# Readout Electronics
**for a**
# Novel Animal PET Scanner
**using**
# Field Programmable Gate Arrays

**System Definition, Implementation and Assessment**

*by*
Jo Inge Buskenes
mail @ joinge.net

Submitted to


University of Oslo
*Faculty of Mathematics and Natural Sciences*

in partial fulfilment of the requirements for the degree of
Master of Science.

*Supervisors:*

Steinar Stapnes
Steinar.Stapnes @ cern.ch

Erlend Bolle
Erlend.Bolle @ fys.uio.no

Ole Rohne
Ole.Rohne @ fys.uio.no

# Abstract

A digital FPGA-based data acquisition system for a novel preclinical PET detector developed at the University of Oslo will be described. The detector, called ComPET, employs an inventive geometry with 600 LYSO scintillator crystals interleaved with 400 wavelength-shifters, grouped into 4 modules and arranged in a rectangular fashion to attain high photon sensitivity and high spatial resolution with minimal shift-variance. By means of APDs and a custom analog front-end the detector response is converted to a digital output, its rising edge and width being a measure of the $\gamma$-photon arrival time and energy, respectively. An FPGA samples up to 84 of these channels with deserialisers clocked at up to 1GHz, computes and stores the event photon arrival time, energy and location, provides a fan-in structure to collect data from these channels, and sends these over Ethernet to a data acquisition system. The system allows for coincidence- and energy-windows to be set for improved contrast resolution, can handle sustained event-rates of 100Mevents/s with full 3D-readout, and is parametrised for ease of maintainability and flexibility.

# Acknowledgments

Man, it just hit me, this is probably the only place in this report where I can go nuts.

With no particular priority in mind, I think I'll start off with my parents. Thanks. Not sure about the questionable remarks with regards to the countless hours I spent writing this thing, but hey, at least the food was nice. :)

Then of course, every ComPET member has my infinite gratitude (not quantifiable, ohh the evil). Cheers to Erlend Bolle for his enthusiastic and steady leadership, and to Ole Rohne for his sharp, yet painfully accurate, voice. Special thanks to Steinar Stapnes, for making this project, and this thesis, possible.

And while we are in the special department; Another special thanks to Michael Rissi, for not fleeing the office when I arrived in the morning (yes, we always arrived in that order). I am sure to miss the physics talks, not to mention having someone to speak English to. :)

And yeah, Martin Brinkmann, thanks heaps for keeping me accompanied during the long summer when the final touch on this thesis was made. The temptations were many, but we prevailed! That is, an exception should possibly be made for all those coffee breaks.

Further thanks to John Williams at PetaLogix for granting us the donation request on PetaLinux, a great software with excellent documentation to get going with an Embedded Linux system. Also, thanks to Xilinx for donating various embedded Linux course material.

And to all you peeps out there reading this thesis (indeed, a man got to have dreams), since you probably read this simply because you have to, you have my sympathy! I have attempted to make it interesting, but hey, this is science, what can you do?

*"If you are out to describe the truth, leave elegance to the tailor"*
Albert Einstein

# Contents

*"It's no longer a question of staying healthy.*
*It's a question of finding a sickness you like."*
Jackie Mason

# 1 Introduction

Technological advances of recent years has created new unique possibilities in medical diagnosis. New novel diagnostic instruments enables us to look inside living bodies (*in vivo*) at internal structures and processes, with impressive level of detail, and repeatedly with insignificant harm.

In the *clinic* these images are used to identify abnormal conditions, study the underlying mechanisms that caused them, and to analyse and aid in the treatment process. Since visual data is easy to interpret, it helps the doctor to set an accurate diagnosis, and assigns the problem an identity to which the patient may relate.

*Preclinical* imaging applications include studying the effect of diseases and new methods of treatment on animals, usually rodents[1]. New detector technologies are frequently realised as animal prototypes due to their relatively low cost and complexity, shorter development cycles, the possible aspect of testing the detectors on animals[2], and because there is a lot of available research material in this field. There is also the economical motivation present; if the effect of drugs and pharmaceuticals on animals can be monitored more accurately these products can hit the market sooner.

Which technologies exists? The next section will introduce some of the most well-established ones, before moving on to focus on a detector technology called Positron Emission Tomography (PET).

[1] Mice and rats.
[2] Due to ethical considerations the use of animals should be avoided if possible.

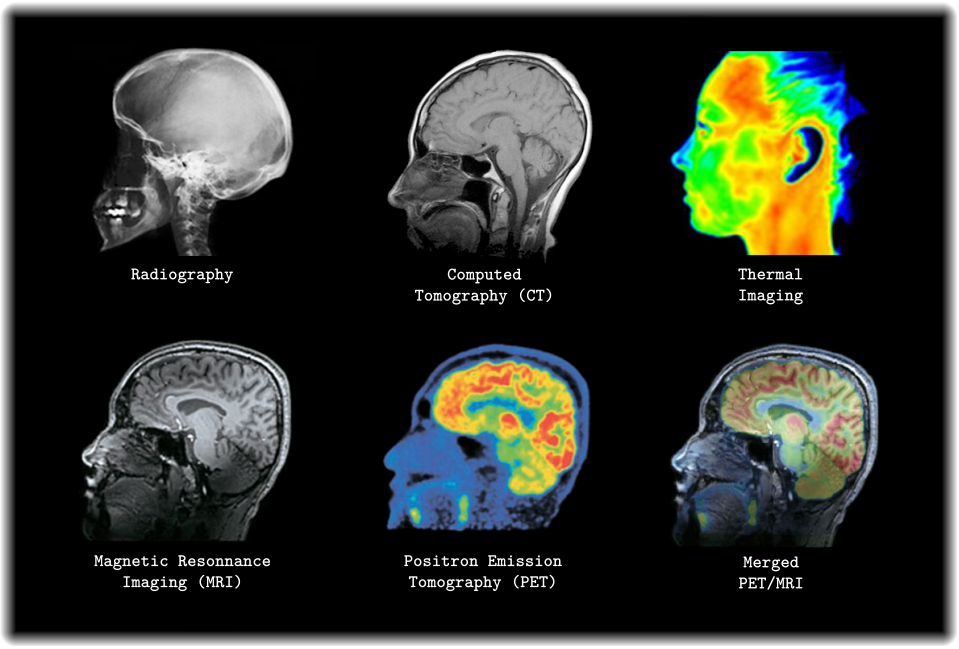



*Figure 1.1 - Imaging technologies compared [23]*

## *Medical Imaging Technologies* 1.1

The first medical imaging technology of modern science - *radiography* - was introduced in 1895, when Wilhelm Röntgen produced the first x-ray image. This scan is conducted by transmitting x-rays through a body - where the level of absorption varies for each substance - and the remaining rays are measured at the other side. The technique offers images with high resolution "silhouettes" of hard materials such as bones, but suffers from low soft tissue resolution and moderate radiation exposure. However, it is still around today (just ask the local dentist) due its relatively low complexity and cost.

*Computed Tomography* (CT) usually refers to the computation of tomography from x-ray images. In tomography several 2D slices are combined to form 3D images with better tissue resolution and better signal-to-noise ratios (SNR). It is widely used, but even though new techniques for reducing the radiation exposure are frequently introduced, the time it takes to conduct a scan is long enough for the accumulated radiation to reach moderate to high levels[3].

Similarly to CT, *Magnetic Resonance Imaging* (MRI) is an imaging technology based on tomography. It is armed with 3 electromagnetic fields; a

[3]We generally do not wish the a medical imaging instrument to cause harm, but exceptions exists (destroying scar tissue, cancer cells, etc.).

very strong static magnetic field which polarises hydrogen molecules in the body, a weaker gradient-field used to measure the position of the polarised molecules, and a radio-frequency (RF) field used to manipulate hydrogen atoms in order to produce detectable signals. An MRI scan is considered non-harmful[4] (and can thus be conducted repeatedly) and has a very good soft tissue resolution (better than CT).

In *Ultrasound* high-frequency sonic waves are sent into the body, reflected, and the echo recorded. By measuring the delay and direction of the incoming waves, the exact point at which it was reflected can be computed[5] - usually in real-time. Due to its live and non-harmful nature (at low intensities) it is widely used to image the foetus in pregnant women, abdominal organs, heart, breasts, muscles, arteries and veins.

Finally, there is *nuclear medicine*, with non-invasive imaging techniques such as *Single Photon Emission Computed Tomography* (SPECT) and *Positron Emission Tomography* (PET) as the main imaging technologies. Both yield 2D/3D-images, rely on similar physics and principles, and exerts moderate radiation exposure.

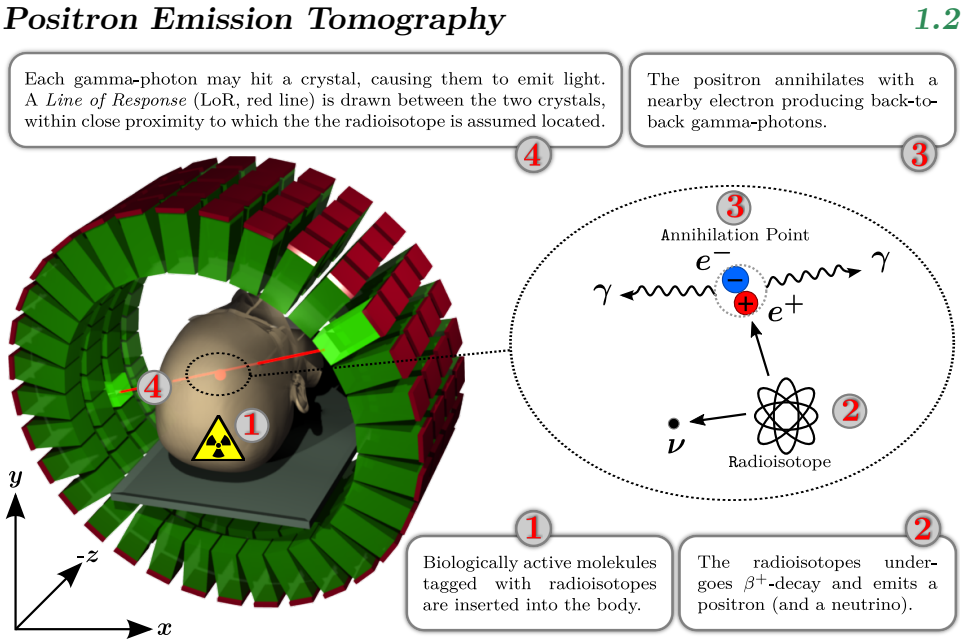



*Figure 1.2 - PET principle, radial geometry*

[4]Studies indicate no long-term bi-effects from the strong static field, although high exposure to RF fields are associated with some health risks.

[5]A single Ultrasound image is 2D, but can be combined to form 3D images.

### The Principle 1.2.1

A PET scan is conducted by injecting the body with low-mass biologically active molecules (e.g. glucose) tagged with short lived radioactive isotopes (usually $^{18}F$), a compound commonly referred to as radiotracer or probe[6]. Then follows a waiting period to allow the probe to become properly distributed in the tissue.

The radioisotopes are subject to $\beta^+$-decay (see fig. 1.2), a process where a positron[7] and a neutrino[8] is emitted[9]. In a body consisting of mainly water the positron will traverse a few millimetres[10] and then annihilate with an electron, producing two 511 keV gamma-photons with an intervening angle close to 180°.

A common means to detect these gamma-rays is with scintillator crystals[11], traditionally arranged radially in trans-axial rings (see fig. 1.2). A scintillator is a dense, transparent material in which the energy of the incoming gamma-rays is absorbed (completely or partially) by ionisation[12] and re-emitted in subsequent processes as light due to de-excitations. While some of the light will escape the crystals due to hitting the crystal sides in a too steep angle, the rest will be reflected off the sides and move towards the crystal edges[13].

Finally, the scintillation photons are converted to electrical energy, usually with either PhotoMultiplier Tubes (PMTs) or Avalanche PhotoDiodes (APDs)[14]. For the following discussion, an electric pulse caused by a gamma-photon interaction will be referred to as simply an *event*, and two of these within a very short temporal interval (ns-range) will be referred to as *coinciding events* (or simply coincidences).

[6] Chosen such that it is unlikely to disturb the natural states of cells and tissues.

[7] A positron is an anti-particle of the electron, and is - as the name suggest - positively charged.

[8] It probability of a neutrino reacting in the detector is extremely small, hence these will be ignored in the further discussion.

[9] $\beta^+$-decay: $energy + p \rightarrow n + e^+ + \nu$

[10] The distance a positron traverses prior to annihilation is continuously distributed, depending on its kinetic energy (for $^{18}F \approx 0.5$mm).

[11] Another common alternative to scintillators is semiconductors, used by some detectors.

[12] A 511 keV gamma-photon can interact with the crystal atoms in either of two ways: through *Compton scattering* or *photoelectric effect*. In either cases part of the gamma-photon energy is transferred to electrons in the atom, but depending on the size of the energy transfer the electron may be rejected from the atom completely (photoelectric), or caused to recoil (Compton scatter).

[13] The scintillator crystals acts as optical waveguides, in close resemblence to optical fibers.

[14] APDs are physically much smaller than PMTs, but has not been able to match the incredible gain of photomultiplier tubes, until the recent development of Geiger-mode APDs (GAPDs) [3]. APDs, and similar components, may also be referred to as Multi-Pixel Photon Counters (MPPCs) or Silicon PhotoMultipliers (SiPMs).

The challenge remains to *detect* these coincidences, and preferrably only those caused by gamma-photons originating from a common radioisotope. When one is found, a *Line of Response* LOR may be drawn between the two interaction points, in close vicinity to which the decaying isotope may be assumed located[15].

The biologic activity in a given volumetric element inside the body may now be inferred from the number of LORs passing through that element. This is essentially how a PET-image is computed.

### Advantages & Disadvantages 1.2.2

Since the radiotracer can be designed to probe into specific biological processes, PET is typically able to provide higher quality sub-process information than optical scanners[16][5]. Even subtle molecular signals deep in the tissue can be resolved with high temporal and spatial resolution and contrast [5].

This information is important because in the event of of a disease, functional changes are likely to appear before, or exceed, structural changes in the body. This makes PET a very important tool to study *cardiac* and *neurologic* diseases, and *cancer*. Preclinical PET systems allows the careful monitoring of the disease development, with the animals acting as their own control, thus drastically reducing the development time of new pharmaceuticals and therapeutic agents allowing them to be put into commercial use sooner.

However, due to being a relatively new imaging technology, and because the most commonly used radioisotopes must be created with a cyclotron [20], PET systems are rather expensive and not found in all hospitals. Also, due to the radioactive exposure, although not persisting for long[17], a patient can only undergo this procedure a limited number of times.

### Performance Parameters 1.2.3

Three important performance parameters of a PET-system is its *photon sensitivity*, *spatial resolution* and *contrast sensitivity*.

The *photon sensitivity* is the probability that a photon emitted from the body is detected. This depends on the solid angle coverage, the inter-crystal

[15] In clinical PET the gamma-photon time-of-flight (TOF) is sometimes measured with extreme precision electronics, from which the position of the radioisotope along the LOR may be inferred. However, due to observing much smaller bodies, this is hardly ever attempted with animal PET scanners.

[16] One noteworthy exception to this is MRI, which can provide some information on biologic activity - for instance by injecting cold water into the bloodstream and measuring where, and how quickly, it heats up.

[17][18] F half-time: 109min.

and module "gaps" (crystal packing fraction), and the conversion efficiency of the crystals. Or, put slightly differently, it depends on the probability of a gamma-ray even hitting the crystals, and the probability of an interaction if it does. Thus photon sensitivity is directly linked to SNR, which ultimately affects the estimation accuracy of the photon *arrival time* and *energy*. Thus improving the photon sensitivity implies that PET scan-time or radiation dosage, or both, may be decreased, without loss in image quality.

In PET, the physical lower limit of *spatial resolution* is given by a convolution of positron range (how far the positron propagates prior to annihilation), annihilation photon non-collinearity (the radial error caused by an annihilation photon incident angle slightly unequal to 180°), and intrinsic detector resolution [4]. The positron range depends on the body and the kinetic energy of the positron, the effect of collinearity depends on the detector diameter, and the intrinsic detector resolution depends on the crystal size and detector geometry.

The *contrast* sensitivity is a measure of how well signals with similar values can be resolved and distinguished from the background noise. This depends on photon scattering in tissue and crystals (Compton), random coincidences, pulse-pileup[18], and the two above mentioned parameters; photon sensitivity and spatial resolution[19].

### Detector Design 1.2.4

One way to increase the *photon sensitivity* in a detector is essentially to add more scintillation material, either to increase the FoV or to add to the crystal thickness or length (yielding better intrinsic detection efficiency). However, extending FoV adds to the cost considerably, making crystals thicker decreases the intrinsic spatial resolution, and making them longer reduces the light yield due to intrinsic crystal attenuation. An alternative is to make the diameter of the detector smaller, but this causes more photons to hit crystals at oblique angles. In traditional PET-systems with radially oriented crystals this translates to a larger parallax error and a higher probability of photons escaping the crystals (refer to 2.2.1 for more information).

To improve the *spatial resolution* one should make crystals thinner, and find the best compromise between non-collinearity and parallax error. Indeed, a smaller detector diameter will reduce the non-collinearity factor, but increase the parallax error, which is basically the same as trading off spatial resolution shift-invariance for improved resolution in the detector centre.

[18] If a crystal is hit by a gamma-ray while it is still responding to a previous hit, this is refered to as pulse-pileup.

[19] Other dependencies also exist, including non-specific targeting of the probe and the image reconstruction algorithms [5], but these are outside the scope of this report.

Making crystals thinner also comes with a trade-off; these are harder to manufacture and thus costly, and less light can be collected from a thinner crystal end [5], thus compromising on photon sensitivity.

Achieving a better *contrast sensitivity* means being able to reduce the level of background, either passively by detector design or actively by means of high-speed front-end electronics and clever trigger algorithms (see 2.2.1/2.4.2 for more details).

## ComPET *1.3*

ComPET (COMpact PET) is a new novel detector with very high sensitivity and spatial resolution[20], achieved with an inventive geometric design and readout system[21]. It is developed at the University of Oslo, in collaboration with AxPET [10].

Each detector module is made up of 150 long LYSO-crystals[22], interleaved perpendicularly with 100 WaveLength Shifters (WLS). These are distributed into 4 blocks (each of which being a LYSO/WLS matrix), and arranged in a rectangular fashion to attain the shape shown in fig. 1.3. Attached to the end of every LYSO-crystal and WLS is a GAPD.

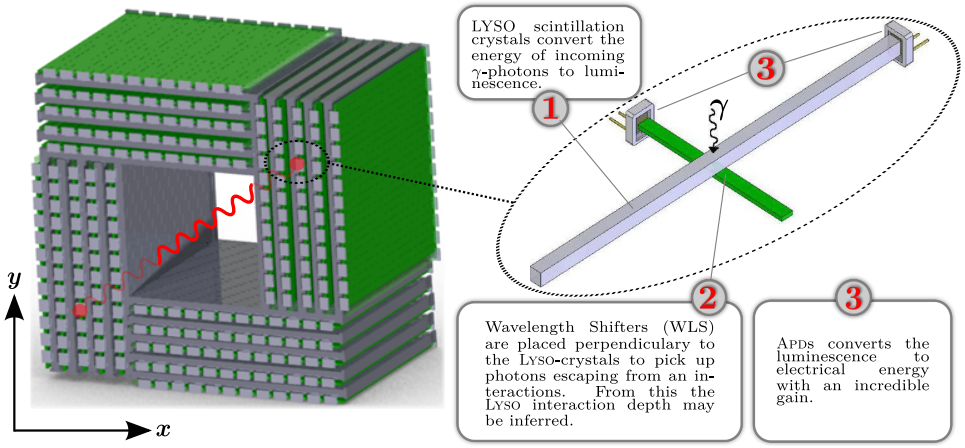



*Figure 1.3* - ComPET *geometry*

[20] Monte Carlo simulations indicate an intrinsic detector sensitivity of up to 16%, and a FWHM spatial resolution of slightly less than 1mm in the centre of the detector field-of-view.

[21] The discussion of the former will be delayed till the next chapter, which will deal with it thoroughly.

[22] $Lu_{1.8}Y_{0.2}SiO_5(Ce)$ (LYSO) crystals are inorganic scintillators with a high atomic number (Z) and density, and a fast, near linear absorption energy to luminescence output response [5, page 129].

Since the response from every crystal and wavelength-shifter is read out, a 3D-representation of the interaction may be computed, as opposed to just a 2D-weighed mean. This extra information can be used to estimate the $\gamma$-ray depth-of-interaction (DOI), and to distinguish Compton scattered events from photoelectric events in the crystals. This largely mitigates the parallax error, allowing the detector diameter to be reduced as much as possible (while still being able to fit the body inside) for improved photon sensitivity and spatial resolution.

Furthermore, ComPET has no inter-crystal and module gaps, is very compact, and MRI-compatible[23]. The latter is interesting because it allows fused images to be computed, with metabolic information from PET and anabolic information from MRI.

## *Thesis Contents* *1.4*

When the work on this thesis started ComPET was not much but a goal; to create a reasonably priced detector with cutting edge performance using the latest advances in detector technologies. This also involved the digital readout system, of which this author was the main responsible. Based on this, an inherent way or partitioning the thesis seems to be as follows:

1. **Introduction**. Introduces medical imaging and some of its prime technologies, with main focus on Positron Emission Tomography (PET). References to additional resources will also be provided.
2. **System Definition**. Aims to identify important design parameters, explains why Field Programmable Gate Arrays (FPGA) were used, and introduces the digital readout system in terms of functional behaviour.
3. **Implementation**. Supplies more technical details on the FPGA readout design.
4. **Results**. Presents various simulations and readouts as a means to verify correct system behaviour.
5. **Discussion**. Elaborates on the importance of the results.
6. **Conclusion**. Wraps up the current status of the design and provides an outlook.

[23] To be MRI-compatible the detector must be able to operate in very strong magnetic fields (5+ [Tesla])

Naturally, some topics do not blend well with this outline, but might be valuable for those seeking to continue this work. The appendices cover some of these:

- **Getting Started**. Believe it or not, getting started is easier said than done if you are not well experienced with the software used in the development of this project. This appendix attempt to ease the process by providing hints and tips on installation and common usage of these applications.
- **Project Management**. As the project tree has become rather large and complex, some strategies for managing it have been developed. This appendix provides documentation in this respect.
- **Embedded Tutorial**. An embedded project were added to the digital readout system to perform various control activities and manage network access. It will hardly be treated in the report, but this appendix may be visited as a how-to on how to get a similar system up and running.
- **ISE/EDK Messages**. ISE and EDK represents the Xilinx FPGA development studio. This appendix elaborates on some of the most common, but less intuitive, messages these programs output when implementing the digital readout design. This to build confidence that the correct FPGA logic is inferred from the HDL description.

This report is best read as a `pdf` file. This way you get to enjoy bookmark navigation and hyperlinks functionality. However, the colorcodes should make it pretty straightforward to read the paper version aswell. Green coloured words are links to the glossary, red coloured words are references to the bibliography. In the bibliography you will find an extensive list of sources and where to get additional information.

## *Additional Resources* *1.5*

Several additional resources exists for the design mentioned in this report, such as source files, images, online documentation and repositories, etc. This will be made available for the reader, either visiting the homepage mentioned a few paragraphs down or by simply clicking the references in this document (if you are reading the electronic version).

The homepage may be accessed by logging into the the ComPET Wiki-page[24] at

`https://wiki.uio.no/mn/fys/compet/`

At this page there should be pointers to

- Git-repositories (see B.2.1) tracking all the project source-code (see B.1.1).
- A `.rar`-file containing the project tree at the time of printing.
- Doxygen-generated documentation for all the HDL source code.
- A cache of support literature.

For those seeking to continue my work, it may also be beneficial to keep the "Project Management" appendix (B) in handy while reading this thesis. It contains a brief overview over the various parts of the project tree, and provides quick tips on managing it with Makefiles and the version control system Git.

[24] The homepage can also be accessed directly at `http://www.joinge.net/compet?ref=wikipage`, but this location may change in the future.

*"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified."*

Donald Knuth

# 2 System Definition

In ComPET I was responsible for *defining* and *implementing* a digital readout system that collects and filters data coming from the analog front-end, "builds" events by coincidence matching, and sends these over an Ethernet network for further processing.

This was a challenging task as the detector system was initially largely underspecified. Several important design characteristics were yet to be decided upon, including detector geometry (and thus the number of channels), analog front-end, and the entire digital trigger- and readout-system.

Thus, this chapter will be dedicated to defining the system, while implementation details will be delayed until next chapter (3). First the analog front-end will be introduced (2.1), then a few general challenges and constraints will be presented (2.2), followed by a discussion of the technology chosen for the implementation (2.3). Finally, in the light of the found conclusions, the readout design will be described in terms of functional behaviour (2.4).

Any mention of the digital readout system should be considered the result of my own work unless otherwise noted. Detector physics, mechanics, and aspects of the analog front-end are mentioned where it makes sense to do so, but these areas were covered by other team members.

## *Introduction* 2.1

Before discussing the digital part of the readout system, a quick description of the analog front-end will be necessary.

### Analog Front-end 2.1.1

Recall that there are only 3 essential parameters to be found for each interaction: the *photon arrival time*, *energy* and *location*. A common way to retrieve this information is by sampling the GAPD-response with ADCs, from which these parameters may be inferred. Recording the raw-data stream is a flexible solution, because no knowledge of the input signal is required except that it fulfills the Nyquist-Shannon criterion. However, ADCs are power-hungry, complicates the PET design, and - because they produce a lot of data - puts tough constraints on the digital readout system.

ComPET, on the other hand, employs an alternative digitisation system that encodes this information in a single digital output pulse; the start of it corresponding to the photon arrival time and the width corresponding to the photon energy. The principle is shown in fig. 2.1, and will be discussed shortly.

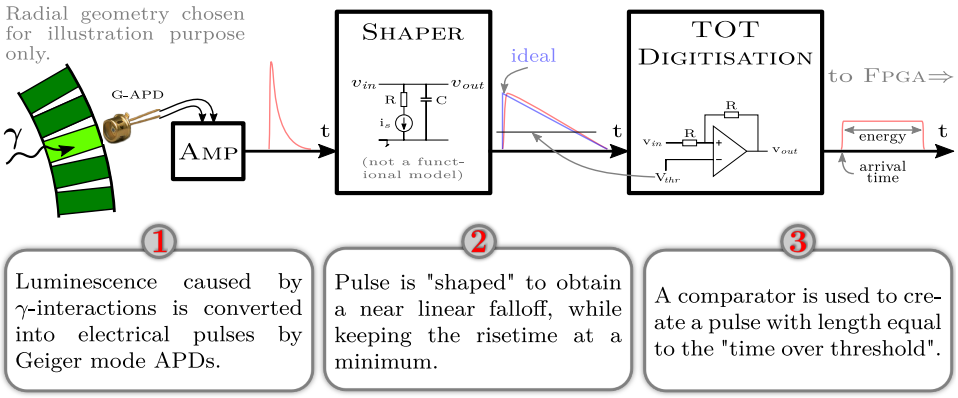



*Figure 2.1* - ComPET *analog front-end*

### Time over Threshold 2.1.2

After being amplified, the signal energy is integrated and re-shaped into a saw-tooth with a charge collecting circuit, linearly discharged through a constant current source. The saw-tooth is then sent to a comparator along with a threshold voltage, producing a digital pulse with a width equal to the *time-over-threshold*. Not surprisingly, this is also the name of the method.

The saw-tooth shape is ideal because of its steep rising edge and linear falloff, which allows for an accurate estimation of the photon arrival time

and energy, respectively. Unfortunately, since the crystal, APD and shaper introduce a time-constant, the rising edge lose some steepness. Furthermore, the analog front-end will not be able to produce a completely linear fall-off.

In short, the TOT-approach facilitates a compact design with low cost, complexity and power consumption, is quick to develop, and provides excellent scalability. However, because all the information is inferred from the rising and falling edge alone, a means to accurately measure these must be introduced.

The technology chosen for this purpose is Field Programmable Gate Arrays FPGAs, which will be introduced in 2.3. However, first a few words regarding the challenges a readout system must handle.

## *Design Challenges* 2.2

What is required of a PET readout system, and of readout systems in general? Let us start with the physics involved, and proceed with matters of higher level abstraction.

### Physics Parameters 2.2.1

How should we design the data acquisition system to improve *photon sensitivity*, *spatial resolution* and *contrast sensitivity*?

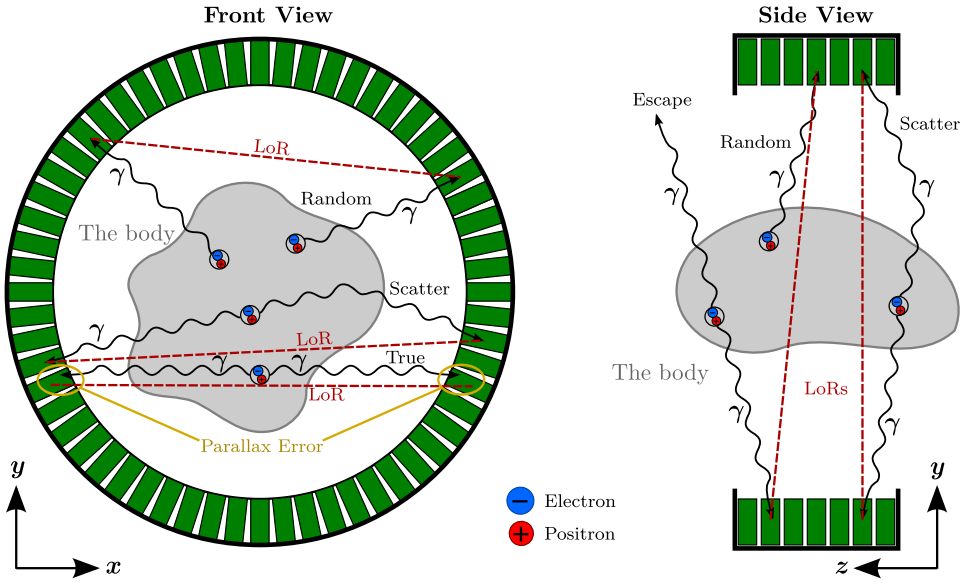



*Figure 2.2 - Some factors affecting PET image quality (radial geometry chosen for illustration purpose only)*

Any event the readout system fails to detect, be it due to faulty *capture* or *processing*, negatively affects the *photon sensitivity*. Capturing an event is only problematic for very short TOT-pulses (low energy), or for subsequent TOT-pulses hitting the same channel with very short temporal isolation[1], and should not be a major concern. Data loss due to buffer overflows will be, however, and must be avoided.

Unlike traditional geometries such as the one shown in fig. 2.2, the ComPET geometric structure of interleaved LYSOs and WLSs allows the $\gamma$-ray depth-of-interaction (radial coordinates) to be found, allowing for the event to be reconstructed in full 3D. This largely mitigates parallax error (hence improves the *spatial resolution*), and the identification of crystal Compton scatters (see below), but puts high demands on the digital readout electronics in terms of number of channels[2] and throughput.

*Compton scatters in tissue and crystals*, and *random coincidences* causes incorrect LORs to be computed and thus affects the *contrast sensitivity* (and indirectly also spatial resolution).

To reduce the effect of *tissue and crystal scatter* an energy threshold is usually applied to each channel to discard events that were not photoelectric[3]. In ComPET, however, the energy threshold is set low enough (around 50keV) to include Compton scatters as well. If the first and second point of interaction is identified and the energies sum to 511keV, then another coincidence is found and the SNR will hence increase. In case the energies sum to less than 511keV, however, the event either scattered in the body tissue or escaped the detector, and is discarded. The energy threshold is set by adjusting the discriminator threshold of the analog front-end.

In conventional PET scanners single events are typically 1-2 orders of magnitude more frequent than true coincidences, because the sensitivity of these detectors tend to be less than 10% [5]. This contributes heavily to the generation of *random* (or "false") *coincidences*, and thus image background. The only way to distinguish these from true coincidences is by time-separation. This dictates the need for high sampling rates, and a means to match events within very short time-intervals (later referred to as coincidence windows).

[1] The minimum time that must be allowed to pass between successive events on *the same channel* before the system can perform the correct distinction, is referred to as *dead-time*.

[2] Some detectors performs a 2D energy-mean in the analog front-end, thus drastically reducing the number of channels and system cost. However, doing so discards information about Compton scatters in the crystals, negatively impacting the spatial resolution and contrast sensitivity.

[3] Scattering causes photons to lose energy depending on the scattering angle.

### Readout System Constraints 2.2.2

These factors allow us to put up the following optimisation criteria for the performance of our data acquisition system:

- *Time resolution.* The finer the time-resolution of the TOT-data, the better the accuracy of estimating the photon arrival time and energy. Thus the *sampling speed* should be as high as possible, ideally more than 1 GHz.
- *Throughput.* The system throughput must be sufficient to avoid losing data due to buffer overflows. The total event-rate in our PET-detector is expected to be a few Mevents/s[4], but for good measure the aim is to handle at least 20Mevents/s.
- *Data compression.* PET-data is extremely sparse, thus the compression potential is huge. Not a single bit should be stored unnecessary, as this implies increased cost and decreased throughput.
- *Dead time.* All channels will be handled concurrently, so the system dead-time will equal the channel dead-time, i.e. the ability of the channel-logic to separate closely separated events. Unless the energy threshold is set very low, these events are rare. Even so, the readout system that will be presented should be capable of handling a dead-time down to 20ns, or maybe even less.

Other considerations, which are not directly performance related, are:

- *Cost.* While the total cost of our detector is well defined, the question remains how to best distribute it over the various detector elements. To do more with less is generally a good idea.
- *Flexibility.* This is still a development project. It is virtually impossible to know in advance the optimal hardware structure, or combination of design parameters. Especially important is scalability, since this design is subject to be used on a wide variety of FPGAs, with varying number of inputs.
- *Portability.* Good design practise dictates writing for reusability to minimise development time, ease maintenance, and promote design reliability. Furthermore, it is never wise to become too technology-dependent, especially not in an early development phase. Solving

[4]The activity in clinical PET-scans using $^{18}F$ tracer is 100-400Mbq [2][Wikipedia]. This would translate to 15-60Mevents/s with a 15% photon sensitivity detector, such as ComPET. However, since rodents are quite small, the activity is much lower. One PET study of mice stated an activity of 7.4MBq [1].

newly discovered problems and challenges may simply be a matter of choosing different hardware, so keeping this option open is a wise idea.

Now that these design requests are defined, let us move on to discuss the technology selection.

## *Technology Selection* *2.3*

A currently very popular technology for realising these types of designs are Field Programmable Gate Arrays (FPGAs). This section will introduce this technology, explain why was chosen, and quickly introduce some off-the-shelf PET-boards with these chips embedded.

### Field Programmable Gate Arrays 2.3.1

The internal structure of a Field Programmable Gate Array (FPGA) may be thought of as isles of Configurable Logic Blocks (CLB[5], see 2.3.2.2.) in a sea of programmable interconnect[6] (see fig. 2.9). It was originally invented by Xilinx to bridge the gap between traditional Programmable Logic Devices (PLDs), which are configurable but do not scale well towards larger devices[7], and Application Specific Integrated Circuits (ASICs). ASICs are fully customisable down to transistor level (if desired), and can realise extremely complicated designs, with a minimum of silicon real estate usage, at a low power consumption, and with superior performance. However, since ASICs are not reprogrammable, and these designs are complex, the development time and cost is very high. Thus the use of these chips are only common in high volume markets, where they are economically feasible, or in designs requiring this extra level of customisability.

However, the combination of hundreds of general purpose IO-pins, extreme concurrency, and re-programmability[8], makes FPGAs ideal in physics experiments, which often employs numerous sensors sampled at high speeds, yielding high data rates and commonly a need for heavy compression. Also, the high demand for these chips, e.g. in the high-volume embedded device market, makes them relatively cheap and readily available in off-the-shelf boards.

[5] This is a Xilinx abbreviation, other vendors use different terms.

[6] More or less a quote from Clive Maxfield [7].

[7] Increasing the size of a traditional PLDs caused interconnect to grow more rapidly than logic [7].

[8] SRAM based FPGAs (from Altera/Xilinx) are ”infinitely” reprogrammable, with the trade-off that the configuration is volatile (i.e. lost when powered down).

| Features | Virtex-6 | Virtex-5 | Spartan-6 | Extended Spartan-3A |
|---|---|---|---|---|
| User I/Os | 320-1 200 | 172-1 200 | 132-576 | 144-519 |
| SERDES support | Yes | Yes | Yes | No |
| Slices[1] | 11 640-118 560 | 3 120- 51 840 | 600- 23 038 | 704-23 872 |
| Look-up Tables[2] | 46 560-472 240 | 12 480-207 360 | 2 400-184 304 | 1 408-47 744 |
| Registers | 93 120-948 480 | 12 480-207 360 | 4 800-368 608 | 1 408-47 744 |
| Clock Management | 3-9 CMTs[3] | 1-6 CMTs[3] | 2-6 CMTs[3] | 2-8 DCMs |
| BlockRAM [kb] | 5 616-38 304 | 936-18 576 | 216-4 824 | 54-2 268 |
| DSP Blocks[4] | 288-2016 | 24-1 056 | 8-180 | 0-126 |
| Multi-Gigabit Serial[5] | 0-48 | 0-24 | 0-8 | - |
| Ethernet MAC[6] | Yes | Yes | No | No |
| PCI Express[7] | Yes | Yes | Yes | No |
| MicroBlaze Support | Yes | Yes | Yes | Yes |

*Table 2.1 - Virtex-5/6 and Spartan-6/3A Comparison [11][12][13][16][18]*

[1] Virtex-6 and Spartan-6 slices each contain 4 LUTs and 8 registers, Virtex-5 slices each contain 4 LUTs and 4 registers, and Extended Spartan 3A slices contain 2 LUTs and 2 registers.
[2] Virtex-5/6 and Spartan-6 use 6-input LUTs, while Extended Spartan-3A use 4-input LUTs.
[3] Virtex-6 Clock Manager Tiles (CMTs) each contains 2 Mixed-Mode Clock Managers (MMCM), which can be used as either PLLs or DCMs; Virtex-5 and Spartan-6 CMTs each contain 2 DCMs and 1 PLLs.
[4] The DSP blocks contain Multiply-ACumulate hard cores (MAC). Virtex-5 and 6 use 25x18 MACs, Spartan-6 and 3A use 18x18 MACs.
[5] Virtex-6 has Multi-Gigabit blocks supporting speeds up to 11+Gbps, the Virtex-5 up to 6.5Gbps, and Spartan-6 up to 3.125Gbps [14].
[6] All but one Virtex-6 model have 2-4 Ethernet MAC cores.
[7] Virtex-6 has hard-core PCI-Express support for generation 1 and 2, with x8 speed; Virtex-5 has hard-core support for gen.1 (x8), and soft-core support for gen.2 (x8); and Spartan-6 has hard-core support for gen. 1 (x1) [14].

The following discussion will stick to *recent generations* of Xilinx FPGAs, since these chips were chosen for this design[9]. Xilinx terminology will be consistently, unless otherwise noted.

### Resources 2.3.2

Tab. 2.1 compares the available logic offered from four currently very popular series of programmable Xilinx devices; the Virtex-6 and Virtex-5 high-performance chips, and the low-cost, low-power Spartan-6 and Spartan-3A chips. A brief description of these resources, along with their importance in this design, will follow.

#### *Input and Output Buffers* 2.3.2.1

The *Input and Output Buffers* (IOB) is the interface between the "external" (outside FPGA) and internal logic. For both inputs and outputs a wide range of signalling standards are supported - for both differential and single ended schemes. Optional input delay elements may be used to synchronise

[9] For our project, the alternative to Xilinx would have been Altera. Both produce chips with similar performance and functionality, and thus the decision was simply made based on what the team had previous knowledge of (to reduce development time).

input data streams, and input impedance can be adjusted either digitally (with Digitally Controlled Impedance, DCI) or by toggling input termination. Each IOB buffer has several registers, for driving the output pins or for clocking input data (even Dual Data Rate - DDR - is supported). However, the data may also just be routed through.

In recent FPGA models[10] the GPIO-tile also contains a silicon SERialisation and DESerialisation (SERDES) block. This facilitates high-speed signalling, and is easy to use. It will be used in this design to capture the TOT-data.

The alternative to SERDES is to bring the high speed signals into the FPGA fabric and perform the deserialisation there, but the internal delays of the fabric makes this very hard. If the goal is to sample data coming from an external synchronous source, dedicated clock resources (PLLs and DCMs) can be used to make it work[11]. However, unless the deserialiser logic is manually placed[12], there is no way to guarantee that the deserialiser performs the sampling linearly - i.e. that bits are uniformly distributed within the dataframes.

#### *Configurable Logic Blocks (CLB) and Slices* *2.3.2.2*

The *Configurable Logic Blocks* (CLBs) are the main logic resource for implementing sequential as well as combinatorial circuitry[13]. Due to its importance in nearly all FPGA-designs it deserves a quick description.

First, have a look the right side of fig. 2.3 and notice what is called a *slice*. A slice is the smallest group of logic in an FPGA. Depending on the technology model (limited to the ones listed in tab. 2.1), a slice may contain 2-4 LUTs, 2-8 registers, a few multiplexers, and some gates, carry-chains and tri-state buffers. A slice is fully configurable, which means that it can be programmed to realise almost any digital circuitry.

Each lookup table (LUT) has 4-6 inputs and can be used as a function generator, read-access memory (*Distributed SelectRAM*) or a shift register[14].

[10] Since Virtex-4 all Virtex-models have SERDES-support, but only some Spartan-models do.

[11] That it "can be made to work", does not mean it is easy. High speed designs are consistently hard to debug, and adding the necessary time- and location-constraints to such a design can easily lead to over-constraining; Unless the implementation tools are allowed a certain minimum of freedom, the implementation process are likely to become very slow, and - in some cases - misbehave or fail completely.

[12] This is referred to as *Floorplanning* the design, and should only be considered in the final stage of the development process. This is because it makes the implementation software less able to perform optimisations, slows the implementation process down (which is too slow already), and may cause problems to appear where they previously did not (like squeezing a balloon).

[13] CLBs represents the "isles of logic" mentioned in 2.3.1.

[14] In Virtex-6 chips only some LUTs can function as Distributed RAM.

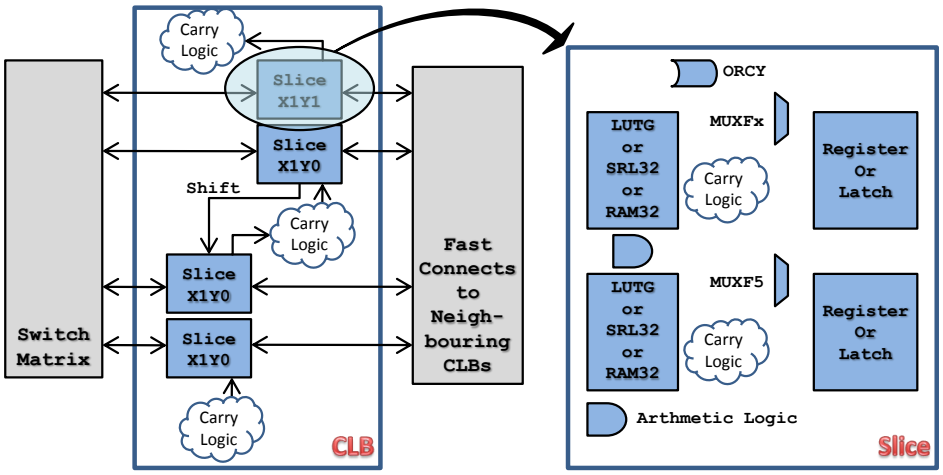



*Figure 2.3 - Configurable Logic Block (simplified)*

A LUT with, say, 6 inputs can realise *any* combinatorial circuitry with 6 input lines. The configuration of Distributed SelectRAM and shift registers are flexible, and more so for every new generation of FPGAs. This allows distinct resources in the FPGA to be used for increasingly complicated tasks, as well as overlap other resources in terms for functionality. Thus, not only will new FPGA-models contain more logic, it will also be used more efficiently[15].

The storage elements can be configured either to realise a level triggered latch or an edge triggered register. The input data can either be supplied from logic inside or outside of the slice.

It is possible to perform "rough calculations" on resource utilisation by referring to the numbers above, but this estimate is never accurate. This is because the implementation software performs a wide range of optimisations when realising the design, making it hard to predict exactly what resources will be used in the end. It might decide that some resources ought to be shared or that the design should be redistributed in order to achieve a better compromise between logic utilisation and performance.

Each CLB is connected to the global routing network through a switch matrix and to adjacent CLBs with fast interconnect (see fig. 2.3). To realise complex logical structures several CLBs can be combined, however, as the complexity of the logic increases so will the hit on performance[16]. Good coding style dictates breaking down any problem into simple functions which can be implemented with as few slices as possible.

[15] This is one of the reasons why comparing logic consumption across FPGA-models is tricky.

[16] This is because routing delays between cells dominates the delays inside the cells.

***Digital Clock Managers*** *2.3.2.3*

Each Virtex-5 Clock Manager Tile (CMT) contains one Phase-Locked Loop (PLL) and two Digital Clock Managers (DCMs). The latter is a self-calibrating and fully digital solution for:

- *Clock distribution.* An excellent fan-out and internal delay-locked loop (DLL) helps preserve signal integrity.
- *Delay compensation.* Using an internal feedback the DCM can deskew all clocks relative to the input clock, thus making the DCM appear "transparent".
- *Frequency synthesis.* Derived clocks can be created with a wide range of possible frequencies.
- *Coarse-grained clock phase shifting.* Supplies output clocks with 0°, 90°, 180° and 270° phaseshifts, respectively.
- *Fine-grained clock phase shifting.* Provides the ability to on-the-fly adjust the clock phase in increments of `T/256`[17] [17, page 49].

PLLs, other other hand, can not adjust the phase as the DCMs do. But they offer something else in return, *jitter* filtering, which is very handy to ensure optimal system performance. Note that the use of these two components may be combined, e.g. first use a PLL to clean up the clock, and then pass it to a DCM. In its current state, this design does not use DCMs, only PLLs, but this is expected to change in the future when external cards must be interfaced[18].

A wide variety of interconnect is available for routing data and clock-signals. For global or high speed clock signals, and reset signals, the global clock routing network should be used. This is low skew interconnect designed for low duty cycle distortion, improved jitter tolerance, low power consumption and high speed clock signalling. The clocks are routed to these global highways with clock multiplexers, which can switch glitch-lessly from one clock to another (see 3.2). The clock multiplexers no longer shares routing resources, thus the bank access restrictions seen in previous FPGA-models no longer applies[19].

[17] Overridden by `DCM_TAP_MIN` and `DCM_TAP_MAX` [17].

[18] To interface external cards, a "source synchronous" clock domain should be created around the IOB-tile connected to the external device to promote transfer-speed and data link reliability. The data may be synchronised with the system clock, i.e. become "system synchronous", by means of a few flip-flops.

[19] The FPGA is segmented into "banks", physical areas sharing routing resources, voltage supplies, etc. Previously only a select few clocks could be routed to the same bank, which easily caused problems if several high speed clock domains were required inside this bank.

#### *Multi-Gigabit Transceivers* 2.3.2.4

The Multi-Gigabit Transceivers (MGTs) provided by Xilinx are called RocketIO. This is a technology aimed to provide serial communication at speeds up to several Gb/s. In recent FPGAs from Xilinx the RocketIO functionality resides in silicon[20] and offers excellent performance.

#### *PowerPC Blocks* 2.3.2.5

*PowerPCs* are "hard" processor cores (implemented in silicon), as opposed to "soft" processor cores (like Microblaze) which is implemented in FPGA-fabric[21]. Due to their hard nature, *PowerPCs* offers better performance and lower energy-consumption than soft-processors[22], but soft-processors are far superior in terms of flexibility. Either can be used to run Embedded Linux, as this design will.

#### *DSP Blocks* 2.3.2.6

The Digital Signal Processing blocks in recent Xilinx FPGAs are variations of, what they referred to as, a DSP48 block[23]. This block contains multiply and accumulate (MAC) circuitry, able to handle up to 25x18 bits multiplications on the Virtex-5.

The combination of these DSP resources and the massive parallelism offered by the FPGA logic has made FPGAs very popular in applications where extreme DSP performance is required. The major drawback has historically been that writing DSP applications in HDL is very cumbersome, but a constant drive towards allowing higher level languages for FPGA-descriptions has made this less of an issue. For example, it is now perfectly possible to write DSP algorithms in Matlab[24], and synthesise these for FPGAs.

Another very potent technology here is Graphics Processing Units (GPUs), which is made up of a large number of small DSP processors, all operating concurrently. This technology is low-priced, the programming language is C, and the GPU designs are highly flexible, scalable and portable. The technology is worth a mention in this context, as ComPET might eventually perform some DSP tasks with FPGAs, and some with GPUs.

Generally, FPGAs will excel GPUs in terms of performance, and most notably so in applications where it is hard to achieve concurrency, or where

[20] This was introduced in the Virtex 2 Pro, but then the latencies were too high for it to become a real success.

[21] The "fabric" of an FPGA is the internal configurable logic, as opposed functionality residing in silicon blocks.

[22] Analogous to ASICs versus FPGAs.

[23] This block was introduced with Virtex-4. Before this, a few multipliers was the best the Virtex series had to offer.

[24] Writing in the Matlab `m`-language typically require 50-100x less code than the equivalent HDL-description.

the concurrent processes must communicate [9]. However, FPGA-designs do not natively support floating-point precision, is time-consuming to develop, and are not (yet?) as portable, scalable and flexible as GPU-designs.

#### *Block Select RAM* *2.3.2.7*

BlockRAM are dual-port silicon RAM blocks of sizes 18-36kb, that may be alternatively configured as FIFOs. A wide range of configurations are available for the port width and depth. While the Distributed RAM provides a small, fast and local buffer, the BlockRAM provides a large global buffer. Hence, these are designed to complement each other, an important factor to consider during the development process.

### Which FPGA to Choose? 2.3.3

Our detector consists of 600 LYSO-crystals and 400 WLSs, adding to a total of 1000 channels. Ideally all these channels should be sampled with Multi-Gigabit Transceiver (MGT) lines at speeds up to 6.5Gbps (see tab. 2.1), but this would become too costly. The top-model Virtex-6 only contains 36 such pins. In comparison, the same model boasts 1200 GPIO-pins. These should only be used if the FPGA-model has SerDes-support (see 2.3.2.1).

The Spartan-3 is ruled out from the beginning, since it is too small, has no SerDes-support, and no Ethernet MAC. Furthermore, Virtex-6 requires ISE version 11 or 12, which this project currently lack a licence for. The options providing the best performance to cost ratio is thus Virtex-5 or Spartan-6, although of these the Virtex-5 is preferred. The performance is generally better, and the extra power consumption of the Virtex-series is of no significance in this detector.

To facilitate low design cost and rapid development boards containing the FPGA and common peripherals are used. Several off-the-shelf solutions exists, including prototype and evaluation-boards from Xilinx used to showcase their FPGAs.

### Evaluation Boards 2.3.4

Evaluation boards facilitates rapid development and low prototyping cost, but as they are designed to showcase certain FPGAs, a large share of the FPGA-pins are routed to various external IO peripherals. Naturally, this leaves less pins free for the front-end channels, which would require buying more cards.

Thus, as long as some basic peripherals are present, such as Ethernet PHY[25], JTAG, RS-232, and some RAM, the aim should be to find evaluation boards with as many GPIO-pins as possible. Some options are presented below.

[25] PHY is the physical layer in the OSI network model.

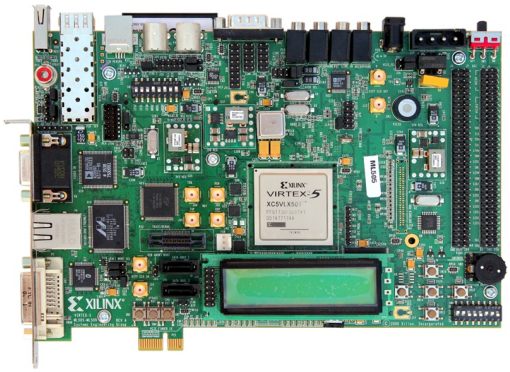
| **Xilinx Virtex-5 LXT Evaluation Platform** (ML505) | **Key Features** |
|---|---|
|  | • Xilinx Virtex-5 LX50T<br>- 28 800 FFs and LUTs<br>- 2 160kb BlockRAM<br>- 4 10/100/1000 Ethernet MAC<br>- 48 DSP48 slices<br>- 12 RocketIO Tranceivers<br>- 1 PCIs Express endpoint<br>• *256MB DDR2 SODIMM*<br>• 32MB Flash<br>• RS-232, JTAG, USB, *Audio Jack In/Out, PS/2, DVI, VGA, SATA*<br>• 1 10/100/1000 Ethernet PHY<br>• SAM connector: 16 LVDS-pairs |

*Table 2.2 - Key Features - Xilinx Virtex-5 LXT Evaluation Platform (ML505)*

The ML505 board is characterised by its large number of peripherals and a lot of memory, but at the cost of only 16 user LVDS-lines. The design described in this thesis was developed solely using this board, but the lack of inputs makes it unfit as a "Readout Card" (see 2.4).

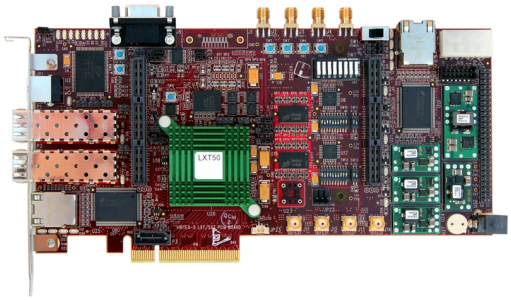
| **Xilinx Virtex-5 LXT PCI Express Development Kit** (V5LX-EVL50-G) | **Key Features** |
|---|---|
|  | • Xilinx Virtex-5 LX50T<br>- 28 800 FFs and LUTs<br>- 2 160kb BlockRAM<br>- 4 10/100/1000 Ethernet MAC<br>- 48 DSP48 slices<br>- 12 RocketIO Tranceivers<br>- 1 PCIs Express endpoint<br>• 64MB DDR2 SDRAM<br>• 16MB Flash<br>• RS-232, JTAG, USB<br>• *2 10/100/1000 Ethernet PHY*<br>• *EXP connector: 84 LVDS-pairs* |

*Table 2.3 - Key Features - Xilinx Virtex-5 LXT PCI Express Development Kit*

A much better choice is thus the V5LX-EVL50-G evaluation kit, which promotes an EXP connector with 84 LVDS-pairs, the same FPGA as the ML505 (the LX50T, see fig. 2.9), and two Ethernet PHYs! The latter is very interesting, because one may be used for high-speed data-transfer, and the other for control activities (through the embedded project, see 2.4.5).

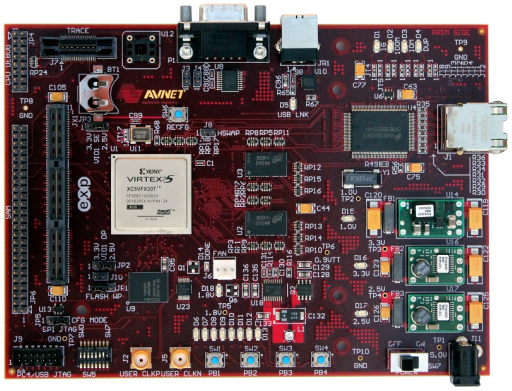
| **Xilinx Virtex-5 FXT Evaluation Kit** (V5FXT-EVL30T) | **Key Features** |
|---|---|
|  | • Xilinx Virtex-5 FX30T<br>- 20 480 FFs and LUTs<br>- 2 480kb BlockRAM<br>- *1 Embedded PowerPC core*<br>- 4 10/100/1000 Ethernet MAC<br>- 64 DSP48 slices<br>- 8 RocketIO Tranceivers<br>- 1 PCIs Express endpoint<br>• 64MB DDR2 SDRAM<br>• 16MB Flash<br>• RS-232, JTAG, USB<br>• 1 10/100/1000 Ethernet PHY<br>• 30 pins SAM connector<br>• $\frac{1}{2}$ EXP connector: 42 LVDS-pairs |

*Table 2.4 - Key Features - Xilinx Virtex-5 FXT Evaluation Kit (ML505)*

Finally, the design needs a common card in charge of triggering and clock distribution, a ”Trigger Unit” (see 2.4.2). The V5FXT-EVL30T might be fit for this job, because of its 42 LVDS-lines, PowerPC core, and higher number of DSP48 blocks. PowerPC based embedded networking are generally faster than the soft-core alternatives, which might come in handy if it is to control a large number of Readout Cards.

These cards were just examples, and may eventually not be used at all, but introducing them gives a general idea of what to expect in terms of functionality. Further note that most of the evaluation platforms comes with variable sized FPGA, where the models just described is in the lower end. Upgrades will be made should it prove necessary as more detector channels are added.

## *Functional Description* 2.4

A functional overview of the digital part of the data acquistion system is shown in fig. 2.4.

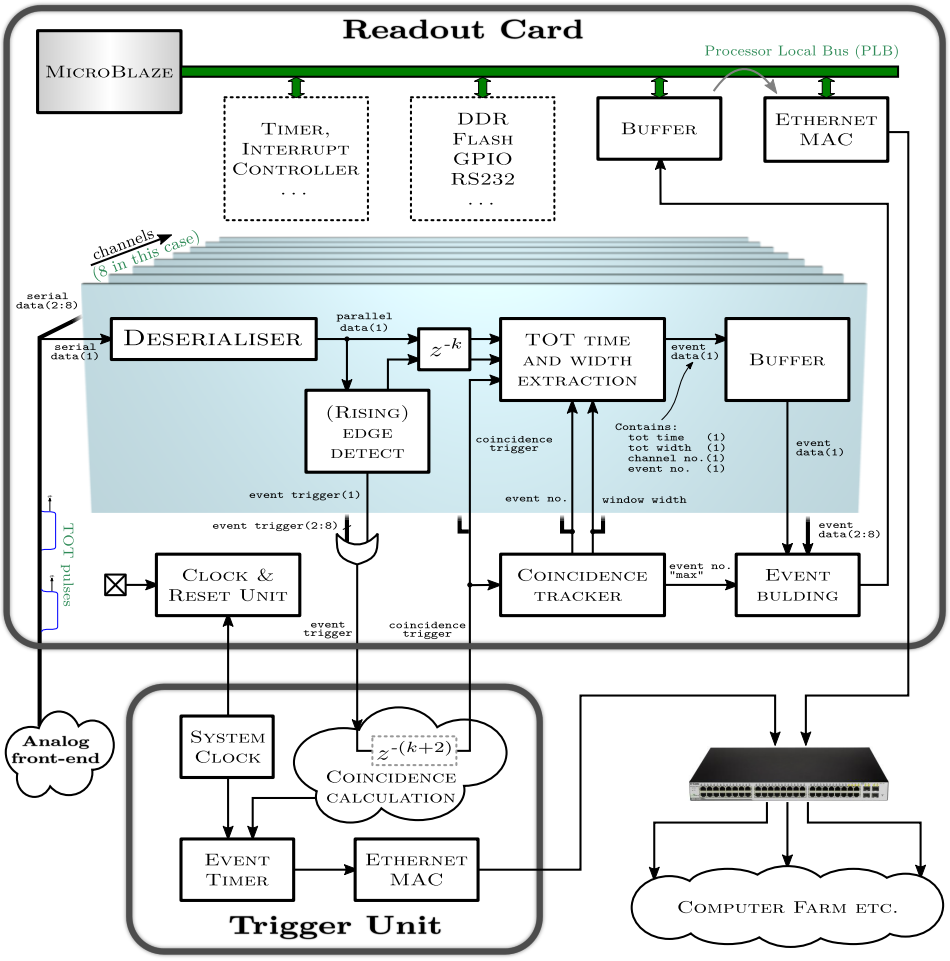



*Figure 2.4* - ComPET *readout system: Functional buildup*

The readout system consists of several distinct physical parts. The detector is split into 4 modules, each with 250 channels. These channels are pre-processed by the analog front-end electronics, and routed to a *Readout Card*. The currently considered evaluation boards provides 84 GPIO-pins, dictating the need for 3 of these per module, or 12 in total. The Readout Card concurrently samples each channel (2.4.1), sends and receives triggers to a Trigger Unit (2.4.2), computes the interaction *time* and *energy* from the TOT-data (2.4.3), fan-in and "builds" an event from the data over all channels (2.4.4), and sends these via Ethernet to a computer farm (2.4.5).

This section will introduce this functionality. Implementation details may be found in the next chapter.

### Data Capture 2.4.1

As shown in fig. 2.4, the TOT-pulses enter the Readout Cards from the analog front-end, one line for every channel (in this and upcoming examples, 8 channels are chosen for simplicity). To avoid high speed signalling in the FPGA-fabric, each of these are then sampled by on-board silicon deserialisers, yielding frames of 10 bits synchronised with the system clock (see 3.3 for details).

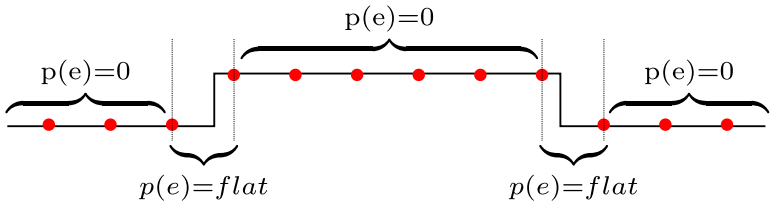



*Figure 2.5 - Visualisation of the TOT sampling accuracy*

The theoretical accuracy of resolving one of these edges, and thus the time-resolution, is given by

$$\alpha^2_{time}(f_s) = \int_{-1/2f_s}^{1/2f_s} e^2\, p(e)\, de = \frac{f_s}{3}\left[e^3\right]_{-1/2f_s}^{1/2f_s} = \frac{1}{12f_s^2}, \tag{2.1}$$

but the energy resolution, due to being inferred from both leading and falling edge, is twice that,

$$\alpha^2_{energy}(f_s) = \frac{1}{6f_s^2}. \tag{2.2}$$

Assuming a sampling frequency of 1 GHz, this translates to a standard deviation of

$$\alpha_{time} \approx 0.29ns \qquad \text{and} \qquad \alpha_{energy} \approx 0.41ns. \tag{2.3}$$

However, these are theoretical figures. Further distortion by *walk* and *jitter* in the analog front-end must be expected.

### Triggering and Windowing 2.4.2

A PET detector, due to employing numerous channels sampled at very high speeds, produces massive amounts of data. Fortunately, this data is also

very sparse; ComPET samples 1000 channels at 1 GHz, yielding a data-rate of 1Tbps, but with an event-rate of, say, 3 Mevents/s, the actual rate of decent data is in the order of 100Mbps (assuming 32 bits per event, see 2.4.3).

This is common to particle physics experiments, and is solved by employing highly concurrent analog or digital electronics to monitor the data, and - when something interesting occurs - ”trigger” data capture. What is interesting is usually defined by a few criteria, and - depending on how long it takes is to evaluate - may be grouped into levels.

The top level trigger, level 0, is a fast (latency typically in the ns-range) real-time trigger used to capture raw data from the sensors. Traditional PET-systems uses either an analog or digital[26] level 0 trigger to enable data capture with ADCs. The equivalent in our system is the voltage threshold in the analog electronics, which discards gamma-rays with an energy less than 50 keV[27].

The next trigger in ComPET, level 1 (tagged event trigger), is asserted when one, or several, TOT channels contains a leading edge (see 3.4.2.1). Combinatorial logic is added to each channel to produce the trigger (the ”Edge Detect” block in fig. 2.4), and an OR of these from all channels are sent to the Trigger Unit. Depending on some coincidence criterion, the Trigger Unit responds by asserting a coincidence trigger window.

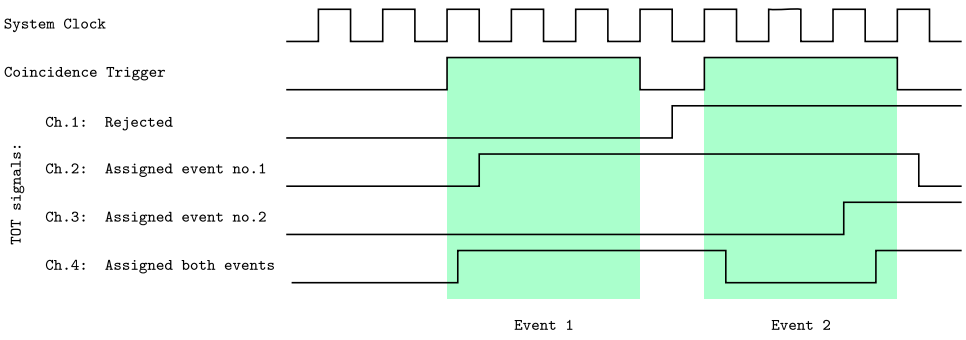



*Figure 2.6 - Coincidence validation*

The coincidence window serves three purposes; it discards events which are too far apart for improved contrast sensitivity, allows all events within the same window to be assigned a unique event number, and can be used to

[26] The level 0 triggers can either be local ”asynchronous” triggers, or global ”system synchronous” triggers.

[27] With 3D-readout this threshold may be set lower in order to include Compton scatters. Clever signal processing algorithms can use this extra information to improve image quality [5].

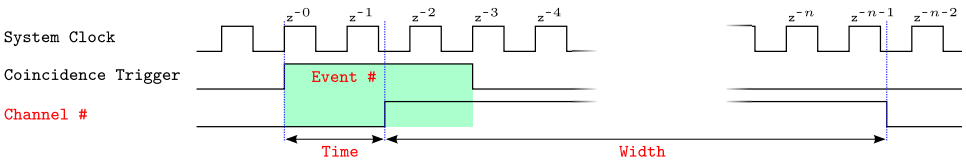



*Figure 2.7 - Event parameters*

specify the relative time of interaction (see 2.4.3). This information is in turn used by the Event Builder (see 3.5) to collect event data from all channels and sort these according to event number. Another fortunate effect of tagging events with event numbers is that the latency of the system becomes largely unimportant.

### Parameter Extraction 2.4.3

When an event occurs, the TOT-*time* and *width* is computed, and the relevant *channel number* and associated *event number* is recorded (tab. 2.7). These 4 parameters combined form an *event packet*, which should be as small as possible, and preferably a power of two for maximum performance. Since Microblaze processor has a 32-bit data-bus, setting the event packet size to 32-bits is probably wise.

To accommodate this requirement, an obvious challenge is that the TOT-time can not be specified absolutely. This was the case in an early version of the design, but then every event required two Ethernet packages - one dedicated to the TOT-time alone. However, the TOT-time is only used to separate events within the same coincidence window, and as such only the relative timing difference of these events are necessary.

Even if the final detector design will not require absolute timings, these should somehow be attained to aid in the detector verification. This is possible, because the Trigger Unit must source a common clock to all Readout Cards in order to attain synchronised coincidence windows and event number counters. Thus, if one of the Readout Cards - or the Trigger Unit - keeps a system time counter running and maintains a look-up table relating event numbers to the corresponding system time, the table may be read out over a dedicated line, or after the scan has completed, to reconstruct absolute timings for all events.

#### *Size Specification* *2.4.3.1*

Let us first consider the *relative time* of the TOT-pulse. The absolute requirement for this value is that it spans an interval large enough to cover two

frames of bits[28], but should naturally cover all values in the coincidence window. By using 7 bits, the relative time spans $2^7/f_s = 128\text{ns}$. It is unlikely for the coincidence window to be wider than this.

Next, the *width* of the TOT-pulse has the same absolute requirement, but should cover the entire TOT-width range. The current analog front-end produces TOT-pulses with widths rarely exceeding $1\mu\text{s}$, in which case a parameter size of 12 bits, or $2^{12}/f_s = 4.096\mu\text{s}$, should suffice.

Setting the *channel number* size is easy, just add enough bits to be able to address the channels the design is compiled for. Currently 6 bits are used, dictating a maximum of 64 channels. The final design will most likely require 84 channels, in which case another bit must be added.

Finally, the size of the *event number* counter is the hardest one to set. It must be sufficiently large to make sure it does not wrap to an event number that is potentially already being processed. Consider a case where one of the channels receives an event with an TOT-width close to $2\mu\text{s}$, while the remaining channels successively receives TOT-pulses with 10ns separation. To achieve a generally high efficiency of the system, short and frequent coincidence windows should be used, but this increases the risk that the event number counter wraps before the descending edge of the very first long TOT-pulse. This will lead to incorrectly assigned event numbers, and possibly cause system-lockup (because the event building structure is not designed to handle this exception, if at all possible).

Thus, if the coincidence window is made shorter or the maximum length of the TOT-pulse is made longer, more bits must be added to the event number. In the case of a shorter coincidence window, less bits are needed to specify the relative time, thus the overall bit requirement should not increase.

The above relations may be condensed into 3 equations:

$$\frac{2^{B_{time}}}{f_s} \overset{optimal}{>} \tau_{co.win.} \overset{absolute}{>} 2\,W \tag{2.4}$$

$$\frac{2^{B_{event\ no.}}}{f_{co.win.}} \overset{optimal}{>} \frac{2^{B_{width}}}{f_s} \overset{absolute}{>} 2\,W \tag{2.5}$$

$$B_{time} + B_{width} + B_{channel\ no.} + B_{event\ no.} < 32 \tag{2.6}$$

Here $\text{B}_x$ is the number of bits for parameter $x$, W is the deserialisation width, $\tau_{co.win.}$ is the length of the desired coincidence window, $f_s$ is the sampling speed, and $f_{co.win.}$ is the rate at which the coincidence window repeats. A preliminary suggestion for parameter sizes is summarised in tab. 2.5.

[28] This is due to how it is implemented.

| Parameter | Bits | Coverage |
|---|---|---|
| Event number | 7 | $2^7/f_s = 128$ns. |
| Relative time | 7 | $2^7/f_s = 128$ns. |
| Width | 12 | $2^{12}/f_s = 4096$ns. |
| Channel number | 6 | $2^6 = 64$ channels. |

*Table 2.5 - Parameter size suggestions*

The parameters may be set in the design file `constants.vhd`, see B.1.1.

### Event Building 2.4.4

At this point the relevant parameters have been extracted from the TOT-pulses and stored in the channel buffers. The question now is what to do with it. Obviously, the data must be *collected*, but it should also be *sorted by event number*.

There are two reasons for this; if the data is sorted in hardware, time is saved in the post-processing, and ”bundling” all events from the same coincidence window together allows for distributed processing of these events (see 2.4.5).

This operation will be referred to as ”building events”. The output of this process should be a single stream of data, with events grouped by coincidence window, each of which appearing chronologically. Ideally, this should be done with 100% occupancy, meaning a new event will be appear at the output at every single clock cycle. This is important, since every event must pass through this structure, its throughput will then determine the system throughput.

The decision tree shown in fig. 2.8 shows how this can be achieved. Each top node represents a channel, and each of these nodes receive the eldest data in the channel buffers (values chosen at random). Due to implementation technicalities presented in 3.5, the decision tree must be implemented in a pipeline.

This means that, in order to achieve 100% occupancy, all decisions must be made on the first level. Instead of making any decisions inside the pipe, care must be taken to ensure that only a single event enters the pipe at any given clock cycle. All nodes inside the pipe are setup with predefined priorities, so there are no ambiguities as to which event will be routed to the output.

### Embedded Networking 2.4.5

As seen in fig. 2.4, the last part of the project the data passes through prior to moving out on the network is an embedded project. The word

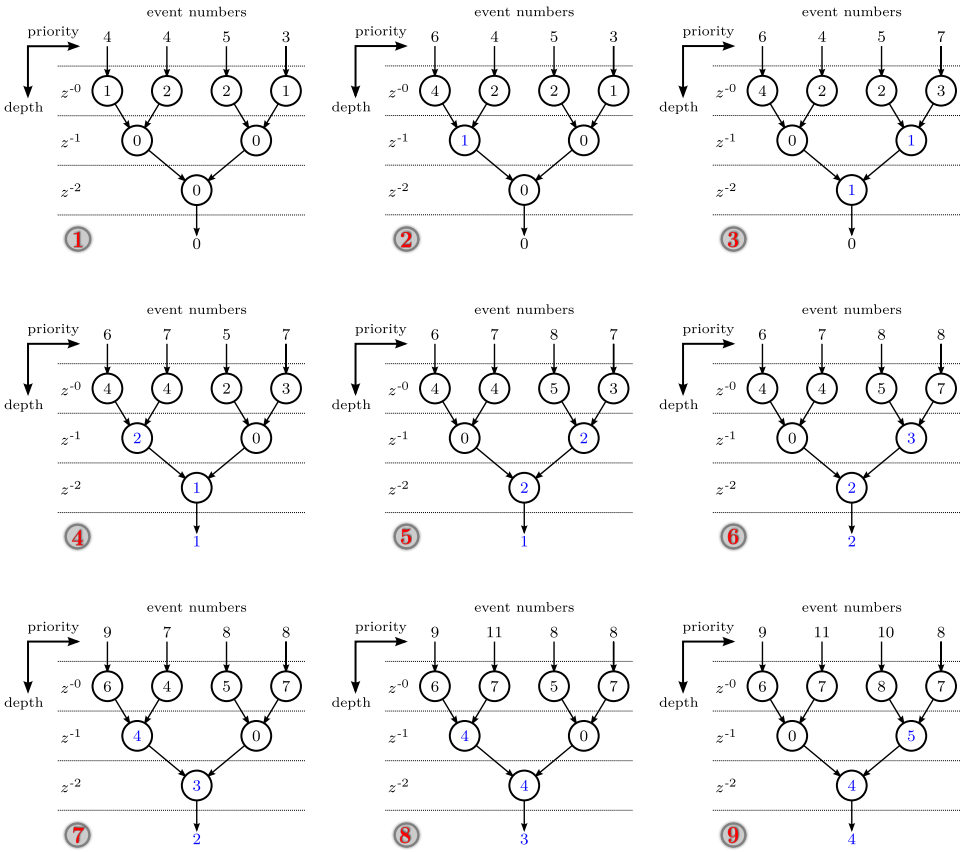



*Figure 2.8 - Event Builder decision tree*

embedded implies the use an microprocessor, and this design runs a ”soft” micro-processor from Xilinx called Microblaze. This is hosting an embedded Linux system called PetaLinux, capable of performing various control and network activities.

When this system sends packets over the network, different destination addresses can be specified. Now the reason for building events like described becomes apparent. If a single network processing node can not handle the amount of data sent to it, the Readout Card can e.g. send even numbered events to node 1, and odd numbered events to node 2, or similar. In other words, this is a way to distribute the load over as many processing nodes as needed in order to avoid that these become the performance bottleneck in the system.

For more information on the embedded project, see 3.6 or have a look at the ”embedded tutorial” (C).

### *Summary* *2.5*

ComPET employs a compact geometric structure of LYSO-WLS matrices that allows for a very high sensitivity, spatial resolution, and contrast sensitivity. The light yield of these is converted to electrical energy by GAPDs, and sent to a custom analog front-end, which "reshapes" it into a pulse with a very steep rising edge and a near-linear falloff. A discriminator compares this pulse with an energy threshold, with a resulting binary output pulse; its rising edge being a measure of the interaction time, and its width - or "Time over Threshold" (TOT) - being a measure of the interaction energy.

These parameters, along with the channel coordinate, are used to identify photoelectric interactions, or the first interaction point of Compton scatters. A Line of Response (LOR) is drawn for each of these interaction pairs, and from these an image is computed. Its quality will depend on the accuracy of each LOR, the number of LORs available, and the probability that each LOR match $\gamma$-rays originating from the same radioisotope.

This impacts the readout system in several ways. To accurately estimate the LORs, the TOT-pulses must be sampled very high speeds. To increase the number of LORs, or the SNR, the throughput should be high enough to avoid buffer overflows, and dead-time as low as possible. And finally, to filter out events that has no coinciding event, a coincidence window must be applied to all channels. This implies communication across channels, and to an external "Trigger Unit", hence adds to structure and timing complexity.

A technology very capable of fulfilling these criteria, Field Programmable Gate Arrays (FPGAs), are chosen. The FPGA design captures the TOT-data using GPIO deserialisers, greatly compresses it by performing coincidence-window validation and parameter extraction, collects and sorts data from all channels, and sends these out over an Ethernet network for further processing.

Next we will look at the design implementation (3).

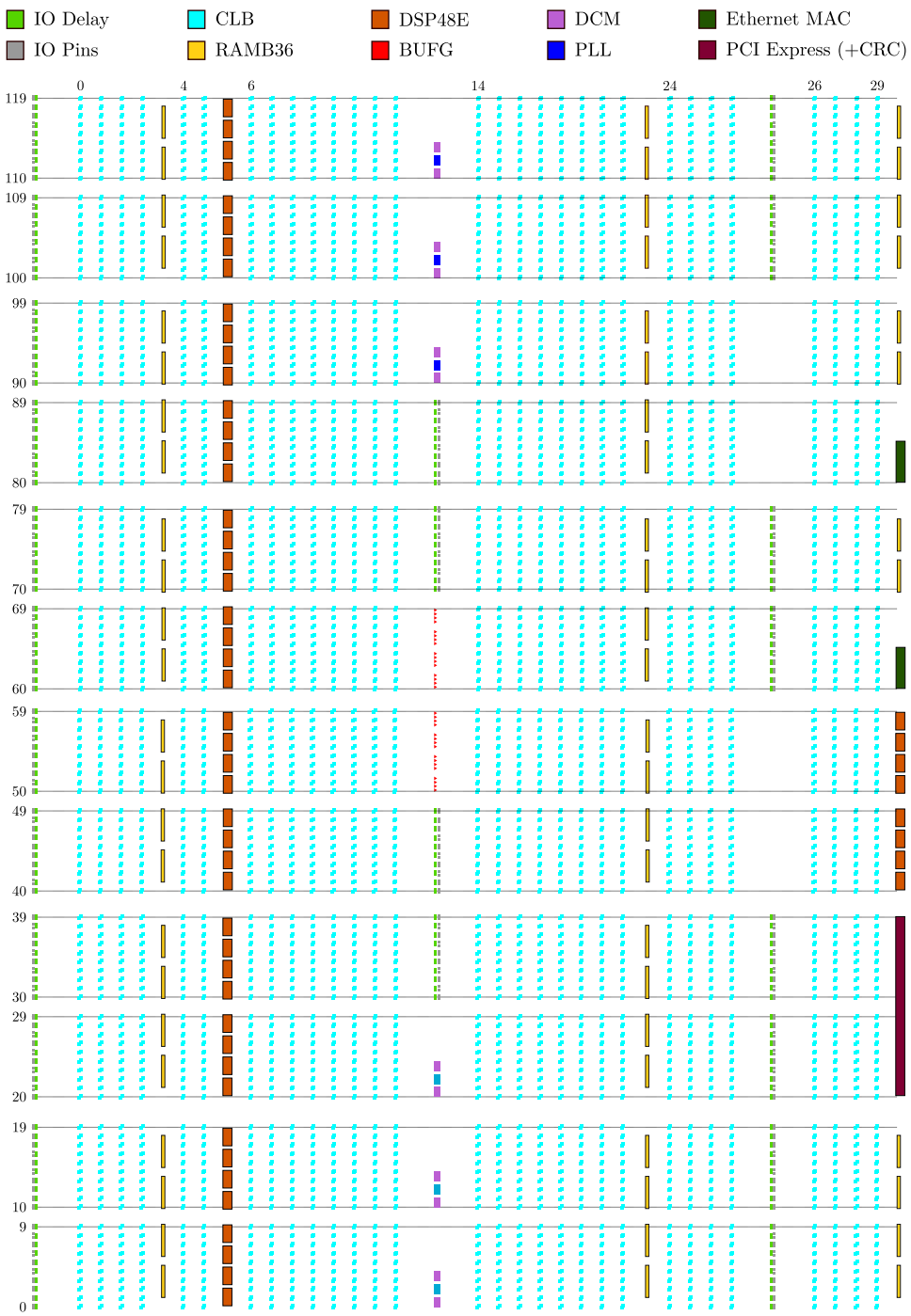



*Figure 2.9 - The Virtex 5 LX50T internal architecture*[29]
*(for curiosity only, to get a feeling of what an FPGA is)*

[29] Made from information provided by the Xilinx ISE Floorplanner.

*"Once a new technology rolls over you, if you're not part of the steamroller, you're part of the road."*

Stewart Brand

# 3 Implementation

Following up on the previous chapter, the basic functionality of the digital readout system should now be known, although implementation details remains to be described. Following the data path, the system must capture data from all sensors (3.3), compress it by means of triggering and parameter extraction (3.4), collect and sort data from all channels (3.5), and finally send this over the network (3.6).

In addition to these topics, a few words will also be dedicated to the clock and reset scheme (3.2), and design control (3.7).

First, however, an introduction to project versions and the Hardware Description Language (HDL) used throughout this project will be presented.

## *Introduction* *3.1*

This design can be split into several parts. Most of the design is written from scratch in HDL (see 3.1.2), with the main exception being the embedded project (see 3.6, C), which is composed of soft and hard Intellectual Property (IPs) cores. The embedded project also runs PetaLinux, which requires a few Linux kernels and the GNU tool-chain to be built. This adds to the project size considerably, which is currently exceeding 100 000 files and 2.7GB(see B).

Thus, for the each implementation description in this chapter the most important design files will be identified. To obtain either of these, use the hyperlinks, or look up the file location in B.1.1. This should be straightforward, as care was taken to ensure important source files never got mixed with output files from the build tools.

### Project Versions 3.1.1

The version control system used to track this project is called Git (see A.4.1, B.2). Git tags commits by default with SHA-hashes, but allows human friendly version numbers be manually specified.

Only two major versions of this project have been specified, version 0.1 and 0.2. These designs are both stable, and represents the current state of the project when the various readouts were conducted (see 4.3).

Version 0.1 was intended as a proof-of-concept; a single channel implementation that were able to extract TOT absolute time and width, store this as 32 bits in a BlockRAM buffer, and send it out over an Ethernet network using the embedded design.

Version 0.2, on the other hand, is a large upgrade over the first design. It can handle an arbitrary number of channels, is able to extract TOT-width and relative time from events, validates and tags these by coincidence, and features the 100% occupancy event-builder and fan-in structure. It was also made very general, to improve flexibility and portability, and reduce the need for maintenance.

Since version 0.2, a few optimisations and corrections have been implemented. This is nothing too severe, but will nonetheless be taken into account in this thesis. To synchronise the project with the contents of this document, all repositories will be tagged version 0.21 at the time of print.

### Hardware Description Language 3.1.2

An FPGA design is described using a Hardware Description Language (HDL). For this design, two prominent alternatives were considered; Verilog 2001 and VHDL-1993. Verilog is a C-like, loosely typed language, developed with emphasis on time to market. VHDL, on the other hand, is a strongly typed language based on Ada, that was originally intended as a documentation language for large gate-arrays. Unlike Verilog, it supports abstract data types and descriptive functions, which makes it very flexible and "formal". For this reason, VHDL was chosen for this project.

It should be noted that more recent versions of these languages, like SystemVerilog 2005/2009, or VHDL-2008, are far more powerful, but is not supported by Xilinx XST[1]. Alternative synthesisers, such as Mentor Graphics Precision or Synopsys Synplify, supports these languages at least partially, but since mixing software adds to the project complexity and licence costs, it was decided against in this project.

For simulations, however, this design use Mentor Graphics Modelsim (see A.3.2), which supports SystemVerilog. Thus, all testbenches are written in SystemVerilog.

As for the choice of code syntax and conventions, this was chosen to conform with the Motorola Freescale HDL conventions [22], which seems to be generally promoted throughout the HDL community.

[1] The current last release of ISE, version 12, still lacks support for SystemVerilog 2005. Support for the even more recent VHDL 2008 and SystemVerilog 2009 seems to be even further off (according to Xilinx forums).

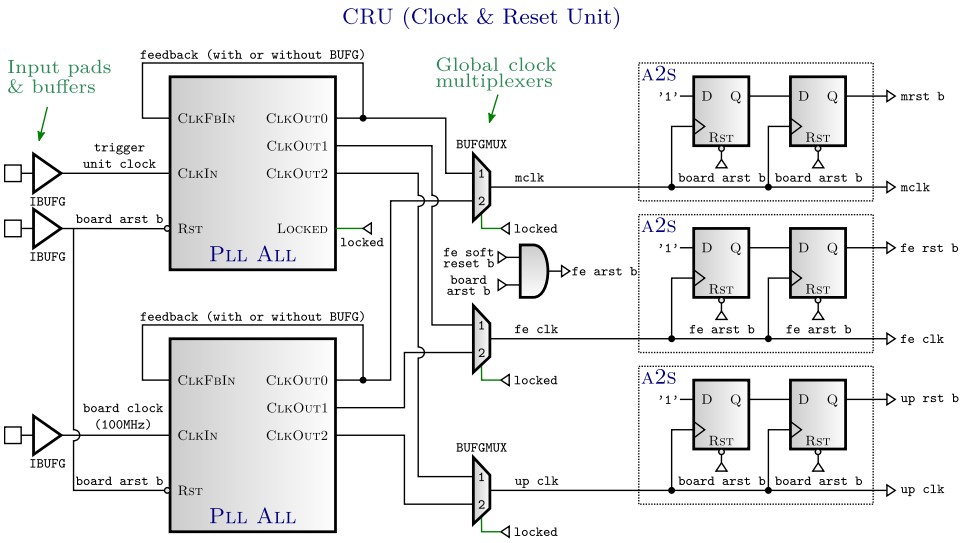



*Figure 3.1 - Clock Reset Unit (on a Readout Card)*

## *Clocks and Resets* 3.2

The ComPET Readout Cards has a common clock domain for most of the logic; a ”master” clock running at 100 MHz. The exceptions are a front-end clock running at 5 times the master clock speed required by the deserialiser[2], and a microprocessor clock running at 125 MHz. The design unit in charge of sourcing these and associated resets is the Clock Reset Unit (CRU, fig. 3.1).

| Design Unit | File |
|---|---|
| top | top.vhd |
| ∴ cru | cru.vhd |
| ∴ pll_all | pll_all.xaw |
| ∴ a2s | a2s.vhd |

*Table 3.1 - CRU - associated design units*

### Motivation 3.2.1

Designing a solid clock and reset scheme is a task whose importance is largely underestimated. Since these signals are global, incorrect implementation may cause errors that are non-repeatable and seemingly non-causal, which makes the debugging process very difficult. Thus it makes sense to group all reset and clock functionality into a single design unit. This also promotes portability (which is nice because this functionality often similar across designs), code cleanness and structure. Basically, the *only* place where clocks and resets signals should be altered is in this unit. It might seem unnecessary to be so strict, but not having absolute control over these signals is a recipe for disaster, and will lead to countless hours of debugging.

[2]This clock rate, while always being a integer multiple of the master clock, depends on the word width and clock mode (DDR/SDR). See [17, page 371, table 8.5].

### Implementation 3.2.2

The Readout Card should use the Trigger Unit clock as a source when available, and when it is not use one of the on-board clocks. The system shown in fig. 3.1 allows this switching to happen automatically. Two PLLs are setup with exactly the same settings, one driven by the internal board clock and the other by the Trigger Unit clock, and the output clocks are sent to global glitch-free clock multiplexers. These multiplexers are controlled with the `locked`-signal from the external clock PLL, such that if this PLL is able to lock on the external clock, it will be used to drive the design. This plug-and-play feature allows cards to be immediately synchronised with a Trigger Unit when it is connected (which is required), but continue operation even if disconnected.

Choosing whether to use asynchronous or synchronous resets is a frequently debated topic. Flip-flops with asynchronous reset is guaranteed to be cleared upon assertion, but unexpected behaviour might arise if the deassertion happens just prior to clock capture. The solution seems to be to use synchronous flip-flop reset inputs, but if the reset is too short it might not get registered. Furthermore, only some vendors (e.g. Xilinx or Altera) support components with synchronous resets[3], and in ASIC-designs this is not supported at all. To promote portability a solution is use the component asynchronous reset input (which is supported by everyone), and make sure that the *deassertion* of the reset lines is *synchronous* [6]. As shown in fig. 3.1, this is achieved by means of a few flip-flops.

It is possible to reset the FPGA front-end logic with a *soft-reset*, which may be controlled through the ChipScope VIO interface (see 3.7.2). It may be used to reset the event-number counter, or to synchronise all the Readout Cards.

### Conclusion 3.2.3

The Clock Reset Unit seems to be working as expected when it is running on a single clock. However, the clock multiplexing has not yet been tested, and will remain as a task to be carried out when the Trigger Unit is being designed. It is likely the design will have to be soft-reset after a clock switch, but this remains to be tested as well.

[3] It should be noted that Xilinx highly recommends the use of synchronous resets altogether, to achieve better performance and slightly less logic usage with their FPGAs. Unless portability is a concern (which it should be), then this is naturally the preferred solution.

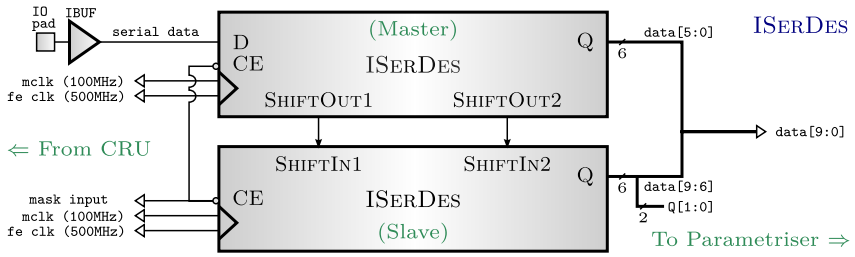



*Figure 3.2 - Deserialiser, 10 channels, DDR-mode*

## *Data Capture* 3.3

To achieve a good time-resolution the TOT pulses are captured and deserialised using the silicon SERDES blocks residing in each GPIO-tile. These silicon-blocks can be run at 1-1.25 Gbps in a Virtex-5 [15, p29][19, p32], depending on speed grade.

| Design Unit | File |
|---|---|
| top | top.vhd |
| ⋮∴ fe | fe.vhd |
| ⋮∴ fe_ch | fe_ch.vhd |
| ⋮∴ fe_ch_iserdes | fe_ch_iserdes.vhd |

*Table 3.2 - ISERDES - associated design units*

### Implementation 3.3.1

A deserialiser always require a bit-clock and a frame-clock, to sample the bits and frames, respectively. To achieve the highest attainable sampling rate the SERDES block is setup in DDR mode, allowing for a sampling speed of 1 GHz with a 500 MHz front-end clock (and a 100 MHz frame-clock)[4]. This mode requires the cascading of two SERDES-modules (fig. 3.2).

A common request is to be able to disable channels, either block misbehaviour, or for testing or calibration purposes. This is possible though the control interface (3.7.2). If a channel is disabled, the SERDES clock enable will be deasserted to save power.

### Conclusion 3.3.2

Using the GPIO-tile SERDES functionality to capture the TOT pulses facilitates reduced detector cost, allows for easy scalability, and yields a time-resolution of less than 1ns. However, as it turns out, the idea is not new. A group at the University of Cheerbrooke, Canada, implemented a similar TDC with a Virtex-4 [8]; The input pulse was copied and sent to 4 GPIO-pins, each running a 800Mbps ISERDES with input clocks shifted a quarter of period relative to each other. This is essentially how the GPIO deserialiser works, they just expanded the functionality it by utilising more pins.

A challenge when spatially oversampling a signal is to ensure linearity, but the mentioned group reports good results here. The question, however,

[4] To see all possible modes, see [17, page 371, table 8.5].

is whether this is cost-efficient. The reason for using GPIO-pins in this design was to reduce cost, but if several GPIO-pins are required to achieve the desired time-resolution, it might be worth reconsidering the use multi-gigabit transceivers (MGT) as well.

Using 10-bits frames, as opposed to e.g. 8 bits, may not seem optimal as this is an "awkward" binary size. However, computations based on these data are performed in Multiply-Accumulate (MAC) blocks, meaning hardly any extra logic will be required. Thus, it seems wiser to use 10-bits frames to keep system clock speed at a minimum, ultimately minimising power consumption and timing complexity.

## *Triggers and Parameters* *3.4*

The generation of triggers and data filtering includes several design units. A per-channel Parameter Extraction module (`fe_ch_pargen`) tracks rising and falling edges, and in case of the former sends an event trigger to the Trigger Unit[5]. Continuously it keeps extracting parameters from the data, and puts these into a delay-line long enough for the output to be synchronised with the coincidence window.

| Design Unit | File |
|---|---|
| top | top.vhd |
| ⋮ fe | fe.vhd |
| ⋮ ⋮ fe_cotrg_proc | fe_cotrg_processing.vhd |
| ⋮ ⋮ fe_ch | fe_ch.vhd |
| ⋮ ⋮ ⋮ fe_ch_pargen | fe_ch_pargen.vhd |
| ⋮ ⋮ ⋮ shift | shift.vhd |

*Table 3.3 - Triggers and parameters - associated design units*

This window is monitored by a global module (`fe_cotrg_proc`), which keeps track of event numbers and the time passed since window assertion. These are important event parameters, thus are kept available for all Parameter Extraction modules.

### Motivation 3.4.1

The complexity of a digital readout system increases with the number of sensors involved, the amount of data to be processed from each, the time-constraints imposed on the system, and the degree of intermediary communication required between channels.

This suggests compressing the data as early as possible in the processing chain, just after the deserialiser, to avoid exhausting the buffer capabilities in the FPGA, and to relax the timing constraints imposed on the remaining part of the design. The challenge is to perform this compression with minimum logic utilisation, to allow the design to scale well towards higher channel numbers. Further adding to the challenge is the aim to create a design that is flexible, easily maintainable, and reliable.

[5] Ultimately, this will be the case, although no Trigger Unit yet exist.

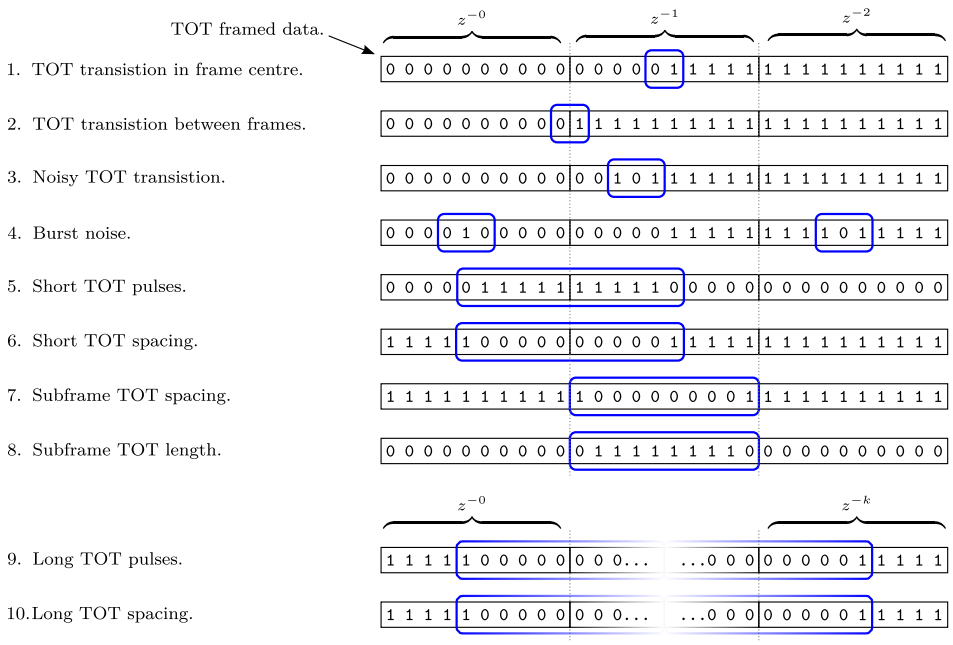



*Figure 3.3 - Edge detection, various conditions*

### Implementation 3.4.2

The first step is to detect the frames containing edges (3.4.2.1), then measure the edge location within these frames (3.4.2.2), and compute all other parameters based on this.

#### *Edge Detection and Triggering* *3.4.2.1*

The presence of an edge within a frame may be detected with either sequential circuitry running at the sampling frequency, or combinatorial logic. High speed sequential solutions tend to require less logic, but are more error prone, harder to debug, and more power-hungry than the equivalent combinatorial solutions. In turn, using combinatorial logic is less intuitive, since a common Boolean equation must be derived to cover the entire range of data input scenarios.

Some of these are illustrated in fig. 3.3. Notice that the simplest case to handle is when a TOT transition occurs in a frame centre (case 1 in fig. 3.3), since here all the information required to perform the correct decision is obtained from the single frame alone. When a TOT transition occurs between two frames (2), however, the data from both are needed to ensure the correct detection. For these scenarios, the following simple implementation should do the job (fig. 3.4).

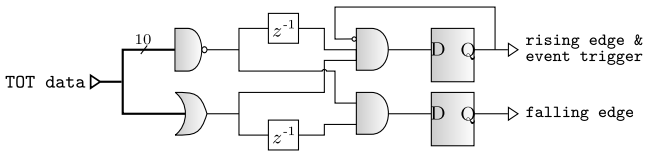



*Figure 3.4 - Edge detect circuit*

This circuit flags a rising edge condition when a current frame contains ones, and the previous one contained zeros. Vice versa for the falling edges. Note that while this approach handles most scenarios depicted in fig. 3.3, it is likely to fail in the case of burst noise (4) and sub-frame edge separation (7,8). However, noise has so far only been observed on the TOT falling edge[6], and the probability of events with less than 10ns separation is very low.

#### *TOT Time and Width* *3.4.2.2*

Since the TOT-data is segmented into frames synchronised with the system clock (fig. 3.3), it makes sense to adapt the terminology of *"coarse"* and *"fine"* timings; the former indicating a timing resolution equal to the system clock, and the latter a resolution equal to the sampling clock (fig. 3.5).

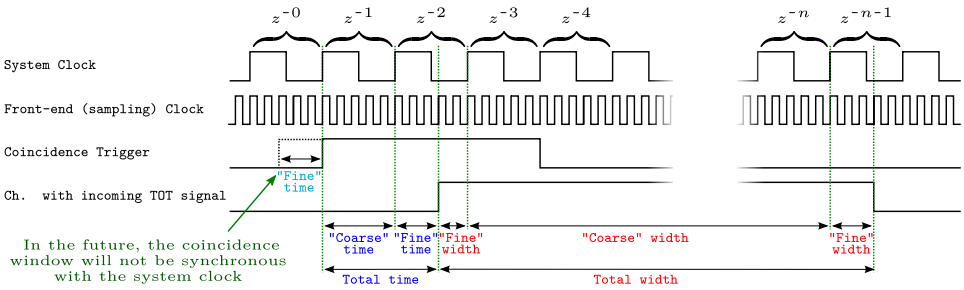



*Figure 3.5 - Parameter extraction*

As may be seen from fig. 3.5, the total time passed between two edges may be decomposed into three parts; the (fine) time of the first edge relative to the end of the first frame, the (fine) time for last edge relative to the start of the last frame, and the (coarse) time of the number of full frames in between. In terms of number of bits, this may be expressed as

$$\begin{aligned} E\{TOT\} &= E\{\tau_{falling\ edge} - \tau_{rising\ edge}\} \\ &= \frac{1}{f_s}\Big(W\, N_{full\ frames} + N_{1\text{'s in first frame}} + N_{1\text{'s in last frame}} + 1^{7}\Big). \end{aligned} \tag{3.1}$$

[6]The tail of the detector response is flat compared to the speed of the sampling clock, making it the falling edge more susceptible to noise. However, a properly adjusted discriminator hysteresis should correct this.

The current implementation computes the TOT-time and width this way; the number of ones in the first and last frame is found by bitwise SUM, and the number of frames is found by simply starting a counter at the rising edge, and stopping it at the falling edge (fig. 3.6).

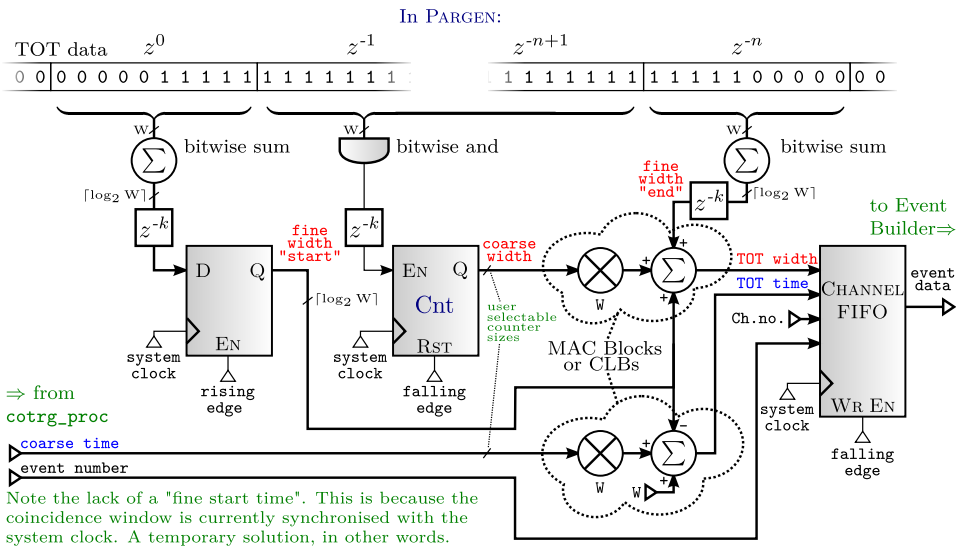



*Figure 3.6 - TOT time and width computation*

Note that while the TOT-width is found by counting the number of 1's in the data-stream, the TOT-time is found by counting the number of 0's between coincidence window assertion and the rising edge. The coarse TOT-time is measured by the `cotrg_proc` block in a similar fashion as the coarse TOT-width above, and the fine TOT-time is found by subtracting the number of 1's in the first frame from the deserialisation width $W$.

Fig. 3.6 also shows that the design calculates the total TOT-width using either Multiply Accumulate (MAC) blocks or CLBs, the decision is left for the synthesiser to make. If logic needs to be saved in the future, or the DSP-slices somehow can be put to better use, the coarse and fine values may be kept separate and merged in the post-processing instead. However, a greater dynamic range can be covered if this calculation is performed in the FPGA[8]. Also, knowing that the network throughput is likely to become

[7] The expectation value of the pulse width equals the number of 1's, plus one.

[8] For example, consider a case with 10-bit frames, where maximum 12 bits can be used to measure the TOT-width. The bitwise sum of two such frames can range from 0 to 20, requiring $\lceil \log_2 20 \rceil = 5$ bits. This leaves 7-bits for the coarse width counter, allowing a maximum TOT-width of $\sim 2^7(+1) = 128(129)$ frames, or $128\,W = 1280$ bits. Really disappointing, knowing that 12 bits can potentially cover $\sim 2^{12} = 4096$ values. The only way to "fix" this is merge the fine and coarse information in logic using a bit of two more than necessary, and only truncate the end result.

a bottleneck in the final system, it is probably wise to make every bit as valuable as possible.

Note that in order to save logic, the bitwise AND, OR, and SUM is delayed instead of the data itself. The delay is introduced using LUT-based shift-registers, each being able to shift a single bit up to 32 clock cycles. For a 10-bit deserialiser the parameters amount to 8 bits, allowing two LUTs to be saved per channel. Additional logic is also saved by reusing the bitwise AND and OR, instead of recomputing these after the delay.

The timings of the data capture in fig. 3.6 is described using a simple Mealy state machine (fig. 3.7), optimised for latency (4.2.1). It captures the TOT time and width, and stores these - along with the associated channel location and event number - into a 32-bit event packet in a channel FIFO (3.5.2). What happens with the data after this is up to the Event Builder (3.5).

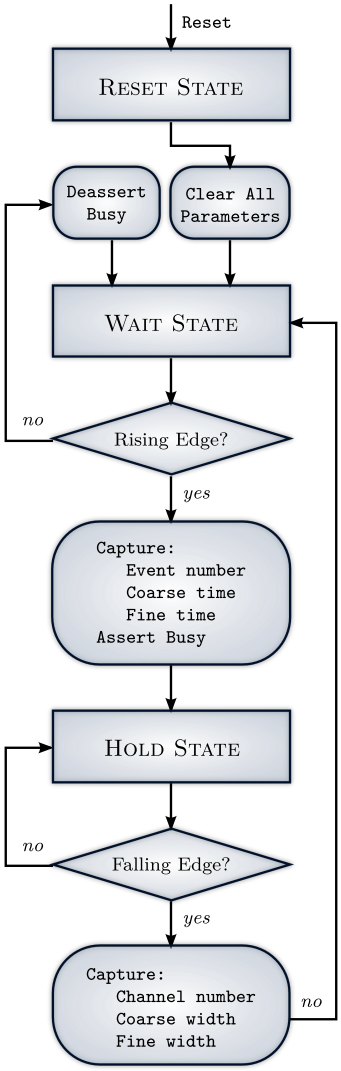



*Figure 3.7 - State Diagram - Parametrisation Filter (simplified)*

***Exception Handling*** *3.4.2.3*

The extraction of parameters from a TOT-signal may seem intuitively easy, but is complicated by the need to handle exception cases, such as when the rising or falling edge of a TOT-pulse ends up between two frames. In these cases the state machine (as it is shown in fig. 3.7) will ”overshoot” into the next frame, causing incorrect parameters to be computed. Creating some sort of hack for this is not too tricky, but finding a generic solution that allows for parameter modifications, without adding any unnecessary design logic, is somewhat less obvious.

The strategy deployed to handle these conditions consists of two parts. First, a timing model where all parameters are correct at any given time is selected. This implies that all fine timings are computed solely with combinatorial

logic, and that the sequential circuitry used to track coarse timings is advanced in time by one clock cycle[9].

Second, some logic for preventing the overshoot, and advance data capture, is added. More specifically,

- the state machine is prevented from overshooting. This is achieved by modifying the "falling edge condition" depicted in fig. 3.4 to take into account the next value of the bitwise OR, which will be 0 when the TOT-edge comes between frames.
- data capture is forced. Aborting the state machine invalidates the data capture condition, thus the data will be lost unless a capture is forced.
- the coarse width counter is stopped. Its running criterion is the same as the state machine, causing it to overshoot as well. The conditions under which it usually increments must now be invalidated.

Note that this will fail if the next frame contains one or more 1's, either due to a new event or noise (5.1.2). This is a problem with the entire design; because a state machine is used to keep track of whether the last edge was rising or falling, it will be very hard to make it handle both subframe pulse widths and subframe event separation. Because the former is considered far more important, a simple `busy`-signal is generated to make sure the two frames following a falling edge is ignored. This will make sure no data is corrupted, and since this condition is unlikely anyhow, the loss of SNR should be minimal.

### Conclusion 3.4.3

Although this design has evolved to become very reliable, there are ways to make it perform better. However, the options are limited if emphasis on low logic usage is to receive the same weight as it has so far.

This emphasis is seen throughout this design with the choice of a very simple edge detection condition, delay of parameters instead of data, and streamlined data extraction procedure. Only a bitwise AND and OR is used as control signals, and all event parameters are extracted using bitwise SUM and a few counters. Even the state machine runs off these variables, requiring very little extra logic.

However, even when being this conservative, the logic consumption is rather high. A rough estimate for the per-channel logic instantiated for 84 channels

[9] For example, event number must be ready at the exact same time as coincidence window assertion, in case the event ends already in the first frame.

indicates that more than 20% of all registers, and 35% of all LUTs available in the LX50T FPGA will be used. The embedded project, event builder, and debugging cores take up most of the remaining logic, so unless some of these are removed there is not much logic left to play with.

The only ”waste” of logic in this design is the use of a bitwise SUM to find edge position, instead of a priority encoder. The reason for selecting the former was to increase noise tolerance. However, so far noise has not been a problem, and if this persists then using a priority encoder is probably a better solution. Using a priority encoder will fix the TOT-time problems when the TOT-width is less than a frame wide (see 4.2.2 and 5.1.2), but break the currently perfect extraction of TOT-width under the same conditions. To make both perfect with this condition, another priority encoder can be utilised to scan the frame from the other end, or the use of bitwise SUM and a priority encoder can be combined.

In this design the event trigger and coincidence window is synchronised with the system clock. In a future design each Readout Card should also mediate the relative position of an edge within the frame to the Trigger Unit, to allow it to produce a coincidence window with a resolution equal to the sampling rate. This can be achieved over a single line using SERDES at each endpoint, or in parallel using 4-5 lines for the trigger and position, respectively. The former is preferable because less logic is required to synchronise two SERDES modules than to multiplex the data-lines, and because no extra data lines will be required.

If someone feels really tempted, there is also a possibility that the state machine here can be completely replaced by combinatorial logic. This would mitigate any timing issues, but the price in terms of logic consumption and design complexity will probably be too high.

### *Event Builder* *3.5*

The Event Builder is connected to all channel FIFOs in the system (`fifo`). These are read by ascending event numbers by the Event Builder (`eb`), fanned-in by a piped multiplexer (`submux`), and finally stored in a BlockRAM shared with the embedded project.

| Design Unit | File |
|---|---|
| top | top.vhd |
| ∴ fe | fe.vhd |
| ∴ fe_ch | fe_ch.vhd |
| ∴ fe_ch_buf | fe_ch_buf.vhd |
| ∴ fe_ch_fifo | fe_ch_fifo.xco |
| ∴ fe_eb | fe_eb.vhd |
| ∴ fe_eb_submux | fe_eb_submux.vhd |
| ∴ shift | shift.vhd |

*Table 3.4 - Event Builder - associated design units*

### Motivation 3.5.1

Several implementation strategies will perfectly well fan-in data from all the individual channels to a single sequential output. The question is what sort of data throughput these can handle, and how priorities should be assigned to each channel.

In low data-rate systems a simple ”scheduler” (state machine) will probably suffice, to read channels in turn according to a predefined priority scheme, and sequentially stack the data into a serial output stream. However, any state machine is limited by its sequential nature; it only checks its inputs and refreshes the output drivers once per clock cycle. This is not an ideal solution in PET where the data is extremely sparse.

A better solution, it seems, is to make a purely combinatorial design which always selects one of the channels containing data. But combinatorial designs become slow when the number of inputs increases. In a test implementation of a MUX with 80 channels, each with a 25-bit data lines, the implementation tools reported a worst-path delay of more than 50ns in a Virtex-5 FPGA[10].

Hence, the MUX must be put in a pipeline. The challenge, however, is to do this without adding delays to the control logic. Or in other words, the data must be routed through a pipe, but the *selection* of data to enter it must be performed in real-time. This is the only way to attain a 100% occupancy.

### Implementation 3.5.2

To be able to see if any channels has data with a particular event number, each of the channel FIFOs is setup in a first-word fall-through mode. This means the first word each of these contains will always be present at the output. The only issue with this solution is when a FIFO is read empty, in which case the output will not clear but keep the last stored value. The obvious solution is to AND the data output line with the already available empty flag (fig. 3.8).

To determine which channels have data to be read, the event number part of the FIFO data is compared with the event number currently requested by the Event Builder. For every channel the event numbers match, an ”event ready flag” is set high, signalling that these channels have data to to be read.

The Event Builder requests event numbers by means of a simple counter. As discussed in 2.4.3.1, an event number can not be addressed before it is certain that no event with this particular number is still being processed

[10] Speedgrade -1.

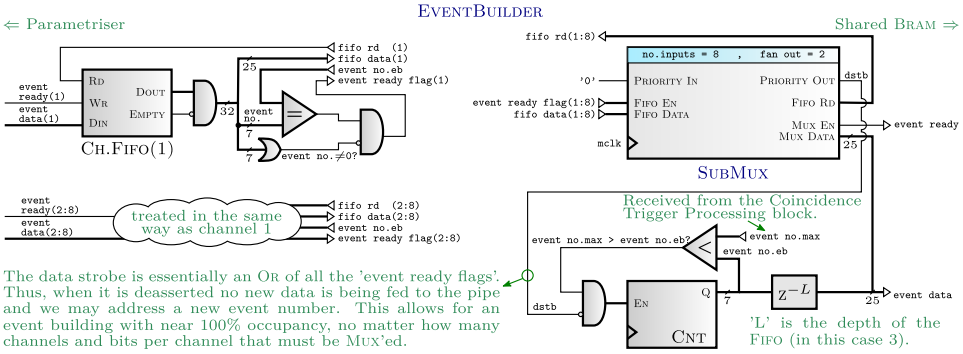



*Figure 3.8 - The Event Builder ($N = 8$, $F = 2$, $L = 3$).*

by the parameter extraction module. The limit is specified with the "event number max" signal, which is basically a delayed version of the event number counter. It is received from the coincidence processing block.

Since the Event Builder knows what event number it is addressing, the event numbers is stripped off all the event packages before these are routed to the multiplexer pipe. The event number is then delayed a few clock cycles to synchronise it with the MUX-pipe output.

Finally, a state machine will fetch these data and write it to a BlockRAM that is shared with the embedded design. All the addresses will be used for data storage, except the first one, which is used to store the number of events the buffer currently contains. When the buffer is full, the state machine simply resets, and the cycle repeats.

#### *SubMux* *3.5.2.1*

How can the multiplexer be put into a pipe in a way that ensures 100% occupancy, no timing violations, and easy adaptability?

An elegant approach would be to implement this using recursive functions, since this is how the hardware is eventually synthesised anyways. To be able to do so the pipe must be segmented into elemental design units with similar functional behaviour and structure.

To make the task easier, a constant fan-in for all levels in the design are chosen. This symmetry allows building the MUXs with a constant number of inputs and outputs. Now, let $N$ be the number of channels per MUX module, $F$ be the constant fan-in, and $L$ be the total number of levels in the pipe. If the following simple relation

$$N = F^L \tag{3.2}$$

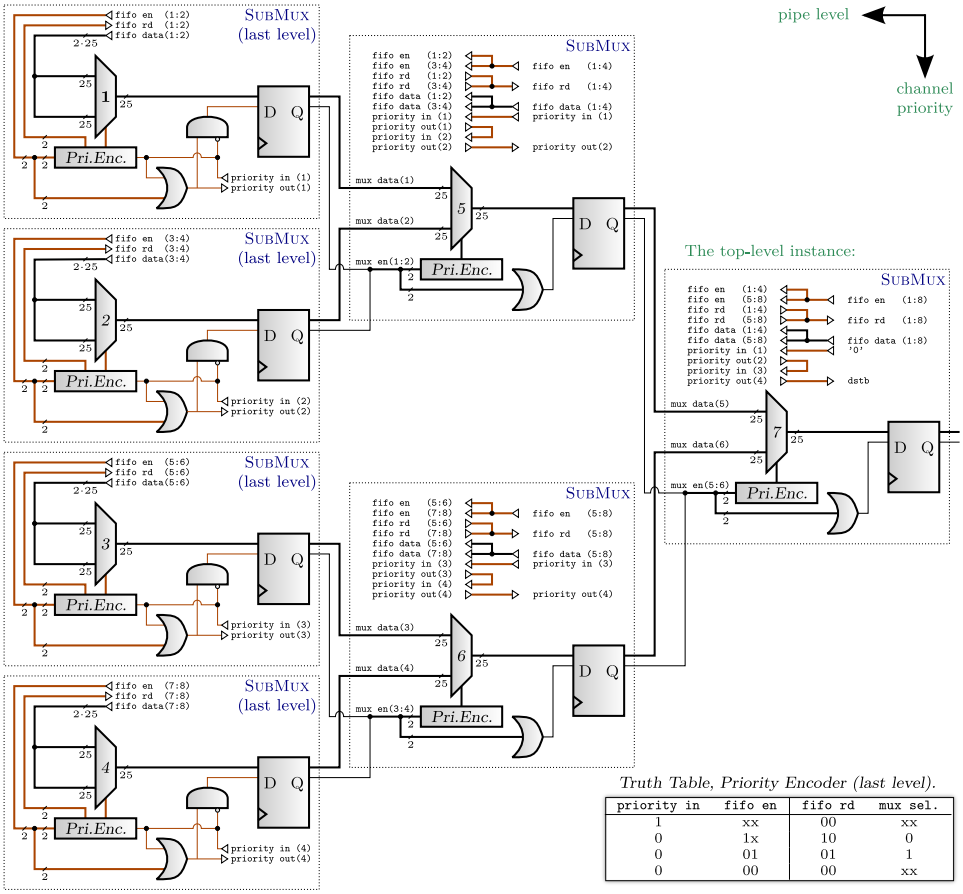



*Figure 3.9 - The recursive MUX-pipe ($N = 8$, $F = 2$, $L = 3$)*

can be satisfied using only positive integers, then the entire MUX-pipe can be built using the ”SubMux” design unit.

Fig. 3.9 depicts the structure a SubMux would instantiate with $N = 8$, $F = 2$, and $L = 3$. If the event number size is 7 bits, then each data-line is 25 bits wide (32-7), thus each multiplexer in the drawing has 50 input lines. How this is implemented internally in the FPGA, whether it be with LUTs or smaller MUXs, or a combination, will be left for the implementation tools to decide. The important factor to consider for the designer is the longest path delay from any of the given input lines through the multiplexer, relative to the system clock speed. If it becomes too high, then the fan-in must be reduced.

The tricky part of designing this logic was to get the routing correct. As the tree is being built from right to left (as it is drawn in fig. 3.9), the channel FIFO data, read, and enable lines is distributed between the SubMux modules as the pipe is being recursively instantiated, and only actually used at

the maximum depth. For all other depths, the routing focus is to descend the hierarchy. To do this correctly a SubMux must know how many channels remains to be distributed, and what the fan-in is. For every instantiation, the ”remaining channels” parameter is reduced accordingly, and when there are less remaining channels than fan-in the recursive process is stopped.

Further adding to the routing challenge is the control paths that must be allowed to move at the bottom level of the pipe (the brown lines in fig. 3.9). A large enable chain (priority in/out signals) is setup from channel to channel at this level, such that only one channel can be addressed at any given time. The chain is routed by collecting lines when ascending the hierarchy, and by re-distribution them when descending the hierarchy.

When a channel is addressed, the enable signal is passed along into the pipe, making sure that SubMux modules higher in the hierarchy allow this data to pass. A ”read” signal is also sent to the channel FIFOs to clear the data when it has been read. The enable line at the top-level SubMux is also used as a data strobe, to allow the Event Builder to constantly feed the pipe with new data. This is the key to attain a 100% occupancy.

It might seem unnecessary to create the data strobe by OR-ing all FIFO enable flags together when the same operation is partially also done in the last level of the event-builder. While this is true, the implementation software will recognise this and remove any unnecessary logic[11].

### Conclusion 3.5.3

It took a while to design it, but the Event Builder now seems to operate as expected at up to 16 channels. As long as eq. 3.2 is satisfied, the MUX-pipe seems able to handle various combinations of channels and fan-ins, and the occupancy is indeed nearly 100%.

While its implementation was somewhat complex, it was developed with ease of use in mind. A designer only needs to specify the number of channels, an appropriate fan-in (as a suggestion, 2 or 3), and connect the data lines - and a channel enable bus - to the SubMux input. That is basically it. There is no need to worry about pipe-depth, because the SubMux will recursively figure this out on its own.

As future work a suggestion is to implement overflow handling, which is currently not present. Also, the design should be modified for the EVL50T board to test the design with channel counts up to 84.

[11] FPGA implementation software usually optimises the design to find the best compromise between logic usage, performance and power usage. A designer may, however, specify an optimisation criteria of particular importance.

## *Embedded Networking* 3.6

The embedded project was added to this design for three main reasons. First, writing control applications using e.g. C is much more efficient than creating state machines, regardless of the HDL language used. Second, a wide range of available IP-cores makes it easy use common peripherals with the Microblaze microprocessor, allowing all of these to be taken into use very rapidly. And third, it allows for the use of Embedded Linux.

| Design Unit | File |
|---|---|
| top | top.vhd |
| ∴ system | system.xmp |

| Source files | Compatibility |
|---|---|
| clean_bram.c | All |
| read_bram.c | v0.1 |
| read_bram2.c | v0.2 |
| read_bram_speed.c | v0.2 |

*Table 3.5 - Embedded project - associated design units/applications*

An Embedded Linux system is usually a light-weight real-time operating system that provides networking functionality, device and file system support, scheduling, and interrupt handling, among other things. Furthermore, it represents a familiar development platform, is scalable, and open source. For this design, an Embedded Linux system from PetaLogix was selected.

Of course, there are also downsides to adding an embedded project, the most prominent possibly being the added design complexity and logic consumption. Roughly 30% of the LX50T logic is utilised by the embedded project, and although this is without attempting to optimise the implementation or slim the embedded project, its logic consumption is likely to remain formidable.

For more information on this project, refer to the ”embedded tutorial” (C), or see tab. C.1 for a quick overview of its various features, and the IPs that provides them.

### C Readout Programs 3.6.1

Since the Microblaze microprocessor is configured without a Memory Management Unit (MMU), reading the BlockRAM - where event data from the Event Builder is stored - is simply a matter of reading the address this RAM-block is set up with in the embedded project.

Three small C applications for reading these data from the BlockRAM, and writing it to file, were developed. For design version 0.1, a readout application exists (`read_bram`) that reads all the data in the BlockRAM, extracts the events parameters from the event data packet, and writes this to an ASCII file.

For the design version 0.2, two applications exists. One is a rewrite of the 0.1 implementation that also allows for the the data to recorded continuously (`read_bram2`). The other is a lighter version that does nothing but copy

data directly from the BlockRAM to the file (`read_bram_speed`), hence can handle slightly higher data rates.

Finally, if the BlockRAM needs to be cleared, `clear_bram` can be run to write '0' to all memory addresses.

Note that for this to work, the file system should be mounted on the embedded system using e.g. the Network File System (NFS, see C.6.1).

## *System Control & Adaptability* 3.7

As mentioned, while creating this design a major focus was put on flexibility, meaning the system was to be easily adaptable to fit new hardware platforms, or to allow experimentation with new features. Some of these parameters are set during compile time, and some may be set during run-time through the JTAG interface.

| Design Unit | File |
|---|---|
| constants | constants.vhd |
| functions | functions.vhd |
| types | types.vhd |
| top | top.vhd |
| ⋮∴ core | core.vhd |
| ⋮∴ ⋮∴ icon | icon.xco |
| ⋮∴ ⋮∴ ila | ila.xco |
| ⋮∴ ⋮∴ vio | vio.xco |

*Table 3.6 - System control - associated design units*

### Compile-time Parameters 3.7.1

To mimimise maintenance, all compile time parameters are specified globally in the design package `constants`. This currently includes the parameters listed below.

- *Number of channels, $N$.* The number of system channels may be changed at will, but naturally can not be set higher than the number of GPIO LVDS-pairs available. With a single channel built LVDS-pair 0 is built, pair 0,1 for 2 channels, ...
- *Deserialisation width, $W$.* The deserialisation width can be set to any of the allowed modes listed in [17, page 371, table 8.5]. However, when changed the PLLs must be rebuild to provide the correct clock frequencies, and furthermore, if $\lceil \log_2 W \rceil$ is changes with the new width, also revisit the event parameters.
- *Size of event number, channel number, and TOT-time and width.* See 2.4.3.1 for more information on these.
- *Maximum Trigger Unit delay.* This is the maximum allowable delay that the Parameter Extraction module is allowed to apply to the TOT-data in order to synchronise it with the coincidence window. Set it large enough to account for Trigger Unit latency when calculating the coincidence window, plus a few extra clock cycles. Lower values will allow logic to be saved, but the exact delay will anyhow be set with the `data_delay` parameter in run-time (see below).

- *SubMux fan-in, F*. The SubMux fan-in should be set to conform with the suggestions in 2.4.4. Increasing the number allows a lower latency, but at the cost of a reduced maximum system clock frequency.
- *Size of VIO and ILA*. The ChipScope logic consumption largely depends on the port widths of the ILA and VIO cores, which should thus be kept at a minimum. Adjusting these parameters takes care of the design part of the equation, but the respective cores must naturally also be rebuilt to fulfil the port change.

### Run-time Control Logic 3.7.2

Initially, the idea was to let the microprocessor control the design completely. However, since it is not yet completely decided whether the Readout Cards should run a microprocessor, the control activities are currently carried out with the help of a ChipScope Virtual Input/Output (VIO) module (see A.2.3).

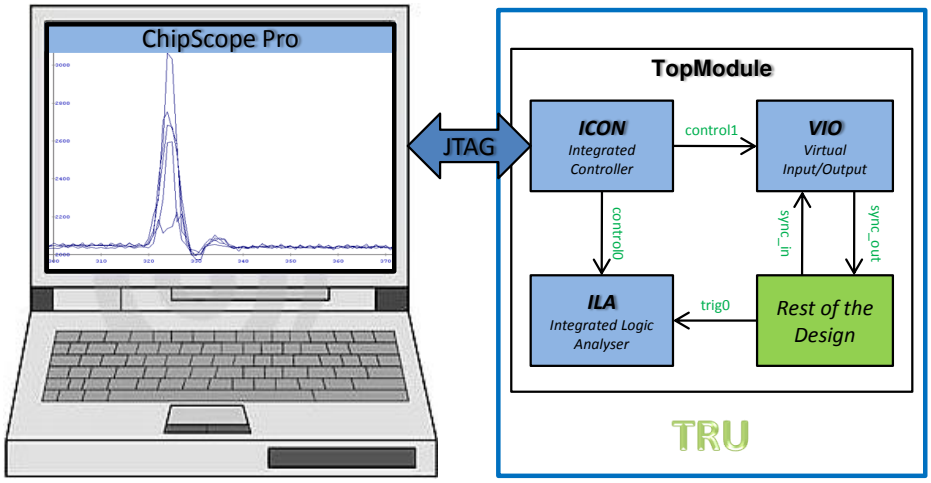



*Figure 3.10 - ChipScope principle*

To improve flexibility, portability and maintainability this ”control line” is setup with its own defined type defined in the global `types` package, and currently contains the following control signals:

- *test enable*. Once asserted the system will use internally generated ”TOT-pulses” instead of the input data pins. This may be used to verify the correct behaviour of the design, or to synchronise it at power-up.

- *readout enable*. Enables the write enable line of the final output buffer. If deasserted the system will keep running but discard all data.
- *fe soft reset*. Active low. Resets all the front-end logic when asserted.
- *coincidence trigger enable*. If an external coincidence trigger is not available, or should not be used, this flag should be deasserted. This will cause all data to be accepted into the system.
- *data delay*. Specifies the delay the is to be applied to TOT-data to synchronise it with the coincidence window received from the Trigger Unit. The implementation is a dynamic shift register of a length specified by a compile time parameter (see 3.7.1).
- *negate inputs(N-1:0)*. The analog front-end is continuously being developed, which - amoung other things - means the data input polarity is not fixed. Asserting bit $n$ of this flag will invert all in the datastream of channel $n$.
- *mask inputs(N-1:0)*. This flag is similar to the one mentioned above, but will mask the input data completely instead of inverting it. This allows the designer to build all channels, and turn these on or off during run-time.

## *Summary* *3.8*

Sampling the TOT data with GPIO deserialisers allows for a system timing resolution of down to 1ns, but makes the subsequent data handling a bit difficult. The exact time of the TOT edges must be inferred from sequential counters synchronised with the system clock, and combinatorial logic. The computation is rather straight-forward when the TOT-pulses can be assumed not to be noisy, too short, or too close to another TOT-pulse on the same channel.

The solution implemented is slightly robust to noise, allows for very short TOT-widths to be resolved accurately, and is very low on logic consumption. However, the TOT-time parameters sometimes get incorrectly calculated with very short TOT-pulses, and the circuit can not handle two TOT-pulses on the same channel with temporal separation less than 20ns. This is taken care of with a channel busy, which effectively sets the channel dead-time to 20ns.

When the TOT time and width is computed, these along with the corresponding event and channel number, are wrapped in an event package and stored in a channel FIFO. These are setup in first-word fall-through mode, meaning the first word in each FIFO always will be visible on the output. The Event Builder compares these data with an event number it requests

to read, and by means of comparators flags each channel to indicate which contains relevant data.

All FIFO data lines, and the channel ready flags, are sent to a recursively implemented MUX-pipe. By keeping control signals outside the delayed clock domains, the Event Builder is able to constantly feed the MUX-pipe with new events, hence promoting a 100% occupancy. Since the Event Builder is synchronised with the system clock, this translates to a throughput of 100Mevents/s.

The embedded system, however, is nowhere near being able to handle this event rate. Thus, another team member (M.Rissi) has investigated whether the embedded project should be swapped with a UDP-core, which should be able to provide much better performance. Initial results looks promising, so a future design will probably use the UDP-core for the data path. However, since the EVL50T boards has 2 Ethernet PHY's, it might also be possible to keep both solutions.

*"People love chopping wood. In this activity one immediately sees results."*

Albert Einstein

# 4 Results

Normally, when creating FPGA designs, the first step toward verification is by means of logical simulations. The design is wrapped in a testbench that provides input stimuli, and a simulator theoretically computes how this affects the design based on the HDL-description.

Four simulations are presented here. Two simulates the Parametriser (4.2.1) and Event Builder (4.2.4), respectively, and are shown as a proof-of-concept. The remaining two simulate varying pulse widths (4.2.2) and rates (4.2.3), to investigate the designs ability to handle various TOT-data input scenarios. If the reader is not accustomed to read such diagrams, feel free to skip these, or look at the relevant part of the discussion instead (5.1.2).

A design can easily work in simulations, but not in reality. A few readouts will be presented to verify that it does. Two readouts of external test pulses of fixed width are shown to investigate the accuracy of estimating the TOT-width (4.3.1). Then the TOT spectrum of a LYSO-crystal with and without exposure to $^{137}$Cs and $^{133}$Ba sources is shown, to investigate the correlation between TOT values and the energy spectra (4.3.2). Finally, two LYSO-crystals were setup around a $^{68}$Ga source, and the effect of applying a coincidence window is presented (4.3.3).

## *Test Setup* *4.1*

The following tests use the latest stable version of this design, i.e. v0.2 (see 3.1.1), compiled with the following parameters:

- System clock set to either 50 or 100 MHz.
- 10-bit deserialisation width, yielding a sampling rate of 500 MHz/1 GHz.
- System compiled with 16-channels (all LVDS lines on the ML505), with any inactive channels masked out.

- Fan-in of multiplexer pipe set to 4, inferring a pipe depth of 3 (amounting to 16 channels).

## *Simulations* *4.2*

When simulating HDL designs for an FPGA there are two common options; cycle based or event based simulation. A cycle based simulation only calculates results at clock edges, ignores inter-phase timing, and usually only considers Boolean states ('1' or '0', not 'high impedance' or similar). This makes it very fast. Event based simulations, on the other hand, attempts to take into account how each signal propagates the described logic, hence is more precise, but slower.

The upcoming simulations are all event driven, created with Mentor Graphics ModelSim. Each simulation is fully scripted, and can be recreated with the `.do`-file specified under each timing diagram[1].

Post-synthesis or post-place&route simulations will be omitted since these are more complex, harder to interpret, slow to simulate, and because most of the "extra" information one might infer from these, in terms of timing parameters, can also be found in the timing reports generated by the implementation software.

To further simplify the interpretation of the simulations only a handful of the most vital signals are added.

[1] For a small guide on how to get started with ModelSim, and how to run the simulations, see appendix A.3.2.

### Parametriser 4.2.1

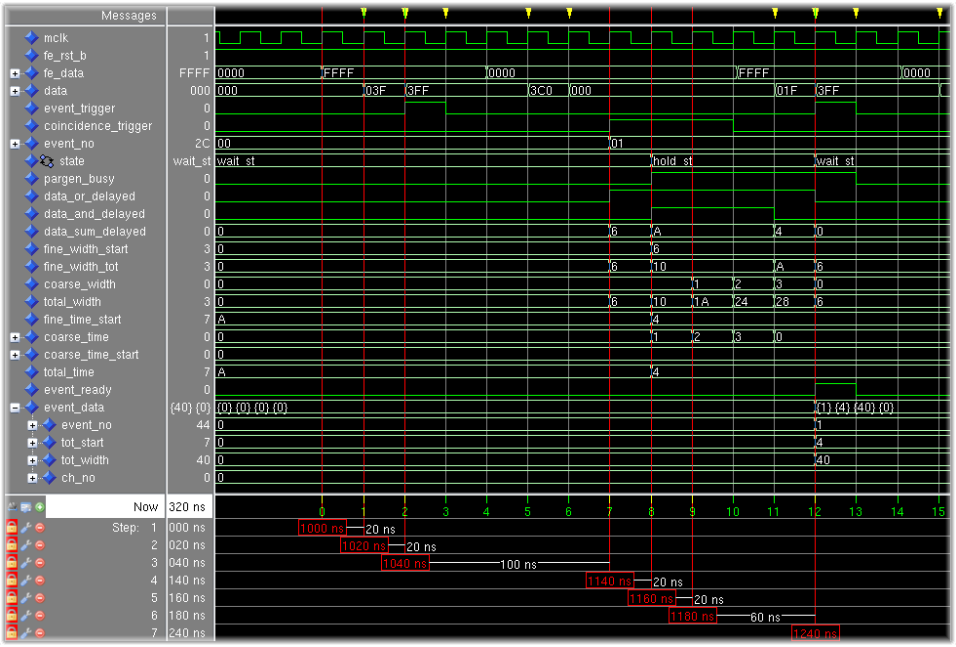



*Figure 4.1 - Chronogram - Parametriser (one channel)*
(generated with `parametriser.do`)

As shown in fig. 4.1, the Parametriser follows these steps:

1. A TOT pulse arrives at the deserialiser input.
2. Synchronised frames of data are ready for interpretation.
3. An event trigger is sent[2] to the Trigger Unit.
4. The Trigger Unit - after having evaluated any event triggers - responds by asserting a coincidence validation window.
   - For each coincidence window an event-number counter increments.
   - The bitwise And, Or and Sum comes out from the delay pipeline and gets validated ("unmasked") by the coincidence window[3].
5. For each rising edge the event number, time since coincidence window start, and the location of the edge within the frame is copied to register memory. All parameters are now known except the TOT length.
   - A "coarse-time" counter starts to keep track of relative timing.
6. While waiting for the falling edge a coarse-width counter starts running to keep track of the TOT length.

[2] To ensure glitch-free output to the Trigger Unit, the event trigger is synchronously driven, hence the one-cycle delay.

7. At the falling edge the value of the coarse-width counter in addition to subframe edge position information (from rising and falling edge) is used to compute the TOT length.
   - All the parameters are copied to an "event-package", and flagged with an "event-ready" signal to indicate the presence of new data.

### Varying Pulse Widths 4.2.2

A simulated "sweep" over small TOT-widths was conducted, starting at 48ns with a decrement of 5ns. For every TOT-length, the TOT phase was incremented with 1ns for a total 10 times to ensure the results were phase-independent. The sampling rate was set to 1 GHz.

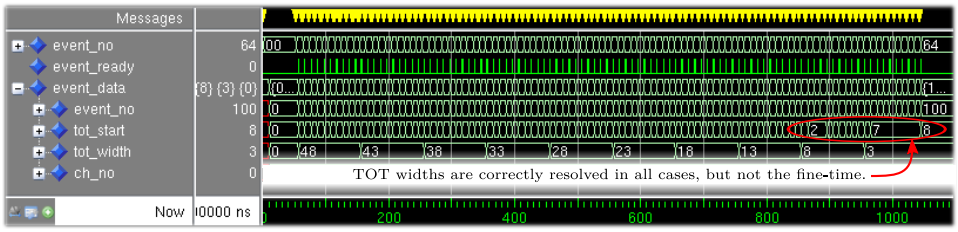



*Figure 4.2 - Chronogram - Parametriser output with input pulses of variable width and phase*

(image `parametriser_pws.svg`, generated with `parametriser.do`)

### Varying Pulse Rates 4.2.3

A simulation with two succeeding pulses was setup, testing numerous combinations of inter-pulse delays and pulse widths (fig. 4.3). As before, for every said combination all possible frame positions was tested to ensure no scenario was missed.

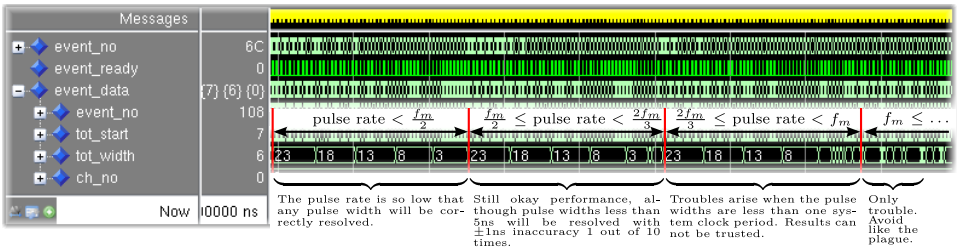



*Figure 4.3 - Chronogram - Parametriser output with input pulses of variable width, rate and phase*

(image `parametriser_prs.svg`, generated with `parametriser.do`)

### Event Builder 4.2.4

Synchronised TOT pulses of width 25 (hex 19) were generated for channels 0-3, parameters extracted and fed into a channel FIFO, and events were built by collecting and sorting these (fig. 4.4). Figure description follows.

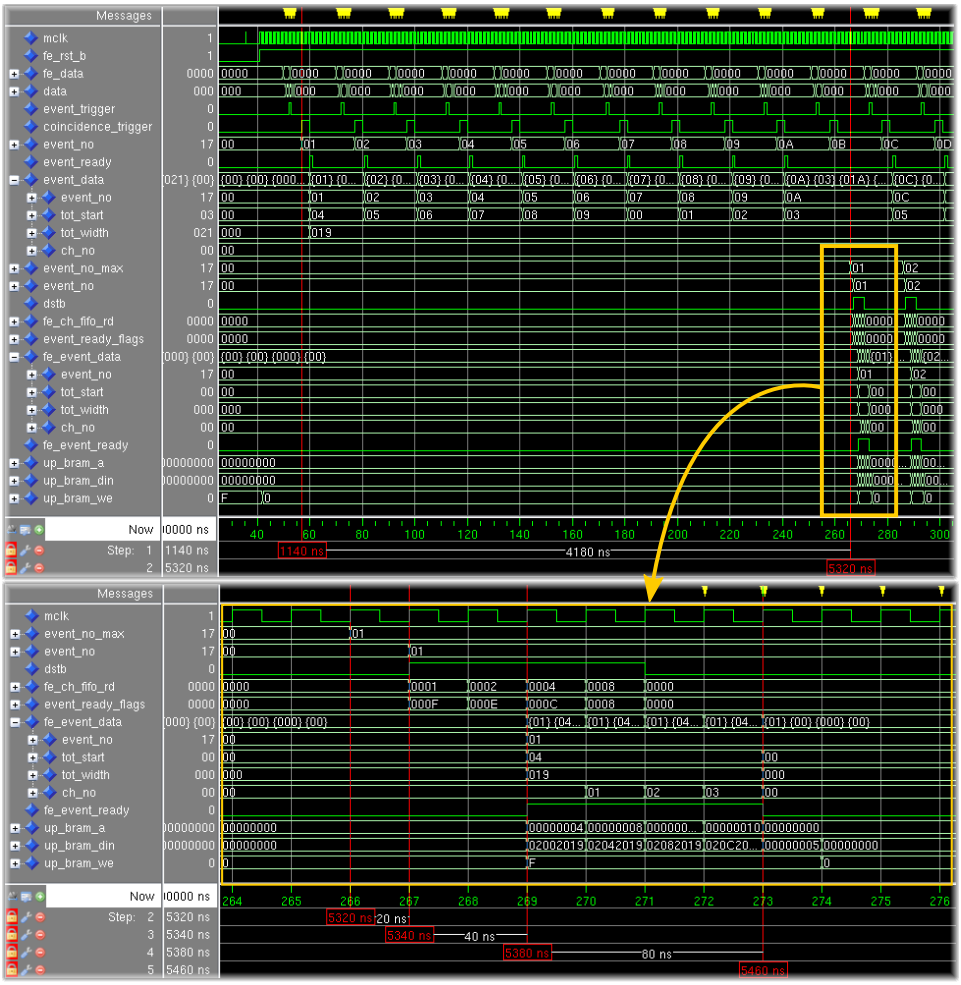



*Figure 4.4 - Chronogram - Event Builder*
(image `event_builder_sim.svg`, generated with `eb.do`)

1. A coincidence window validates the first event.
2. To ensure no event is attempted read before all channels are guaranteed to have stored it, a waiting period slightly longer than the maximum TOT-time commences. An "event number max" counter indicates the largest event-number the Event Builder is allowed to address[4].

[4]Exceptions applies when the counters wraps.

3. The Event Builder addresses event 1 using combinatorial control logic (thus no introduced delay).
   - "Event ready flags" indicates which channels have data with the addressed event number.
   - A data strobe is asserted as long as either of the channels flags the presence of valid data.
   - Only when an event from a channel FIFO is sent to the multiplexer pipe will the channel be "read".
4. The first event, first channel, comes out from the multiplexer pipe.
   - An "event ready" data strobe validates the output data of the multiplexer pipe, and acts as a write enable for the shared BlockRAM between the readout logic and embedded design.
   - Each event is stored in a new BlockRAM location[5].
5. All events are read out.

## *Readout Tests* *4.3*

To verify the actual FPGA-implementation of this design, a series of readouts were conducted with input data either from a pulse generator or from a few LYSO-crystals. The ML505 evaluation board were used in the study, its LVDS inputs set to receive the pulses, and its Ethernet MAC connected directly to a receiving computer.

Only recent readouts will be presented, to reflect the current state of the project.

[5]The shared BlockRAM is byte-addressed, hence the increment of 4 for every event.

### External Test Pulses 4.3.1

A Tektronix AFG3022 Dual Channel Function Generator was connected to a custom LVDS-converter to produce test-pulses (fig. 4.5), in turn read by the FPGA, and finally parameter spectra were computed (fig. 4.6).

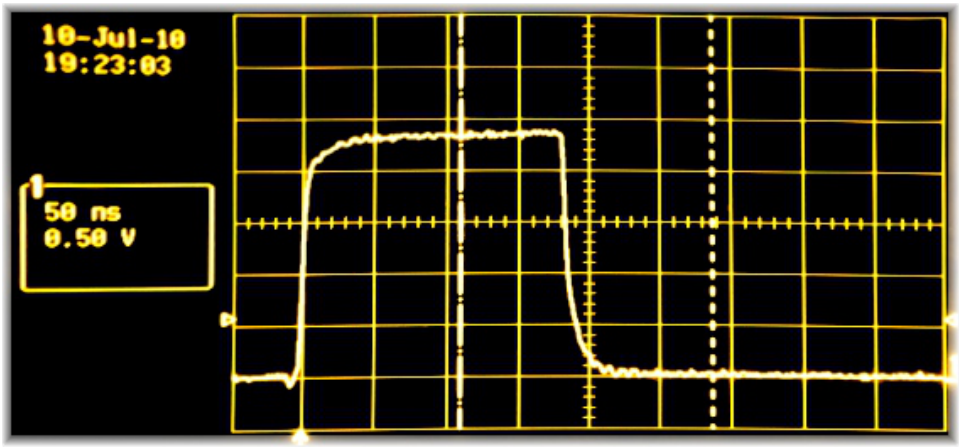



*Figure 4.5 - Asserted test pulses (LVDS-driver output)*

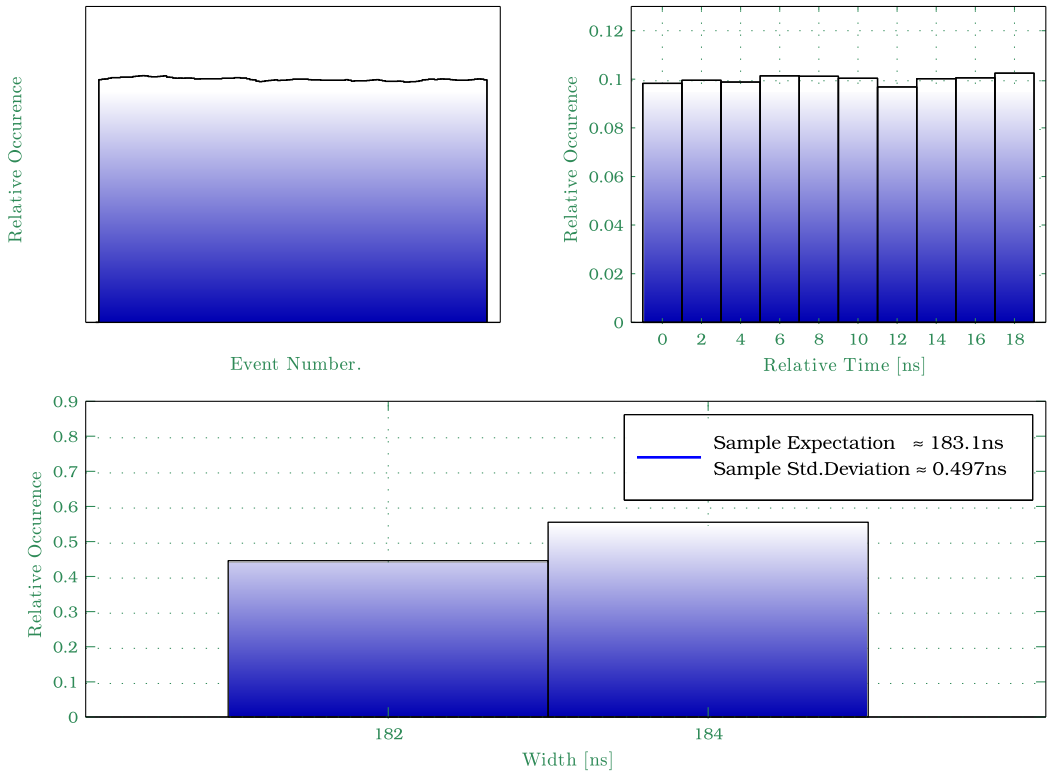



*Figure 4.6 - Parameter spectrum, ∼100k test pulses, 1 channel sampled at 500 MHz*
(generated with `run_2010_07_10.m`)

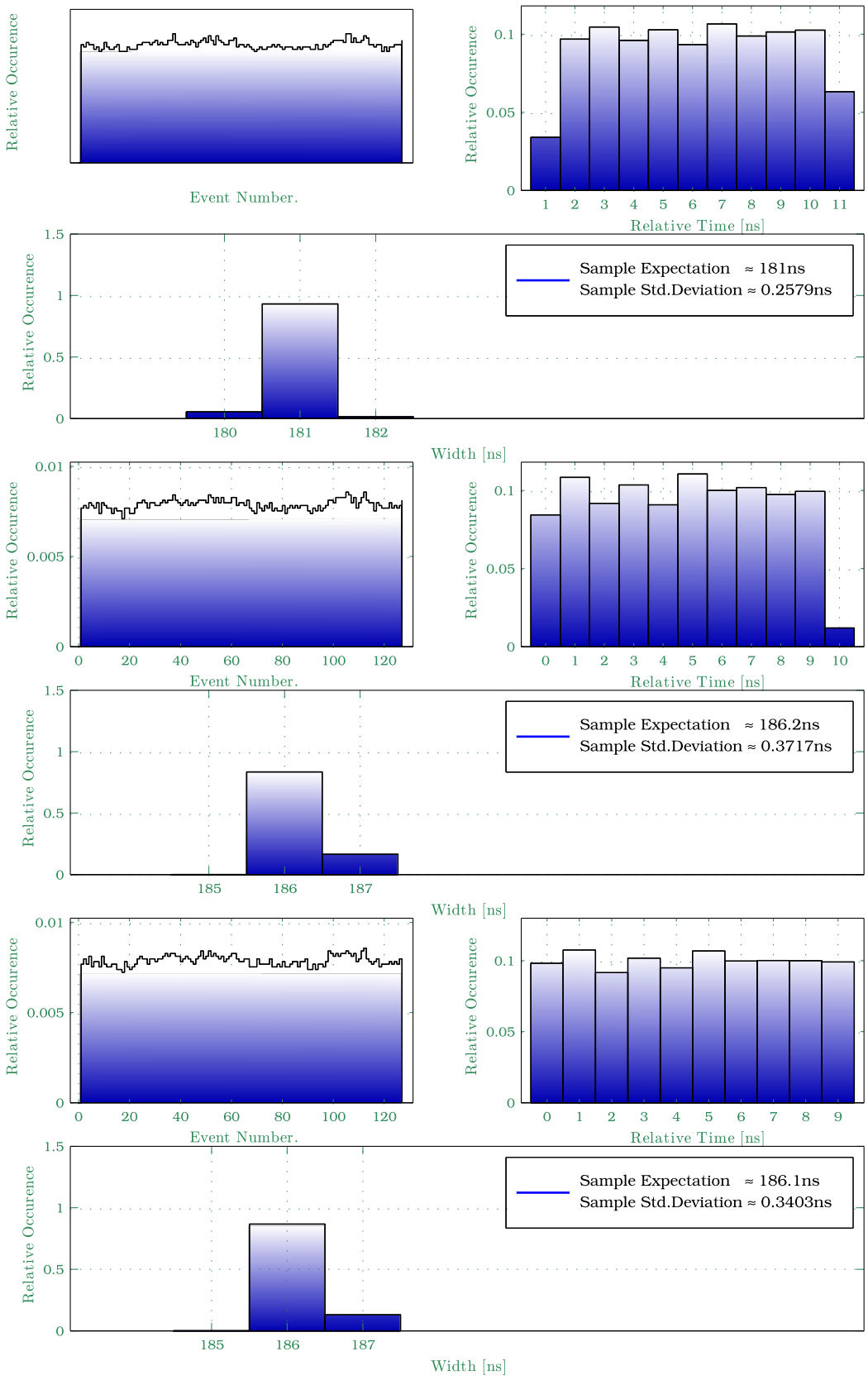





*Figure 4.7 - Parameter spectrum, ∼100k test pulses, 3 channels sampled at 1 GHz*
(generated with `run_2010_07_10_100mhz_4ch.m`)

### LYSO Spectrum - Intrinsic,Ba133,Cs137 4.3.2

Three readouts were conducted[6], one measuring the LYSO-crystal intrinsic activity, the other two measuring the LYSO-response when exposed to a $^{133}$Ba or $^{137}$Cs source. The crystal was wrapped in a material with minimal ambient light penetration, connected to the analog front-end (GAPD-voltage set to 71.5V), in turn connected to a ML505 evaluation board. Each TOT-width spectrum was computed, and normalised with the intrinsic activity (fig. 4.8). For a list of expected emission energies, see tab. 5.1 (page 75).

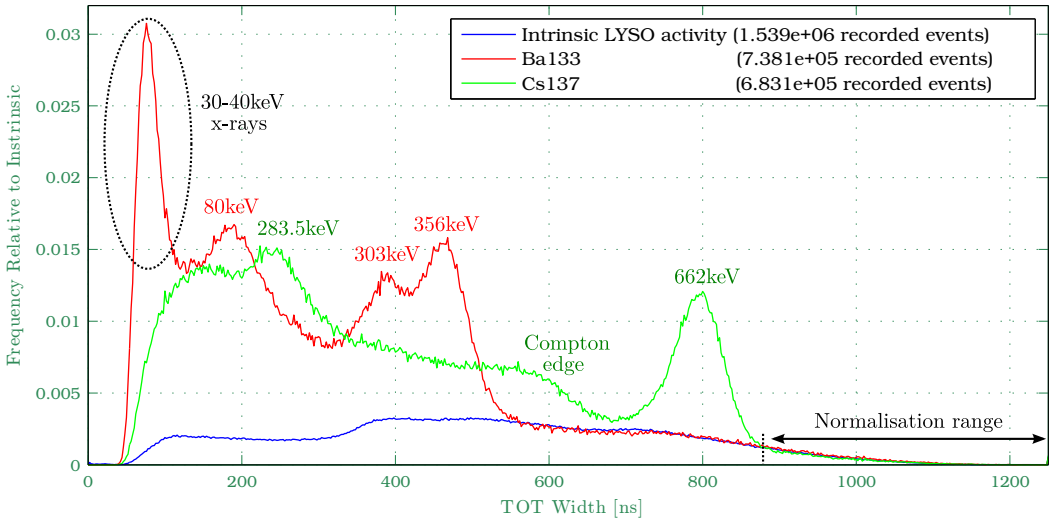



*Figure 4.8 - TOT spectrum from a LYSO-crystal*
(generated with `run_2010_07_12_intrinsic_ba133_cs137.m`)

#### *Linearity* *4.3.2.1*

The peaks in the $^{133}$Ba and $^{137}$Cs spectra have known energies (see 5.1), allowing the relation between TOT-values and energy to be investigated. To do so, the spectra was upsampled by low-pass interpolation, and a sweep was conducted to find the maximums. Combining the position of the maximums with the known peak energies, a linear least-squares approximation was conducted to measure the detector linearity (fig. 4.10).

### Coincidence Processing 4.3.3

Two LYSO-crystals were set up around a positron emitting $^{68}$Ga source[7], independently treated by two front-end boards, and read out with a ML505 Readout Board running with a sample rate of 500 MHz. Coincidence windows of varying widths were later applied in software (fig. 4.9).

[6] These readouts were performed by E.Bolle and M.Rissi using the readout design v0.1. The data analysis, however, is the result of my own work.

[7] The daughter product of $^{68}$Ga is $^{68}$Ge, which is the actual positron emitter.

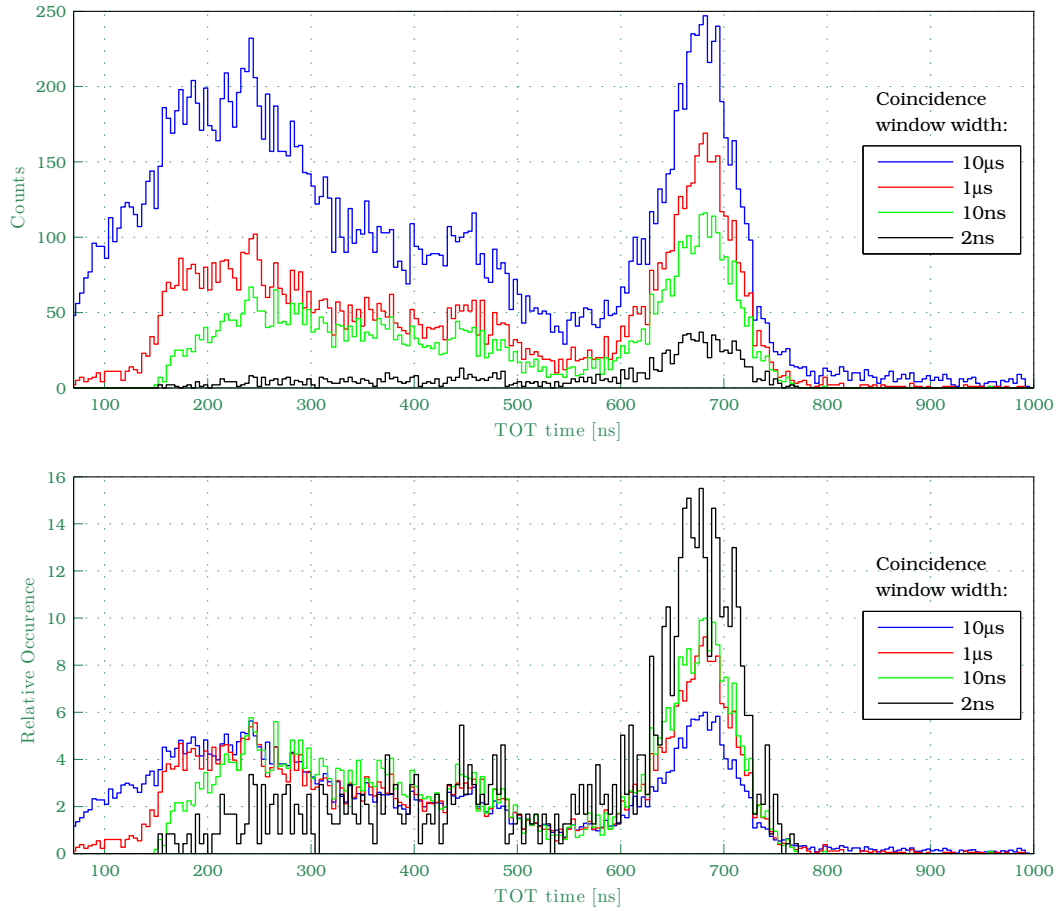



*Figure 4.9 - TOT spectra with software coincidence windowing*
(top: no normalisation, bottom: normalised)
(generated with `run_2010_07_12_coincidence.m`)

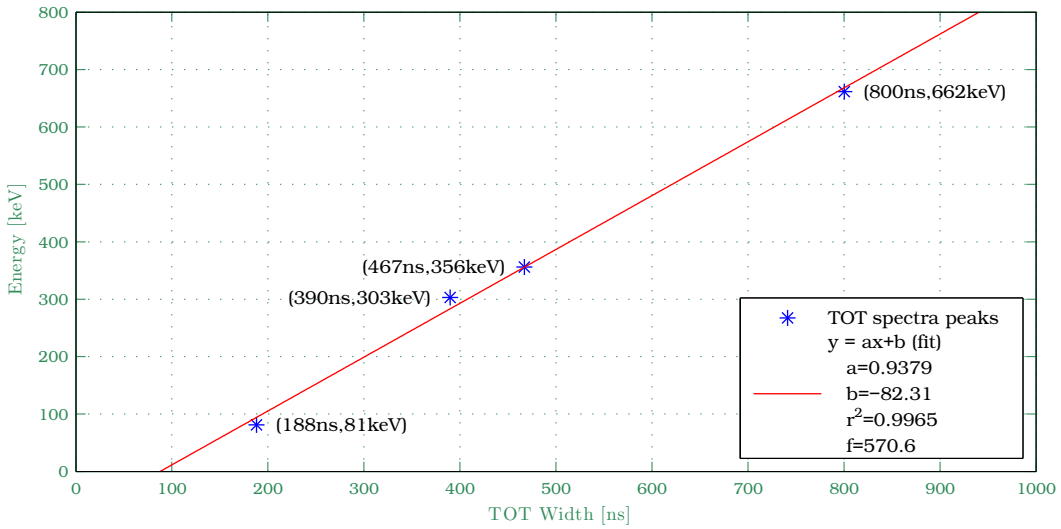



*Figure 4.10 - Detector linearity with Ba-133 and Cs-137 sources*
(generated with `run_2010_07_12_intrinsic_ba133_cs137.m`)

***Energy Resolution*** *4.3.3.1*

In order to derive the energy resolution the standard deviation of the energy peak must estimated. This was achieved by fitting a Gaussian to the 2ns windowed spectra (fig. 4.11).

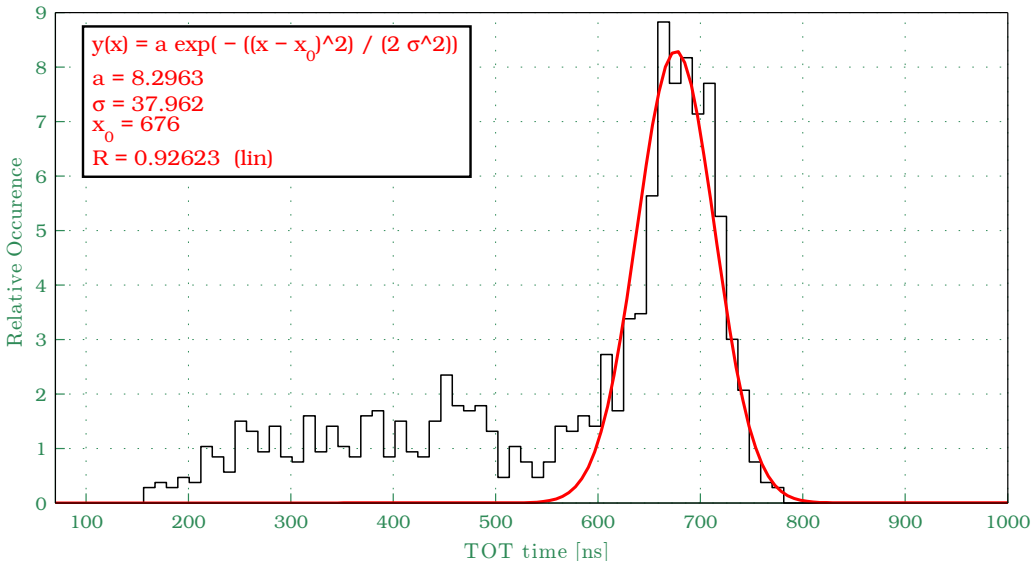



*Figure 4.11 - Gaussian fitted TOT spectra obtained with a 2ns coincidence window* (generated with `run_2010_07_12_coincidence.m`)

The univariate Gaussian used here is described by the well known function

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma_x} \exp\left[-\frac{(x-\mu_x)^2}{2\sigma_x^2}\right]. \tag{4.1}$$

Substituting $\mu_x = 0$, setting the exponential equal to 0.5, and solving for $x$ yields

$$x = \pm\sqrt{2\ln 2}\sigma_x. \tag{4.2}$$

From this the Full Width of Half Maximum (FWHM) energy resolution may be found[8], either absolutely or relatively,

$$FWHM = 2\sqrt{2\ln 2}\sigma \approx 89.4\text{ns } (15.0\%). \tag{4.3}$$

[8] Actually, since the independent variable in this case is time, a better term might be Full *Duration* at Half Maximum (FDHM).

*"However beautiful the strategy, you should occasionally look at the results."*

Winston Churchill

# 5 Discussion

## *Simulations* 5.1

Often there seems to be a "gap" between the results acquired by HDL-simulations and the results actually seen in the implemented FPGA-design. However, this is usually not caused by simulation inaccuracies, but rather a failure to recognise and embed all real-world scenarios that may occur into the testbench[1].

Fortunately, in this design, creating the input stimuli is rather straightforward; All possible variations of TOT-pulses, including varying phases, widths or pattern across channels, should be tested, but the testbench may be oblivious to how the design actually responds to this (as opposed to e.g. simulations of buses, where the entire protocol must be simulated). This is left as a manual task in this design by studying timing diagrams.

### Parameter Extraction 5.1.1

To make the parametrisation logic able to handle every possible combination of TOT-pulses, whether they are extremely short or alternates in between deserialisation frames, the system must effectively perform decisions using every bit in every frame. To make this work, an aggressive timing model must be selected for the synchronous logic controlling the parameter extraction. Or in simpler terms, synchronous logic should be removed from any decision making process, or advanced in time, in order remove unnecessary decision "lag". Do not wait for decisions to be made, but align data from the future, present and past in a fashion that allows *any* essential decision to be made at *any given time*.

Consider the simulation of the parametriser (4.2.1), which displays the parameter extraction principle. Here the TOT-pulse (`data`) is neither short

[1] A testbench is a "wrapper" for the FPGA-design, designed to generate input stimuli and (sometimes also) validate these with the outputs, in which case the testbench is referred to as "self-testing".

nor alternates between frames, making it an ”easy case”. However, note the following: There is *no* delay[2] between the data from which parameters need to be computed, and the parameters themselves.

As described in 3.4.2.3, this is *the* clue to remember in order to increase the systems ability to handle exception cases. Make sure parameters are ready at any given time, and the only remaining challenge is to decide when to read them. In fact, as indicated by the ”variable TOT-width” simulation (4.2.2), this approach is able to handle almost all combinations of TOT-widths and phases that adheres to the parameter size constraints (see 2.4.3.1). The exceptions are mentioned next.

### Variable Pulse Widths and Rates 5.1.2

When the pulse repetition rate is low, the system has no problems resolving the TOT-widths, even those less than a frame wide (fig. 4.2). However, for these widths, the relative TOT-time may be incorrectly resolved. This happens because the relative time is effectively found by counting the number of 0's in the first frame, an erroneous approach when the tailing bits in the first frame are not all 1's. Finding the time using a priority encoder will solve this, but will in turn decrease system tolerance to TOT-edge noise.

The system responds less well, however, to an increase in pulse repetition rate (fig. 4.3). When the repetition rate is less than half the system clock (i.e. less than 50 MHz), the design behaves as described above. But if increased past this point the error rate will follow, and when set higher than the system clock speed the design is quite useless. Thus, if the interval between the falling and rising edge of two successive TOT-pulses hitting the same channel is less than two frames, both events are likely to be corrupted (but the system is stable, and will self-recover).

The reason this is not handled is because the exception logic was designed to primarily handle short TOT-pulses, which was considered more important than high repetition rates. Actually, by simply inverting the data, one realises that the problem is essentially the same in either case. Thus, by duplicating the parameter extraction logic, and have this duplicate treat and inverted version of the data instead, the *length* of the pulse spacing can be measured instead, allowing even these scenarios to be handled. However, this is probably not necessary, and the logic can likely be put to better use elsewhere in the design.

Thus, to ensure no data corruption occurs, a channel `busy` is asserted for two clock periods after the rising edge of an event (introduced in version

[2] No visible delay in the simulation, that is. In reality there is a small delay, just nowhere near the system clock period.

0.21 of this design), in which period no other rising edge will be accepted. This logic is simple and reliable, and effectively sets the channel dead-time to 2 clocks, i.e. 20ns with 100 MHz system frequency.

### Event Building 5.1.3

The Event Builder was designed from the very beginning to handle event-rates up to the system clock speed, i.e. 100 Mevents/s when running at 100 MHz. Since it is completely recursive, and the number of inputs and fan-in per multiplexer may be adjusted at will, the upper performance limit of this structure should be way higher than the data link from the FPGA to the outside world.

The principle is shown in fig. 4.4, displaying the multiplexing of 4 channels with data, each being repeated with rates low enough to see some ”idle-time” in the simulation. The only known weakness is if the channel FIFO buffers are saturated before read out, in which case the event number counter may ”catch up” with the Event Builder (see 2.4.3.1) and the data will be corrupted. However, this will also be temporary, and the design will eventually self-recover.

Even if overflows are caused by unlikely high data rates, the system should eventually be able to cope with it, or at least flag an overflow condition. The channel FIFOs has `overflow` flags ready to use, and even `almost full` flags, thus this should be fairly quick to implement. Although this design employs the use of BlockRAM shared with a microprocessor, and a free-running circular counter feeding it, the end design will probably utilise a firmware UDP-block for improved networking performance, in which case a FIFO - and more overflow logic - must also be added.

Finally, the system was never tested with more than 16 channels compiled in, due to the lack of LVDS-pins on the ML505 board. The full potential of the Event Builder will be discovered once the design is scaled to 84 channels.

## *Readout Tests* 5.2

Generally speaking, even if simulations show the most promising results imaginable, the FPGA-design may very well be completely unusable. This especially applies to high speed parts of the design, since HDL-simulations never take into account parasitic effects that will cause high-speed circuitry to behave as if not entirely digital.

The clue is to make sure the synthesiser infers the logic described in the HDL-code, but as even ”text-book” descriptions of hardware may be synthesised incorrectly, one should never assume, always verify. With properly inferred

logic, good clock and reset handling, and comfortable timing margins, the design are likely perform as intended.

That being said, this design essentially works. Great care was taken to ensure design stability and flexibility, and this seems confirmed by the readout tests. The design was left unattended for up to 12 hours while acquiring data, with not flaws in the data observed[3].

### External Test Pulses 5.2.1

When studying the histograms of the parameters extracted from external test pulses (fig. 4.6, fig. 4.7), one may conclude that the readout system essentially behaves as it should. The plots contain no extreme values, and the distribution of *event numbers*, *relative timings* and *TOT-widths* all seem sane.

To start with the *event numbers*, these should be flatly distributed. All events were accepted in this test, and assigned event numbers in increments of one. This gives reason to expect a perfectly flat distribution, but the plots indicate otherwise. The reason seems to be network saturation; if the pulse rate is increased above the maximum of what the embedded design can handle, events are simply dropped, leading to a more erratic event number distribution.

No external logic for generating the coincidence window has yet been made, but some internal logic is currently used to generate a coincidence window every time a TOT-pulse enters the system. By looking at fig. 4.7 one may notice that the first channel has *timing values* slightly higher than the other two, with the last channel being the only one with events occurring no later than 9ns after coincidence window assertion. This indicates that the last channel receives the pulse first, although by a very narrow margin. A further study might be necessary to investigate the origin of this delay, whether it be the test setup or the FPGA. Regardless of the cause, the phase differences may be compensated for by adjusting the delay of each channel deserialiser. In the end this will probably be necessary to achieve the timing accuracy this system is designed for.

The *TOT-spectra* looks promising, at first eyesight a bit too promising, perhaps. Consider the first channel in fig. 4.7, and notice the calculated sample variance of 0.26ns. Recalling that the sampling accuracy (see 2.4.1, eq. 2.3) was calculated to 0.41ns, how can the measured standard deviation be better?

[3] The only issue was the FPGA became rather hot, even with just a few channels capturing data.

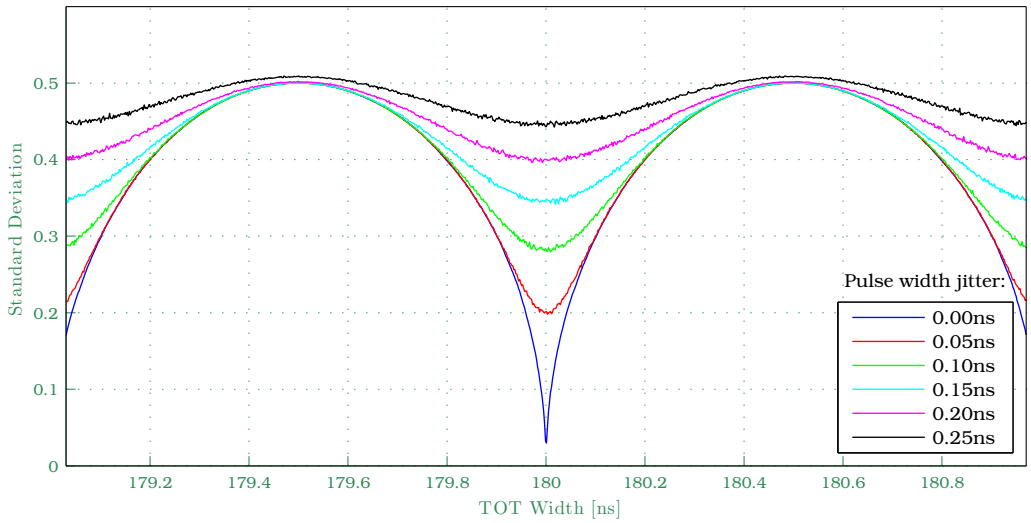



*Figure 5.1 - Standard deviation of estimating TOT-widths in the sub-ns range, with varying degree of Gaussian jitter*
(generated with `variance_plot.m`)

The reason is that in this case one can not assume that the position of the leading and falling edge is independent. In fact, if the width of the pulse is known with minimal uncertainty, the two parameters are strongly correlated, which reduces the uncertainty of computing the difference of the two. To investigate this effect, a Monte Carlo simulation was conducted to find the TOT standard deviation as a function of sub-ns pulse width variations with varying amounts of Gaussian jitter. (fig. 5.1).

As indicated by the simulation, the standard deviation depends strongly on the pulse width. Those being a multiple of the sampling period yield the best measurement accuracy, and vice versa for pulses being of a multiple plus one half a sampling period wide. Adding jitter decorrelates the rising and falling edge position within the sampling interval, and thus flattens the variance curve. Due to the measured standard deviation of 26ns, it should be safe to assume a jitter of less than 0.15ns.

Finally, as for why one channel end up measuring a shorter TOT pulse than the other two, this is not known. However, this channel is the only one that receives an inverted test pulse, which may be something to look into. In general, a note should be made regarding the widths of all TOT-pulses; Counting the number of 1's in the sampled data, and multiplying this with the sampling period, will yield a figure one sampling period shorter than the width expectation value. It may not matter, however, as a system calibration will easily be able to mitigate the effect of constant offsets.

### LYSO Spectrum - Intrinsic,Ba133,Cs137 5.2.2

To match TOT-values with $\gamma$-ray energies, the detector must be calibrated. This may be done by using a source emitting $\gamma$-rays of known energies, collecting enough data to attain a statistically significant TOT-spectrum, and match the peaks in this spectrum with the known energies.

The sources used in this study (4.3.2), $^{133}$Ba and $^{137}$Cs, emits a variation of $\gamma$-rays at discrete energies (tab. 5.1), but some energies are emitted more frequently than others. Indeed, because of the photoelectric effect, the spectrogram will thus peak at these energies (fig. 4.8).

For instance, Compton scatters causes $\gamma$-rays to lose energy, thus contributes to lower energy levels of the spectrum. The spectra low-end is largely contaminated by X-rays and detector noise, and the characteristics of this areas is not well known. Thus, in this study only energies above 80keV (the first $^{133}$Ba-peak) was considered. All energies of large intensities (see bold figures in tab. 5.1) are shown in the spectra, which is good news.

The linearity plot (fig. 4.10) seems promising as well, although with just a single run these findings can hardly be assumed significant. A curiosity is the $^{137}$Cs energy peak at $\sim$250ns, of which the cause is unknown.

Naturally, the study will have to be repeated in order to conclude with greater certainty. However, given the preliminary data at hand, the future looks bright for ComPET.

### Coincidence Processing 5.2.3

Even if coincidence windowing can not yet be fully carried out in hardware (due to the lack of a Trigger Unit), it may very well be performed in the post-processing.

As seen from the TOT spectra with various coincidence windows applied (fig. 4.9), reducing the window length increases the energy resolution, but decreases the photon sensitivity. Out of more than 7 million events only $\sim$1200 was accepted with a window length of 2ns, and $\sim$5800 with a length of 10ns. Increasing the window width over this level accumulated mostly single counts, thus no more than $\sim$20600 events were accepted even with a window length of 10$\mu$s.

The Full Width at Half Maximum (FWHM) energy resolution indicates the minimum energy separation at which two events are guaranteed to be resolved, as defined by the Rayleigh criterion. The measured energy resolution of 15.0% is very promising, especially since this is without optimisations.

| | Energy (keV) | Type | Intensity (%) | Origin |
|---|---|---|---|---|
| $^{133}$Ba emissions | 4.75 | X-rays | 16.0 (8) | $^{133}$Cs |
| | 30-36 | X-rays | 119.6 (14) | $^{133}$Cs |
| | 53.16 | $\gamma$-rays | 2.14 (3) | $^{133}$Cs |
| | 79.61 | $\gamma$-rays | 2.65 (5) | $^{133}$Cs |
| | **81.00**[1] | $\gamma$-rays | 32.9 (3) | $^{133}$Cs |
| | 160.61 | $\gamma$-rays | 0.638 (4) | $^{133}$Cs |
| | 223.24 | $\gamma$-rays | 0.45 (3) | $^{133}$Cs |
| | 276.40 | $\gamma$-rays | 7.16 (5) | $^{133}$Cs |
| | **302.85**[1] | $\gamma$-rays | 18.34 (13) | $^{133}$Cs |
| | **356.01**[1] | $\gamma$-rays | 62.05 (19) | $^{133}$Cs |
| | 383.85 | $\gamma$-rays | 8.94 (6) | $^{133}$Cs |
| $^{137}$Cs emissions | 4.88 | X-rays | 0.9 (5) | $^{137}$Ba |
| | 31-38 | X-rays | 6.9 (41) | $^{137}$Ba |
| | 283.5 | $\gamma$-rays | 0.00058 (8) | $^{137}$Ba |
| | **661.66**[1] | $\gamma$-rays | 84.99 (20) | $^{137}$Ba |

*Table 5.1 - Cs-137 and Ba-133 emissions [21]*

[1] Corresponds to the peaks in the TOT-spectra (fig. 4.8), and the points used in the linearity study (fig. 4.10).

# 6

# Conclusion

The primary aim of ComPET is to create a preclinical PET scanner with performance on-par with, or exceeding, that of commercially available products, at a reasonable price. The use of an inventive geometry combining LYSOs and WLSs, coupled with a custom analog front-end that encodes interaction parameters in a binary TOT-pulse, allows for a detector photon sensitivity up to 16%, while maintaining a FWHM spatial resolution of less than 1mm in the centre of FoV. In preclinical PET, this is a very potent combination.

To accommodate this potential it is important that the readout system acquires the data with sufficient precision, and without loss. Ideally, the system should introduce no extra noise, never lose events, be able to handle all possible combinations of events, at any rate, and at any time. In other words, it should seem transparent, to ensure focus is kept on research rather than technicalities.

This suggests using a technology that is highly capable of performing concurrent operations, is flexible, re-programmable, and available off-the-shelf at a reasonable price. FPGAs and evaluation boards are chosen for this reason. The FPGA design presented here samples the TOT-data at 1 GHz with the SERDES functionality residing in the GPIO-tile, and compresses it by performing parameter extraction and coincidence validation. A fan-in structure collects and sorts events from all channels, while promoting a throughput of 100Mevents/s with a 100MHz system clock. Finally, the data is sent out over a network with the use of an embedded system.

Does it work? Preliminary tests look very promising. Simulations were carried out to ensure the readout system was theoretically able to cope with various worst-case scenarios, which it did, except when two pulses were separated by less than 20ns (assuming 1 GHz sampling frequency). To avoid data corruption, a channel busy is asserted in this time interval to ignore

new events in that time span. Since all channels are treated concurrently, the system dead-time will equal the front-end shaping time plus channel dead-time, the latter being equal to the busy length of 20ns.

Furthermore, the results from a readout of externally generated fixed-width test-pulses suggested that the total pulse width jitter was less than 100ps. The setup did not include the analog front-end, but nonetheless indicates a very high accuracy of the digital readout electronics. Readouts were also conducted with LYSOs, the custom analog front-end, and gamma-emitting and positron-emitting sources. These results indicated a promising detector linearity, and when a software coincidence window of 2ns were applied the energy resolution was measured to 15.0%, which is a very decent result without having optimised for light collection.

Note that these results are all preliminary, a further systematic studies are definitely needed. However, they seem to indicate that the readout system extracts the TOT time and width correctly, and tags events with proper event and channel numbers. However, what remains to be verified is the hardware coincidence matching. At this point the Trigger Unit is merely "simulated" inside the readout design, but this is just a temporary solution.

## *Outlook* 6.1

Some suggestions for future work is listed below.

- *Fine resolution coincidence windowing.* A high priority goal should be to verify the hardware coincidence windowing, and add support for adjusting these in steps equalling the sampling period.
- *Scalability.* The readout design has been tested running at 100 MHz core frequency, 1 GHz sampling frequency, and up to 16 channels. This is because only 16 GPIO LVDS-pins were available on the ML505 evaluation board. The design should be modified to fit to the `EVL50G`-board (fig. 2.3), and the effect of implementing up to 84 channels investigated.
- *Throughput.* The per-channel logic and event building process are able to handle a sustained system throughput of 100 Mevents/s. However, the embedded project congests at speeds less than 1% of that. Another project member has tested the design with a custom logic UDP-core, thus being able to greatly enhance the networking speed.
- *Drop the embedded project*? The embedded project adds flexibility and features to the design. However, if it turns out that only a UDP-core is needed for each Readout Card, or extra logic must be saved (e.g.

to sample 84 channels at 1 GHz), then the embedded design should be removed.

- *RocketIO deserialiser.* Even if these resources are costly, some are available on almost every FPGA. Unless these are used for other purposes, a few could be used to sample the TOT-pulses at much higher speeds, to investigate its performance potential, and whether it could be used to measure $\gamma$-ray Time-of-Flight (ToF).
- *User Friendliness.* Although the design can be controlled using ChipScope and the VIO module, this is hardly a user friendly solution. A common front-end for performing these, and related tasks, should eventually be created.
- *System calibration.* In the future several Readout Cards and a Trigger Unit will be working together, which will require some synchronisation mechanisms. The Clock Region Unit has support already for clock domain switching, but additional logic will have to be added to trigger soft-reset of the front-end, and to synchronise the SERDES links between each Readout Card and the Trigger Unit.
- *On chip image processing.* In a 3d PET system like ComPET, the image reconstruction algorithms will demand a lot of processing power. A possibility is to offload some of this to the FPGA, or some to a GPU, or even do both. As a curiosity, with enough processing power there is also the possibility of attempting 4D PET, or PET-video. This was gained relevance lately, as it would be useful in brain surgery.

# A

# Getting Started

How to go about to get the various project software up and running? Can it be run on Linux, or even unsupported Linux platforms? The answer is yes, with a few exceptions. This appendix aims to provide hints and tips on this matter.

Why Linux? Because the implementation tools are essentially just front-ends to a large number of build scripts native to Linux, and interacting with Embedded Linux designs it is simply easier when the development system also runs Linux. The only hassle on an unsupported Linux platform is to get USB-drivers for the Xilinx Programmer Dongle to work, but solutions exists here as well.

## *Development Environment* *A.1*

The Integrated Software Environment (ISE) provides all the tools required to develop FPGA-designs for Xilinx FPGAs, including a text editor, the XST synthesiser, implementation tools (like translate, map and place&route) and a binary configuration file (`.bit`) generator. The Embedded Development Kit (EDK), on the other hand, is a graphical front-end to a collection of tools and IP-cores intended to ease the process of building an embedded design.

### Xilinx ISE and EDK A.1.1

To install this software, just execute the setup binaries shipped with the ISE and EDK software. Go with the default options, but do not choose to install the cable drivers (see A.2.1). When done, source either `settings.sh` or `settings64.sh` (in the root install path), which will add these environment variables (or append to already existing ones):

```
1 PLATFORM=lin
2 XILINX="<ISE installation path>"
3 XILINX_EDK="<EDK installation path>"
4 LMC_HOME="${XILINX}/smartmodel/lin/installed_lin"
5 PATH="${XILINX}/bin/lin:${XILINX_EDK}/bin/lin"
```

This assumes a node-locked licence was used in the setup process. If a network licence is required, a variable pointing to must naturally also be added.

### Conflicting or Missing Libraries A.1.2

With an unsupported Linux system it is highly unlikely that all the libraries XST and EDK expects to find are actually present on the system. The problem is usually that the recompiled libraries bundled by ISE/EDK attempts to access a very specific version of a system library, and fails when it can not be found. There is two possible ways to solve this, either by removing the conflicting ISE/EDK library completely, thereby forcing the use of system defaults, or to substitute it with a symlink pointing to the closest matching library on the system:

```
1 >> cd <EDK or ISE path>/lib/lin  (assuming 32-bit architecture)
2 >> mv <conflicting_lib_name>.so  <conflicting_lib_name>.so.bak
      (try this, see if it works. in case it does not, continue)
3 >> ln -s -T /path/to/system/lib_location/<best_matched_version_lib_name>.
      so  <conflicting_lib_name>.so
```

Does it work? In most cases, yes. Mostly the libraries in question are ”common” (i.e. to for instance libstdc++), and most versions will work, but there is simply no guarantee.

## *Accessing the FPGA and Configuration Memories* A.2

To configure the FPGA, view its internal signals, or setup configuration memories, a direct link between the host computer and the board must be setup. A popular solution is to use a JTAG USB-cable, but it is not straight-forward to get this to work on an unsupported Linux platform.

### USB Cable Drivers A.2.1

At the time of writing, the tools needed to upload the `.bit`-files to the FPGA or configuration memories[1] are available for free off the Xilinx webpage [14]. Recent versions contains a *built-in* user-space USB-driver called `libusb.so`[2]. This is the *only viable* solution to getting the Xilinx cable drivers to work on an unsupported Linux[3]. Thus, fetch and install.

Once installed, these tools can be accessed by executing the graphical front-end *iMPACT*. Make sure it loads `libusb.so` by setting the following environment variable:

```
1 XIL_IMPACT_USE_LIBUSB=1
```

Now, instead of putting the binaries and libraries of the newly obtained programming tools to the global environment, and risk all sorts of system conflicts, do it locally:

```
1 >> cd <EDK or ISE path>/bin/lin        (assuming 32-bit architecture)
2 >> mv _impact  _impact.bak             (make a backup)
3 >> touch _impact                       (create a new file)
4 >> chmod a+x _impact                   (make it an executable)
```

Fire up the favourite editor and enter the following into this file:

```
1 #! /bin/bash
2 LD_LIBRARY_PATH="/path/to/programming_tools/lib/lin" /path/to/
      programming_tools/bin/lin/_impact $1 $2 $3 $4 $5
```

Note that now, when ISE or EDK attempts to program the the FPGA, the new version of Impact will be used. The libraries required by this version of iMPACT is specified in the `LD_LIBRARY_PATH`, thus promotes a safe "cross-compilation" environment - with decent cable driver support.

[1] These are non-volatile memories from which the FPGA can fetch its configuration upon power-up.

[2] Support added from ISE version 10.1.01 (although it has improved in more recent releases). http://www.xilinx.com/support/answers/29310.htm

[3] A few other approaches exists, including compiling support into the Linux kernel. But unless the system is supported, i.e. it uses Red Hat, I highly discourage attempting to make this work. It takes a lot of effort, and every kernel update will cause it to break. It is really too bad, as having the FPGA showing up as a character device in `/dev` would have made it truly easy to stream files to it.

### iMPACT A.2.2

Sending a `.bit`-file to the FPGA or configuration memory should be easy as anything. Simply use the ISE/XPS interface to do so. But in the - not so unlikely - event that this does not work, this is how the solution. Start iMPACT and perform the following steps[4]:

- **iMPACT Project** is the title of the pop-up window you should get when launching iMPACT. Select `create a new project (.ipf)` and hit `OK`.
- **iMPACT - Welcome to iMPACT** is the title of the next window you will get. Select `Prepare a PROM File` and click `Next`.
- **iMPACT - Prepare PROM Files**. Check that *I want to target a...* is set to `Xilinx PROM`, *PROM File Format...* is set to `MCS`, and *Checksum Fill Value (2 Hex Digits)...* is set to `FF`. Finally you need to fill the *PROM File Name...* field, and set the location to where you want the `MCS` file stored. When done, hit `Next`.
- **iMPACT - Specify Xilinx PROM Device**. Make sure the *Select a PROM (bits)...* leftmost rolldown box says `xcf`, and the rightmost rolldown box contains the name of the configuration device you want to program. When these boxes are set hit `Add`. Make sure all the checkboxes on this page are unchecked, and click `Next`.
- ***iMPACT - File Generation Summary***. Verify the settings and click `Finish`.

A small pop-up box informs you that you should now adding files to the project. Click `OK`, then select the `.bit` file you want to program the configuration device with. Answer `NO` to the next box asking you whether you want to add more devices or not. Next doubleclick the "flow" named `Boundary Scan` in the topleft "Flows" window. The "main" window should now say *Right click to Add Device or Initialise JTAG chain.* Do the latter and select the proper `.bit` file and `.mcs` file when confronted with this. In the same window you should now be able to see two icons, respectively representing the configuration device and the FPGA.

To start the programming procedure hit arrow tagged `Program` in the "iMPACT Processes" window. In the window that appears check the box reading `Erase Before Programming` (under *General CPLD And PROM Properties*); likewise with the box labelled `Load FPGA` (under *PROM Specific Properties*). All the other checkboxes I tend to leave unchecked. When done hit the `OK` button and the programming should commence. That is it. Optionally, the FPGA can also be configured with ChipScope.

[4] Approach may be slightly dependent on iMPACT version.

### ChipScope A.2.3

ChipScope Pro is a JAVA based application providing an Integrated Logic Analyser (ILA) and a Virtual Input/Output (VIO) module, which may be used to study the relationship between internal signals or provide input stimuli, respectively. These cores are optimised for size and performance, but when operated at speeds exceeding 200 MHz, or in a design with logic utilisation exceeding 80%, the signals can often not be trusted. The ChipScope cores utilise a fair bit of *Distributed SelectRAM* (see 2.3.2.2), thus its buffer size should never be set higher than necessary.

Unfortunately, ChipScope does not fly well with Linux, mostly due to not being able to take advantage of the user-space `libusb.so` USB-drivers (A.2.1). It is, however, very light-weight, so running it in a Virtual Machine (VM) with Windows will do the trick[5].

For more information, see the implementation section on "control logic" (3.7.2).

## *Simulation* *A.3*

To get going simulating the design with Mentor Graphics ModelSim, one must first compile a set of HDL simulation libraries for all the Xilinx primitives (such as DCMs, PLLs, etc.).

### Xilinx Simulation Libraries A.3.1

HDL-libraries of all primitives in the FPGA are required in order to fully simulate the FPGA design. The most important ones are `Unisim` and `XilinxCoreLib`. Compile them as follows:

```
1  Backup environment and substitute the proper libraries:
2  >> OLDLIBS=$LD_LIBRARY_PATH
3  >> export LD_LIBRARY_PATH="/lib:/usr/lib:<ISE path>/lib/lin:<EDK path>/
       lib/lin""
4  Compile Unisim and XilinxCoreLib for ModelSim SE, VHDL+Verilog support:
5  >> compxlib -s mti_se -arch virtex5 -l verilog -lib unisim
6  >> compxlib -s mti_se -arch virtex5 -l vhdl -lib unisim
7  >> compxlib -s mti_se -arch virtex5 -l verilog -lib xilinxcorelib
8  >> compxlib -s mti_se -arch virtex5 -l vhdl -lib xilinxcorelib
9  Optional, compile the XST libraries aswell:
10 >>
11 Reset the environment:
12 >> LD_LIBRARY_PATH=$OLDLIBS
13 For a list of other possibilities:
14 >> compxlib -help
```

[5] However, if someone knows how to run it in Linux, please let me know.

Now, reference these in the Modelsim (see A.3.2) `.ini` file, `<ModelSim path>/modeltech/modelsim.ini`:

```
[Library]
...
unisim = <path to ISE>/vhdl/mti_se/unisim
XilinxCoreLib = <path to ISE>/vhdl/mti_se/XilinxCoreLib
...
```

If you want the XPS libraries as well (to simulate the embedded project), open XPS, then click `Simulation ▶ Compile Simulation Libraries...` In case a custom IP was created, the simulation library may be obtained by:

```
As before, setup the environment then type:
>> compedklib -s mti_se -lib <IP name> -X <path to EDK project>/edk/
    pcores
```

### ModelSim A.3.2

Mentor Graphics ModelSim is one of the leading HDL simulation tools on the market, and was used extensively in the development of this design. Obtain the "SE" version (has SystemVerilog support) off the Mentor Graphics homepage:

http://www.model.com/downloads/default.asp

When you are past the registration process, you get a temporarily username and password which you may use to download ModelSim. In my case, downloading ModelSim 6.5 on an x86 Linux box required me to fetch the following files:

```
>> wget ftp://<username>:<password>@ftp.model.com/SE/6.5/

INSTALL_NOTES             (Install notes for current release)
RELEASE_NOTES             (Release notes for current release)
modelsim_se_install.pdf   (Licensing and installation details)

install.linux             (Install executable for x86 or x86_64 Linux)
modelsim-base.mis         (Base functionality for all platforms)
modelsim-docs.mis         (Documentation, i.e manuals, tech. notes, ...)
modelsim-linux.mis        (Installation files specific to Linux Redhat)
```

When setup, add or append the following environment variables:

```
PATH="<installation path>/modeltech/linux"
MGLS_LICENSE_FILE="<licence host>@<licence server>"
```

Having obtained a running copy of ModelSim, I suggest performing these initial procedures:

- *Clean Start.* If ModelSim has been used to simulate the project before, I recommend heading over to the project location and deleting the old `work` library (folder), the `modelsim.ini` file, and the `<projectname>.mpf` before doing anything else. Sometimes ModelSim will misbehave unless these steps are carried out.
- *Set Project Directory.* `File` ▶ `Change Directory...`. Select a location for ModelSim output files.
- *Create Working Library.* `File` ▶ `New` ▶ `Library...`. Use default library name "work", check that *a new library and a logical mapping to it* is selected and hit `OK`. Make sure the library listing also contains the `Unisim` and `XilinxCoreLib` libraries (see A.3.1).
- *Create the Project.* `File` ▶ `New` ▶ `Project...`. Select a proper name for it, check that the project location is set right and make sure "work" is set as the default library. Do *not* copy settings from a previous `modelsim.ini` file, as we want to start blank in order to know things are under control. Make sure that field is blank, and that the option for copying library mappings is checked. When done, hit `OK`.
- *Add items to the Project.* This is the label of the window appearing after you created the project. You may want to start making some folders to set up a hierarchy. When adding files I suggest you choose to reference to your files instead of copying them (at least I want it to simulate my source files and not some "backup" I made).

When this is all done, you will need to compile the files the "ModelSim way" and add them to a simulation. This is best accomplished by using a .do file. It is more reliable, much quicker and features a wide variety of options for customising the simulation to better suit your preferences.

```
1  onerror {abort all}  (in case of error, do not attempt to continue)
2
3  do compile.do        (call another .do-file to compile all sources)
4
5  vlog +acc -sv      <path to testbench>/<filename> (SystemVerilog/Verilog)
6  vcom -93 -explicit <path to testbench>/<filename> (VHDL)
7
8  quit -sim            (quit any ongoing simulations)
9  vsim -voptargs=+acc -t 1ps tb (start simulation)
10
11 (Optional) Position the wave window (example values shown):
12 view -undock -height 900 -width 1400 -x 0 -y 0 wave
13
14 do wave.do           (run a .do to populate the wavediagram with signals)
15 run 100000 ns        (run the design)
16 radix hex            (set radix to hex)
```

To create the `wave.do` file mentioned above, run the simulation, then add add signals to the ModelSim Wave-diagram, and export it to a `.do`-file with `File ▶ Save Format ...`

The designer looking to spend some serious time with ModelSim should also read the following (located in `<ModelSim path>/modeltech/docs/`):

```
1 pdfdocs/modelsim_se_tut.pdf   (Get started tutorial)
2 pdfdocs/modelsim_se_user.pdf  (User manual)
3 technotes/sysvlog.note        (SystemVerilog support note)
```

## *Server Setup* *A.4*

This project use two different servers, one to hold the Git project repositories, and one to source a public webpage (see 1.5 and A.4.2) with real-time pointers to these. This section briefly describes the setup of these two servers. For more information on Git, and example usage when working with this design, see B.2.

### Git A.4.1

For convenience, it is nice to have a central server to push changes to (even though with Git you do not have to). Assuming the server is accessible[6], log onto it and setup the Git repository:

```
1 >> adduser git                (add a 'git' user)
2 >> mkdir /home/git            (create a home folder for this user)
3 >> chown git:git /home/git    (make sure 'git' owns its own home folder)
4 >> passwd git                 (set the password, e.g. to 'compet')
5 >> su git                     (login as 'git')
6 >> pwd (optional)             (check that the path is indeed '/home/git')
7 >> mkdir compet.git           (create a folder for the repository)
8 >> git --bare init            (setup a 'bare' repository)
```

The `--bare` option tells git that this will be a server repository, allowing external users to push to it without permission issues. The repository folder suffix `.git` is simply a common way of indicating that the repository is in fact ”bare”.

[6] This tutorial will not deal with the issues of accessing a computer across the internet, although this project did. Suffice to say, somehow the DNS problems must be overcome, or a virtual connection must be made (e.g. with Hamachi).

### The Webpage A.4.2

The project webpage is currently served and maintained using the following software:

| | | |
|---|---|---|
| `Apache` | 2.2.14 | A web server (remember to open port 80). |
| `MySql` | 5.0 | An open source database. |
| `PHP` | 5.2.12 | For dynamic contents on the webpage. |
| `phpMyAdmin` | 2.11.9.6 | To handle databases and ease PHP interaction. |

To install, refer to the extensive online documentation specific to your distribution. These will explain in detail how to setup and configure these tools in order source a basic homepage, which in this case is the one provided in `software/user-apps/web`. However, if you wish to use Webgit, keep reading.

#### *Webgit* *A.4.2.1*

From the project one may graphically navigate all the current repositories, and check out the status of each. The program making this possible is *Webgit*. It is natively available in Git if support for 'ctags' and 'perl' is compiled in. To host this project, create or open the file `/etc/gitweb.conf` and enter the following:

```
1 $projectroot = '/home/git/gitweb';  (the project repositories)
2 $sitename = 'ComPET Git Repos';     (custom title)
3 $project_maxdepth = 2;              (the maximum tree display depth)
```

Here the `/home/git/gitweb` folder simply contain symlinks to the actual project repositories. Just point to the repositories in one way or another. Now, in addition you will need to setup Apache with 'CGI' support. Edit `/etc/apache2/vhosts.d/default_vhost.include` to include the following:

```
1  DocumentRoot "/var/www/localhost"
2
3  # This should be changed to whatever you set DocumentRoot to.
4  <Directory "/var/www/localhost">
5                      (...)
6          Options Indexes FollowSymLinks ExecCGI
7           <Files gitweb.cgi>
8                SetHandler cgi-script
9           </Files>}
10                     (...)
11         AllowOverride All
12                     (...)
13         Order allow,deny
14         Allow from all
15 </Directory>
16
17 SetEnv GITWEB_CONFIG /etc/gitweb.conf
```

# B

# Project Management

My project consists of more than 100 000 files, mostly due to relying on the Linux kernel source tree, Petalinux source and the GNU toolchain. ISE and EDK projects are rather verbose too, so it is easy to get slightly lost in the management of this project in the beginning.

There are 3 main challenges I would like to point out. First, as the project complexity increases so does the challenge of maintaining it. Second, a project of this size benefits greatly by being put under version control, but it is cumbersome to version control projects consisting of mostly binary files. Third, compiling FPGA-code is a slow and resource-demanding process, thus a great deal of time can be saved by outsourcing the ”job” to faster computers.

To solve these challenges one must first know what parts make up the project. Thus this appendix will start off by introducing the *project file structure* (B.1.1). Then some experiences with keeping it under version control with *Git* will be shared, along with some hints and tips to avoid a few major pitfalls (B.1.1). Finally, to ease project maintenance and allow for remote building of the HDL-source, a few self-composed *makefiles* will be presented (B.3).

## *Directory Structure* *B.1*

The project file-structure is not random, but chosen at such that it fits well with the PetaLinux source code tree, is easy to maintain, and is logical (to the extent this is achievable). The table below lists a few selected folders alphabetically, with a short description.

| | |
|---|---|
| `compet` | Main project folder (may have any name). |
| + `doc` | The project documentation. |
| + `bin` | Scripts used to customise the Latex build. |
| + `common` | My self-made all-in-one Latex style package. |
| + `doxygen` | Doxygen source and generated documents. |
| + `gfx` | Graphics for this document. |
| + `pdf` | Compiled `pdf`'s go here. |
| + `tmp` | Most temporary build files end up here. |
| + `hardware` | All hardware and firmware. |
| + `edk_user_repository` | PetaLinux Autoconfig BSP generation tools. |
| + `fs-boot` | A bootloaders used by PetaLinux. |
| + `user-platforms` | *All the HDL-code!* See tab. B.2. |
| `matlab` | Recorded data and plots. |
| `share` | Shared with the Trigger Card[1]. |
| + `software` | Various software. |
| + `linux-2.6.x-petalogix` | The Linux 2.6 kernel source tree. |
| + `petalinux-dist` | The PetaLinux kernel source tree[2]. |
| + `uClinux-2.4.x` | The Linux 2.4 kernel source tree[3]. |
| + `user-apps` | Self-composed applications. |
| + `shared` | The C-programs used to readout data[4]. |
| + `web` | The web-page source code. |
| + `user-modules` | Self-made Linux modules (hardly working). |
| + `tools` | Various PetaLinux tools. |
| + `common` | PetaLogix helper-scripts. |
| + `linux-i386` | The GNU toolchain. |

*Table B.1 - Project folder list*

The blue entries are separate Git-repositories, see B.2. The main folder of all the above is the `user-platforms`, since it contains all the project HDL-code and embedded projects. Its subfolders are structured as follows:

[1] This is an NFS export mount.
[2] Depends on the Linux 2.6 kernel source tree, *including* some of its binaries.
[3] Currently not used.
[4] This folder is mounted in the global SHARE-folder as well, to be included in the NFS-mount. This way all these programs may be launched from inside the PetaLinux environment.

| | |
|---|---|
| `user-platforms` | All the HDL code. |
| `+ <ISE project>` | HDL-source. See tab. B.3. |
| `  + ise` | A folder where ISE can put its junk binaries. |
| `  + bit` | A collection of old bit-files. |
| `  + logs` | Logs from the automation scripts (TCL). |
| `  + tcl` | TCL-scripts to automate builds (see B.3). |
| `  + vsim` | ModelSim simulation files. |
| `+ ml505peta` | The main XPS project. |
| `  + __xps` | XPS temporaries[1]. |
| `  + blkdiagram` | Project block diagram[1]. |
| `  + bootloops` | Bootloop used to keep uP busy[1]. |
| `  + bsp` | Board Specific Packages, if any[1] |
| `  + data` | Contains a template `.ucf`-file. |
| `  + etc` | A few scripts, including one to download `.bit`-file. |
| `  + hdl` | HDL-code that may be used for simulation[1]. |
| `  + implementation` | Resulting binaries, including `.bit`-file(s)[1]. |
| `  + pcores` | Cores created with "Add Peripheral"-wizard. |
| `  + synthesis` | Synthesis temporaries[1]. |
| `  + <application>` | Every EDK-application gets a distinct folder. |

*Table B.2 - Hardware sub-folder list*

To keep the project clean, source code is grouped after functionality, and implementation and simulation files are kept in subfolders, as is shown in the table above. The main groups of functionality are:

| | |
|---|---|
| `_top_final` | This is the top-most ISE project, containing the entire hardware project, including source files from all the below listed folders. |
| `chipscope` | Contains the ChipScope Core, Integrated Logical Analyser (ILA) and Virtual Input/Output (VIO) modules, and the HDL-source which includes these. |
| `cru` | All the HDL source for the Clock Reset Unit (CRU). |
| `front-end` | All the source for the readout logic, including - but not limited to - the deserialiser, parametriser and event-builder. |
| `shared` | This contains global design units, such as a highly flexible FIFO and global design constraints, types and functions. |
| `ml505peta` | Contains the entire XPS embedded design. |

*Table B.3 - ISE and EDK projects.*

[1] All these folders are not essential and will be recreated by the implementation tools if missing.

### Source Files B.1.1

A few source-files of particular importance are described below.

#### *Hardware* *B.1.1.1*

Knowing the essential files out of the masses makes life easier. A list of some of the most important ones may be found below (most of these are referred to in this document).

| | |
|---|---|
| _top_final/ | |
| + ise/ | |
| + bit/ | A collection of old bit-files. |
| top_download-0.1.bit | Precompiled design version 0.1. |
| top_download-0.2.bit | Precompiled design version 0.2. |
| top.tcl | Used to build and manage the ISE-project. |
| + tcl/ | Custom TCL-scripts to automate ISE. |
| ise.tcl | Used to open and build the ISE-project. |
| settings.tcl | A common script to set ISE-settings. |
| wrapper.sh | Script-wrapper for environment setup etc. |
| top.vhd | This is the top-module of the design. |
| top.ucf | The constraints-file for the design. |
| tb_*.sv | Testbenches (written in SystemVerilog). |
| *.do | ModelSim simulation scripts. |
| compile.do | Builds the ModelSim work library. |
| Makefile | The main Makefile. Used to create, build or clean the project. |
| chipscope/ | ChipScope files. |
| + projects/ | ChipScope project files. |
| v0.1-1channel.cpj | For design version 0.1. |
| v0.2-8channels.cpj | For design version 0.2, up to 8 channels. |
| icon.ngc | Integrated Controller (black box). |
| ila.ngc | Integrated Logical Analyser (black box). |
| vio.ngc | Virtual Input/Output (black box). |
| core.vhd | Wrapper for the black boxes above. |
| core_sim.vhd | In simulations the file above is substituted with this one. |
| cru/ | Clock Reset Unit. |
| pll_all.xaw | The PLL used in the design (black box). |
| cru | Clock Reset Unit, top file. |

*Table B.4 - Essential hardware files (part 1)*

| | |
|---|---|
| front-end/ | The digital readout front-end. |
| ·· fe.vhd | Front-end, top file. |
| ·· fe_cotrg_processing.vhd | Coincidence tracker. |
| ·· fe_ch.vhd | Wrapper for the per-channel logic. |
| ·· fe_ch_iserdes.vhd | Channel deserialiser. |
| ·· fe_ch_pargen.vhd | Channel parameter extraction. |
| ·· fe_ch_fifo.xco | Channel FIFOs (black box). |
| ·· fe_ch_buf.vhd | Wrapper for the FIFOs black-box. |
| ·· fe_eb.vhd | The Event Builder. |
| ·· fe_eb_submux.vhd | The Recursive SubMux. |
| ·· fe_input_stimuli.vhd | Internal pattern generator, for testing. |
| ·· fe_input_stimuli_sim.vhd | In simulations the file above is substituted with this one. |
| shared/ | "Global" design units. |
| ·· a2s.vhd | A parametrised FIFO for synchronisations. |
| ·· constants.vhd | Design parameters. |
| ·· functions.vhd | A collection of custom functions. |
| ·· types.vhd | Global types (buses, etc.). |
| ·· shift.vhd | A shift-register description that ensures implementation with LUTs. |
| ml505/ | The embedded project (bad choice of name). |
| + edk/ | |
| ·· ·· system.bsb | The Board System Builder project file. |
| ·· ·· system.mhs | Hardware settings for the embedded project. |
| ·· ·· system.mss | Software settings for the embedded project. |
| ·· ·· system.xmp | General embedded project settings. |
| ·· system.vhd | Simulation "dummy". |

*Table B.5 - Essential hardware files (part 2).*

***Software (Various)*** *B.1.1.2*

Furthermore, for the ones looking to continue either the documentation or the embedded project software, be aware of the below listed files.

| | |
|---|---|
| `doc` | |
| `+ bin/` | Custom scripts to build the documentation. |
| `·· la2pdf` | Wrapper for `pdflatex`/`xelatex`/etc. |
| `·· laFigure` | An extension to graphics inclusion in Latex. |
| `+ common/` | |
| `·· mytemplate.cls` | Latex document type extension. |
| `+ doxygen/` | Doxygen documentation. |
| `·· hw-template.cfg` | Doxygen config template. |
| `·· Makefile` | Custom makefile for Doxygen. |
| `+ gfx/` | All the graphics in this document. Nuff said. |
| `+ pdf/` | Last compiled version of this document. |
| `·· *.tex/*.bib` | Sources for this document. |
| `software` | |
| `+ petalinux-dist/` | The PetaLinux kernel tree. |
| `·· .config` | The kernel config-file. |
| `+ user-apps/` | Applications for the embedded design. |
| `·· share/` | Sources for the programs in the share export. |
| `·· clean_bram.c` | Wipes the BlockRAMs contents. |
| `·· read_bram.c` | Readout program for design version 0.1. |
| `·· read_bram2.c` | Readout program for design version 0.2. |
| `·· read_bram_speed.c` | A lightweight version of the one above. |
| `·· web/` | All files needed to host the webpage. |
| `matlab/` | A few runs with scripts to treat them. |
| `·· run_*.txt` | Data from the runs. |
| `·· run_*.m` | Scripts to recreate plots from the runs. |
| `·· *.eps/*.svg` | Plots. |
| `settings.sh/settings64.sh` | Environment setup scripts. |

*Table B.6 - Essential software files.*

## *Git* B.2

Git is an open source, *distributed* version control system (VCS) designed to handle everything from small to very large projects with speed and efficiency. Every Git clone is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server. Branching and merging are fast and easy to do. It is the VCS of choice in CompPET.

It is worth mentioning that ISE has a ”snapshot” feature for making backups, but these take up way too much space (from a few tens to hundreds of MBs). Besides, it is not nearly as powerful as Git, nor is it open source.

### What is tracked? B.2.1

As highlit in tab. B.1, this project is composed of 8 repositories. There is a top repository called (`compet`), containing the `matlab` and `share` folder, a repository tracking the documentation, one tracking user-generated hardware, and a final one for the user-generated software. This amounts to four.

The remaining four repositories tracks the Linux kernel sources, and the GNU toolchain (see C.1.1.1). These are kept apart from the rest of the design because they hardly every change (except the PetaLinux source when the kernel is recompiled), and because they amount to an excess of 100 000 files and 2.7GB. Also, categorising the repositories this way makes the tracking swifter, and change logs cleaner.

Before moving on, one thing should be said. The developers of the Linux kernel also use Git, and thus a large number of `.gitignore`-files are present in numerous kernel tree locations. These specify what files Git is allowed to track, and inherently binaries are often left out. This makes sense, but unfortunately PetaLinux *depends* on some of these files. Thus, the kernel 2.6 Git-info had to be slightly altered for this project[1].

Sound good, but how to interact with the repositories? Coming up!

### Initial Procedures B.2.2

Say you just installed Git, and do not have the slightest clue what to do with it. This is a good time for introducing yourself to Git:

```
1 >> git config --global user.name "Your Name"
2 >> git config --global user.email your.email@your.domain
```

This stage will never have to be repeated.

[1] More specifically, the troubled `.gitignore`-files was localised in the 2.6 kernel and put into *exclude*, an ”ignore” construct of higher priority provided by Git. This is not elegant, but it works.

### Synchronising Repositories B.2.3

Now, proceed by fetching a few of the project repositories. Since this is the first time the repository is downloaded, it must be *cloned*. To clone the top repository from the server (setup as explained in A.4.1), proceed as follows:

```
>> git clone git@<server name>:compet.git ./compet
```

Why not clone the rest while you are at it?

```
>> cd compet
>> git clone git@<server name>:
     compet.git/doc.git doc
     compet.git/hardware.git hardware
     compet.git/tools.git tools
     compet.git/software.git software
>> cd software
>> git clone git@<server name>:
     compet.git/software.git/uClinux-2.4.x.git uClinux-2.4.x
     compet.git/software.git/linux-2.6.x-petalogix.git linux-2.6.x-petalogix.git
     compet.git/software.git/petalinux-dist.git petalinux-dist
>> cd ..
```

This syntax is admittedly awkward. Fortunately, the clone is performed only once. Later on, one instead 'pulls' to fetch changes,

```
>> cd doc                (enter the repository to update)
>> git pull master       (merges changes from the 'master' branch)
```

If you do not know whether you want the changes to be merged with your local copy, simply take a "peek" by typing

```
>> git fetch master
>> git log -p HEAD..FETCH_HEAD
```

The log-command means "show everything reachable from FETCH_HEAD, but exclude everything reachable from HEAD". HEAD is the current state of the local repository, and FETCH_HEAD is the state of the fetched project.

Finally, to merge the fetched repository with the local one, use *git merge*:

```
>> git merge master              (Merge <user>/master into current branch)
```

### Making Changes B.2.4

Say you altered some files, what now? First you must tell Git which files it should include in a commit (refered to as adding them to the *index*):

```
>> git add <files or folders>
```

To see what files are added to the index, check the status:

```
>> git status
```

Red entries are files that are either not tracked by Git or not added to the index, while green entries are. To commit the changes of the files added to the index, simply type

```
>> git commit
```

If you want all tracked files to be commited, and not just a selection, add the `-a` switch:

```
>> git commit -a
```

Before you commit, should you ever wish to see what changes will actually be included, type

```
>> git diff --cached
```

You can also call `git diff` (without the –cached) in order to see any changes that you have made but not yet added to the index. Finally, you may eventually want to *push* the changes back to the server, updating the repository there:

```
>> git push master
```

### Branches B.2.5

When two people work on the same repository, these two will *branch*, i.e. the repository for each user started off alike, but will start to diverge. But even as a single user one might want to create branches, e.g. to test some experimental feature without risking to compromise the main branch (called 'master'). To create and switch to a new branch do the following:

```
>> git branch experimental     (Creates a new branch called experimental)
>> git branch                  (Prints a list of all existing branches)
>> git checkout experimental   (Switch to the experimental branch)
```

At the point of creating a branch from an initial one, the two will start to diverge. To merge them, first make sure you are on the initial branch, then type

```
>> git merge experimental      (Merges experimental with current branch)
```

In case the merging fails, run `git diff` to see where the problem resides. At any time, to finish off a branch, you may type

```
>> git branch -d experimental  (Delete branch, ensuring changes are
    merged)
>> git branch -D experimental  (Delete branch, discarding changes)
```

### Error Correction B.2.6

Ahh, you screwed up, what now? Well, depends really.

A common error is to commit too quickly, realising some files were left out or that the commit message was simply silly. This can be fixed by simply continuing adding the files you lack, then typing

```
>> git commit --amend
```

which will effectively overwrite the last commit. Changes way back in the history are trickier, but fixable (naturally, be cautious here). Say, for instance, that you added a large chunk of binaries or temporary files which simply has no value. You can ”undo” the history by using the git-filter-branch command:

```
>> git filter-branch --index-filter "git rm -r --cached --ignore-unmatch
    <reg.exp.>" HEAD
```

This removes it from Git history, but does not clean up Gits ”temporary” cache of these files. This means the .git repository maintains its previous size. To slim it you can use the - not so recommended - approach below:

```
>> rm -rf .git/refs/original/ && git reflog expire --all &&  git gc --
    aggressive --prune
```

However, a smarter choice is simply to clone your own repository, which will leave any temporary files out.

## *Makefiles* *B.3*

Navigate to `hardware/user-platforms/_top_final` to find the main Makefile in this project. It can be used to clean the entire project tree, rebuild the ISE project, and compile the HDL code.

### Remote compilation B.3.1

Compiling FPGA-code is resource-demanding, and implementing large designs can take several hours. To offload the computer used to develop the code, and allow the code to be compiled on a potentially faster computer, the following procedure may be used.

1. Make sure all source files are tracked by Git, and *commit* changes (see B.2).

2. Either *push* the repository to the external machine[2], or log on to the external machine and *pull* changes from the development machine (preferable).
3. When logged onto the external machine, enter `_top_final` (see B.1.1) and simply type `make`.

[2]Unless the external repository is "bare", pushing changes must be forced (`--force`). This overwrites potential changes in the external repository, and should be avoided.

# C

# Embedded Tutorial

Quite a few steps are required to build an embedded micro-processor and run embedded Linux. To ease the process for those seeking to do so, this tutorial should come in handy. It explains how to build and run a complete system with

- Xilinx Microblaze 7.10.d microprocessor, running
- PetaLogix PetaLinux 0.40 final, built for the
- ML505 Evaluation board.

The design is implemented using Xilinx ISE and EDK v10.1.03.

The tutorial is based on the official PetaLinux documentation [24], the Xilinx OpenSource Wiki [27] (and various pages linked to from that site), Xilinx Embedded Linux on Xilinx Microblaze Workshop [25], and the various experiences made while working on this design.

## *Preparing the Host Computer* C.1

Some software must be obtained and installed prior to developing the embedded project (in addition to all implementation software, see A). These will briefly be mentioned here.

### PetaLinux C.1.1

PetaLinux from PetaLogix is based on uClinux, but designed specifically for the Microblaze soft-processor. To get going with it, one must obtain the PetaLinux source. Version 0.40 (final) is available for free through a University donation program, which this project applied for and kindly received.

Note that the free 0.40 version only supports ISE/EDK 10, for newer releases[1] PetaLinux is more advanced but not free. For these versions one might want to check out the OpenSource Xilinx Linux instead [26].

#### *Directory Structure* C.1.1.1

When the PetaLinux source is obtained and unpacked, the directory structure contains the `hardware`, `software`, and `tools` folder listed in B.1.1.

The *hardware* folder contains 3 sub-folders; `edk_user_repository`, which contains several auto-configuration tools used to merge PetaLinux with the embedded hardware; *fs-boot*, a simple first-stage bootloader used to get the micro-processor up and running with basic functionality after power-up; and *user-platforms*, a folder intended to be used to store the entire FPGA-design. See B.1.1 for more details.

The *software* folder contains 5 sub-folders; *linux-2.6.x-petalogix*, *uCLinux-2.4.x*, and *petalinux-dist*, the kernel source tree for Linux 2.6, 2.4 and PetaLinux; and *user-apps* and *user-modules*, initially empty folders intended to contain modules and applications designed to be run with the embedded Linux.

Finally, the *tools* folder contains PetaLogix *helper scripts* and the *GNU toolchain*. These are only added once to the design, and never changed.

### Ethernet IP-address C.1.2

This embedded system will assume a static IP-address of the host system of 192.168.1.5. Force the host Ethernet connection to use this IP, and an IP-cable can be connected directly from the host-computer to the ML505 Evaluation Board. The physical layer of the ML505 MAC supports both crossed and normal IP-cables (automatic switching).

[1] The version 12 of ISE and EDK is the currently most recent.

### Tftp C.1.3

The Trivial File Transfer Protocol (TFTP) is, as the name suggests, simple. Stated simply, it means not much can go wrong, but patience will be required. The Linux kernel image provides native support for TFTP, so setting the host computer up with TFTP allow it to exchange files with the embedded system straight after downloading the `.bit`-file.

Start by setting up the root `/tftpboot`-folder, and assign full permissions.

```
1 (as root)
2 >> mkdir /tftpboot
3 >> chmod a+rwx /tftpboot
4 >> chown nobody /tftpboot (optional)
```

Make sure the TFTP directory is setup with `/tftpboot`. In the distribution used throughout this development (Gentoo), ATFTP (Advanced TFTP) was used, and the path was setup in `/etc/conf.d/atftpf`. Once done, start the TFTP-service[2], or add it to the default run-level to make it start upon computer reboot [3]. Some distributions may do this automatically.

### NFS C.1.4

The Network FileSystem (NFS) will be used to share development files and Linux kernel binaries between the host-computer and embedded system. Install it, and specify the folder(s) that is to be shared in `/etc/exports`:

```
1 /tftpboot 192.168.1.10(rw,sync,all_squash,no_subtree_check,anonuid=500,
      anongid=500)
2 /share    192.168.1.10(rw,no_subtree_check)
```

Of these options, make sure to include 'rw' (permit read and write) and 'no_subtree_check' (makes the NFS-connection slightly more robust). Now, start (or restart) the NFS-daemon, which should make these folders visible on the network.

### RS232 Interface C.1.5

A RS232 serial link will be used for basic communication with the embedded system. Thus, the host-system will need a program to control the serial link. For Linux Kermit may be used, for Windows Hypertransport or Hercules, all being free and light-weight. The latter was used in this project. Obtain, and set the serial link with the parameters listed in tab. C.1.

[2] Gentoo: `/etc/init.d/atftpd start`
[3] Gentoo: `>> rc-update add atftpd default`

### Telnet C.1.6

To log onto the embedded system over Ethernet, `telnet` is the easiest way (SSH (Secure Shell) can be made working, but requires some effort). It is supported by the PetaLinux kernel by default, and will work as soon as the embedded system is up and running. When it is, login with

```
>> telnet 192.168.1.10
  Trying 192.168.1.10...
  Connected to 192.168.1.10.
  Escape character is '^]'.
  login:    root
  Password: compet
#
```

## *Hardware Setup* *C.2*

This project uses the Microblaze soft micro-processor from Xilinx. Due to its soft nature, it is very customisable, which is advantageous but adds complexity to the design procedure. Out of the myriad of options, it is easy to get lost.

### Base System Builder C.2.1

The Base System Builder is a Wizard providing basic embedded systems for common evaluation and prototype boards, including the ML505.

Navigate to `hardware/user_platforms` and create a folder for the embedded design, e.g. `ml505peta`. Launch Xilinx XPS and start the Base System Builder. Go for "I would like to create a new design" and save the project, e.g. as `ml505peta/system.xmp`. Select the appropriate evaluation board, e.g. ML505. For the remaining part of the wizard, go with the options specified in tab. C.1 (page 107).

With the Base System Builder run with the settings mentioned, the design is coming together. However, a few minor changes are required; the Microblaze needs a *Barrel Shifter* and "full" Processor Version Register. In XPS, open the Microblaze IP, where these options may be found in the "Barrel Shifter" and "PVR" tab.

### Modifying Project Files C.2.2

All important settings for the embedded project are set in either the project `xmp`-file, `mss`-file or `mhs`-file[4].

To make the project compatible with PetaLinux, edit the `.mss`-file to add operating system information:

[4] Thus, these are the key embedded files being tracked with Git.

| Design Unit | | Feature | Settings |
|---|---|---|---|
| Processor | | Type | Microblaze |
| | | PLB frequency | 125 MHz |
| | | Reset | Active low |
| | | Debug I/F[1] | On-chip H/W debug module |
| | | Memory[2] | Data and instruction memory, use 8kB BlockRAM for each |
| | | MMU[3] | No |
| | | Cache | Enabled |
| | | Floating Point Unit | Disabled |
| IO devices | RS232 (Uart1) | IP | Xps_UartLite |
| | | Baud Rate | 115 200 |
| | | Databits | 8 |
| | | Parity | None |
| | | Use interrupts | Yes |
| | Uart2 | Disabled | |
| | GPIO (LEDs, Push Buttons, DIP Switches) | IP | Xps Gpio |
| | | Use interrupts | No |
| | Iic Eeprom[1] | IP | Xps Iic |
| | | Use interrupts | Yes |
| | Flash | IP | Xps Mch Emc |
| | Sram | Disabled | |
| | PCI Express | Disabled | |
| | Hard Ethernet MAC | IP | Xps LL Temac |
| | | Scatter-Gather DMA[4] | Yes |
| | | Use interrupts | Yes (default with DMA) |
| | DDR2 SDRAM | Use MPMC | Yes |
| | SysACE CompactFlash | IP | Xps SysACE |
| | | Use interrupts | Yes |
| Peripherals | Timer[5] | IP | Xps Timer |
| | | Counter width | 32 |
| | | Number of timers | 2 |
| | | Use interrupt | Yes |
| Cache | Instruction | Size | 2kB |
| | | Use cache for | DDR2 SDRAM |
| | Data | Size | 4kB |
| | | Use cache for | DDR2 SDRAM |
| Standard Input/Output | | | RS232 Uart1 |
| Boot memory | | | ilmb_cntlr (BlockRAM) |

*Table C.1 - Settings for Base System Builder*

[1] This functionality is implemented, but not used in this design. Remove if desirable.
[2] The data and instruction memory consumes a fair bit of BlockRAM. However, assigning less will cause PetaLinux to fail (with cryptic address space messages).
[3] To increase performance this design does not utilise a Memory Management Unit.
[4] Gather-Scatter DMA is currently not used by the design, should be implemented for improved networking performance.
[5] A timer is required in order to run embedded Linux.

```
BEGIN OS
 PARAMETER OS_NAME = petalinux
 PARAMETER OS_VER = 1.00.b
 PARAMETER PROC_INSTANCE = microblaze_0
 PARAMETER STDIN = RS232_Uart_1
```

*Listing C.1 - hardware/user-platforms/ml505peta/edk/system.mss*

The auto-configuration scripts (residing in `edk_user_repository`) are designed to be called from XPS when the hardware project is built. This is achieved by adding this path to the module search path in the `.xmp`-file:

```
VerMgmt: 10.1.03
IntStyle: ise
ModuleSearchPath: ../../../edk_user_repository/
MHS File: system.mhs
MSS File: system.mss
```

*Listing C.2 - hardware/user-platforms/ml505peta/edk/system.xmp*

Make sure all paths in these files are relative! Sometimes absolute paths are introduced by XPS (or sub-programs), which will break the project if moved to another folder or computer. Considering that this project is entirely tracked by Git and designed to be remote compiled, sticking with relative paths is certainly wise.

### FS-Boot C.2.3

FS-Boot is a first-stage bootloader used to boot the embedded Linux. Enter the XPS "Add Software Application Project" menu and set it up as follows:

| Feature | | Settings |
|---|---|---|
| Project name | | fs-boot |
| Processor | | microblaze_0 |
| Add files | Sources | `hardware/fs-boot/fs-boot.c`<br>`hardware/fs-boot/srec.c`<br>`hardware/fs-boot/time.c` |
| | Headers | `hardware/fs-boot/fs-boot.h`<br>`hardware/fs-boot/srec.h`<br>`hardware/fs-boot/time.h` |
| Compiler options | Application mode | executable |
| | Output ELF (binary) file | Use default |
| | Linker script | Use default |
| | Stack size | 1K |
| Optimisation level[1] | | Size optimised (-Os) |
| (Advanced) compiler options[2] | | -Wall |

*Table C.2 - FS-Boot setup*

[1] Naturally, keeping the binary size down is important to preserve resources on the embedded device.
[2] Gcc will used. -Wall (-W) enables verbose gcc output.

### Software Settings C.2.4

Before building libraries and software for the embedded project, the kind of memory in which these can run must be selected. Look up *Software Platform Settings*▶*OS and Libraries* and specify the following:

```
lmb_memory: dlmb_cntlr
flash_memory: FLASH
main_memory: DD2_SDRAM
```

## *PetaLinux Setup* *C.3*

### Sourcing Settings C.3.1

Before compiling anything, or using any of the PetaLogix binaries, a cross-compilation environment must be setup (i.e. one that differ from the native host-machine environment). An easy way to do this is to "source" the `settings.sh` file[5] in the project root folder

```
>> cd <project root>
>> source settings.sh       (or settings64.sh if on a 64-bit achitecture)
```

The custom Makefiles described in B.3 will source these settings when necessary, e.g. when remote compiling the project (see B.3.1).

### MenuConfig C.3.2

Compiling the PetaLinux kernel source is performed in a similar way as other distributions. Setting up a kernel from scratch is a cumbersome process, but fortunately the PetaLinux default kernel configuration is rather sane. Thus, only modifications will be listed here. Enter the `software/petalinux-dist` folder and type

```
>> make menuconfig
```

which should bring up the PetaLinux kernel configuration menu. To navigate, use the *arrow keys*, to toggle an option use the *space bar*, and ascend or descend the hierarchy with *enter* (when 'OK' or 'Exit' is highlit, respectively). Now, setup the *Vendor/Product* information

```
Vendor/Product Selection  --->
  --- Select the Vendor you wish to target
  (Xilinx) Vendor
  --- Select the Product you wish to target
  (ML505-ll_temac-sgdma-edk101) Xilinx Products
```

[5] A few tweaks have been added to `settings.sh` to add binaries needed to build the documentation to path as well.

Next we want to customise some kernel settings. Toggle the following option

```
Kernel/Library/Defaults Selection  --->
  (linux-2.6.x) Kernel Version
  (None) Libc Version
  [ ] Default all settings (lose changes) (NEW)
  [*] Customize Kernel Settings (NEW)
  [ ] Customize Vendor/User Settings (NEW)
  [ ] Update Default Vendor Settings (NEW)
```

Now hit `Exit` ▶ `Exit` ▶ `Save new kernel configuration...`, which should launch a new menu (the uClinux 2.6 kernel menu).

***uClinux Kernel Settings*** *C.3.2.1*

This design will use NFS extensively. The kernel is pre-configured for NFS version 3, but this version seemed troubled with bugs. Version 4 seems healthier.

```
--- Linux Kernel Configuration ---

File systems  --->
  Network File Systems  --->
    [*]   Provide NFSv4 client support (EXPERIMENTAL)
```

To be able to load custom modules, select these options

```
Loadable module support  --->
  [*] Enable loadable module support
  [*] Module unloading
```

When done exit the configuration menu with successive `Exit`'s and save the configuration.

***Vendor/User Settings*** *C.3.2.2*

Re-enter the initial menu by typing `make menuconfig`, but this time toggle `Customize Vendor/User Settings` in the `Kernel/Library/Defaults Selection` sub-menu. Exit and save.

This menu allows further customisation of the Linux system, as what settings to use and programs and library to compile in and load. Start by setting up networking address and specify host-name and root-password:

```
System Settings  --->
  Network Addresses  --->
    Static IP address: "192.168.1.10"
    Server IP address: "192.168.1.1"
  Default host name: "compet"
  Default root password: "compet"
```

There is also the possibility of setting up DHCP, for dynamic request and acquisition of an IP-address. However, for simplicity only static IP-addresses are currently used.

*Insmod* will be required be able to load custom modules. Furthermore, *sleep* is handy as well to be able to "pause" programs[6].

```
BusyBox -->
  [*] insmod
  [*] insmod: lsmod
  [*] insmod: rmmod
  [*] insmod: 2.6 and above kernel modules
  [*] sleep
```

When done, exit MenuConfig and save changes.

## *Building the Embedded Project* *C.4*

### Implementing the Hardware C.4.1

Xilinx XPS is simply a front-end to a library of scripts and executable binaries. When the project are to be built one may either execute "build" options in the XPS *Software* and *Hardware* menu, or launch the behind-the-scenes makefiles directly. The latter will be described here.

Enter a terminal and navigate to the embedded project folder (i.e. where the `.xmp`/`.mss`/`.mhs`-files are stored). Then, request the project makefiles from XPS (do not forget to source `settings.sh` first):

```
>> xps -nw system.xmp xps -nw system.xmp  (substitute <system> with the
    relevant project name)
% save make
% exit
```

Continue by building the bitstreams (hardware), libraries, FS-Boot, and initialise the BlockRAM. This may take a while.

```
>> make -f system.make bits                    % Generate bitstreams
>> make -f system.make libs                    % Generate library files
>> make -f system.make program                 % Build FS-boot
>> make -f system.make init_bram               % Initialise BRAM
```

[6] For example, in C simply use `system("sleep 5");` to pause a program for 5s.

### Board Specific Package C.4.2

During the hardware implementation process a Board Specific Package (BSP) was created (called `Kconfig.in` for kernel 2.6), which describes the hardware platform. This information is required by the bootloader and Linux kernel in order to correctly boot the system. Navigate to the embedded project root folder and type

```
>> petalinux-copy-autoconfig
```

to copy the BSP-file to the Linux kernel source tree.

### Compiling the Linux Kernel C.4.3

To compile the kernel, and go with with all the default answers to any question that might be asked, navigate to the `petalinux-dist` folder and type:

```
>> yes "" | make oldconfig dep all
```

When successful, the following files will be copied to `/tftpboot` (see C.1.3) and `software/petalinux-dist/images`:

| File Name | | Description |
|---|---|---|
| Linux Kernel | `image.bin` | Linux kernel image and root filesystem (binary format) |
| | `image.elf` | Linux kernel image and root filesystem (ELF format) |
| | `image.srec` | Linux kernel image and root filesystem (SREC format) |
| | `image.ub` | Linux kernel image and root filesystem (U-Boot format) |
| | `linux.bin` | Linux kernel only, no filesystem (binary format) |
| | `romfs.img` | The ROMFS image in binary format |
| U-Boot | `u-boot.bin` | The U-Boot image in binary format |
| | `u-boot.srec` | The U-Boot image in SREC format |
| | `u-boot-s.bin` | The relocatable U-Boot image in binary format |
| | `u-boot-s.elf` | The relocatable U-Boot image in ELF format |
| | `u-boot-s.srec` | The relocatable U-Boot image in SREC format |
| | `ub.config.img` | U-Boot platform configuration script in binary format |

*Table C.3 - PetaLinux kernel binaries [24]*

The embedded system is now built. What remains is to download it to the evaluation board and boot it.

## *Booting the Embedded System* C.5

When the hardware project was built, a `download.bit` file appeared in

the `implementation`-folder in the embedded project[7]. Ship this this to the FPGA, and observe the output from FS-Boot over RS323 (see C.1.5). Shortly after configuring the FPGA FS-Boot will request the image to load the next-stage bootloader, U-Boot.

```
1  ================================================
2  FS-BOOT First Stage Bootloader (c) 2006 PetaLogix
3  Project name: edk
4  Build date: Mar  5 2010 14:55:10  FS
5  Serial console: Uartlite
6  ================================================
7  FS-BOOT: System initialisation completed.
8  FS-BOOT: Booting from FLASH. Press 's' for image download.  % Press 's'
       here
9  FS-BOOT: Waiting for SREC image....
```

### U-Boot C.5.1

By not clicking 's' here, FS-Boot will attempt to boot from Flash, which is eventually the desired solution (see below). By some means provided by the RS232 interface, send the `u-boot.srec` file to the ML505.

```
1  FS-BOOT: Waiting for SREC image....
2  FS-BOOT: Use new image.
3  FS-BOOT: Booting image...
4  SDRAM :
5  Enabling caches :
6  Icache:OK
7  Dcache:OK
8  U-Boot Start:0x9ffc0000
9  Malloc Start:0x9ff80000
10 Board Info Start:0x9ff7ffd0
11 Boot Parameters Start:0x9ff6ffd0
12 FLASH: 32 MB
13 ETHERNET: MAC:00:0a:35:00:22:01
14
15 *** Warning - bad CRC, using default environment
16
17 Hit any key to stop autoboot:  0       % Hit 'b' here
18 U-Boot>
```

#### *Putting U-Boot in Flash* *C.5.1.1*

Before doing anything, one might want to run `flinfo` to see the Flash-memory layout. If TFTP is setup correctly, U-Boot may be put set to Flash by simply typing

[7] When ISE is used to build the embedded project, `<topmodule name>_download.bit` appears in the ISE project root folder instead. This is the case with the design described in this thesis.

```
U-Boot> run update_uboot
  eth0: Xilinx XPS LocalLink Tri-Mode Ether MAC #0 at 0x81C00000.
  1000BASE-T/FD
  TFTP from server 192.168.1.5; our IP address is 192.168.1.10
  Filename 'u-boot-s.bin'.
  Load address: 0x90000000
  Loading: ######################
  done
  Bytes transferred = 107008 (1a200 hex)
  .. done
  Un-Protected 2 sectors

  .. done
  Erased 2 sectors
  Copy to Flash... done
U-Boot>
```

This unprotects the first 2 blocks of Flash memory, and copies the image there. Note that the server and embedded system IP-address are those specified in the kernel configuration (C.3.2.2). To re-enable the protection (not necessary, but recommended) type

```
U-Boot> protect on 1:0-1              % General syntax: protect <on/off>
    <bank>:<block(s)>
  Protect Flash Sectors 0-1 in Bank # 1
  .. done
U-Boot>
```

Depending on the U-Boot image size, more than 2 blocks of memory may be required. For the ML505 evaluation board, only bank 1 exists.

Now the card will boot straight to the U-Boot menu every time the FPGA is configured with a `.bit`-file containing an embedded project and the FS-Boot bootloader.

### PetaLinux C.5.2

To run PetaLinux the kernel image must be downloaded to the onboard DDR2-RAM, and executed. To upload the image, use e.g. NFS:

```
U-Boot> nfs 0x90000000 192.168.1.5:/tftpboot/image.bin
  File transfer via NFS from server 192.168.1.5; our IP address is
      192.168.1.10
  Filename '/tftpboot/image.bin'.
  Load address: 0x90000000
  Loading: len bad 46 < 241
  ############################ ...
                              ...
  ################################################################
  done
```

```
  Bytes transferred = 4292752 (418090 hex)
U-Boot>
```

0x90000000 is the start address of the DDR2-RAM in the design in this project. These addresses equals the addresses listed in XPS aslong as the Microblaze is setup without a MMU. Now, execute the image:

```
U-Boot> go 0x90000000
( lots of output )
       ...
eth0: XLlTemac: We renegotiated the speed to: 1000
eth0: XLlTemac: speed set to 1000Mb/s
eth0: XLlTemac: Send Threshold = 24, Receive Threshold = 4
eth0: XLlTemac: Send Wait bound = 254, Receive Wait bound = 254
Starting portmap:
Starting thttpd:

compet login:                                    % Type 'root'
Password:                                        % Type 'compet'

#
```

Congratulations! This is, in essence, a working system!

***Putting Linux Kernel in Flash*** *C.5.2.1*

Yet again, use `flinfo` to see the Flash-layout. By default U-Boot is setup to look up the Linux kernel image at 0x8c080000. This may be changed by using the U-Boot environment commands, e.g. `setenv kernel_addr <address>` (or editing the PetaLinux configuration), but there is no reason to do so now. To send the kernel image to Flash, the normal procedure consists of first transferring the file to DDR2-RAM via NFS/TFTP, and then byte-copy it into Flash.

```
U-Boot> nfs 0x90000000 192.168.1.5:/tftpboot/image.ub
  File transfer via NFS from server 192.168.1.5; our IP address is
      192.168.1.10
  Filename '/tftpboot/image.ub'.
  Load address: 0x90000000
  Loading: ##################### ...
                                ...
  ################################################################
  done
  Bytes transferred = 2990288 (2da0d0 hex)
U-Boot>
```

Pay attention to the number of bytes transferred, as this number must be supplied to the byte-copy program. Optionally, one might want to verify the kernel image

```
U-Boot> imi 0x90000000

  ## Checking Image at 90000000 ...
    Image Name:   PetaLinux Kernel 2.6
    Image Type:   Microblaze Linux Kernel Image (uncompressed)
    Data Size:    2990224 Bytes =  2.9 MB
    Load Address: 90000000
    Entry Point:  90000000
U-Boot>
```

The ML505 Flash-memory is segmented into 256(+2) 128kB blocks of memory. In this project, the kernel image (`image.ub`) size is slightly more than 4MB, thus clearing 4-40 Flash-memory should be more than enough. Disable the protection, and erase, these blocks:

```
U-Boot> protect off 1:4-33
  Un-Protect Flash Sectors 4-40 in Bank # 1
  ............................. done

U-Boot> erase 1:4-40
  Erase Flash Sectors 4-40 in Bank # 1
  ............................. done

U-Boot>
```

Finally, perform the byte-copy and re-enable the protection:

```
U-Boot> cp.b 0x90000000 0x8c080000 0x4180d0     % Syntax: cp.b <from> <to
    > <size>
  Copy to Flash... done

U-Boot> protect on 1:4-40
              ...
```

Woila! The system should now boot PetaLinux without any manual interventions.

## *Hints and Tips* C.6

### NFS Development Share C.6.1

A very handy way of developing embedded-applications is by sharing the cross-compilation binaries between the host- and embedded system. If, say, `/share` is common for both the host and embedded system, then the development flow could be similar to this

| *Terminal 1* (host machine) | *Terminal 2* (development board) |
|---|---|
| 1. Write the code.<br>2. Compile and put binaries in `/share` | 1. Navigate to `/share`<br>2. Run the binary file. |

A way to do this is by directly editing the PetaLinux filesystem, situated in `petalinux-dist/romfs`. All "S-files" in `petalinux-dist/romfs/etc/rc.d` are launched upon boot, so create a file there called `S99nfs` with the following contents:

```
1 echo '  NFS: Mounting 192.168.1.5:/share as /mnt'
2 mount -t nfs -o tcp,nolock 192.168.1.5:/share /mnt
```

Assign full permissions to this file to rule that out of the list of potential party-crashers:

```
1 >> chmod 777 petalinux-dist/romfs/etc/rc.d/S99nfs
```

Remake the kernel, upload it to the card, and watch the RS232 output:

```
1  U-Boot> go 0x90000000
2  ( lots of output )
3         ...
4  Starting thttpd:
5    NFS: Mounting 192.168.1.5:/share as /mnt
6
7
8  compet login:  root
9  Password:      compet
10
11 # cd /mnt
12 # ls
13   ( list of all binaries )
```

This is the link currently used to launch applications on the embedded device, and to send data back. The data acquisition programs store the data in the `/share`-folder, which is actually located on the host-machine. This is by means a high-performance solution, but it simple and reliable.

### Cross-Compilation C.6.2

To compile code for the embedded system, first setup the cross-compilation environment by sourcing `settings.sh`. Then compile as usual with the `microblaze-uclinux-*` commands. E.g. to compile `hello.c`, and put the binary in `/share`, do

```
1 >> microblaze-uclinux-gcc -W hello.c -o /share/hello
2 >> chmod a+x /share/hello
```

Now launch it from the embedded environment (e.g. by first logging in with `telnet`), and watch the magic.

# D

# ISE/EDK Messages

ISE and EDK outputs a rather verbose list of info-, warning- and error-messages[1]. While the latter halts the implementation process completely, and always must be resolved, the two former are of less severity and may often be ignored. However, good design practise dictates that these are carefully evaluated to ensure that synthesis and simulation coincide. When the simulation shows promising results but the FPGA implementation fails miserably, the reason may usually be found somewhere in the mentioned messages. Despite my continuous efforts to keep the code clean and unambiguous, some messages simply will not vanish. I intend to comment on a selection of these here.

## *Warnings* D.1

⚠*VHDL not supported as a language.*
This seems only to affect Linux systems, and *may* cause the implementation process to fail. Clear the `LANGUAGE` environment variable[2] prior to starting ISE/EDK to bypass this bug.

⚠*ProjectMgmt - Circular Reference: Architecture/fe_ch_submux/rtl*
This is caused by the recursive nature of the SubMux (see 3.5.2.1), which - according to AR#20480[3] - seems not to be supported by the Project Navigator, but should cause no problems with XST.

⚠*No primary, secondary unit in the file ".../icon.vhd" (or "ila.vhd"/vio.vhd" . Ignore this file from project file ".../top_vhdl.prj".*
These are simulation files for the ChipScope cores, automatically generated by CoreGen in the IP creation process. The warnings seems pointless (these files are never added to the project, nor ever used), and can be safely ignored.

[1] ISE and EDK versions used: 10.1.03.

⚠*Xst:2211 - "/path/to/somefile.vhd" line XX: Instantiating black box module <module name>.*

Look into AR#9838[4] if you wish to suppress these, which are there for notification only.

⚠*Xst:819 - "/path/to/somefile.vhd" line XX: One or more signals are missing in the process sensitivity list...*

The sensitivity list should - if the logic is synchronous - contain the clock and the reset (if asynchronous), and - if the logic is combinatorial - all input signals. However, Xst will usually implement correct logic even if the list is incomplete. In some cases, it makes sense leave it incomplete. Consider an array of records, for instance, and you wish a process to be sensitive to all signals in this array. Writing all of these into the sensitivity list is inflexible, tedious, and makes the code less readable[5].

⚠*Xst:2404 - FFs/Latches <signal name> (without init value) have a constant value of 0 in block <block name>.*

Signals where one or more bits have a constant value does not need logic, but can simply be tied to static logical '0' or '1'. The "channel number" variable in the event data is a typical "victim" of this treatment since it is set in synthesis, and never changed afterwards.

⚠*Due to other FF/Latch trimming, FF/Latch <signal name> (without init value) has a constant value of 0 in block <block name>. This FF/Latch will be trimmed during the optimization process.*

Pay particular attention to these messages, as they often indicate a broken signal path somewhere.

## *Errors* D.2

⛔*coreutil - License for component <xps_ll_temac_v1> not found*

This is caused by a bug in EDK. The ComPET readout project uses a hard Ethernet MAC that should not require a licence, but due to this bug it needs one anyhow. Fortunately, it is free. See AR#32054[6].

# List of Tables

# List of Figures

# Bibliography


## Articles

[1] Barbara J. Fueger et al. "Impact of Animal Handling on the Results of 18F-FDG PET Studies in Mice". In: *Nuclear Medicine* 47 (2006).

[2] M. Hoffman et al. "18F-fluoro-deoxy-glucose positron emission tomography (18F-FDG-PET) for assessment of enteropathy-type T cell lymphoma". In: ().

[3] Armin Kolb et al. "Evaluation of Geiger-mode APDs for PET block detector designs". In: *Physics in Medicine and Biology* 55.7 (2010), p. 1815.

[4] Craig S Levin and Edward J Hoffman. "Calculation of positron range and its effect on the fundamental limit of positron emission tomography system spatial resolution". In: *Physics in Medicine and Biology* 44.3 (1999), p. 781.

[5] Craig S. Levin and Habib Zaidi. "Current Trends in Preclinical PET System Design". In: 2.2 (Apr. 2007), pp. 125–160. ISSN: 1556-8598. DOI: 10.1016/j.cpet.2007.12.001.

## Books

[6] Steve Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007. ISBN: 0470054379.

[7] Clive Maxfield. *The Design Warrior's Guide to FPGAs*. Orlando, FL, USA: Academic Press, Inc., 2004. ISBN: 0750676043.

## Inproceedings

[8] L. Arpin et al. "A Sub-Nanosecond Edge Detection System using embedded FPGA fabrics". In: *Real Time Conference, 2009. RT '09. 16th IEEE-NPSS*. Oct. 2009, pp. 299 –303. DOI: 10.1109/RTC.2009.5321977.

[9] Shuai Che et al. "Accelerating Compute-Intensive Applications with GPUs and FPGAs". In: *SASP '08: Proceedings of the 2008 Symposium on Application Specific Processors*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 101–107. ISBN: 978-1-4244-2333-0. DOI: http://dx.doi.org/10.1109/SASP.2008.4570793.

## Online

[10] *AxPET Collaboration Twiki*. URL: https://twiki.cern.ch/twiki/bin/view/AXIALPET/WebHome.

[11] Xilinx. *Extended Spartan-3A Family Overview v1.0.1*. Jan. 2010. URL: http://www.xilinx.com/support/documentation/data_sheets/ds706.pdf.

[12] Xilinx. *Extended Spartan-3A FPGA Product Table*. URL: http://www.xilinx.com/images/s3a/extendedspartan3a-product-table.jpg.

[13] Xilinx. *Spartan-6 Family Overview v1.4*. Mar. 2010. URL: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf.

[14] Xilinx. *The Xilinx homepage*. URL: http://www.xilinx.comSili.

[15] Xilinx. *Virtex-5 Data Sheet v5.3: DC and Switching Characteristics*. May 2010. URL: http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf.

[16] Xilinx. *Virtex-5 Family Overview v5.0*. Feb. 2010. URL: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf.

[17] Xilinx. *Virtex-5 User Guide v5.3*. May 2010. URL: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[18] Xilinx. *Virtex-6 Family Overview v2.2*. Jan. 2010. URL: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[19] Xilinx. *XAPP860: 16-Channel, DDR LVDS Interface with Real-Time Window Monitoring*.

### *Phd. Thesis'*

[20] DL Bailey. "Quantification in 3D Positron Emission Tomography". PhD thesis. University of Surrey, United Kingdom, 1996.

# Glossary

| Notation | Description |
|---|---|
| ADC | Analog to Digital Converter. |
| APD | Avalanche Photodiode. |
| ASIC | Application Specific Integrated Circuit. |
| BSP | Board Specific Package. |
| CLB | Configurable Logic Block (see 2.3.2.2). |
| CMT | Clock Manager Tile. |
| CRU | Clock-Reset Unit. |
| CT | Computed Tomography. |
| DCI | Digitally Controlled Impedance (see 2.3.2.1). |
| DCM | Digital Clock Manager (see 2.3.2.3). |
| DDR | Dual Data Rate. |
| DHCP | Dynamic Host Configuration Protocol. |
| DLL | Delay-Locked Loop . |
| DMA | Direct Memory Access. |
| DNS | Domain Name System. |
| DOI | Depth-of-Interaction. |
| DSP | Digital Signal Processing. |
| EDK | Embedded Development Kit. |
| FF | Flip-flop. |
| FIFO | First In, First Out. |
| FoV | Field of View. |
| FPGA | Field Programmable Gate Array. |
| FWHM | Full Width at Half Maximum. |
| GAPD | Geiger-mode Avalanche PhotoDiode. |
| GPIO | General Purpose IO. |
| GPU | Graphics Processing Unit. |
| HDL | Hardware Description Language. |
| ILA | Integrated Logical Analyser. |
| IOB | Input/Output Block (see 2.3.2.1). |
| IP | Intellectual Property. |

| Notation | Description |
|---|---|
| ISE | Integrated Synthesis Environment. |
| ISERDES | Input SERialiser/DESerialiser. |
| JTAG | Joint Test Action Group. |
| LED | Light Emitting Diodes. |
| LOR | Line Of Response. |
| LUT | LookUp Table (see 2.3.2.2). |
| LVDS | Low Voltage Differential Signaling. |
| LYSO | $Lu_{1.8}Y_{0.2}SiO_5(Ce)$. |
| MAC | Medium Access Controller. |
| MAC | Multiply-Accumulate. |
| MGT | Multi-Gigabit Transceiver. |
| MMU | Memory Management Unit. |
| MPMC | Multi-Port Memory Controller. |
| MRI | Magnetic Resonance Imaging. |
| MUX | MUltipleXer. |
| NFS | Network FileSystem. |
| PCB | Printed Circuit Board. |
| PCI | Peripheral Component Interconnect. |
| PET | Positron Emission Tomography. |
| PLB | Processor Local Bus. |
| PLD | Programmable Logic Device. |
| PLL | Phase Locked Loop. |
| PMT | PhotoMultiplier Tube. |
| RAM | Read Access Memory. |
| RF | Radio Frequency. |
| SDR | Single Data Rate. |
| SERDES | SERialiser/DESerialiser. |
| SNR | Signal to Noise Ratio. |
| SPECT | Single Photon Emission Computed Tomography. |
| SRAM | Static Random Access Memory. |
| SSH | Secure Shell. |
| TCL | Tool Command Language. |

| Notation | Description |
|---|---|
| TDC | Time-to-Digital Converter. |
| TFTP | Trivial File Transfer Protocol. |
| ToF | Time of Flight. |
| TOT | Time Over Threshold. |
| UDP | User Datagram Protocol. |
| USB | Universal Serial Bus. |
| VCS | Version Control System. |
| VHDL | Very high speed Hardware Description Language. |
| VIO | Virtual Input/Output. |
| WLS | WaveLength Shifter. |
| XPS | Xilinx Platform Studio. |
| XST | Xilinx Synthesis Technology. |