAs) and GPU vendors. Directives-based programming model (such as OpenACC) and GPU libraries (such as Thrust) allow to invoke GPU computing during serial code execution, to parallelize specific task, in a user transparent way. For educational reasons, in this lab, we will concentrate on CUDA, as it allows to better understand the GPU programming philosophy and the interplay with the hardware architecture.

Exercise 1: Discover the GPU performances

The CUDA developer SDK provides examples with source code and utilities to get started writing software with CUDA. In the CUDA samples directory, you can find several programs ranging from basic to advanced level. In particular, in the directory "`1_Utilities`" two simple applications are ready to be compiled (using `make` for the moment): `deviceQuery` and `bandwidthTest`. Both of them provide information about the system, type, and characteristics of GPU (if correctly recognized).

Try to run them and write down the features that seem most important to you.

Exercise 2: Parallel "Hello World!"

The code to program the GPU is substantially subdivided into two parts: **HOST** and **DEVICE**. The HOST part is executed in the PC while the DEVICE part (called KERNEL) runs on GPU. In the HOST part, together with the serial part of the program, a GPU related part must be present, such as data preparation and instructions to copy data on device (the GPU). Let's try to write a "Hello World!" program (see *Snippet 1*)

```
#include <stdio.h>
__global__ void mykernel(void) {
  printf("Hello World from GPU! (block: %d thread: %d)\n"   ,blockIdx.x,threadIdx.x);
}
int main(void) {
  mykernel <<<1,5>>>();
  cudaDeviceSynchronize();
  printf("Hello World from Host!\n");
  return 0;
}
```

Snippet 1

In this simple program, there are two key points of GPU programming: *kernel definition* and *kernel launch*. The kernel is essentially a C function defined through a qualifier. The qualifier (`__global__`) tells the compiler that the function is callable from host to be executed on the device. The syntax to launch the kernel in CUDA is: `mykernel<<<DimGrid, DimBlock>>>(arg)`, where `mykernel` is the name of the kernel defined above with, eventually, the arguments of the function between the brackets (), while the variables `DimGrid` and `DimBlock` define how to use the GPU resources at run time. We will discuss this point later, for the moment we just note the kernels have access to two built-in variables (`threadIdx`, `blockIdx`) that allow threads to distinguish among themselves. The CUDA SDK toolkit provides tools for compilation and debugging. The C compiler is "`nvcc`", very similar to cc/gcc in several aspects. You can compile the "Hello world" code just with:

```
nvcc file_name.cu –o HelloWorld –arch=compute_50 –code=sm_50
```

Try to compile and run the code: Which is the output? Try to play a little bit with the parameters in the kernel launch (1 and 5 in this example), what happens?

Exercise 3: Vector Add and GPU optimization

In order to illustrate the structure of parallel computing, we start with a somewhat more complex example. Suppose you have two very big vectors (let's say with 1048576 elements) to add up. The task is quite simple since each element in the vector result is just the sum of the corresponding elements in primary vectors:

c[i]=a[i]+b[i]

but must be repeated a huge number of times. This problem is particularly suitable for parallelization: each element in the sum is independent of the other. As a starting point let's try to write a serial version of the code as suggested in snippet 2

```c
#include <stdio.h>
#define N 1048576
void RandomVector(int *a, int nn){
  for (int i=0;i<nn;i++) {
    a[i]=rand()%100+1;
      }
 }
//serial sum
void VecAddSerial(int *a, int *b, int *c){
  for (int i=0;i<N;i++){
    c[i] = a[i]+b[i];
  }
}
int main(void) {
  int *h_a, *h_b, *h_c;
  int size = N*sizeof(int);
  float time;
  cudaEvent_t start,stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  //Alloc in Host (and filling)
  h_a = (int *)malloc(size);
  h_b = (int *)malloc(size);
  h_c = (int *)malloc(size);
  RandomVector(h_a,N);
  RandomVector(h_b,N);
```

Snippet 2.a

```c
//start time
  cudaEventRecord(start);
  //Launch Serial Sum on CPU
  VecAddSerial(h_a,h_b,h_c);
 //stop time
  cudaEventRecord(stop);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&time, start, stop);
 //Print Result
 //  for(int i=0;i<N;i++){
  //    printf ("%d) h_a:%d h_b:%d h_c:%d\n",
  //    i,h_a[i],h_b[i],h_c[i]);
  // }
  //print time
   printf("Time: %3.5f ms\n",time);
 //Cleanup
  free(h_a);
  free(h_b);
  free(h_c);
  return(0);
}
```

Snippet 2.b

In this case, the vector sum is done in a standard C function (`VecAddSerial`). Some CUDA function is added to measure the execution time with the same mechanism we will use in the GPU version. For this reason, this code must be compiled with `nvcc`, even if it will run only on the host. Let's try to modify this code to exploit the GPU computing power for the vector sum. Before starting we discuss a moment as the parallelism works in the GPU. After the kernel launch, the CUDA runtime system generates a *grid of threads* organized in a two-level hierarchy. Each grid is organized into an array of *thread blocks*, which will be referred to as *blocks*. All blocks of a grid are of the same size; each block can contain up to 1,024 threads.

Go back to exercise 1 to discover the maximum dimensions of grid and blocks in the GPU we are using.

```
<skip>
//kernel
__global__ void VecAddGpu(int *a, int *b, int *c){
  c[blockIdx.x] = a[blockIdx.x]+b[blockIdx.x];
}
<skip>
//Alloc in Device
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);
//Copy input vectors form host to device
  cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
<skip>
//Launch Kernel  on GPU
  VecAddGpu<<<N,1>>>(d_a,d_b,d_c);
  cudaDeviceSynchronize();
<skip>

//Copy back the results
  cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
<skip>
//Cleanup
  free(h_a);
  free(h_b);
  free(h_c);
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
```

Snippet 3

The total number of threads in each thread block is specified by the host code when a kernel is launched. Each thread is identified by the thread number within the block (`threadIdx.x` we saw in exercise 2) and the block number within the grid (`blockIdx.x`). Another important aspect in preparing the GPU modification of the serial code is that the user must provide the data to the GPU. This means that, before the kernel launch, the data must be explicitly copied in the GPU. When the computation on GPU ends, the results must be copied back from device to host by the user. This is done using the PCI express bus, slower with respect to the processor memory direct connection. One can imagine that the movement of data from host to device (and back) is one of the main contributions to the total latency in heterogeneous application. This overhead in time execution can be eventually "masked" with the gain in the computational part. Now, try to modify the serial code by using the suggestions in snippet 3. The `cudaMalloc` function is used to allocate memory on the device, while data from the host to the device are copied by using the function `cudaMemcpy`. After the kernel launch, another `cudaMemcpy` will copy back the results.

Try to compile and run.

The result is not what we expected. Although we are using a GPU with 1 TFlops and our problem is very parallelizable (and simple), the execution time is worse than the serial code version!

Take a minute to have a close look at the code.

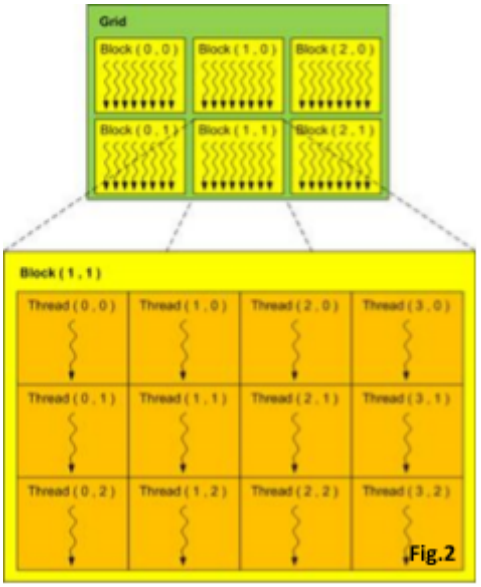


Fig.2

The reason for this failure is that we are using the GPU in a very inefficient way. In fig. 2 the hierarchy of the threads described above is shown. Roughly speaking each block is executed in a multi-processor (this is not completely true), that, in hardware, is a group of single CUDA cores. The GTX750, we are using, has 512 computing cores grouped in just 4 multi-processors. In the kernel launch in the snippet above we define the structure of the parallelism with `VecAddGpu<<<N,1>>>`, this means that we asked for 1048576 blocks with one thread each. Since the number of multi-processors is limited, only 4 threads are executed concurrently in the GPU. In each block, one thread is working while 127 threads are in idle, with a lot of blocks waiting for their turn to be executed.

Then let's try to change the point of view and rewrite the code (Snippet 4) to fully exploit the threads.

```
<skip>
//Launch Kernel  on GPU
  VecAddGpu<<<1,N>>>(d_a,d_b,d_c);
  cudaDeviceSynchronize();
```

Snippet 4

In this case, we are using one single block and a huge number of threads.

Let's try to compile and run.

Now the execution time is very very small. Try to compute the speed-up factor with respect to the serial code. As a rule of thumb a well-written code can give a factor of 100, but no more than that. But here it is even bigger. This is quite suspicious! **Try to print out the result of the kernel execution (after the copy on the host) to figure out what is happening.** Probably you can notice that there is

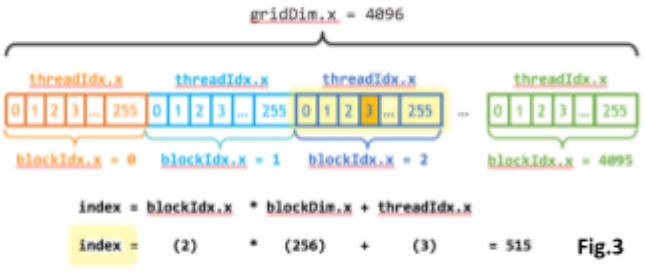


Fig.3

something wrong. Another possibility to spot the error is to use the error managing system of CUDA. Try to add:

```
cudaError_t KernelError=cudaGetLastError();
printf("Error %s\n",cudaGetErrorString(KernelError));
```

just after the kernel call.

What message do you get? Another possibility is to use CUDA debugger. It is very similar to the c gdb debugger with additional support to investigate the behavior of single threads. Try with:

`cuda-gdb <executable>` (remind to compile with the flag –g).

As suggested by debugging tools we are running the GPU out of his resources. Try to have a look (exercise 1) at the maximum number of threads per block allowed for the board we are using. The winning strategy is to use both blocks and threads to exploit the maximum number of resources

```
<skip>
#define THREADS_PER_BLOCK 128

<skip>
//kernel
__global__ void VecAddGpu(int *a, int *b, int *c){
  int index = threadIdx.x + blockIdx.x*blockDim.x;
  c[index] = a[index]+b[index];
}
<skip>
//Launch Kernel  on GPU
VecAddGpu<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a,d_b,d_c);
cudaDeviceSynchronize();
```

Snippet 5

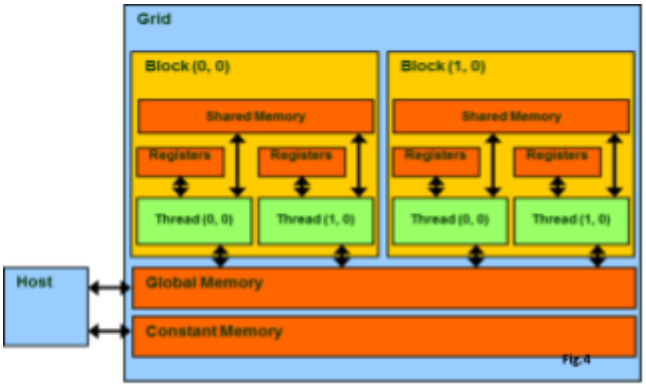
available. In doing that we need to change two things (Snippet 5): first in the kernel call we have

explicitly configured the GPU architecture to have the correct number of blocks in the grid and the correct number of threads for each block, second we have to change the kernel in order to be sure that each of the N threads running operates on different vector elements (fig.3 is just a generic example on how to define the index in the kernel).

Try to compile and run the code. How much is the execution time now?

Exercise 4: Memory management

In the previous exercise, we learned how to organize the GPU resources to use a huge number of threads, concurrently. Although we saw how to optimize threads and blocks to get better performances, this is not the end of history. The CUDA kernels that we have learned so far will



likely achieve only a tiny fraction of the potential speed of the underlying hardware. The poor performance is due to the fact that in GPU programming it is responsibility of the user to organize data in the proper way inside the memory. Let's try to understand better this point with an example. In our previous example for each operation (vector elements sum), there are 2 accesses to the memory to extract the sum addenda. Thus the ratio of floating point calculation to memory access is 1:2. We refer to this ratio as *compute-to-global-memory-access ratio*, defined as the number of floating-point calculations performed for each access to the memory. For our GTX 750 the memory bandwidth is (exercise 1) 80 GB/s, this means that with 4 bytes for each single-precision floating point operation one can expect to load 80/(2*4) =10 Giga single-precision operands per second. Assuming a compute-to-global-memory-access of 0.5 the maximum number of operations per seconds is 10 GFlops that is quite far from the nominal computing power of 1 TFlops of our GPU. The programs in which execution speed is limited by the memory access throughput are called *memory bound* programs. CUDA provides a number of additional resources and methods for accessing memory that can remove the majority of traffic to the memory. We will briefly discuss the use of CUDA memories and how to increase the timing performances, through a simple example. In GPU there are several types of memories that can be accessed from different levels of the parallelism hierarchy. In fig.4 you can see the structure of the two main memories: the global and the shared memory. The global memory is accessible by each multi-processor and each thread running in single cores of a multi-processor. It is physically implemented in DDR and is connected through the PCI express bus to receive data coming from the host: when you perform a standard cudaMemcpy operation you are copying data in this memory, that is, usually, quite big (2GB in GTX 750). On the other hand, the scope of the shared memory is limited to a single block. Only the threads running in that block can use the same shared memory, that is implemented on the GPU chip. The quantity of shared memory per block is very limited (49kB on GTX 750) but the speed is very high (at a level of 1 TB/s). This structure of memories is exploited to increase the peak performance in the GPU in several problems.

Let's try to understand better how to exploit the shared memory with a new example: the matrix multiplication. Take a look at the snippet presented on the next page (Snippet 6). The structure is similar to the one analyzed above, even if the code is organized in a different way. **Try to understand what the code is intended to do by yourself.** In this version of the code data are copied in the global memory from host and then each thread read the data directly from the global memory. A simple improvement in memory managing is achieved by observing that the global memory access is coalesced. This means that when all threads of a warp (a group of 32 threads in

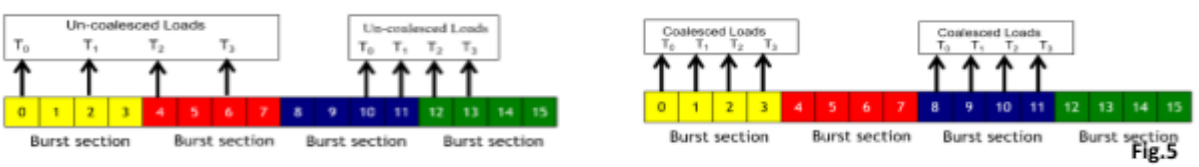


Fig.5

the same block) execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced, as described in fig.5.

```
<MatrixMultiplication_global.h>
#include <stdio.h>
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row *
M.width + col)
typedef struct {
int width;
int height;
float* elements;
} Matrix;
// Thread block size
#define BLOCK_SIZE 16
__global__ void MatMulKernel(const Matrix,
const Matrix, Matrix);
<MatrixMultiplication_global.cu>
#include "MatrixMultiplication_global.h"
// Matrix multiplication - Host code
void MatMul(const Matrix A, const Matrix B,
Matrix C) {
  // Load A and B to device memory
Matrix d_A;
d_A.width = A.width;
d_A.height = A.height;
size_t size = A.width * A.height *
sizeof(float);
cudaError_t err = cudaMalloc(&d_A.elements,
size);
 printf("CUDA malloc A:
%s\n",cudaGetErrorString(err));
 err = cudaMemcpy(d_A.elements, A.elements,
size, cudaMemcpyHostToDevice);
printf("Copy A to device:
%s\n",cudaGetErrorString(err));
Matrix d_B;
d_B.width = B.width;
d_B.height = B.height;
size = B.width * B.height * sizeof(float);
err = cudaMalloc(&d_B.elements, size);
printf("CUDA malloc B:
%s\n",cudaGetErrorString(err));
err = cudaMemcpy(d_B.elements, B.elements,
size, cudaMemcpyHostToDevice);
printf("Copy B to device:
%s\n",cudaGetErrorString(err));
  // Allocate C in device memory
Matrix d_C;
d_C.width = C.width;
d_C.height = C.height;
size = C.width * C.height * sizeof(float);
err = cudaMalloc(&d_C.elements, size);
printf("CUDA malloc C:
%s\n",cudaGetErrorString(err));
  // Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid((B.width + dimBlock.x - 1) /
dimBlock.x,(A.height + dimBlock.y
```

Snippet 6.a

```
// Read C from device memory
err = cudaMemcpy(C.elements, d_C.elements,
size,
cudaMemcpyDeviceToHost);
printf("Copy C off of device:
%s\n",cudaGetErrorString(err));
// Free device memory
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by
MatMul()
__global__ void MatMulKernel(Matrix A,
Matrix B, Matrix C) {
 // Each thread computes one element of C
 // by accumulating results into Cvalue
 float Cvalue = 0.0;
int row = blockIdx.y * blockDim.y +
threadIdx.y;
int col = blockIdx.x * blockDim.x +
threadIdx.x;
if(row > A.height || col > B.width) return;
 for (int e = 0; e < A.width; ++e)
Cvalue += (A.elements[row * A.width + e]) *
(B.elements[e * B.width + col]);
C.elements[row * C.width + col] = Cvalue;
}
// Usage: multNoShare a1 a2 b2
int main(int argc, char* argv[]){
Matrix A, B, C;
int a1, a2, b1, b2;
 // Read some values from the commandline
a1 = atoi(argv[1]); /* Height of A */
a2 = atoi(argv[2]); /* Width  of A */
b1 = a2;           /* Height of B */
 b2 = atoi(argv[3]); /* Width  of B */
A.height = a1;
A.width = a2;
A.elements = (float*)malloc(A.width *
A.height * sizeof(float));
B.height = b1;
B.width = b2;
B.elements = (float*)malloc(B.width *
B.height * sizeof(float));
C.height = A.height;
C.width = B.width;
C.elements = (float*)malloc(C.width *
C.height * sizeof(float));
for(int i = 0; i < A.height; i++)
     for(int j = 0; j < A.width; j++)
          A.elements[i*A.width + j] =
(float)(random() % 2);
 for(int i = 0; i < B.height; i++)
      for(int j = 0; j < B.width; j
```

Snippet 6.b

To exploit the coalescence, the memory reading should be organized in a proper way. The next level of improvement is to exploit the shared memory. The general idea is to subdivide the matrix into sub-matrices and to assign each sub-matrix to a thread block. The data are copied once from the global memory to the shared memory of the block and then re-used according to the needs of the matrix multiplication algorithm. The shared memory is defined with the keyword `__shared__` followed by type and dimensions. Also the shared memory, similarly to the global memory, is organized in banks (16 in older GPU, 32 in newer GPU) that can be accessed simultaneously to achieve high memory access. In your working directory, you should have a file `MatrixMultiplication_shared.cu` file, where the implementation with shared memory is done.

Try to take a look to understand how it works and modify the code to try different solutions and to measure the timing (adding the "events" statement properly).

Exercise 5: Fitting rings with GPU

In this last exercise, you can use the code in the directory "`rings`" in your working directory. This is a simple example of possible use of GPU for pattern recognition. The so-called *Crowford algorithm* is used to search for center and radius of a limited number of hits on a circle in, for instance, a Cherenkov Ring counter. You are invited **to try to optimize the code** in order to increase the speed of the total process. A good way to understand what is happening is to use the *profiler* provided in the CUDA toolkit. You can invoke it just with `nvprof` and the name of the executable for the textual version and `nvvp` and the name of the executables for the GUI version (fig. 6).

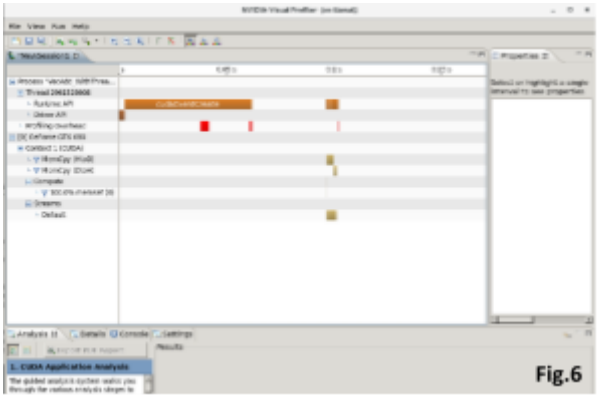


Fig.6

GLOSSARY

GPGPU (General Purpose computing on Graphics Processing Units): is the idea to use GPU for standard computing usually performed on CPU, by exploiting the intrinsic parallelism of graphics processors.

Threads: A thread is a simplified view of how a processor executes a sequential program in modern computers. A thread consists of the code of the program and the values of its variables and data structures. A CUDA program initiates parallel execution by launching kernel functions, which causes the underlying runtime mechanisms to create many threads that process different parts of the data in parallel.

Blocks: Groups of threads.

CUDA core: the smallest arithmetic computing unit in a GPU (a.k.a. single core).

Multi-processors: a group of single cores.

Built-in Variables: Many programming languages have built-in variables. These variables have a special meaning and purpose. The values of these variables are often pre-initialized by the runtime system and can be referenced in the program.

Memory Space: Memory space is a simplified view of how a processor accesses its memory space is usually associated with each running application. The data to be processed by an application and instructions executed for the application are stored in locations in its memory space. Each location typically can accommodate a byte and has an address. Variables that require multiple bytes – 4 bytes for float and 8 bytes for double are stored in consecutive byte locations. The processor gives the starting address (address of the starting byte location) and the number of bytes needed when accessing a data value from the memory space.

SIMD (Single Instruction Multiple Data): It is a computing model in a parallel architecture. It described processing units where the same instructions are executed simultaneously on a different set of data.

SIMT (Single Instruction Multiple Threads): The computing item is the thread that is implemented as a sequence of SIMD operation